UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI
SPECIALIZAREA: TEHNOLOGIA INFORMAȚIEI

LUCRARE DE DIPLOMĂ

Coordonator științific
S.L. Dr. Inginer Andrei Stan

Absolvent
Petrică Daniel Toderică

Iași, 2016

UNIVERSITATEA TEHNICĂ „Gheorghe Asachi” din IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DOMENIUL: CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI
SPECIALIZAREA: TEHNOLOGIA INFORMAȚIEI

Sistem de Operare în Timp Real

pentru procesoare cu două nuclee

LUCRARE DE DIPLOMĂ

Coordonator științific
S.L. Dr. Inginer Andrei Stan

Absolvent
Petrică Daniel Toderică

Iași, 2016

DECLARAȚIE DE ASUMARE A AUTENTICITĂȚII LUCRĂRII DE LICENȚĂ

Subsemnatul(a) Toderică Petrică Daniel,
legitimat(ă) cu CI seria VS, nr. 507095, CNP 1921107375503,
autorul lucrării Sistem de Operare în Timp Real pentru procesoare cu două nuclee ,

elaborată în vederea susținerii examenului de finalizare a studiilor de licență organizat de către Facultatea de Automatică și Calculatoare din cadrul Universității Tehnice „Gheorghe Asachi" din Iași, sesiunea iulie a anului universitar 2015-2016, luând în considerare conținutul Art. 34 din Codul de etică universitară al Universității Tehnice „Gheorghe Asachi" din Iași (Manualul Procedurilor, UTI.POM.02 – Funcționarea Comisiei de etică universitară), declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române (legea 8/1996) și a convențiilor internaționale privind drepturile de autor.

Data Semnătura

30.06.2016

Cuprins

Sistem de Operare în Timp Real

pentru procesoare cu două nuclee

Petrică Toderică

Rezumat

Este bine știut că principala componentă a unui sistem de operare este planificatorul, care poate influența performanțele și caracteristicile comportamentale ale platformei fizice pe care este executat codul.

În cazul unui sistem de operare în timp real, planificatorul este proiectat pentru a asigura un timp de execuție determinist. Acest lucru este necesar, în special, pentru sistemele încorporate deoarece aceste sisteme trebuie să răspundă la evenimente in timpi foarte bine definiți(eng. deadline). Pentru asigurarea unei astfel de metrici, multe dintre sistemele de operare în timp real(ex. FreeRTOS) folosesc un sistem de prioritizare al *task*-urilor(eng.)(pentru acest tip de sistem de operare, programele sau firele de execuție sunt denumite *task*-uri), dând utilizatorului astfel posibilitatea de a asigna fiecărui task o prioritate la execuție.

Planificatorul dezvoltat in cadrul acestei lucrări folosește un astfel de sistem de prioritizare. Așadar, fiecare task are o prioritate de execuție, un identificator numeric, un bloc de control sau un context(unde sunt salvate valorile registrelor procesorului) și o stare (*disponibil* - task-ul este disponibil pentru a fi executat, *in execuție* - task-ul deține controlul procesorului, *suspendat* – când task-ul se autosuspendă și *in asteptare* – când task-ul așteaptă pentru o resursă care nu este disponibilă).

Task-urile disponibile pentru a fi executate sunt ținute într-o coadă de priorități ce folosește un *array*(eng.) și operații de inserare/ștergere bazate pe proprietatea de *min-heap* a unui vector. Task-urile care nu sunt disponibile sunt ținute într-o coadă de așteptare obișnuită. Când un task din lista de așteptare devine disponibil, el este scos din aceasta listă și introdus în coada de task-uri disponibile. Componentele principale ale planificatorului sunt reprezentate de cele două cozi menționate și o referință către task-ul care este executat la un anumit moment de timp.

Planificatorul verifică periodic dacă există un task mai prioritar disponibil pentru execuție. Verificarea se face pe funcția ce tratează întreruperea *SysTick Timer*-ului(eng.). Această întrerupere are loc la o perioadă bine definită, dar care poate fi ajustată(valoarea actuală este de 10ms).

Inițial, toate task-urile sunt considerate pregătite spre a fi lansate în execuție. Se alege task-ul cel mai prioritar și se lansează in execuție. Dacă la un anumit moment de timp task-ul curent așteaptă pentru o resursă care nu este disponibilă sau dacă task-ul își autosuspendă execuția pentru o anumită perioadă de timp, atunci acesta este pus în lista de așteptare până când execuția îi va putea fi reluată(resursa pentru care așteaptă devine disponibilă sau expiră perioada de autosuspendare). Se salvează contextul acestui task iar apoi se extrage următorul task disponibil. Se restaurează contextul acestui task și astfel se continuă execuția. Dacă nici un task nu este disponibil, atunci se consideră că task-ul Idle va fi următorul task executabil. Task-ul Idle este un task care nu execută practic nici o operație complexă, ci doar așteaptă la infinit și care este adăugat automat de către planificator, cu prioritatea cea mai mică din sistem și care este rulat atât timp cât nici un alt task nu este disponibil.

Așa cum se menționează în titlul lucrării, se dorește dezvoltarea unui sistem de operare in timp real care să ruleze pe două nuclee. Platforma hardware folosită este bazată pe un micro procesor dual LPCXpresso43S37(un nucleu este de tipul ARM Cortex M4 iar celălalt de tipul ARM Cortex M0), care operează la o frecvență de 204 MHz, dispune de o memorie flash de 1MB, 136 KB SRAM, interfețe GPIO, seriale, analogice și digitale. Având în vedere arhitectura diferită a celor două nuclee, decizia optimă a fost ca fiecare nucleu să aibă propriul planificator.

Pentru fiecare planificator, sunt dezvoltate componente specifice accesului sincronizat (mutecși și semafoare), cărora le sunt atașate cozi de așteptare. De asemenea sunt dezvoltate instrumente pentru comunicare între task-uri(bazate pe conceptul de *shared memory(eng.)* și sincronizare cu semafoare), funcții pentru accesul la zone de cod critice precum și instrumente pentru accesul controlat al resurselor IO (afișaj LCD, joystick, senzor de temperatura, senzor inerțial, porturile seriale).

Fiecare nucleu al sistemului de operare este împărțit in 7 module: Dispatcher, IO, Library, Scheduler, Syncronization, System, TaskManager. Fiecare dintre aceste module are un rol bine definit, iar o modificare în unul dintre module are un impact foarte mic asupra celorlalte.

În mare parte, codul este scris in limbajul C++ pentru a putea fi folosite anumite facilități ale acestuia precum încapsularea și existenta constructorilor(ex: se inițializează automat blocul de control al fiecărui task). Există porțiuni de cod scris in limbajul C, în special codul care este oferit de furnizorii platformei de dezvoltare, și chiar porțiuni de cod scrise in limbaj de asamblare(codul care salvează contextul unui task precum și cel care îl restaurează). Acest cod este diferit pentru fiecare dintre cele două nuclee.

Pentru testarea codului implementat, s-au creat diferite task-uri ce folosesc facilitățile oferite de sistemul de operare: folosirea semafoarelor și a memoriei comune, comunicare între task-uri sau afișaj LCD.

Proiectul respectă cerințele impuse inițial, însă oricând acesta poate fi îmbunătățit prin adăugarea de funcționalități și servicii noi, extinderea suportului pentru diferite procesoare sau optimizarea planificării.

Capitolul 1. Fundamentarea teoretică și documentarea biografică pentru tema propusă

1.1. Sisteme de operare de uz general

Sistemul de operare este un pachet de programe [1] care asigură gestionarea eficientă a resurselor fizice și logice ale unui sistem de calcul precum și o interfață între utilizator și acel sistem simplificând astfel accesul la acel sistem și extinzând setul de operații disponibile.

Funcțiile unui sistem de operare sunt:

- Interfața cu utilizatorul
- Gestiunea fișierelor
- Gestiunea perifericelor
- Gestiunea memoriei
- Gestiunea proceselor
- Tratarea erorilor
- Gestiunea sistemului

Interfața cu utilizatorul determină în mare măsură acceptarea de către utilizatori a unui sistem de operare. Această funcție componentă este reprezentată pe de o parte de comenzile suportate și pe de altă parte de apelurile se sistem care definesc practic arhitectura unui sistem de operare și care ajută la implementarea aplicațiilor sistem.

Fișierele reprezintă forma în care sunt păstrate informațiile într-un sistem de calcul. Sistemul de operare trebuie să suporte operații pentru crearea, ștergerea, citirea și scrierea fișierelor precum și organizarea și controlul accesului la fișiere.

Perifericele reprezintă unul dintre modulele care realizează comunicarea sistemului fizic cu utilizatorul. Sistemul de operare trebuie să realizeze operații de transfer de informație între periferice și sistem dar și tratarea posibilelor erori.

Memoria este componenta cea mai sensibilă a unui sistem. O parte este rezervată sistemului de operare, însă cea mai mare parte este disponibilă pentru programele utilizator. Sistemul de operare trebuie să asigure protecția si partajarea memoriei între programele solicitante.

Legat de tratarea erorilor, sistemul de operare trebuie sa reacționeze la o diversitate de erori(chiar și fizice) și nu trebuie sa aibă mai multe erori decât programele utilizator.

Componenta principală a unui sistem de operare o reprezintă *kernel*-ul(eng.) sau nucleul. Nucleul conține funcții și proceduri care tratează planificarea proceselor, tratarea erorilor, tratarea inițială a apelurilor de sistem. Nucleul ocupă o zonă fixă a memoriei, incluzând adresele cele mai mici deoarece aici se găsesc vectorii de întrerupere. Planificarea proceselor este realizată de către planificator, rolul acestuia fiind de a decide ce program poate rula la un moment dat in sistem. Sistemul de operare trebuie să asigure execuția tuturor proceselor. Un proces este un program încărcat și rulat de către procesor, cu scopul de a obține/prelucra informație.

Este bine știut faptul că se dorește ca mai mult de un program să ruleze la un moment dat. Considerăm un exemplu cotidian: pe stația de lucru vrem să navigăm pe diferite site-uri în timp ce ascultăm muzică și primim notificări pentru emailuri. În acest exemplu avem trei programe care trebuie să fie executate în paralel: *browser*-ul(eng.), clientul de email și *player*-ul(eng.) audio. Această nevoie poate fi satisfăcută folosind un sistem cu un procesor multi nucleu. Însă ce se întâmplă când

numărul de procese depășește numărul de nuclee fizice ale sistemului de calcul ? Intervine sistemul de operare.

1.2. Paralelism fizic versus paralelism virtual

Termenul de paralelism se referă la execuția simultană a mai multor procese. Așa cum am menționat mai sus, din punct de vedere fizic acest lucru este posibil doar dacă sistemul deține o unitate de prelucrare cu mai multe nuclee. Fiecare program este executat de către un nucleu fizic, realizând așadar paralelism fizic.

Necesitatea existenței unui sistem de operare este imediat observabilă, spre exemplu, când numărul programelor ce se dorește a fi lansate în execuție depășește numărul de nuclee fizice. Sistemul de operare creează iluzia că sistemul fizic deține un număr nelimitat de nuclee făcând astfel posibilă execuția în paralel a tuturor programelor dorite. Sistemul de operare realizează o virtualizare a procesorului, realizând astfel paralelismul virtual. Cum este realizată virtualizarea ?

Rulând un proces, apoi oprindu-l și rulând altul și așa mai departe, sistemul de operare realizează virtualizarea procesorului. Această tehnică de bază, cunoscută ca *partajarea timpului* (eng. *time sharing*) procesorului, dă posibilitatea utilizatorului de a rula un număr mare de procese in același timp. Însă există o penalizare de performanță, fiecare proces rulând mult mai greu. Putem observa în Figura **Error! No text of specified style in document.**1.1 penalizarea de performanță:

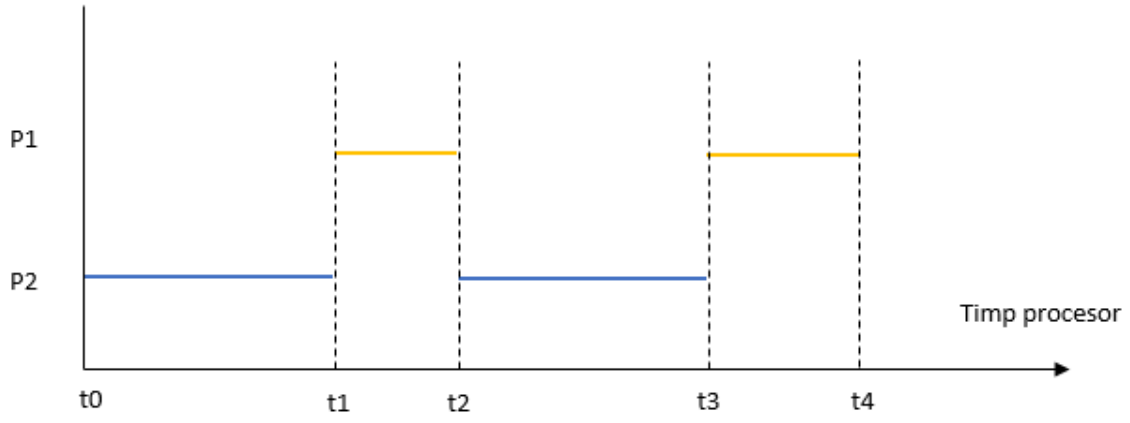

*Figura **Error! No text of specified style in document.**1.1 Paralelism virtual*

Dacă ar fi executat independent, durata de execuție a procesului P1 ar fi egală cu suma dintre timpii scurși intre momentele t1 si t2 și momentele t3 și t4. In condițiile virtualizării, se adaugă și timpul de așteptare pentru rularea lui P2, dintre momentele t2 și t3.

Partajarea timpului procesorului este tehnica cea mai utilizată de sistemele de operare pentru a împărți o resursă. Pentru a implementa virtualizarea procesorului, un sistem de operare trebuie să definească mecanisme *low-level*(eng.), care implementează funcționalități atomice, utilizate in special pentru a implementa schimbarea de context(*context switch* eng.) sau pentru a proteja secțiuni critice de cod.

Schimbarea de context reprezintă procedeul prin care sistemul de operare decide suspendarea execuției unui proces la un moment dat, înainte de terminarea efectivă a acestuia, pentru a permite reluarea execuției altui proces. După cum am menționat, un nucleu fizic poate executa la un moment dat doar instrucțiunile asociate unui singur proces. Decizia de a opri temporar execuția unui proces aparține componentei inteligente a sistemului de operare numită planificator. Aceasta are rolul de a decide ce program/proces are dreptul de a fi executat de către procesor la un moment dat. În funcție de politica folosită, sistemul poate avea un comportament diferit. Politica unui planificator

se referă la algoritmul care decide cantitatea de timp de procesor alocată fiecărui proces precum și momentele când acest timp este alocat.

1.3. Algoritmi de planificare

Pentru un utilizator, primul criteriu de performanță al unui sistem îl reprezintă durata de execuție a unui program. În cartea sa [1], Remzi numește această durată ca timp de execuție, care matematic este durata de timp scursă între momentul în care programul este executat complet și momentul în care acesta a ajuns in sistem. Timpul de execuție este așadar o metrică de performanță. În funcție de algoritmul folosit de planificator pentru minimizarea acestei metrici, există o serie de abordări pentru implementare:

- FIFO – *First In First Out*(eng.) – sau FCFS – *First Come First Served*(eng.) – prin traducere liberă, primul venit primul servit, acest algoritm presupune că procesele vor fi executate complet, fără a fi întrerupte, in ordinea in care au ajuns in sistem. Această abordare este una extrem de simplă din punct de vedere al implementării și al înțelegerii. Acest tip de abordare funcționează optim pentru programe cu durate de execuție mici. În cazul în care durata de execuție a unui proces este extrem de mare, atunci execuția proceselor care vor ajunge in sistem după cel menționat va fi întârziată extrem de mult, ceea ce duce la probleme de performanță.
- SJF – *Shortest Job First*(eng.) – cel mai scurt proces disponibil va fi executat, in totalitate, primul. Performanțele acestui tip de abordare sunt clar superioare celui precedent. Însă ce se întâmplă dacă la sosirea unor noi task-uri, un task cu o execuție foarte lungă este deja pornit in sistem ? Vor apărea din nou probleme de performanță.
- STCF – Shortest Time to Completion First – acest algoritm rezolvă problema precedentă prin suspendarea execuției task-ului curent, dacă acesta are o durată foarte mare, și lansarea in execuție a task-urilor mai scurte. Apare o nouă metrică de performanță și anume timpul de răspuns. Timpul de răspuns este timpul scurs intre momentul sosirii procesului in sistem si momentul in care acesta este planificat pentru a fi lansat in execuție. Din punctul de vedere al acestei metrici, STCF nu este un algoritm optim, deoarece procesul cel mai lung va avea un timp de răspuns extrem de mare.
- RR – *Round Robin* – fiecărui proces ii este alocată o cuanta de timp de procesor. Se lansează primul job in execuție, se execută o cuantă de timp iar apoi este suspendat. Se rulează apoi alt proces o cuantă de timp și se suspendă. După ce tuturor proceselor le-au fost alocate cate o cuantă de timp, se reia execuția fiecăruia alocându-se câte o nouă cuantă de timp. Se repetă procedeul până la terminarea tuturor proceselor.

Tipul de planificator folosit determină performanțele și comportamentul sistemului de operare: pe sistemele Linux, planificatorul asigură fiecărui utilizator/program un timp potrivit din procesor. Pe Windows, planificatorul asigură utilizatorului un sistem care reacționează prompt.

1.4. Sisteme de operare în timp real

Un sistem de operare in timp real este un sistem de operare special construit pentru a asigura aplicațiilor un timp de execuție determinist.

În cazul unui sistem de operare în timp real, planificatorul este cel proiectat pentru a asigura un timp de execuție determinist. Acest lucru este necesar, în special, pentru sistemele încorporate deoarece aceste sisteme trebuie să răspundă la evenimente in timpi foarte bine definiți(*deadline*): un calculator care controlează sistemul de protecție airbag al mașinii trebuie să analize diferite semnale primite de la senzori. Acțiunea acestui sistem trebuie să fie imediată pentru a nu pune in pericol viața

unor persoane. Aici este necesară existența unui sistem de operare de timp real. Pentru a putea viziona pe un dispozitiv mobil un videoclip, prelucrarea întârziată a unor pachete multimedia duce la un ușor disconfort din partea utilizatorului în ceea ce privește experiența de vizualizare. Aici , un sistem de operare de uz general este suficient.

Funcțiile unui sistem de operare în timp real:

- Execuția corectă a tuturor proceselor(task-urilor)
- Asigurarea unui timp de răspuns strict corelat cu cerințe de timp impuse din afara sistemului de timp real
- Posibilitatea de execuție simultană(paralelism virtual) a unor procese secvențiale care cooperează și rulează pe principiul distribuirii timpului de execuție al procesorului
- Realizarea protecției resurselor comune ale sistemului de timp real
- Realizarea comunicației între task-uri
- Sincronizarea proceselor secvențiale
- Interfața dintre sistemul de timp real și utilizator prin intermediul unor periferice

Aceste funcții de bază constituie nucleul sistemului de operare de timp real. După cum am menționat mai sus, sistemul de operare de timp real trebuie să permită paralelism virtual, așadar acest tip de sistem de operare este unul *multitasking*(eng).

Componentele principale ale unui sistem de operare de timp real:

- Planificatorul – componenta care decide care este task-ul care trebuie executat
- Dispecerul – componenta care realizează schimbul de context
- Structuri de date: cozi, liste, semafoare, mutexi, comunicație intre procese etc

Contextul unui task sau blocul de control al unui task reprezintă totalitatea datelor ce sunt salvate înaintea schimbării de context, specifice unui anumit task, pentru a putea fi folosite pentru reluarea corectă a execuției task-ului. Ex.: Presupunem că procesorul se află în mijlocul unui calcul matematic complex. Valorile unor variabile sunt stocate în registrele procesorului. Planificatorul decide că un alt task trebuie executat, astfel încât task-ul curent este suspendat. După terminarea execuției task-ului nou, se revine la executarea primului task. Dacă valorile registrelor au fost modificate, atunci calculul matematic va da rezultate eronate. De aceea este nevoie de salvarea contextului unui task înainte de a fi suspendat de către planificator. Așadar, contextul unui task este reprezentat practic de valorile din registrele microprocesorului, atât generale cât și specifice.

În contextul unui sistem de operare în timp real, un task este o funcție, cu o bucla infinită de obicei, care este executată independent și care se comportă ca un fir de execuție. În interiorul acelei bucle infinite, se apelează funcții de sistem care pot permite și altor task-uri să ruleze.

Procedeul prin care unui task ii este suspendată temporar execuția în favoarea altui task se numește preempție. Despre un sistem de operare care suportă un astfel de comportament se spune că este preemtiv, iar despre un task care a fost suspendat se spune ca a fost preemptat.

1.4.1. Sistemul de operare în timp real Ucos

Sistemul de operare de timp real uCOS(Micro Controller Operating System) a fost dezvoltat de către Jean J. Labrosse, prima versiune fiind publicată in 1991. Este un sistem de operare preemptiv, bazat pe priorități, scris în mare parte folosind limbajul C, fiind destinat sistemelor încorporate.

Această versiune a sistemului de operare permite definirea de funcții care se pot executa independent ca fire de execuție independente. Fiecărui fir de execuție sau task i se asignează o prioritate. Totdeauna task-urile de pioritate mai mică pot fi preemptate de task-uri de prioritate mai mare, acestea din urmă folosind funcții de sistem cum ar fi delay() pentru a da șansa de a fi executate și task-urile mai puțin prioritare. Sunt definite servicii pentru administrarea task-urilor, comunicarea între task-uri, administrarea memoriei dar și servicii de temporizare.

Există o constrângere pentru task-urile care rulează sub acest sistem de operare: fie sunt scrise ca o buclă infinită care nu returnează niciodată, fie la sfârșitul execuției task-ul apelează o funcție de sistem prin care este șters de către nucleul sistemului de operare.

După succesul avut cu această versiune, sunt dezvolte alte două versiuni ale sistemului se operare: uC/OS-II și uC/OS-III .

uC/OS-II a fost introdus in anul 1998 ca fiind un produs comercial și un sistem de operare portabil, scalabil, preemptiv, de timp real, determinist și *multitasking*(eng.). Suportă până la 255 de fire de execuție și ocupă un spațiu ajustabil prin scoaterea funcționalităților care nu sunt necesare(de la 5 KB până la 24 KB).

Portabilitatea sistemului de operare este asigurată prin rescrierea in limbaj de asamblare a bucăților de cod specifice unui anumit microprocesor. În general, codul scris in limbaj de asamblare este codul care realizează salvarea contextului precum și reîncărcarea acestuia, dar și funcții de sistem care permit mascarea întreruperilor sau activarea lor. Având in vedere această caracteristică, acesta este folosit in domenii precum aviație, echipamente medicale, telefonie mobilă, robotică industrială sau construcții de mașini.

În contextul acestui sistem, task-urile pot avea următoarele stări: Inactiv, Disponibil, În execuție, In așteptare, Întrerupt.

Nucleul este responsabil de administrarea task-urilor și de comunicarea între task-uri. Serviciul fundamental furnizat de acesta este schimbul de context, acest serviciu executându-se la comanda unui planificator. Fiind un sistem de operare bazat pe priorități, totdeauna procesorul va fi alocat task-ului cu prioritatea cea mai mare.

Alegerea priorității task-urilor este făcută de către dezvoltatorul aplicației. Decizia se bazează pe faptul că task-urile cu rata de execuție cea mai mare trebuie să aibă prioritatea cea mai mare și invers. De asemenea, dezvoltatorii pot folosi instrumente pentru crearea task-urilor, ștergerea acestora, verificarea stivei alocate unui task, obținerea de informații precum și schimbarea stării unui task.

Pentru a evita fragmentarea memoriei, aplicațiile pot folosi doar blocuri de dimensiune fixă obținute dintr-o singură partiție continuă. Toate blocurile au aceeași dimensiune. Acest mic dezavantaj este introdus pentru a realiza operațiile de alocare și dealocare in timp constant bine determinat.

uC/OS-III a fost introdus in anul 2009 ca o funcționalitate adusă precedentei versiuni. Această versiune oferă aceleași funcționalități, singura diferență majora fiind numărul de task-uri

disponibile. Acesta acoperă un număr nelimitat de priorități și task-uri, toate fiind constrânse doar de dimensiunea memoriei.

Pentru a rezolva problema planificării task-urilor cu aceleași priorități, se folosește algoritmul Round Robin.

Comunicarea intre procese a fost îmbunătățită. In general, task-urile pot comunica prin variabile globale sau prin trimitere de mesaje. Utilizarea variabilelor globale implică accesul exclusiv la resurse, realizat prin folosirea semafoarelor. Trimiterea de mesaje se poate face direct, fiecărui task din această versiune a sistemului de operare fiindu-i atașată implicit o coadă de mesaje, sau prin intermediul unei cozi de mesaje externe care împarte mesajele către task-urile corespunzătoare.

Așadar, funcțiile principale oferite de sistemul de operare de timp real uC/OS sunt enumerate în continuare: crearea task-urilor, modificarea priorității unui proces aflat in execuție, distrugerea task-ului aflat in execuție, comunicație intre procese și asigurarea accesului exclusiv la resurse.

1.4.2. Sistemul de operare în timp real FreeRTOS

FreeRTOS [2] este un sistem de operare de timp real pentru dispozitive încorporate, care a fost portat pe mai mult de 35 de microprocesoare și care o fost utilizat chiar și în spațiu. Așa cum se subînțelege din denumire, acesta este distribuit sub GPL(General Public License), însă cu o singură excepție: codul care aparține utilizatorului rămâne privat cu condiția de a considera codul nucleului public.

FreeRTOS este proiectat pentru a fi simplu și redus ca spațiu ocupat. Nucleul este implementat în 4 fișiere. Limbajul de programare folosit pentru implementare este limbajul C, dar, așa cum este de așteptat, există porțiuni de cod scrise în limbaj de asamblare care realizează operații specifice arhitecturii platformei fizice folosite(rutine folosite de planificator).

Alături de funcționalitatea de bază, administrarea procesorului, sistemul de operare FreeRTOS pune la dispoziție obiecte de sincronizare(mutexi, semafoare) dar și temporizatoare. Ca și în cazul sistemului uC/OS, planificarea task-urilor se face folosind priorități.

Dezavantajele acestui sistem de operare sunt reprezentate de lipsa unor facilități precum existența *driver*-elor(eng.), administrarea avansată a memoriei, existența comunicației de rețea sau posibilitatea de a exista mai mulți utilizatori. Scopul acestui sistem de operare a fost constituit de existența unui cod cât mai compact și de o viteză de execuție excepțională.

FreeRTOS-ul implementează *multithreading*-ul(eng.) prin folosirea unor funcții de sistem care suspendă task-ul curent și oferă posibilitatea celorlalte task-uri de a rula. Planificatorul folosește un algoritm bazat pe algoritmul Round Robin luând în considerare prioritatea task-urilor. Cuanta de timp alocată fiecărui task poate fi configurată prin configurarea temporizatorului fizic folosit.

Așadar principalele caracteristici și funcționalități ale acestui sistem de operare sunt:

- Timpi de execuție foarte buni
- Spațiu foarte mic ocupat in memorie
- Planificatorul poate fi configurat pentru operații preemptive dar și cooperative
- Suport pentru corutine(task-uri care ocupă un spațiu extrem de redus din stivă)

Despre avantajele și dezavantajele sistemului de operare FreeRTOS se vor face referiri de-a lungul lucrării.

1.5. Justificarea Temei

Temă prezentei lucrări este reprezentată de implementarea unui sistem de operare de timp real dedicat unui procesor cu 2 nuclee folosind facilitățile limbajului C++.

Am ales această temă datorită interesului personal față de acest subiect dar și evoluția limbajului de programare C++ precum și neexistența unui sistem de operare in timp real care să evolueze pe două nuclee diferite din punct de vedere arhitectural. Acesta este o idee nouă și poate cunoaște direcții de dezvoltate de viitor având în vedere diversitatea arhitecturală a procesoarelor.

Acest sistem de operare va putea fi folosit si de alți utilizatori, scopul său fiind unul didactic și nu comercial.

1.6. Cerințe și așteptări

Produsul final trebuie să constituie un sistem de operare de timp real scris in mare parte folosind limbajul C++ și care să suporte paralelism fizic dar și paralelism virtual la nivel de nucleu fizic. Paralelismul virtual la nivel de nucleu va fi realizat la cererea planificatorului care, pentru algoritmul de planificare, va combina algoritmul Round Robin cu prioritățile taskurilor.

Caracteristicile așteptate sunt:

- Dualitate/paralelism fizic(să suporte două nuclee fizice)
- Preemptivitate
- Paralelism virtual
- Planificatorul folosește Round Robin și priorități
- Sunt definite funcțiile de bază pentru un sistem de operare în timp real(*delay*(eng.), suspendare, acces la secțiuni critice)
- Este implementat suport pentru operațiile de bază cu task-urile(creare, adăugare)
- Suport pentru un număr nelimitat de task-uri, singura constrângere fiind dimensiunea memoriei ocupate.
- Suport pentru comunicare intre task-uri
- Suport pentru sincronizarea task-urilor(semafoare) precum si pentru accesul exclusiv la resurse(mutex)
- Suport pentru diferite structuri de date(cozi, liste)
- Suport pentru modul DMA
- Suport pentru dispozitive periferice: comunicație serială, LCD etc.
- Administrarea memoriei
- Task-urile vor fi definite ca fiind bucle infinite in care se apelează funcții de sistem pentru a permite rularea altor task-uri
- Task-urile definite precum și sistemul de operare vor face parte dintr-un singur pachet, neexistând o barieră intre codul de sistem si codul de utilizator.

Sistemul de operare dezvoltat va trebui să fie unul de **timp real**. Acest lucru va fi realizat prin existența posibilității de a asigna fiecărui task o prioritate de execuție.

Nucleul sistemului de operare trebuie sa fie **dual**. Există două abordări posibile: fie va exista un singur nucleu al sistemului de operare iar acesta va trebui sa aloce fiecărui nucleu fizic un task, fie vor exista două nuclee ale sistemului de operare sau mai specific două planificatoare, fiecare administrând nucleul fizic corespunzător.

Sistemul de operare va fi **preemptiv**. Totdeauna task-ul cel mai prioritar va fi următorul care se va executa. În momentul în care task-ul curent va aștepta un semafor sau un mutex, procesorul va fi cedat următorului task cel mai prioritar disponibil. Verificarea task-urilor disponibile se va face periodic, alocând cuante de timp fiecărui task. În acest fel se va realiza **paralelismul virtual** sau virtualizarea nucleului fizic.

După cum am menționat, fiecărui task i se va aloca o cuantă de timp dar întotdeauna se va alege cel mai prioritar task disponibil. În acest mod se va implementa un planificator care folosește **Round Robin** și **priorități**.

Toate task-urile care vor fi executate in cadrul acestui sistem de operare vor trebui definite ca **bucle infinite**, in care periodic se vor apela **funcții** de sistem specifice pentru a da posibilitatea altor task-uri să fie executate. Aceste funcții specifice vor fi definite alături de alte funcții: `delay`, care va supenda temporar execuția task-ului curent, accesul zonelor critice, activarea și dezactivarea întreruperilor.

Codul implementat va trebui să suporte operații de **administrare** a task-urilor. Un utilizator care va folosi acest sistem de operare va avea la îndemână instrumente pentru a adăuga un task, pentru a-l șterge sau pentru a defini o prioritate specifică acestui task. Sistemul va suporta un **număr nelimitat** de task-uri. Singura constrângere va fi spațiul de memorie utilizat. Trebuie de asemenea să se aibă în considerare că dacă numărul de task-uri este destul de mare, atunci există posibilitatea ca task-urile de priorități mici să nu primească timp de procesor.

Sistemul de operare va trebui să suporte **comunicație intre procese**. Acest lucru se va realiza folosind *shared memory* și sincronizare cu semafoare sau vor fi dezvoltate module specifice cum ar fi cozi de mesaje.

Așadar, se vor defini obiecte pentru sincronizarea task-urilor, **semafoare**. De asemenea se vor defini obiecte pentru accesul exclusiv la resurse cum ar fi **mutecsi**. Vor fi implementate structuri de date cum ar fi liste sau cozi.

Un modul **DMA** reprezintă un dispozitiv integrat care realizează transferul rapid de date între diverse componente fizice cum ar fi intre memorie si periferice sau intre memorii și chiar intre periferice. Acest transfer se realizează fără implicarea procesorului, așadar se poate executa în paralel cu executarea task-urilor. Acest sistem de operare trebuie să suporte sau să folosească facilitățile oferite de un astfel de dispozitiv.

De asemenea, sistemul va trebui să implementeze funcții care să faciliteze accesul și comunicarea cu dispozitivele periferice: porturi seriale sau afișaj LCD.

Una dintre funcțiile de bază ale unui sistem de operare o constituie administrarea memoriei. Acest sistem de operare de timp real va oferi suport pentru administrarea memoriei, însă aceasta administrare va fi una simplă, care să mențină sistemul funcțional.

În mod normal, sistemele de operare sunt dedicate unei clase variate de utilizatori. În acest context, codul sistemului de operare si codul clientului sunt două entități disjuncte. Un sistem de operare de timp real este destinat unei clase de utilizatori care de obicei lucrează in domeniul programării sau un domeniu conex. Având în vedere acest aspect precum și dimensiunea redusă a memoriei, codul aferent sistemului de operare precum si codul utilizator, in care vor fi definite diferite task-uri specifice, vor fi practic înglobate într-un singur obiect final care va fi scris pe dispozitivul fizic.

1.7. Tehnologii folosite

Așa cum s-a menționat în descrierea temei, Pentru implementare se va folosi limbajul C++. Este bine știut că pentru implementarea modulelor specifice arhitecturii fizice folosite se va folosi limbajul de asamblare. În cadrul acestor module se vor defini funcții de bază pentru suspendarea execuției unui task dar și pentru reluarea acestuia, precum și definirea de funcții sau rutine care activează/dezactivează întreruperile, marcând astfel zonele critice.

Mediul de programare folosit este constituit de LPCXpressoIDE, un mediu de programare bazat pe eclipse la care au fost adăugate module pentru a realiza o interfațare ușoară cu microprocesorul utilizat. Mediul de programare este independent de sistemul de operare, așadar orice sistem de operare este suportat. Singurele cerințe ale mediului de dezvoltare sunt existența porturilor USB. Existența porturilor USB este necesară pentru conectarea JTAG-ului utilizat pentru scrierea codului implementat pe dispozitivul fizic.

Interfațarea fizică cu platforma de dezvoltare folosită va fi realizată utilizând comunicație serială.

Capitolul 2. Proiectarea aplicației

2.1 Platforma hardware

Proiectul este destinat platformelor *hardware*(eng.) incorporând un procesor cu două nuclee și care să respecte cerințele de utilizare ale unui sistem de operare in timp real. În acest scop, am considerat că plăcuța de dezvoltare LPCXpresso43S37 este ideală pentru acest proiect.

LPCXpresso43S37 este o plăcuță de dezvoltare, de cost redus, pusă la dispoziție de NXP și care include *microcontroller*-ul(eng.) LPC43S37JET100, un *microcontroller* cu două nuclee fizice bazate pe arhitectura ARM Cortex-M: un nucleu/*core*(eng.) este de tipul Cortex-M4 iar celălalt de tip Cortex-M0.

LPC43S37JET100 are o arhitectură pe 32 de biți si operează la o frecvență de 204 MHz. Acesta dispune de 1 MB memorie flash unde poate fi stocat sistemul de operare și de o memorie de lucru de o capacitate de 136 KB SRAM. De asemenea, dispune de o memorie EEPROM de 16 KB, interfațare SPI, componente periferice, interfață SGPIO(Serial General Purpose I/O), *controller*-e USB de mare viteză, interfață Ethernet, *controller* de memorie extern dar și interfață pentru dispozitive digitale sau analogice.

Nucleul Cortex M4 rulează, așa cum am descris mai sus la o frecvență de până la 204MHz. Acesta are incorporat un modul de protecție a memoriei(*Memory Protection Unit*(eng.)) și un modul de administrare al vectorilor de întrerupere(NVIC – *Nested Vectored Interrupt Controller*(eng.)). *Core*-ul(eng.) oferă suport pentru virgulă mobilă dar și pentru depanare folosind JTAG sau SWD(*Serial Wire Debug*(eng.)). De asemenea, printre principalele facilități oferite de acest nucleu se numără și existența unui temporizator de sistem (*System Tick Timer*(eng.)), care va fi folosit pentru alocarea cuantelor de timp.

Nucleul Cortex M0 este practic un coprocesor capabil să preia din sarcinile lui M4, dacă este configurat corespunzător. Acest nucleu lucrează la aceeași frecventă, 204 MHz. Acesta oferă suport pentru depanare folosind doar JTAG, însă, la fel ca și nucleul principal, oferă suport pentru NVIC.

Memoriile sistemului:

- 1 MB memorie flash împărțită in două bancuri
- 16 KB memorie EEPROM pentru stocare de date
- 136 KB SRAM, memorie de lucru
- 64 KM ROM conținând codul de pornire și *driver*-ele de sistem
- memorie One-Time Programmable(OTP) - 64 bit
- două bancuri de OTP cu suport pentru criptare.

Această plăcuță de dezvoltare oferă suport pentru criptarea și decriptarea AES atât a imaginii de pornire cât și a altor date folosind transferul datelor printr-un modul DMA.

Interfețe seriale:

- *SPI Flash* oferind viteze de până la 52 MB pe secundă
- Conexiune *Ethernet* cu suport DMA pentru asigurarea unei performanțe optime a procesorului.
- Conexiune USB(*Host/Device/OTG*) de înaltă viteză cu suport DMA
- Conexiune USB(*Host/Device*) de înaltă viteză cu suport DMA
- UART 550 cu interfață modem și suport DMA

- Trei interfețe USART 550 de asemenea cu suport DMA
- Controler SPI
- Interfață I^2C și interfață I^2S

Această plăcuță de dezvoltare oferă suport pentru dispozitive periferice digitale : controler de memorie extern, controler LCD cu suport DMA ce suportă rezoluții de până la 1024 x 768, interfață pentru carduri de memorie(SD/MMC), 164 pini GPIO cu suport DMA, 4 temporizatoare , generator de semnal PWM pentru controlul motoarelor, temporizator de tip *watchdog.* Alte detalii privind modul de funcționare al procesoarelor vor fi prezentate la momentul oportun, pentru a avea un context favorabil înțelegerii subiectului.

În Figura 2.1 este ilustrată structura registrelor celor două procesoare precum și structura stivei după apelul unei rutine de întrerupere.

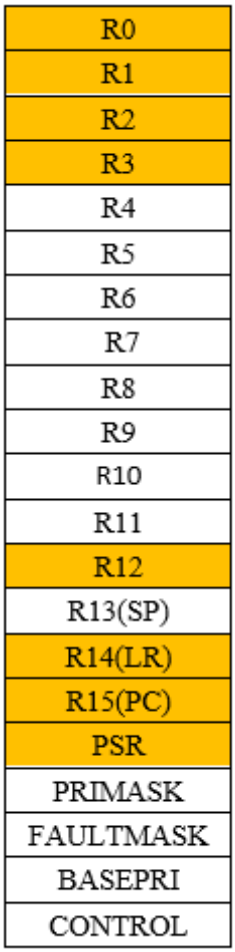

Figura 2.1 Registrele procesorului LPC43S37

Fiecare dintre cele două procesoare, M0 și M4, conțin 13 registre de uz general R0-R12, 5 registre speciale (xPSR, Primask, Faultmask, Basepri, Control) și 3 registre de lucru: SP(Stack Pointer) care reține adresa vârfului stivei, LR(Link Register) care reține adresa de returnare pentru subrutine sau funcții, PC(Program Counter) care reține adresa ultimei instrucțiuni executate și xPSR(Program Status Register) care salvează informații despre stiva folosită și modul de lucru al procesorului.

2.2 Structura nucleului sistemului de operare

Având în vedere funcțiile și caracteristicile unui sistem de operare de timp real, nucleul sistemului dezvoltat va cuprinde următoarele module :

- *Dispatcher*
- *Scheduler*
- *Task Manager*
- *IO Module*
- *Synchronization Module*
- *System Module*
- *Library Module*

Dispatcher-ul va avea ca principală responsabilitate schimbul de context. Acesta va trebui să salveze, la cerere, contextul task-ului actual și să restaureze contextul task-ului următor care va fi lansat în execuție.

Scopul *Scheduler*-ului este de a decide care task va fi următorul executat. De asemenea, având în vedere faptul că în planificarea task-urilor se va folosi algoritmul Round Robin, acesta va trebui să decidă când un task și-a consumat cuanta de timp alocată. Planificatorul va comunica direct cu *Dispatcher*-ul. De asemenea, planificatorul va oferi suport pentru adăugarea task-urilor în coada de execuție.

Task Manager-ul este componenta care va oferi utilizatorilor instrumente pentru crearea de task-uri, asignarea priorităților pentru task-urile create dar și inițializarea contextului unui task.

Modulul IO va implementa și va oferi suport pentru comunicarea cu diferite dispozitive IO cum ar fi porturile seriale, joystick-ul, led-urile plăcuțelor sau senzorul de temperatură.

Una dintre componentele importante ale sistemului este modulul de sincronizare. Acesta va pune la dispoziție instrumente pentru acces exclusiv la resurse, sincronizarea task-urilor, partajarea și accesul memoriei comune dar și comunicarea între task-uri, atât la nivelul aceluiași nucleu fizic al procesorului, cât și la nivelul ambelor nuclee.

Modulul System va avea ca principală responsabilitate pornirea coprocesorului M0. Suportul pentru structuri de date cum ar fi liste sau cozi va fi oferit de către Library Module.

Așa cum s-a menționat în descrierea platformei fizice, cele două nuclee ale procesorului sunt diferite atât din punct de vedere al setului de instrucțiuni suportat, cât și din punct de vedere al facilităților oferite de fiecare. Astfel, nucleul M4 suportă un set mai larg de instrucțiuni, care cuprinde atât setul de instrucțiuni al procesorului M0, cât și integrarea instrucțiunilor pentru operații in virgulă mobilă sau procesare de semnal. De asemenea este notabilă lipsa unor facilități ale nucleului M0 precum temporizatorul de sistem sau accesul la diferite periferice. De asemenea, cele două procesoare mapează zone diferite de memorie.

2.3 Nuclee fizice diferite – avantaje și dezavantaje

Având în vedere diferențele de arhitectură ale celor două nuclee fizice, proiectarea nucleului sistemului de operare poate cunoaște două abordări:

- Pentru fiecare nucleu fizic sunt definite componente independente ale nucleului sistemului de operare
- Ambele nuclee fizice folosesc același nucleu al sistemului de operare .

Pentru cea de-a doua abordare, există o serie de avantaje dar și o serie de dezavantaje. Ca și avantaj ar fi faptul că task-urile ar putea fi prelucrate de oricare dintre cele două nuclee. Din acest mod de proiectare rezultă un dezavantaj și anume imposibilitatea de a folosi *cache*-ul(eng.) în situațiile când un task trebuie să cedeze procesorul curent altui task și să primească procesorul celuilalt task. În acest caz, datele trebuie copiate de pe memoria unui nucleu pe memoria celuilalt. De asemenea, codul trebuie compilat pentru a utiliza setul de instrucțiuni cel mai general adică cel al nucleului M0, ceea ce ar duce la scăderea performanțelor și neutilizarea la capitate maximă a nucleului M4. Această situație se poate rezolva prin introducerea unui modul care să transforme codul compilat pentru un anumit nucleu in cod care poate fi executat optim de celălalt. Această metodă însă ar fi consumatoare de timp.

Pentru prima abordare, dezavantajul cel mai mare constă în imposibilitatea de a permite unui task să fie executat dinamic pe oricare dintre nuclee, astfel că task-urile vor fi alocate static la nivelul compilării, direct de către utilizator. Un alt dezavantaj va fi existența codului duplicat. Printre avantaje, se poate menționa utilizarea eficientă și performantă a celor două nuclee dar și simplitatea implementării.

Mediul de programare ales și utilizat în implementare nu permite compilarea identică a codului pentru cele două nuclee și impune utilizarea a două proiecte diferite pentru fiecare dintre ele cu mențiunea că proiectul aparținând nucleului M0 este considerat o componentă a proiectului M4 la momentul editării legăturilor. De asemenea, o simplă modificare în cod induce o recompilare totală a codului.

Având în vedere cele prezentate mai sus, soluția aleasă va consta în proiectarea a două nuclee ale sistemului de operare diferite pentru cele două procesoare. De menționat este faptul că doar unele module vor fi diferite: dispecerul, planificatorul și modulului IO. Pentru celelalte componente, codul va fi identic, dar va trebui să existe pentru ambele nuclee.

2.4 Proiectarea modulelor

2.4.1 Modulul Dispatcher

Dispecerul este componenta sistemului de operare care realizează salvarea și încărcarea contextului unui task in memorie. Această componentă constă din câteva funcții care vor fi implementate în limbaj de asamblare. Structura și proiectarea celor două versiuni va fi identică însă tocmai din cauza implementării folosind limbajul de asamblare, implementarea va trebui să fie diferită pentru cele două nuclee.

Principalele funcții implementate în acest modul:

- `void CallDispatcher(void)`
- `__attribute__ ((naked)) void Dispatch(void)`
- `__attribute__ ((naked)) void StartDispatcher(void)`
- `void DispatchContext(void)`
- `void DispatchFirstTask(void)`
- `__attribute__(( naked )) void MaskInterrupts(void)`
- `__attribute__(( naked )) void EnableInterrupts(void)`

Specificarea secvenței „__attribute__((naked))” specifică compilatorului faptul că funcția respectivă nu va modifica registrele procesorului. Această operație este un artificiu introdus cu scopul de a realiza schimbul de context.

În condiții normale, schimbul de context este inițiat prin apelul funcției `CallDispatcher`. Această funcție activează o întrerupere prin modificarea valorii registrului ICSR(*Interrupt Control Status Register*), registru responsabil de administrarea întreruperilor la nivelul procesorului. Întreruperea este activată imediat la ieșirea din funcție. Rutina asociată acestei întreruperi este chiar funcția `Dispatch`, care va realiza salvarea contextului procesorului și va încărca un nou context returnat de funcția `DispatchContext`. La momentul ieșirii din funcție, procesorul va conține noul context.

O întrerupere reprezintă un semnal sincron sau asincron ce semnalizează apariția unui eveniment care trebuie tratat de către procesor. Tratarea întreruperii are ca efect suspendarea firului normal de execuție al unui task și lansarea in execuție a unei rutine de tratare a întreruperii (RTI).. Înainte de a lansa in execuție o rutină de tratare a întreruperii, procesorul salvează valorile registrelor modificabile de către rutină pe stiva programului. La sfârșitul execuției rutinei, starea anterioară a registrelor este refăcută, astfel task-ul este continuat din punctul în care a rămas. Prin modificarea valorilor de pe stivă, procesorul încarcă practic în memorie alt task și astfel se realizează schimbul de task-uri.

Pentru a păstra codul scris în limbaj de asamblare la nivelul unui singur fișier, se vor defini și funcții pentru activarea și dezactivarea globală a întreruperilor. Acest lucru este necesar în momentul in care procesorul execută instrucțiuni critice considerate atomice.

2.4.2 Modulul TaskManager

Acest modul cuprinde câteva clase și structuri care vor facilita crearea de task-uri noi precum și operațiile de bază ale lucrului cu task-uri: suspendarea execuției, reluarea execuției sau salvarea contextului.

În contextul unui sistem de operare de timp real, un task reprezintă un fir de execuție sau un proces. Fiecare task are asociat un bloc de control sau un context. Acest context este reprezentat de informațiile necesare procesorului pentru ca task-ul să poată fi reluat cu succes după ce acestuia i-a fost suspendată execuția. Practic, blocul de control este compus din valorile registrelor procesorului.

La declanșarea unei întreruperi, procesorul folosit salvează automat, dacă nu este specificat altfel, valorile registrelor R0-R3, R12, LR, PC și xPSR. Acestea vor fi grupate într-o structură numită `HW_StackedRegisters`. Registrele care vor fi salvate manual de către dispecer vor fi grupate într-o altă structură numită `SW_StackedRegisters`. Clasa `Context` conține doi membri pointeri `ptopStack` și `pBaseStack`, care vor reține vârful stivei și baza stivei care reprezintă adresa maximă alocată task-ului curent. Pentru fiecare task se specifică inițial dimensiunea stivei alocate. Cei doi membri se vor actualiza in funcție de această valoare. Dacă valoarea din registrul SP este in afara acestor limite, atunci apare fenomenul de *stack overflow*(eng.). Inițial, pe stivă sunt puse valorile implicite ale unor registre cum ar fi PC, care este inițializat cu adresa funcției care se va executa.

Clasa Task încapsulează informațiile folosite în general de Planificator despre task-ul curent:

- taskIdM – un identificator unic,
- eStateM – starea task-ului, o variabila de tip enum care poate lua valorile Ready,

Running, Suspended sau Waiting, în funcție de starea corespunzătoare a task-ului. Mai multe detalii în Figura 2.2

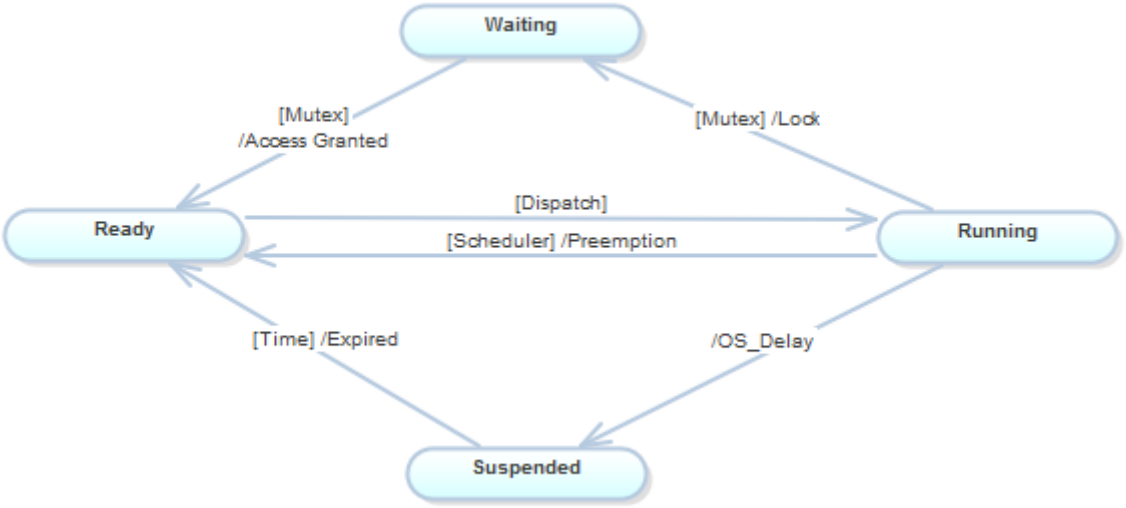

Figura 2.2 Stările task-urilor

- priorityM – prioritatea în cadrul sistemului de operare
- pContextM, un pointer către contextul task-ului curent. Obiectul este de tipul Context definit mai sus.
- pCodeM - un pointer către funcția care se conține bucla de execuție
- nTicksM şi nTickSuspeneded - numărul de cuante alocate fiecărui task respectiv numărul de cuante in care task-ul va fi suspendat.

În figura 2.3 este prezentată diagrama de clase pentru clasa Task.

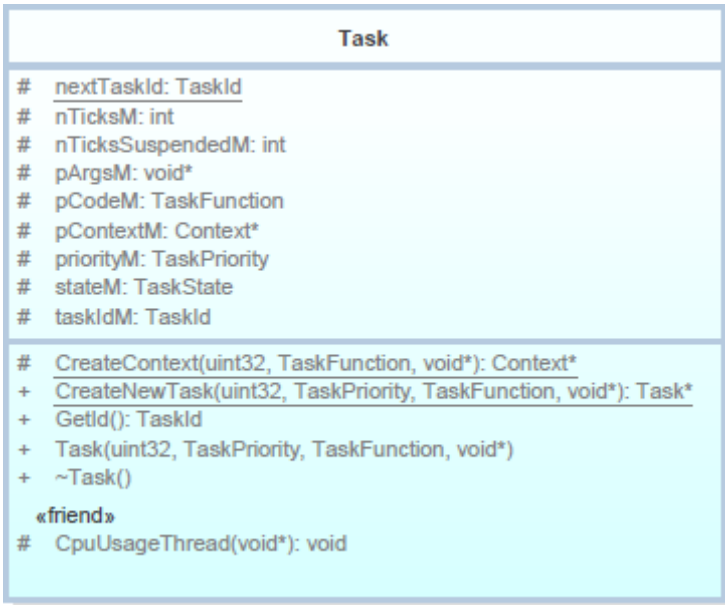

Figura 2.3 Diagrama clasa Task

De asemenea, clasa conţine un membru static nextTaskId, cu ajutorul căruia se vor genera identificatorii task-urilor.

Folosind șablonul de proiectare Metoda Fabrică, clasei i s-a adăugat o metodă publică statică de creare de task-uri noi pe baza informațiilor de intrare primite.

Pentru păstrarea referințelor către task-urile crate, au fost implementate două clase de tip *container*(eng.): TaskArray și TaskQueue.

Clasa **TaskArray** folosește pentru stocarea referințelor(pointer la clasa Task) un vector alocat static cu o dimensiune fixă. Această dimensiune este configurabilă prin setarea valorii în fișierul `Config.h` . Structura de date prezentă implementează operații specifice accesării elementelor unui vector prin supraîncarcarea operatorului [] (`inline Task* operator[](int i)`). De asemenea sunt implementate metode de inserare sau ștergere. Mai multe detalii se pot observa in figura 3, unde sunt prezentate diagramele de clase ale celor două containere.

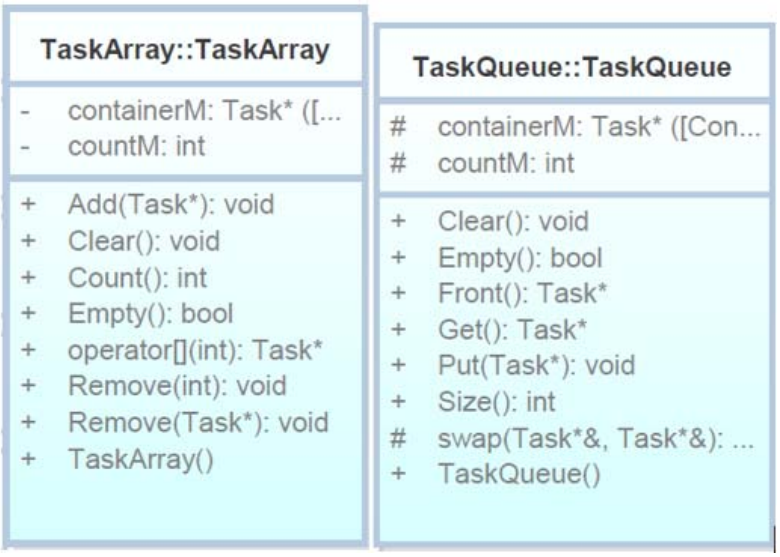

Figura 2.4 Diagrama clase container

Clasa **TaskQueue** implementează operații specifice unei cozi: inserare, stergere, numărare sau accesarea primului element din coadă. Proprietatea acestei structuri de date constă în faptul că folosește, pentru inserare și ștergere, operații specifice unui arbore de tip min heap implementat static. Această metodă de abordare este necesară pentru implementarea algoritmului de planificare al task-urilor, care trebuie să folosească priorități. Ordonarea elementelor implică folosirea valorii priorităților. Task-urile cu prioritatea cea mai mică vor fi inserate mereu pe pozițiile din capul cozii. Se consideră că task-urile cu valoarea proprietății priorityM cea mai mică sunt task-urile cele mai prioritare. Modul de efectuare al operațiilor de inserare și ștergere poate fi consultat la Anexa 2 iar structura clasei în Figura 2.4.

2.4.3 Modulul Scheduler

Acest modul conține o singură clasă care implementează operațiile specifice planificatorului. Având în vedere faptul că există doar un singur planificator asociat unui nucleu, metodele acestei structuri de date sunt declarate statice.

Figura 2.5 Diagrama clasa Scheduler

Atributele acestei clase sunt:

- eStateM – starea planificatorului care poate lua una din valorile prezentate in partea dreaptă a imaginii x.
- nTicksM – contorizează numărul total de cuante de timp alocate tuturor task-urilor. Această valoare este utilă pentru calcularea gradului de încărcare al procesorului.
- pCurrentTaskM – referință către task-ul curent aflat in execuție
- pIdleTaskM – referință către task-ul Idle.

- readyTasklistM – obiect de tipul TaskQueue, care conține task-urile disponibile pentru a primi controlul procesorului.
- waitingTaskListM – obiect de tipul TaskArray ce stochează referințe ale task-urilor suspendate sau care asteaptă deblocarea unei resurse.
- u8TasksM – un contor ce reține numărul total de task-uri adăugate.

Metoda AddTask permite adăugarea unui task în coada de priorități aferentă task-rilor pregătite pentru a fi lansate in execuție. Inițial se consideră că toate task-urile se află în această situație. Se menționează, ca și constrângere, faptul că această metodă poate fi apelată doar înainte de a porni efectiv planificatorul. În caz contrar, această metodă va returna o eroare.

Metodele `CheckSuspendedTasks și CheckWaitingTasks` iterează lista de task-uri waitingTaskListM și verifică dacă există task-uri ce pot fi adăugate în coada de priorități pentru a putea fi lansate în execuție. Prima metodă este apelată după expirarea fiecărei cuante de timp tocmai pentru a verifica expirarea timpului de suspendare. Cea de-a doua funcție este apelată, atunci când este cazul, din obiectele de sincronizare, în momentul eliberării unor resurse. Cazurile de apelare vor fi discutate pe larg în momentul prezentării modulului aferent.

Metoda `GetNextReadyTask`() este apelată de dispecer pentru a primi referința următorului task ce va primi controlul procesorului. Această metodă scoate din coada de priorități task-ul cu prioritatea cea mai mare disponibil pentru execuție și îl marchează ca deținând controlul procesorului. De asemenea, incrementează valoarea nTicksM pentru a putea fi ulterior utilizată în calcularea gradului de încărcare al procesorului.

Metoda `Initialize`() inițializează planificatorul. Aceasta setează prioritățile întreruperilor temporizatorului precum și a întreruperii care realizează schimbul de context. Totdeauna prioritatea întreruperii care realizează schimbul de context trebuie să fie mai mare pentru a nu putea fi întreruptă de tratarea întreruperii temporizatorului. De asemenea, tot aici este adăugat in coada de task-uri disponibile pentru execuție și task-ul Idle.

Task-ul Idle, a cărui funcție de execuție este reprezentată de metoda `IdleFuncțion`, are prioritatea cea mai mică din sistem. El este executat atunci când nici un alt task nu este disponibil pentru a primi controlul procesorului. Funcția aferentă nu execută nici o instrucțiune complexă ci doar așteaptă la infinit declanșarea unei întreruperi generată de nevoia schimbării unui context.

Metoda `Start()` este metoda care pornește efectiv planificatorul. Este pornit temporizatorul de sistem și se lansează în execuție primul task. Temporizatorul de sistem folosit pentru fiecare din cele două nuclee este diferit, fapt datorat diferenței de arhitectură a celor două nuclee. Astfel, pentru nucleul M4, temporizatorul folosit este Systick iar pentru nucleul M0 este Timmer0.

Temporizatorul de sistem SysTick este un temporizator pe 24 de biți special proiectat pentru utilizarea lui în cadrul unui sistem de operare de timp real pentru a genera excepții la intervale bine definite. Acest temporizator numără descrescător iar la atingerea valorii 0 este activată întreruperea aferentă. Acest temporizator poate fi setat pentru a utiliza o sursă de generare a semnalului de ceas externă sau poate folosi chiar sursa procesorului. Pentru proiectarea curentă se va folosi sursa de semnal a procesorului deoarece aceasta are o precizie ridicată și de asemenea se poate calcula cu ușurință valoarea maximă folosită pentru numărare. Valoarea maximă ce poate fi setată este 0xFFFFFF. Frecvența de lucru a procesorului este de 204MHz, ceea ce înseamnă că temporizatorul descrește valoarea numărată la fiecare 1/204 000 000 secunde. Sistemul de operare curent impune folosirea generării unei cuante de timp de 10 milisecunde (= 1 / 100 secunde), așadar valoarea setată pentru acest temporizator va fi de 204 000 000 / 100 . De asemenea, această valoare poate fi configurată în fișierul `Config`.h .

Temporizatorul Timer0 este un temporizator de lucru disponibil pentru ambele procesoare. Acesta este folosit pentru generarea excepțiilor pentru procesorul M0, deoarece din arhitectura sa

lipsește un temporizator de sistem cum este SysTick. Acest temporizator este inițializat pentru a genera excepții cu aceeași periodicitate de 10 milisecunde.

În momentul generării unei excepții, din rutina de tratare a întreruperilor este apelată metoda `SwitchRequired`(), care returnează valoarea booleană `true` dacă un schimb de context este necesar, respectiv false in caz contrar. În cazul în care există un alt task ar putea prelua controlul procesorului, În rutina de tratare a întreruperii temporizatorului este apelat dispecerul care va realiza schimbul de context imediat după terminarea rutinei. Practic, dispecerul activează excepția PendSV care a fost setată cu o prioritate mai mare decât cea a temporizatorului. În metoda planificatorului se verifică dacă există task-uri ale căror perioadă de suspendare a expirat. Dacă există, acestea sunt adăugate în coada de priorități aferentă task-urilor disponibile pentru a fi rulate. Dacă nici un task nu este disponibil pentru a fi executat, atunci task-ul Idle este adăugat automat in coadă. Dacă pe prima poziție din coada de priorități se află un task cu o **prioritate mai mare sau egală** decât a ultimului task care a deținut controlul procesorului, atunci se consideră că este necesar un schimb de context. Se folosește proprietatea mai mare sau egal tocmai pentru a implementa algoritmul Round Robin, în care fiecare task primește procesorul o cuantă de timp, a cărei valoare este de 10 milisecunde în cadrul sistemului de operare dezvoltat. Implementarea acestei funcții este disponibilă in Anexa 3.

Metodele `SetCurrentTaskWait` și `SuspendCurrentTask` adaugă task-ul curent în lista de task-uri suspendate. Aceste metode sunt apelate din cadrul modulului de sincronizare. Pentru task-urile suspendate o perioadă de timp, se calculează numărul de cuante în care procesorul nu va fi cedat acelui task folosind funcția TicksForPeriod, ce transformă o valoarea din milisecunde în cuante de timp folosind valoarea frecvenței de lucru a procesorului și valoarea maximă a temporizatorului.

2.4.4 Modulul Syncronization

Acest modul cuprinde clase si funcții utile pentru accesul exclusiv al task-urilor la resurse partajate(clasa Mutex), pentru sincronizarea task-urilor (clasa Semaphore) precum și pentru comunicarea între task-uri aflate pe același nucleu(clasele ShMemSegment și ShDmaMemSegment), dar și pentru comunicarea task-urilor aflate in nuclee diferite. Pentru ultimul caz se folosește o bibliotecă *open-source* [3], pusă la dispoziție de producătorul plăcuței de dezvoltare.

Clasa **Mutex** permite accesul exclusiv la o resursă. Aceasta are două atribute, `pOwnerM` care reține o referință la task-ul care a apelat cu succes metoda Lock() și un atribut waitingQueueM, unde sunt reținute referințe către task-urile care încearcă să acceseze resursa critică. Un mutex poate avea două stări : blocat și liber.

Prin apelul metodei `Lock`, dacă starea internă este liber, atunci task-ului apelant îi este permis accesul la resursa critică modificând starea in blocat. Dacă mutexul este blocat, atunci task-ul apelant este adăugat în coada de așteptare. De asemenea, acesta este suspendat pentru a permite altor task-uri să preia controlul procesorului printr-un apel la dispecer.

Deblocarea mutex-ului se face prin apelul metodei `Release`. Dacă în coada de așteptare se găsesc task-uri, atunci primul task din coadă este marcat ca fiind pregătit pentru a putea fi executat. De precizat este faptul că doar task-ul care a blocat mutex-ul îl poate debloca. Eliminarea condițiilor de *dead-lock* se face prin suspendarea controlului procesorului task-ului apelant și oferirea controlului altor task-uri, eventual al celui care a blocat inițial mutex-ul.

Clasa Semaphore implementează conceptul de sincronizare al task-urilor. Un task poate aștepta un eveniment prin apelarea metodei `Wait` . Un alt task poate genera declanșarea unui eveniment prin apelarea metodei `Release.`

Pentru implementare se folosește o variabilă contor inițializată cu valoarea 0. Metoda Wait este blocantă dacă valoarea contor este 0. În acest caz, task-ul apelant este adăugat în coada de așteptare și suspendat de planificator până la semnalarea unui eveniment.

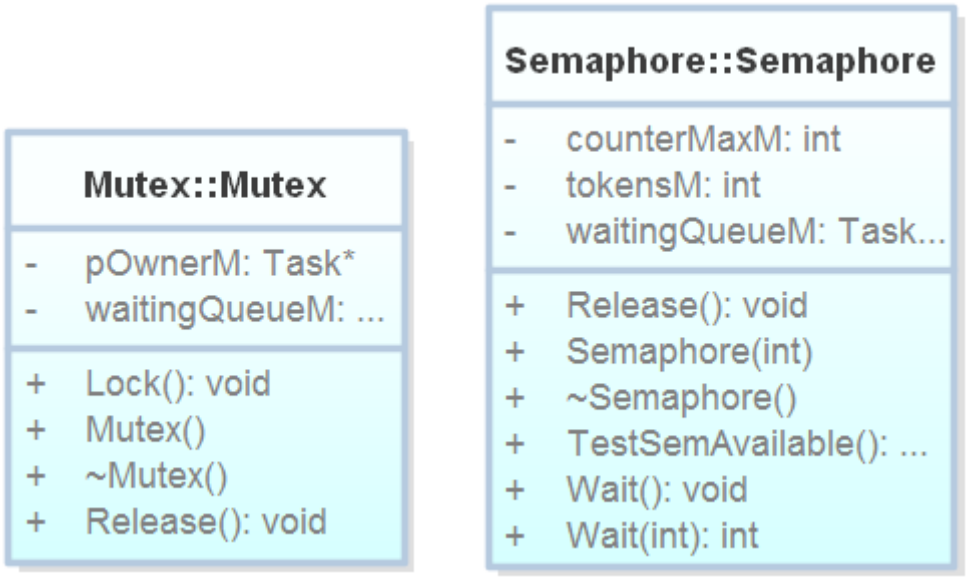

Figura 2.6 Diagrama clase Mutex și Sempahore

Clasele ShMemSegment și ShDMAMemSegment folosesc conceptul de memorie partajată folosit pentru comunicarea proceselor. Aceste clase au fost implementate cu scopul de a încapsula obiectele folosite pentru o comunicare eficientă și sincronizată a task-urilor. Elementele necesare implementării conceptului de memorie partajată sunt variabilele globale și semafoarele, care sunt setate ca atribute ale acestor clase.

Printr-un apel al metodei GetData(), un proces așteaptă primirea datelor. În momentul primirii datelor, acestea sunt returnate task-ului prin copierea lor din container la o adresă specificată. Acest lucru este necesar deoarece alte task-uri pot suprascrie datele salvate anterior.

Printr-un apel al metodei WriteData(), un task pune datele necesare în containerul comun și semnalează acest lucru prin apelul metodei Release() a semaforului.

Diferența dintre cele două clase este reprezentată de modul în care datele sun copiate din zona privată a task-ului și zonă comună, adică containerul. Clasa ShMemSegment folosește instrucțiuni ce necesită utilizarea procesorului, în timp ce clasa ShDMAMemSegment folosește capabilitatea sistemului de a utiliza un controler DMA ce poate efectua copierea datelor fără implicarea procesorului, realizând astfel un transfer mai rapid.

Dimensiunea containerului comun se poate specifica prin modificarea valorii din fișierul de configurare.

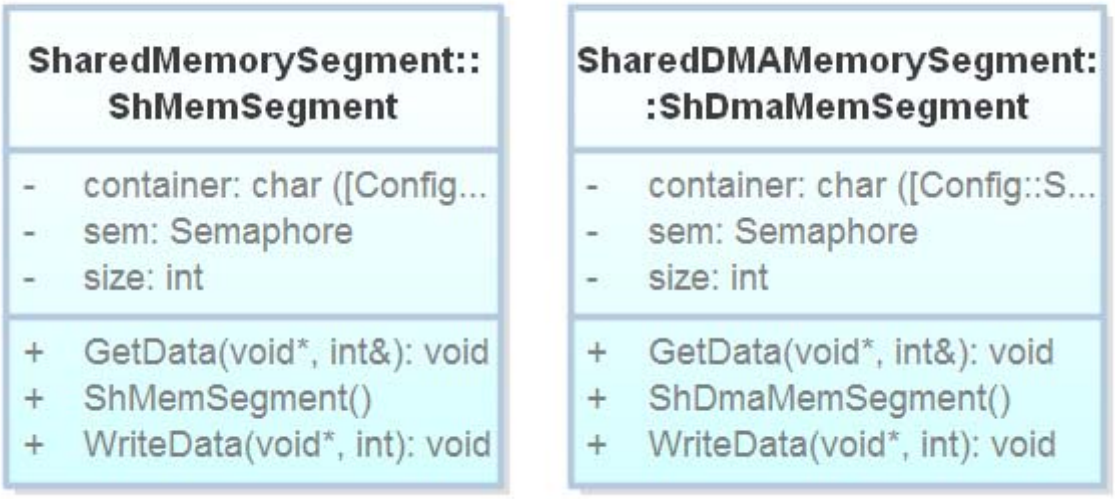

Figura 2.7 Diagrama clase obiecte memorie partajată

De asemenea, acest modul conține și câteva funcții ce specifică accesul la zone critice de cod precum și funcția specifică sistemelor de operare de timp real, OS_Delay, care permite suspendarea task-ului curent și cedarea controlului procesorului, realizând astfel paralelismul virtual.

2.4.5 Modulul IO

Acest modul conține câteva clase ce intermediază accesul la dispozitivele IO precum portul serial, joystick-ul, led-urile sau senzorul de temperatură. Acest modul vine în ajutorul programatorului oferind soluții rapide de integrare a acestor dispozitive. In Figura 2.8 sunt prezentate diagramele de clasa referitoare acestor dispozitive:

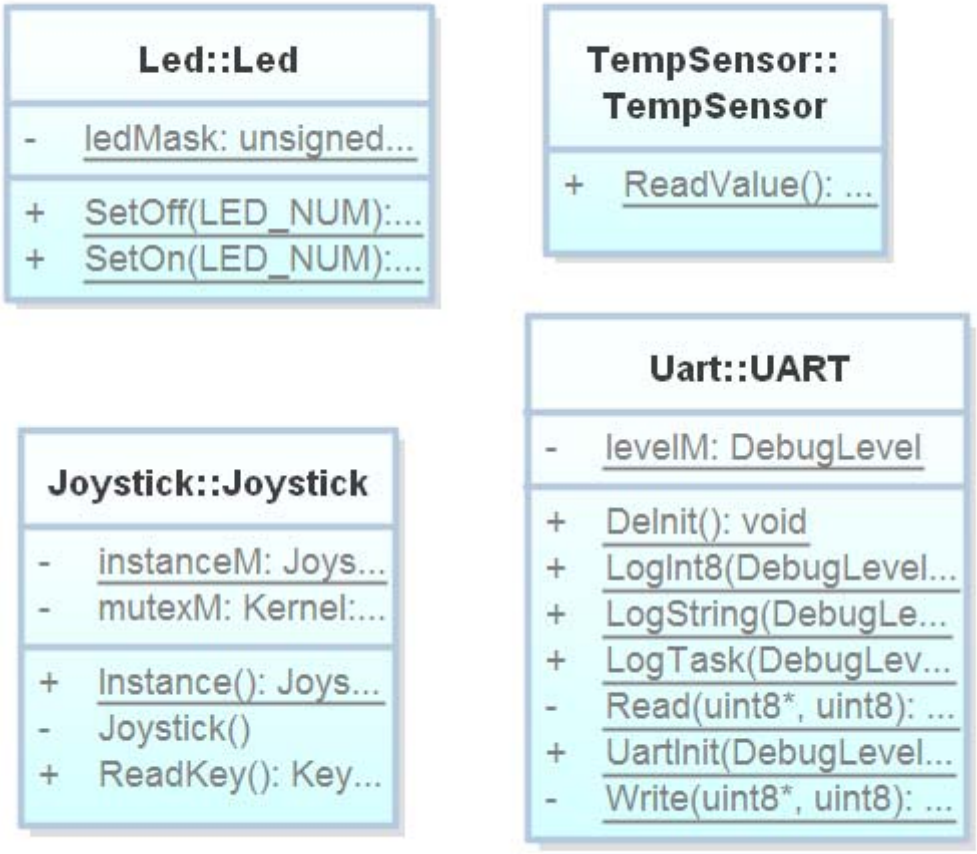

Figura 2.8 Diagrama clase IO

2.4.6 Modulul System

Acest modul este responsabil de inițializarea coprocesorului M0, precum și a comunicării între nuclee. Codul eferent acestui modul este in mare parte generat automat la momentul creării proiectului.

2.4.7 Modulul Library

În cadrul acestui modul sunt dezvoltate structuri de date utile programatorului cum ar fi liste sau cozi.

2.5 C++ - un limbaj util

Bjarne Soustroup susține odată cu apariția cărții sale [4] că limbajul C++ este la fel de rapid ca limbajul C și poate fi folosit fără nici o ezitare în implementarea aplicațiilor incorporate.

Pe lângă viteza de execuție, acest limbaj oferă o facilitate importantă și anume încapsularea datelor. Este foarte utilă gruparea datelor într-o clasă sau într-un namespace deoarece se evită astfel folosirea intensă a variabilelor globale.

Bineînțeles că acest limbaj oferă și alte instrumente precum polimorfismul sau moștenirea, însă folosirea acestora la acest nivel este destul de periculoasă din punctul de vedere al fiabilității sistemului și managementului memoriei. Pentru implementare se va opta pentru folosirea memoriei heap doar înainte de pornirea efectivă a planificatorului pentru a evita diversele probleme ce pot apărea din cauza unui management defectuos al memoriei.

2.6 Constrângeri

Printre constrângerile importante ale acestui mod de proiectare se numără lipsa utilizării memorie heap după pornirea planificatorului și imposibilitatea de a muta un task de pe un nucleu pe altul.

De asemenea, numărul maxim de task-uri ce pot fi executate este limitat prin configurarea acestuia in fișierul special. Indiferent de această valoare, se poate ajunge în situația în care unele task-uri nu vor mai putea fi niciodată executate, dat fiind faptul că se folosesc priorități in planificarea task-urilor.

Un alt aspect ce poate fi menționat se referă la faptul că unele resurse fizice trebuie împărțite exclusiv între cele două nuclee: afișajul LCD folosește pentru utilizare o matrice a cărei copie este deținută de ambele nuclee. Modificările unui nucleu nu sunt vizibile la nivelul celuilalt, de aceea se poate ajunge la suprapunerea conținutului.

Capitolul 3. Implementarea aplicației

3. 1 Modul de folosire și funcționare al sistemului de operare

În figura 3.1 este prezentată diagrama de secvență unde este descris modul de utilizare al sistemului de operare precum și modul de comunicare al principalelor module: Scheduler, Dispatcher și TaskManager.

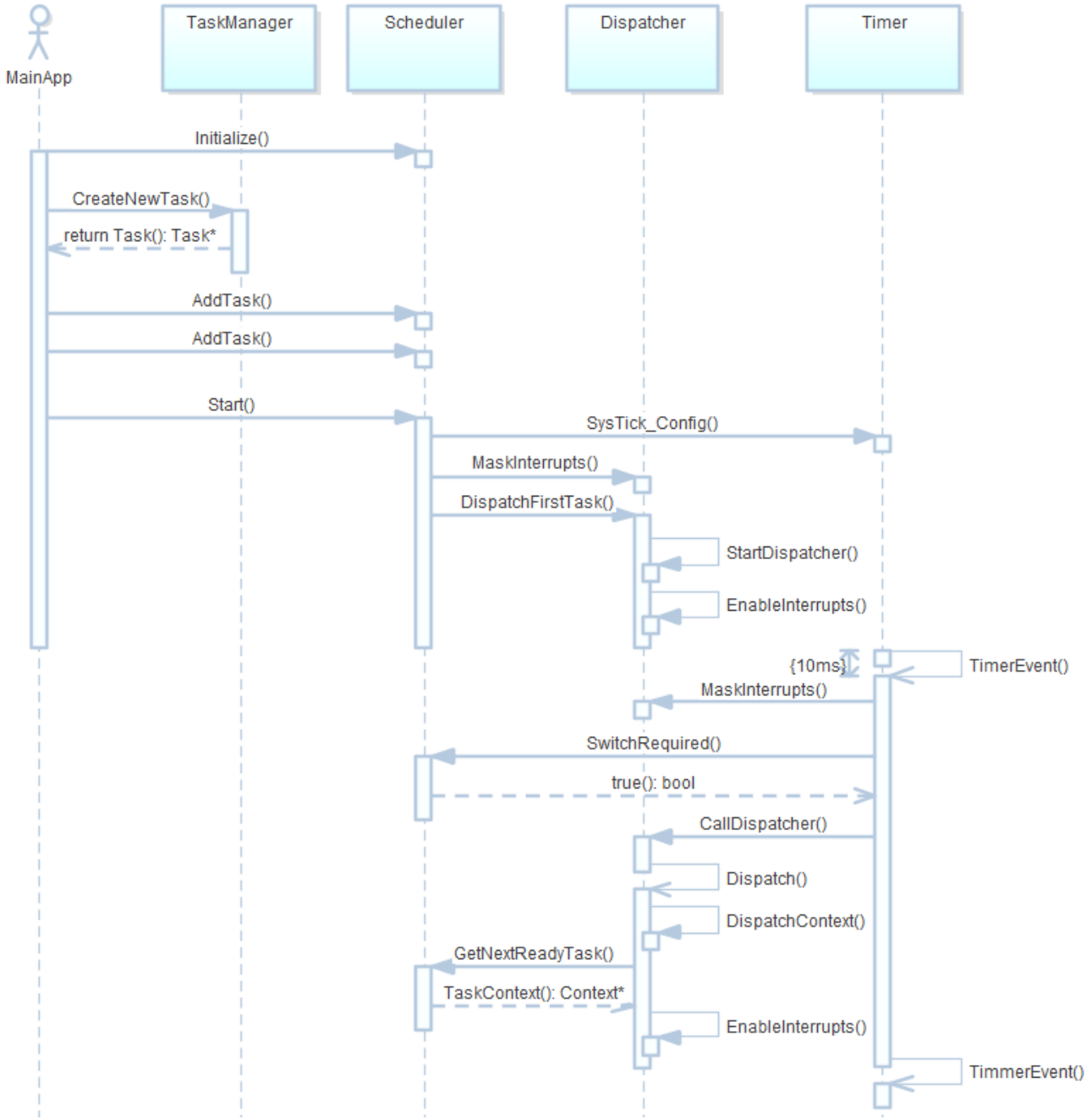

Figura 3.1 Diagrama de secvență pentru principalele componente

După inițializarea componentelor fizice, utilizatorul apelează metoda Scheduler::Initialize(), care inițializează planificatorul. Pentru a putea porni planificarea task-urilor, este nevoie ca acestea să fie create și adăugate în lista de task-uri a planificatorului. În acest sens, utilizatorul poate apela metoda statică Task::CreateNewTask(…) care crează și inițializează un task precum și contextul acestuia.

Pentru a adăuga un task in lista planificatorului, se folosește metoda Scheduler::AddTask(Task* pTask). După ce au fost adăugate toate task-urile utilizatorului, se poate apela metoda Scheduler::Start(). În cadrul acestei metode, inițial se configurează și se pornește temporizatorul. După efectuarea acestui pas, se dezactivează global întreruperile iar apoi se apelează metoda DispatchFirstTask(). În cadrul acestei metode sunt reactivate întreruperile. Activarea și dezactivarea întreruperilor sunt necesare pentru a marca intrarea într-o zonă de cod critică, operație ce trebuie executată atomic.

La fiecare 10 milisecunde este apelată rutina de tratare a întreruperii temporizatorului de sistem. Această rutină este considerată o secțiune critică, așadar va fi marcata ca atare. Se apelează metoda Scheduler::SwitchRequired(). Folosind algoritmul de planificare implementat, se returnează o valoare care indică necesitatea unui schimb de context. În cazul în care este necesar un schimb de context, se apelează metoda CallDispatcher(). Aceasta activează întreruperea PendSV, a cărei rutină de tratare coincide cu funcția Dispatch(). În cadrul acestei metode este cerut planificatorului blocul de control sau contextul task-ului ce va fi executat. La ieșirea din rutină, se va continua execuția task-ului precedent .

3.2 *Implementare mutex*

În figura 3.2 este prezentată modul de funcționare al unui mutex . Un task care apelează metoda Lock() pentru un mutex care anterior a fost luat, va fi suspendat de planificator pentru a da șansa altor task-uri să deblocheze acel mutex. În momentul apelului metodei Release de către deținătorul mutexului, task-ul care a fost anterior suspendat va fi adăugat în coada de task-uri disponibilă a planificatorului.

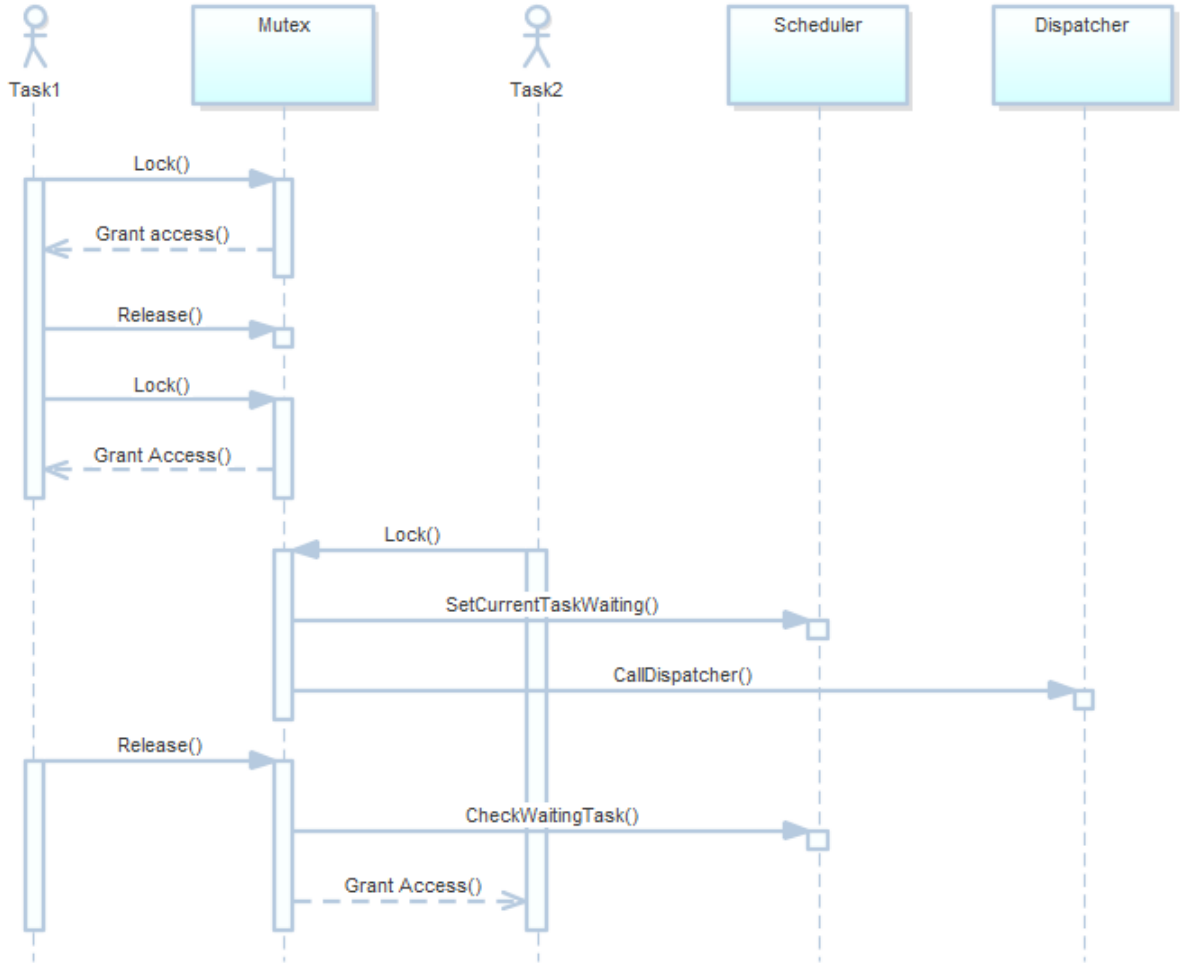

Figura 3.2 Diagrama Secvență Mutex

3.3 *Implementarea schimbului de context*

Așa cum s-a menționat anterior, în momentul în care apare o excepție în sistem(când se apelează rutina de tratare a unei întreruperi), procesorul salvează automat pe stivă valorile din registrele considerate modificabile de către rutina respectivă.

Procesorul M4(dar și M0) salvează în mod automat pe stivă valorile registrelor R0-R3, R12, LR, PC și xPSR. După tratarea rutinei, acesta încarcă valorile salvate anterior pe stivă înapoi în registre, eliminând astfel posibilitatea modificării accidentale a valorilor registrelor.

Schimbul de context se realizează in cadrul rutinei de tratare a întreruperii PendSV și este practic constituit din două faze: salvarea contextului task-ului actual și reîncărcarea contextului task-ului următor.

Contextul unui task este compus din toate valorile registrelor. Salvarea contextului este inițiată practic de către procesor înainte de apelarea rutinei, prin salvarea pe stivă a registrelor mai sus menționate. În momentul apelării rutinei, stiva arată ca in figura 3.3.

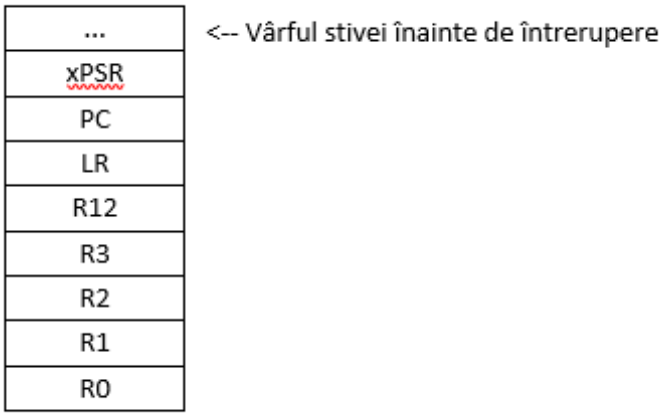

Figura 3.3 Conținutul stivei

Sistemul de operare trebuie acum să salveze valorile celorlalte registre manual. Se pun pe stivă valorile noilor registre și se actualizează valoarea noului *stack pointer*. Această valoare este salvată la adresa indicată de pointerul pTopStack pentru a putea fi folosită ulterior.

Se cere planificatorului adresa contextului următorului task pentru a putea încărca în registrul PSP(xPSR) valoarea conținută de noul context la adresa indicată de același pointer pTopStack. Sunt încărcate de pe nouă stivă valorile registrelor care au fost anterior salvate manual. După ieșirea din rutină procesorul încarcă de pe stiva valorile registrelor salvate anterior apelării rutinei. Dar adresa stack pointer-ului a fost modificată pentru a indica către stiva noului task, așadar se vor încărca valorile salvate intr-un apel anterior la pasul 1. După ce încarcă aceste noi valori, practic procesorul conține contextul noului task.

3.4 *Diverse probleme, idei și rezolvări*

Procesorul M4 suportă instrucțiuni pentru accesul simultan la registre printr-o singură instrucțiune în limbaj de asamblare. Din cauza arhitecturii diferite, procesorul M0 nu suportă aceste operații. Din această cauză, codul care realizează schimbul de context a fost modificat și adaptat procesorului M0. Dacă numărul de linii de cod care realizează schimbul de context pe procesorul M4 este de 24, pe procesorul M0 acesta atinge valoarea 48, tocmai din cauza instrucțiunilor ce trebuie să refere acum doar un singur registru.

De asemenea, plăcuța de dezvoltare suportă depanare în timp real pentru nucleul M4 folosind direct mediul de programare LPCXpresso. Din păcate, pentru nucleul M0 acest suport nu este oferit. Soluția la problema depanării nucleului M0 a constat în executarea codului lui M0 direct pe nucleul M4, având in vedere că setul de instrucțiuni al nucleului M4 cuprinde și setul lui M0.

Modul de realizare al schimbului de context este asemănător celui folosit de sistemul de operare de timp real FreeRTOS.

Producătorul plăcuței de dezvoltare oferă cod suport pentru lucrul cu perifericele procesoarelor. Fiecare dintre cele două procesoare au o copie a librăriilor oferite. Spre exemplu, pentru afișajul pe ecranul LCD, fiecare procesor are o copie a matricei folosite. Modificările la nivelul unui nucleului nu sunt vizibile și la nivelul celuilalt. Dacă două task-uri vor scrie în aceeași zonă a ecranului, atunci vor apărea suprapuneri. Pentru soluționarea aceste probleme, se alege un nucleu care

va administra ecranul. Task-urile celuilalt nucleu vor comunica cu task-uri de pe nucleul gazdă pentru a actualiza afișajul folosind biblioteca pusă la dispoziție pentru comunicarea intre procesoare.

Actualul proiect confirmă faptul că limbajul C++ poate fi folosit pentru implementarea aplicațiilor de nivel jos, care pot comunica direct cu mediul fizic. Poate că utilizarea limbajului C++ în implementarea unui sistem de operare de timp real nu este o idee originală, cu siguranță există și alte versiuni implementate folosind acest limbaj de programare. Însă cu siguranță este o idee reușită ce oferă performanțe ridicate.

Pentru a veni in ajutorul programatorului, au fost definite metode rapide de blocare și deblocare pentru un mutex : clasa Autolock. Această clasa apelează in constructor metoda Mutex::Lock() iar în destructor metoda Mutex::Release(). Așadar secțiunea critică este identică cu vizibilitatea variabilei de tip Autolock. Codul aferent se poate consulta la Anexa 4.

Capitolul 4. Testarea aplicației și rezultate experimentale

Pentru testarea sistemului de operare și a facilităților oferite de către acesta, au fost dezvoltate o serie de task-uri care testează diferite funcționalități ale sistemului. Așadar testarea aplicației dezvoltate s-a realizat prin scriere de cod și verificarea rezultatelor acestuia. De asemenea, sistemul de operare a fost testat prin depanare utilizând facilitățile oferite de producătorul plăcuței de dezvoltare.

Rezultatele obținute prin rularea task-urilor dezvoltate reflectă modul normal de funcționare al sistemului de operare precum și performanțele sistemului.

4.1 Performanțele sistemului

Sistemul de operare a fost proiectat pentru a ușura măsurarea gradului de încărcare al procesorului : au fost definite variabile ce măsoară numărul de cuante de timp alocate fiecărui task care pot fi accesate de o funcție specială numită CpuUsage. Practic această funcție conține bucla infinită a unui task ce execută periodic, la fiecare 5 secunde, câteva calcule ce redau procentul de utilizare al procesorului.

Formula de calcul utilizată este de forma valoare = 100 – cuante_idle / total_cuante. Deoarece sistemul nu a fost dezvoltat pentru a realiza operații in virgulă mobilă, valoarea este amplificata cu 100 in formula finală. Valorile gradului de încărcare al procesorului sunt afișate pe ecran, fiind actualizate la fiecare 5 secunde.

Valoarea afișată pentru nucleul M4 pentru un singur task adăugat este de 0,20%, o valoare extrem de mică având in vedere performanțele fizice ale sistemului. Cantitatea de memorie consumată nu poate fi măsurată în prezent, însă poate fi aproximată având în vedere numărul de task-uri și dimensiunea stivei.

4.2 Testarea mutexului

Pentru testarea funcționalității mutex-ului au fost implementate două task-uri care citesc periodic temperatura returnată de senzorul de temperatură și printează valoarea citită pe portul serial.

Cele două task-uri nu au priorități egale însă partajează o variabilă globală de tip mutex. În bucla de execuție, unul dintre task-uri apelează metoda Mutex::Lock(). După returnarea acestei metode, este citită valoarea temperaturii și este printată. La sfârșitul unei iterații, task-ul apelează metoda OS_Delay() și apoi Mutex::Release(), pentru a-și suspenda execuția timp de 500 milisecunde.

La fiecare iterație, cel de-al doilea task apelează metoda Mutex::Lock(). Task-ul este forțat să aștepte deblocarea mutexului la fiecare iterație, așadar acesta își poate continua execuția după expirarea timpului de suspendare al task-ului precedent. După preluarea accesului la variabila mutex, este printată pe portul serial valoarea temperaturii preluată de la senzor pentru ca în final, mutex-ul să fie deblocat.

Pe ieșirea portului serial se poate observa cum la fiecare 500 de milisecunde sunt printate câte două seturi de valori ale temperaturii, ceea ce demonstrează modul normal de funcționare al mutexului.

4.3 Testarea semafoarelor și a utilizării memoriei comune

Așa cum este bine știut, semafoarele sunt un mod de comunicare între task-uri ce funcționează ca un sistem de pseudo semnale : când un task apelează metoda Semaphore::Wait(), el practic se înregistrează pentru primirea unui semnal. Când se apelează metoda Semaphore::Release(), atunci task-ul aflat în coada de așteptare a semaforului este anunțat că își poate continua execuția.

Așadar, pentru testarea semaforului a fost nevoie de dezvoltarea a două task-uri folosind paradigma consumator-producător.

Task-ul producător citește periodic starea joystick-ului. La fiecare apăsare, este aprins un led pe plăcuță corespunzător tastei apăsate. De asemenea, valoarea tastei apăsate este copiată într-o zonă de memorie comună folosind un obiect de tipul ShDMAMemSeg. După fiecare copiere este apelată metoda Semaphore::Release(), așa cum a fost prezentată în capitolul 3.

Task-ul consumator este reprezentat de algoritmul din spatele unui mic joculeț. Se generează un obiect pe ecran iar utilizatorul trebuie să se deplaseze la acel obiect în cel mai scurt timp posibil. Acest task apelează metoda GetData aparținând obiectului ShDMAMemSeg și astfel așteaptă deblocarea semaforului și practic citirea valorii tastei apăsate. În funcție de această valoare, se efectuează deplasarea obiectului pe ecran.

4.4 Testarea clasei ShDMAMemSeg

Pentru testarea aplicației s-a folosit o implementare asemănătoare celei precedente. Task-ul producător formează un mesaj cu un identificator și îl scrie folosind un obiect de tipul ShDmaMemSegment la fiecare două secunde.

Task-ul consumator așteaptă primirea datelor și printează valoarea folosind portul serial. Se poate observa cum la fiecare două secunde este printat un mesaj aparținând acestui task.

4.5 Testarea comunicării între task-uri din nuclee diferite

Într-un capitol precedent s-a menționat despre constrângerea legată de utilizarea afișajului pe ecran pe cele două nuclee. Întrucât am ales ca afișarea să se facă folosind nucleul M0, toate comenzile de scriere pe ecran vor fi direcționate către celălalt procesor.

Librăria oferită de producătorul plăcuței de dezvoltare oferă posibilitatea trimiterii de mesaje între cele două nuclee. Un mesaj este format din identificatorul nucleului destinație, dintr-un identificator de task destinație precum și două valori întregi care pot reprezenta un pointer și dimensiunea memoriei oferite.

Pe nucleul destinație rulează un task *listener* care verifică periodic dacă au fost primite mesaje noi. In cazul primirii unui mesaj nou, acestea îl redirecționează către task-ul destinație, care poate prelucra informațiile oferite prin pointer si dimensiunea zonei alocate.

Redirecționarea mesajelor se face prin utilizarea semafoarelor dar și a obiectelor de tip memorie comună. În aceste fel este testată și funcționalitatea acestor obiecte pe ambele nuclee.

4.5 Testarea performanței planificatorului

Pentru testarea performanței planificatorului, am folosit un task care afișează la fiecare secundă timpul scurs de la pornirea sistemului în minute și secunde.

Prin folosirea unui cronometru, se poate observa că timpul afișat de planificator este aproximativ identic cu timpul afișat de cronometru. Așadar planificatorul nu introduce nici o întârziere iar task-urile sunt executate prompt, ținând cont de prioritățile acestora.

4.5 Testarea dispozitivelor IO

Dispozitivele IO au fost testate prin folosirea lor în cadrul task-urilor dezvoltate. S-a observat o funcționare normală a acestora, singurele constrângeri fiind cele legate de utilizarea lor pe nuclee diferite. Aceste probleme însă se pot rezolva printr-o utilizare sincronizată.

4.6 Scenarii de test

Pentru a testa buna funcționare a sistemului de operare, acesta a fost supus unor teste negative:

- Testarea planificatorului cu lista de task-uri vidă
  În mod implicit, task-ul Idle este adăugat automat în lista de task-uri. Sistemul execută doar acest task.
- Testarea sistemului folosind doar un nucleu
  Procesorul M0 este inițializat mereu de procesorul M4. Sistemul a fost testat neinițializând nucleul M0. Nucleul M4 are un comportament normal, task-urile fiind executate in mod normal.
- Testarea sistemului prin nepornirea planificatorului

  În acest caz, nucleul respectiv execută bucla de eroare. Celălalt nucleu rămâne perfect funcțional.

4.3 Utilizarea plăcuței de dezvoltare

Utilizarea sistemului de operare a fost prezentată în capitolul anterior. În continuare se va prezenta pe scurt utilizarea plăcuței de dezvoltare.

Pentru a putea executa codul implementat, utilizatorul are nevoie de o plăcuță de dezvoltare identică(LPCXpresso43S37) sau compatibilă, de un cablu USB - microUSB pentru conectarea acesteia dar și de mediul de programare LPCXpresso. Acest mediu de programare se poate descărca gratuit de pe site-ul oficial.

Se pornește mediul de programare și se încarcă proiectele aferente care conțin codul sistemului de operare folosind opțiunea File → Import. Se implementează codul corespunzător nucleului dorit.

Se construiește imaginea ce va fi scrisă în memoria flash a plăcuței folosind butonul de *Build* din interfața grafică sau apăsând click dreapta pe numele proiectului iar apoi alegându-se opțiunea corespunzătoare.

Pentru a scrie în memoria flash imaginea obținută, este nevoie ca plăcuța să fie conectată la stația de lucru folosind cablul USB. În funcție de sistemul de operare, poate fi necesar instalarea unor driver-e, însă acestea pot fi descărcate în mod automat odată cu mediul de programare. Se apasă butonul Program Flash și apoi se apasă butonul OK. După terminarea procesului, noua funcționalitate adăugată poate fi testată. După scrierea imaginii, procesorul M4 este restartat automat, ceea ce duce și la restartarea procesorului M0, deci practic restartarea întregului sistem.

Capitolul 5. Discuții și concluzii

Scopul proiectului este reprezentat de implementarea unui sistem de operare de timp real care să folosească pentru planificare un algoritm bazat pe Round Robin și priorități iar ca limbaj de programare să fie utilizat limbajul C++.

În mare parte proiectul a fost implementat conform cerințelor inițiale. Poate că există un aspect care ar putea fi menționat și anume că nu a fost dezvoltat un sistem de administrare a memoriei mai complex. Însă aceasta ar putea fi o direcție de dezvoltare pentru viitor.

Proiectul actual poate avea o mulțime de direcții de dezvoltare, printre acestea menționând extensibilitatea, suportul unei variate game de procesoare sau optimizarea planificatorului.

Suportul pentru diferite procesoare poate fi extins fără a avea un impact mare asupra structurii sistemului de operare. Proiectarea face ca o eventuală modificare de procesor să afecteze doar modulul Dispatcher, modul responsabil de salvarea și reîncărcarea contextului unui task în memorie. Acest modul execută cod specific platformei utilizate, așadar suportul pentru diferite procesoare poate fi ușor implementat.

Extensibilitatea sistemului se referă la dezvoltarea sau integrarea de componente și module noi ce pot fi oferite utilizatorului. Se pot enumera adăugarea unui sistem de fișiere sau adăugarea suportului pentru stiva TCP/IP pentru a putea realiza conectarea la internet.

De asemenea, planificatorul poate fi optimizat sau modificat pentru a implementa alți algoritmi sau altă modalitate de a planifica task-urile. Se poate adăuga suport pentru task-uri aperiodice sau suport pentru ștergerea unui task din lista de execuție, la cererea acestuia spre exemplu.

Comparativ cu sistemul de operare FreeRTOS, prezentului proiect îi lipsesc anumite funcționalități cum ar fi existența unor instrumente de management a temporizatoarelor sau un sistem avansat de protecție și administrare a memoriei.

Implementarea actualului proiect folosind limbajul C++ reprezintă o caracteristică ce conferă utilizatorului posibilitatea de a folosi cod cu suport pentru paradigma orientată obiect. Este bine știut faptul că un cod scris în limbajul C, așa cum este scris FreeRTOS sau uC/OS este mult mai greu de înțeles.

De asemenea, o notă de unicitate este oferită și de suportul pentru paralelism fizic, ce permite utilizarea paralelă a două nuclee fizice. Bineînțeles, și în acest caz, se pot aduce îmbunătățiri in sensul permiterii migrării task-urilor de pe un nucleu pe altul. În acest moment, acest lucru nu este posibil, însă se poate datora și platformei fizice alese ce oferă două nuclee fizice diferite.

Capitolul 5. Discuții și concluzii

Scopul proiectului este reprezentat de implementarea unui sistem de operare de timp real care să folosească pentru planificare un algoritm bazat pe Round Robin și priorități iar ca limbaj de programare să fie utilizat limbajul C++.

În mare parte proiectul a fost implementat conform cerințelor inițiale. Poate că există un aspect care ar putea fi menționat și anume că nu a fost dezvoltat un sistem de administrare a memoriei mai complex. Însă aceasta ar putea fi o direcție de dezvoltare pentru viitor.

Proiectul actual poate avea o mulțime de direcții de dezvoltare, printre acestea menționând extensibilitatea, suportul unei variate game de procesoare sau optimizarea planificatorului.

Suportul pentru diferite procesoare poate fi extins fără a avea un impact mare asupra structurii sistemului de operare. Proiectarea face ca o eventuală modificare de procesor să afecteze doar modulul Dispatcher, modul responsabil de salvarea și reîncărcarea contextului unui task în memorie. Acest modul execută cod specific platformei utilizate, așadar suportul pentru diferite procesoare poate fi ușor implementat.

Extensibilitatea sistemului se referă la dezvoltarea sau integrarea de componente și module noi ce pot fi oferite utilizatorului. Se pot enumera adăugarea unui sistem de fișiere sau adăugarea suportului pentru stiva TCP/IP pentru a putea realiza conectarea la internet.

De asemenea, planificatorul poate fi optimizat sau modificat pentru a implementa alți algoritmi sau altă modalitate de a planifica task-urile. Se poate adăuga suport pentru task-uri aperiodice sau suport pentru ștergerea unui task din lista de execuție, la cererea acestuia spre exemplu.

Comparativ cu sistemul de operare FreeRTOS, prezentului proiect îi lipsesc anumite funcționalități cum ar fi existența unor instrumente de management a temporizatoarelor sau un sistem avansat de protecție și administrare a memoriei.

Implementarea actualului proiect folosind limbajul C++ reprezintă o caracteristică ce conferă utilizatorului posibilitatea de a folosi cod cu suport pentru paradigma orientată obiect. Este bine știut faptul că un cod scris în limbajul C, așa cum este scris FreeRTOS sau uC/OS este mult mai greu de înțeles.

De asemenea, o notă de unicitate este oferită și de suportul pentru paralelism fizic, ce permite utilizarea paralelă a două nuclee fizice. Bineînțeles, și în acest caz, se pot aduce îmbunătățiri in sensul permiterii migrării task-urilor de pe un nucleu pe altul. În acest moment, acest lucru nu este posibil, însă se poate datora și platformei fizice alese ce oferă două nuclee fizice diferite.

Capitolul 6. Bibliografie

[1] A. Cristian, *Curs Sisteme de operare,* -, 2016.

[2] A.-D. Remzi and A.-D. Andrea, Operating Systems: Three Easy Pieces, UK: Arpaci-Dusseau Books, LLC, 2015.

[3] FreeRTOS, „FreeRTOS Main Page," FreeRTOS, 2016. [Interactiv]. Available: http://www.freertos.org. [Accesat 2016].

[4] NXP, „Main," NXP, [Interactiv]. Available: http://www.nxp.com/products/software-and-tools/software-development-tools/software-tools/lpc-microcontroller-utilities/lpcxpresso-ide:LPCXPRESSO?code=LPCXPRESSO&nodeId=015210BAF75070863D&fpsp=1&tab=Design_Tools_Tab. [Accesat 2016].

[5] B. Stroustrup, The C++ programming language(4th edition), Bell Labs, 2014.

[6] G. Butazzo, Hard Real Time Computing Systems, Pisa: Springer.

[7] M. Barr, Programming Embedded Systems in C and C++, O'Reilly, 1999.

[8] J. Labrosse, MicroC/OS-||, Audpublishing.

Capitolul 7. Anexe

Anexa 1. Modulul dispatcher

- Fișierul Dispatcher.h

```
#include <Kernel/TaskManager/TaskContext.h>
#include <Kernel/Scheduler/Scheduler.h>

#include <Kernel/Sync/IPC.h>
#include <Kernel/Io/Uart.h>

/* Redefine main interrupt handlers - Trick */
#define Dispatch          PendSV_Handler
#define StartDispatcher   SVC_Handler


namespace Kernel
{

/* Sets a PendSV interrupt */
void CallDispatcher(void);

extern "C"
{

/* Global variable - updated in Scheduler */
extern Context * volatile pCurrentTaskContextG;

/* Mask(disable) interrupts */
__attribute__(( naked )) void MaskInterrupts(void);

/* UnMask(enable) interrupts */
__attribute__(( naked )) void EnableInterrupts(void);

/* PendSV Irq Handler */
__attribute__ ((naked)) void Dispatch(void);

/* SVC(Supervisor call) Irq handler */
__attribute__ ((naked)) void StartDispatcher(void);


void DispatchContext(void);

} /* extern "C" */

} /* namespace Kernel */
```

- Fișierul Dispatcher.cpp

```
#include <Kernel/Scheduler/Scheduler.h>
#include <Kernel/Dispatcher/Dispatcher.h>
```

```
#include <Kernel/Io/Uart.h>

namespace Kernel
{

#define NVIC_ICSR                ( * ( ( volatile uint32 * ) 0xe000ed04 ) )
#define PENDSVSET_BIT            ( 1UL << 28UL )
#define INTERRUPT_PRIORITY       (5 << ( 8 - 3))

void CallDispatcher(void)
{
    //UART::LogString("pends");
    NVIC_ICSR = PENDSVSET_BIT;

    __asm volatile( "dsb" );
    __asm volatile( "isb" );
}

extern "C"
{

void DispatchContext(void)
{
    pCurrentTaskContextG = Scheduler::GetNextReadyTask();
}

void MaskInterrupts(void)
{
    __asm volatile(
        "   mrs r0, basepri             \n"
        "   mov r1, %0                  \n"
        "   msr basepri, r1             \n"
        "   bx lr                       \n"
        :: "i" ( INTERRUPT_PRIORITY ) : "r0", "r1"
    );
}

void EnableInterrupts(void)
{
    __asm volatile
    (
        "   msr basepri, r0             \n"
        "   bx lr                       \n"
        :::"r0"
    );
}


void Dispatch(void)
{
    /* Save PSP to R0 */
    __asm volatile("  mrs r0, psp                   \n");
    /* Flush */
    __asm volatile("  isb                           \n");

    /* Load the address of the currentTaskContext */
    __asm volatile("  ldr r3, pCurrentTaskContextM  \n");
    /* and get the top of the stack */
    __asm volatile("  ldr r2, [r3]                  \n");

    /* Push the registers which are not saved
     * automatically on the stack
```

```
     */
    __asm volatile("  stmdb r0!, {r4-r11, r14}      \n");

    /* Store the updated PSP into pCurrentTaskContextM->Top */
    __asm volatile("  str r0, [r2]                  \n");

    /* Push pCurrentTaskContextM->Top into the stack */
    __asm volatile("  stmdb sp!, {r3}               \n");


    __asm volatile("  mov r0, %0                    \n"::
            "i"(INTERRUPT_PRIORITY));
    /* disable interrupts */
    __asm volatile("  cpsid i                       \n");
    __asm volatile("  msr basepri, r0               \n");

    /* call DispatchContext */
    __asm volatile("  bl DispatchContext            \n");

    /* Flush and enable interrupts */
    __asm volatile("  dsb                           \n");
    __asm volatile("  isb                           \n");
    /* enable interrupts */
    __asm volatile("  cpsie i                       \n");
    __asm volatile("  mov r0, #0                    \n");
    __asm volatile("  msr basepri, r0               \n");

    /* load the new "pCurrentTaskContextM->Top" into SP */
    __asm volatile("  ldmia sp!, {r3}               \n");

    /* get the address of the new top */
    __asm volatile("  ldr r1, [r3]                  \n");
    __asm volatile("  ldr r0, [r1]                  \n");

    /* restore the registers which are manually saved */
    __asm volatile("  ldmia r0!, {r4-r11, r14}      \n");

    /* load the new top into PSP */
    __asm volatile("  msr psp, r0                   \n");
    /* Flush */
    __asm volatile("  isb                           \n");

    /* return using the new context */
    __asm volatile("  bx r14                        \n");

    __asm volatile("  .align 2                      \n");
    __asm volatile("  pCurrentTaskContextM: .word pCurrentTaskContextG  \n");
}

void StartDispatcher( void )
{
    /* Load the first task context. */
    __asm volatile ("  ldr  r3, pFirstTaskContext       \n");
    /* Use pFirstTaskContext to get the pCurrentTaskContextG address. */
    __asm volatile ("  ldr r1, [r3]                     \n" );
    /* Load into r0 the top of the stack for task */
    __asm volatile ("  ldr r0, [r1]                     \n");
    /* Pop the registers that are not automatically saved on interrupt */
    __asm volatile ("  ldmia r0!, {r4-r11, r14}         \n");
    /* Load the task stack pointer into PSP. */
    __asm volatile ("  msr psp, r0                      \n");

    /* "Flush" */
```

```
    __asm volatile ("  isb                                  \n");

    /* Enable Interrupts */
    __asm volatile ("  mov r0, #0                           \n");
    __asm volatile ("  msr  basepri, r0                     \n");
    /* Return. On the registers we have the first stack context,
      so after this call we will run the first task */
    __asm volatile ("  bx r14                               \n");
    __asm volatile ("  .align 2                             \n");
    __asm volatile ("  pFirstTaskContext: .word pCurrentTaskContextG    \n");
}

void DispatchFirstTask()
{
    __asm volatile(
            /* Use the NVIC offset register to locate the stack.
             * 0xE000ED08 = M4 Vector Table Offset Register (memory register)
             */
            " ldr r0, =0xE000ED08   \n"
            " ldr r0, [r0]          \n"
            " ldr r0, [r0]          \n"
            /* Set the msp back to the start of the stack. */
            " msr msp, r0           \n"

            /* Enable interrupts. */
            " cpsie i               \n"
            " dsb                   \n"
            " isb                   \n"

            /* Call first task. */
            " svc   0               \n"
            " nop                   \n"
        );
}

} /* extern "C" */

} /* namespace Kernel */
```

Anexa 2. Modulul TaskManager

- Fișierul TaskQueue.h

```
#include <Kernel/TaskManager/Task.h>

namespace Kernel
{

/*
 * This class is a priority queue.
 *
 * It is used a static array to keep the elements(the tasks).
 * The queue can store at maximum Config::RTOS_MAX_TASKS_NUMBER tasks.
 * The Insert and remove from the queue are done in a min heap mode.
 * The piority of an element is actually the priority of the task.
 * So, we can get the most prior task every time from first position.
 */
class TaskQueue
{
public:
```

```
    inline TaskQueue()
    {
        countM = 0;
    }

    /*
     * Add task to queue, then heapify the queue
     */
    inline void Put(Task* task)
    {
        if(countM >= Config::RTOS_MAX_TASKS_NUMBER)
        {
            return;
        }

        /* Add task to queue */
        containerM[countM++] = task;

        /* heapify the queue */
        int parent = countM / 2 - 1;
        int child  = countM - 1;

        while(parent >= 0)
        {
            if(containerM[parent]->priorityM > containerM[child]->priorityM)
            {
                swap(containerM[parent], containerM[child]);
                child = parent;
                parent = child / 2 - 1;
            }
            else
            {
                parent = -1;
            }
        }
    }

    /*
     * Return the root element from the heap. Also removes it from the heap.
     */
    inline Task* Get()
    {
        if(countM <= 0)
        {
            return 0;
        }

        /* Get root element */
        Task* item = containerM[0];

        /* Replace the root with the last element from the queue */
        containerM[0] = containerM[--countM];

        int parent = 0;
        int child  = 2 * parent + 1;
        while(child < countM)
        {
            /* Choose the most prior child */
            if(child + 1 < countM && containerM[child]->priorityM >
containerM[child+1]->priorityM)
            {
                ++child;
```

```
            }

            /* swap if needed */
            if(containerM[child]->priorityM < containerM[parent]->priorityM)
            {
                swap(containerM[parent], containerM[child]);
                parent = child;
                child = parent * 2 + 1;
            }
            else
            {
                child = countM + 1;
            }
        }

        return item;
    }

    /* Return the root element, but keep it to queue */
    inline Task* Front()
    {
        return (countM > 0 ? containerM[0] : 0);
    }

    /* return the number of elements */
    inline int Size()
    {
        return countM;
    }

    inline void Clear()
    {
        countM = 0;
    }

    inline bool Empty()
    {
        return countM <= 0;
    }

protected:

    int   countM;
    Task* containerM[Config::RTOS_MAX_TASKS_NUMBER];

protected:

    inline void swap(Task*& a, Task*& b)
    {
        Task* aux = a;
        a = b;
        b = aux;
    }
}; /* class TaskQueue */

}  /* namespace Kernel */
```

Anexa 3. Modulul Scheduler

- Fișierul Scheduler.h

```
#include <SysTypes.h>
#include <Config.h>

#include <Kernel/Dispatcher/Dispatcher.h>

#include <Kernel/TaskManager/TaskQueue.h>
#include <Kernel/TaskManager/TaskArray.h>
#include <User/CpuUsage.h>

namespace Kernel
{

extern "C" void DispatchContext(void);

extern "C" void DispatchFirstTask(void) __attribute__ ((naked));

class Scheduler
{
public:

	/*
	 * Initialize the Scheduler
	 *    * Set Irq priorities
	 *    * Add Idle task
	 */
	static void Initialize();


	/*
	 * Start the scheduler
	 *    * Configure timer
	 *    * Choose the first task to execute
	 *    * Call first task
	 */
	static void Start();

	/*
	 * Stop the Scheduler. Clear all data & disable interrupts
	 */
	static void Stop();

	/*
	 * Add a new task to the list of the tasks will run.
	 * Initialize Scheduler if needed.
	 */
	static bool AddTask(Task* task);

	/*
	 * Choose the next task which will run.
	 * Method is called from timer interrupt.
	 * @return true if a context switch is required or false otherwise.
	 */
	static bool SwitchRequired();

	/*
	 * Suspend all tasks.
	 */
	static void SuspendAll();

	/*
	 * Resume all tasks
	 */
	static void ResumeAll();
```

```
      /* Scheduler possible states */
    enum State { UNKNOWN, INITIALIZED, STARTED, CRITICAL_ENTERED, SUSPENDED,
STOPPED };

private:

      static Task*   pCurrentTaskM;
      static Task*   pIdleTaskM;

    static State   eStateM;
      static uint8   u8tasksNumM;

      static TaskQueue readyTasklistM;
      static TaskArray waitingTaskListM;

      friend void DispatchContext(void);

      friend void OS_Delay(int msec);
      friend Task::TaskId GetID();

      friend class Mutex;
      friend class Semaphore;

      static int nTicksM;
      friend void Userspace::CpuUsageThread(void*);

protected:

      /*
       * Callback for Idle task
       */
      static void IdleFunction( void* );

      /*
       * return the new task which will run
       */
      static Context* GetNextReadyTask();

      /*
       * Suspend current task execution.
       *    * Calculates the number of ticks the task will be suspended.
       *    * Add the task to delayed tasks list.
       *    * Add Idle task  to ready tasks list if no task id ready to run.
       */
    static void SuspendCurrentTask(int msec);

    /*
     * Change the current task state to waiting.
     *    * Add the task to waiting tasks list.
     *    * Add Idle task  to ready tasks list if no task id ready to run.
     */
    static void SetCurrentTaskWaiting();

      /*
       * This function is called when Systick Interrupt occurs.
       *    * decrement the number of suspended ticks.
       *    * if the number of suspended tick is zero,
       *      then the task is added to the list of ready tasks
       */
      static void CheckSuspendedTasks();

      /*
```

```
        * Check if some tasks waiting for a mutex/semaphore can run now
        */
       static void CheckWaitingTasks();

       /*
        * Calculates the number of ticks based on period in ms
        */
       static uint32 TicksForPeriod(uint32 msec);

};

}
```

- Ff

```
#include "board.h"

#include <Kernel/Io/Uart.h>

#include <Kernel/Dispatcher/Dispatcher.h>

#include <Kernel/Scheduler/Scheduler.h>

#include <Kernel/Sync/os_functions.h>

using namespace Kernel;



#define TimerEvent        SysTick_Handler



/* Initialize static members here */

uint8 Scheduler::u8tasksNumM     = 0;

Task* Scheduler::pCurrentTaskM   = 0;

Task* Scheduler::pIdleTaskM      = 0;

int   Scheduler::nTicksM         = 0;


TaskQueue Scheduler::readyTasklistM;

TaskArray Scheduler::waitingTaskListM;


Scheduler::State Scheduler::eStateM = Scheduler::UNKNOWN;


/* declared "extern" in "InterruptsHandlers.h" */

Context * volatile pCurrentTaskContextG = 0;


extern "C" void TimerEvent(void)

{

    /* Critical section */

    MaskInterrupts();
```

```
    /* A context switch is required */
    if(Scheduler::SwitchRequired() == true)
    {
        UART::LogString(UART::VERBOSE, "Context Switch");

        /* Start Pending Context Switch Request */
        CallDispatcher();
    }

    EnableInterrupts();
}


// Public Interface

/*
 * Initialize
 */
void Scheduler::Initialize()
{
      UART::LogString(UART::INFO, "Initialize kernel");

      // Set Priorities : SysTick should never be preempted by PendSV
      NVIC_SetPriority(SysTick_IRQn, Config::SYSTICK_NVIC_PRIORITY);
      NVIC_SetPriority(PendSV_IRQn, Config::PENDSV_NVIC_PRIORITY);

      readyTasklistM.Clear();
      waitingTaskListM.Clear();

    u8tasksNumM  = 0;
    eStateM      = INITIALIZED;

      /* Add Idle Task */
    pIdleTaskM = Task::CreateNewTask(
            Config::DEFAULT_STACK_SIZE,
            Kernel::RTOS_IDLE,
```

```
            Scheduler::IdleFunction,
            0);
    Scheduler::AddTask(pIdleTaskM);
}

/*
 * Start kernel
 */
void Scheduler::Start()
{
    if(eStateM != INITIALIZED)
      {
        UART::LogString(UART::ERROR,"Kernel NOT Initialized");
        ErrorExit();
      }

    UART::LogString(UART::INFO, "Starting kernel");

      /* Get the first ready task */
      pCurrentTaskContextG = GetNextReadyTask();

      /* Disable Interrupts while setting timers
       * Interrupts will get enabled when first task will be started
       */
      MaskInterrupts();

      /* Start the tick timer */
      SysTick_Config(Config::SYSTICK_HZ_RATE);

      UART::LogString(UART::INFO, "Kernel started");
      eStateM = STARTED;

      /* Start first task */
      DispatchFirstTask();

      /* NEVER GO HERE */
      ErrorExit();
```

```
}

/*
 * Stops kernel. Should not go here.
 */
void Scheduler::Stop()
{
    if(eStateM != STARTED)
    {
        UART::LogString(UART::ERROR, "Error stopping kernel");
    }

      UART::LogString(UART::INFO, "Stopping kernel");

      /* Disable interrupts */
      MaskInterrupts();

      pCurrentTaskM = 0;
      u8tasksNumM = 0;

      pCurrentTaskContextG = 0;

    ErrorExit();
}

/*
 * Add a new task
 */
bool Scheduler::AddTask(Task* task)
{
    /* For the moment, no task can be added after the kernel is started and running
*/
      if(eStateM != INITIALIZED)
      {
          UART::LogString(UART::ERROR, "Kernel not initialized or suspended");
            return false;
      }
```

```
	/* Memory protection */
	if(u8tasksNumM >= Config::RTOS_MAX_TASKS_NUMBER || !task)
	{
		UART::LogString(UART::ERROR, "Error adding task");
		return false;
	}

	++u8tasksNumM;

	/* Add task to the list */
	task->stateM = Task::READY;
	task->nTicksSuspendedM = 0;
	task->nTicksM = 0;

	/* We suppose that all the tasks can run initially.
	 * We could add for the future some checks.
	 * Some task could run later, could be started from another tasks etc
	 */
	readyTasklistM.Put(task);

	UART::LogString(UART::INFO, "Task added");

	return true;
}

/*
 * Decide if a context switch is required
 */
bool Scheduler::SwitchRequired()
{
    /* Used for CPU Usage */
    ++nTicksM;

    /* No context switch is possible if kernel is not started */
	if(eStateM != STARTED)
	{
	    UART::LogString(UART::ERROR, eStateM == CRITICAL_ENTERED ?
```

```
                "Already critical entered" : "State change not permitted");
        return false;
    }

    eStateM = CRITICAL_ENTERED;

    /* Check for new ready tasks */
    Scheduler::CheckSuspendedTasks();

    /* If no task is ready to run, then Idle Task is definitely ready(if not
already running)*/
    if(readyTasklistM.Empty() && pCurrentTaskM != pIdleTaskM)
    {
        readyTasklistM.Put(pIdleTaskM);
    }

  /* By default, no context switch is required */
    bool fContextSwitch = false;

    /* If we have a task more prior to run than actual, then a context switch
is needed */
    if(!readyTasklistM.Empty()       &&       pCurrentTaskM->priorityM       >=
readyTasklistM.Front()->priorityM)
    {
        /* Change the state of the task to ready */
        pCurrentTaskM->stateM = Task::READY;
        readyTasklistM.Put(pCurrentTaskM);

        fContextSwitch = true;
    }
    else /* Used for CPU Usage */
    {
        pCurrentTaskM->nTicksM++;
    }

    eStateM = STARTED;

    return fContextSwitch;
}
```

```
/*
 * Suspend all tasks
 */
void Scheduler::SuspendAll()
{
      UART::LogString(UART::INFO, "Stopping kernel");
      eStateM = SUSPENDED;
      MaskInterrupts();
}

/*
 * Resume all tasks
 */
void Scheduler::ResumeAll()
{
      UART::LogString(UART::INFO, "Resuming kernel");
      eStateM = STARTED;
      EnableInterrupts();
}


// Private Interface

/*
 * Choose the next task which will run
 */
Context* Scheduler::GetNextReadyTask()
{
    /* We assured that readyTasklistM will never be empty
         when be called from here. */
      pCurrentTaskM = readyTasklistM.Get();

      /* Used for CPU Usage */
      pCurrentTaskM->nTicksM += 1;

      /* Update the state of the task will run. */
```

```
      pCurrentTaskM->stateM = Task::RUNNING;

      /* return the context of the task which will run next. */
      return pCurrentTaskM->pContextM;
}

/*
 * Suspend current task
 */
void Scheduler::SuspendCurrentTask(int msec)
{
    if(eStateM == CRITICAL_ENTERED)
    {
        UART::LogString(UART::ERROR, "Already entered critical");
        return;
    }

    eStateM = CRITICAL_ENTERED;
    UART::LogTask(UART::VERBOSE, pCurrentTaskM->taskIdM, "Suspend");

    /* Get the number of ticks that task will be suspended */
    pCurrentTaskM->nTicksSuspendedM = TicksForPeriod(msec);

    /* Change the state */
    pCurrentTaskM->stateM = Task::SUSPENDED;

    /* Add the task to suspended list */
    waitingTaskListM.Add(pCurrentTaskM);

    /* If no task is ready to run, then Idle Task is definitely ready. */
    if(readyTasklistM.Empty())
    {
        readyTasklistM.Put(pIdleTaskM);
    }

    /* Adding this, it makes possible to delay some tasks more/less than needed.
*/
    //SysTick_Config(Config::SYSTICK_HZ_RATE);
```

```
    eStateM = STARTED;
}

/*
 * Enqueue the current task to waiting queue.
 */
void Scheduler::SetCurrentTaskWaiting()
{
    if(eStateM == CRITICAL_ENTERED)
    {
        UART::LogString(UART::ERROR, "Already entered critical");
        return;
    }
    eStateM = CRITICAL_ENTERED;

    /* Change the state */
    pCurrentTaskM->stateM = Task::WAITING;

    /* Add the task to suspended list */
    waitingTaskListM.Add(pCurrentTaskM);

    /* If no task is ready to run, then Idle Task is definitely ready. */
    if(readyTasklistM.Empty())
    {
        readyTasklistM.Put(pIdleTaskM);
    }

    /* Adding this, it makes possible to delay some tasks more than needed. */
    //SysTick_Config(Config::SYSTICK_HZ_RATE);

    eStateM = STARTED;
}

/*
 * Checks for new ready task from timers
 */
```

```
void Scheduler::CheckSuspendedTasks()
{
    int i = 0;
    while(i < waitingTaskListM.Count())
    {
        Task* task = waitingTaskListM[i];
        if(task->stateM == Task::SUSPENDED && --(task->nTicksSuspendedM) <= 0)
        {
            /* Add the task to ready list */
            readyTasklistM.Put(task);
            /* Remove the task from suspended list */
            waitingTaskListM.Remove(i);
        }
        else
        {
            /* go to next */
            ++i;
        }
    }
}

/*
 * Check for new ready task from Mutexes/Semaphores
 */
void Scheduler::CheckWaitingTasks()
{
    int i = 0;
    while(i < waitingTaskListM.Count())
    {
        Task* task = waitingTaskListM[i];
        if(task->stateM == Task::READY) /* state updated from mutex release */
        {
            /* Add the task to ready list */
            readyTasklistM.Put(task);
            /* Remove the task from suspended list */
            waitingTaskListM.Remove(i);
        }
```

```
        else
        {
            /* go to next */
            ++i;
        }
    }
}



/*
 * For example :
 *     SYSTICK_HZ_RATE is set to a period of 10 ms
 *     msec value is 1000ms
 * The result will be 100 : we need 100 * 10ms periods to acquire 1000ms.
 */
uint32 Scheduler::TicksForPeriod(uint32 msec)
{
    return msec / ( Config::SYSTICK_HZ_RATE * 1000 / CLOCK_FREQUENCY );
}

/*
 * Idle Callback function
 */
void Scheduler::IdleFunction( void* )
{
      for(;;)
      {
            UART::LogString(UART::VERBOSE, "Idle");
          __WFI();
      }
}
```

Anexa 4. Modulul Syncronization

- Fișierul Mutex.h

```
#include <Kernel/TaskManager/TaskArray.h>

namespace Kernel
```

```
{

/*
 * This class provide methods to synchronize(serialize)
 *    the access to critical sections.
 * Communicates directly to Scheduler.
 */
class Mutex
{
public:

    /* Initialize Mutex object. State is Available. */
      Mutex();

      /*
       * Try to get the lock.
       * If mutex is available, then the lock is taken. The calling task
       *    is set as owner of the lock.
       * Otherwise, the calling task is enqueued in a waiting queue
       *    until the lock will be released.
       * Also, if the calling task is added into the waiting queue,
       *    it will be suspended by the Scheduler until the lock will be
       *    available for it.
       */
      void Lock();

      /*
       * Release the current lock.
       * Only the owner of the lock can unlock the mutex.
       * If the calling task is the owner, then the first task from waiting
queue, if any :
       *    - is set as the new owner
       *    - is removed from from the waiting queue.
       *    - is resumed by the Scheduler.
       */
      void Release();

      /*
       * Should not be called
       * Removes all the threads from waiting list in order to be resumed by the
kernel
       */
      ~Mutex();

public:

      /*
       * This class provides fast methods, from code size point of view,
       * to Lock and Release a Mutex. Used like following :
       *  Mutex mutex;
       *  {
       *      Autolock lock(&mutex);
       *      ...
       *  }
       */
      class Autolock
      {
      public:
          inline Autolock(Mutex* mutex)
          {
              pMutexM = mutex;
              if(pMutexM != 0)
              {
```

```
                pMutexM->Lock();
            }
        }

        inline ~Autolock()
        {
            if(pMutexM != 0)
            {
                pMutexM->Release();
            }
        }
    private:
        Mutex* pMutexM;
    };

private:

    enum { Available, Locked } stateM; /* Mutex states */

  Task*     pOwnerM;       /* Stores the owner of the lock */
    TaskArray waitingQueueM; /* waiting queue */

}; /* Mutex */

} /* Kernel */
```

- Fișierul Mutex.cpp

```
#include "board.h"

#include <Kernel/Scheduler/Scheduler.h>

#include <Kernel/Io/Uart.h>
#include <Kernel/Sync/Mutex.h>
#include <Kernel/Sync/os_functions.h>

using namespace Kernel;

/*
 * Mutex Initialization
 */
Mutex::Mutex()
{
    OS_EnterCritical();

    stateM  = Available;
    pOwnerM = 0;
    waitingQueueM.Clear();

    OS_ExitCritical();
}

/*
 * Get the Lock
 */
void Mutex::Lock()
{
    UART::LogTask(UART::VERBOSE, Scheduler::pCurrentTaskM->taskIdM, "Lock");

    OS_EnterCritical();

    if(stateM == Available)
    {
```

```
        /* Update the state and the owner */
        stateM  = Locked;
        pOwnerM = Scheduler::pCurrentTaskM;
        waitingQueueM.Clear();
    }
    else
    {
        /* Add current task to waiting list */
        waitingQueueM.Add(Scheduler::pCurrentTaskM);

        /* Change current task state to waiting */
        Scheduler::SetCurrentTaskWaiting();

        /* Set PendSV Irq on order to run other task */
        CallDispatcher();

        /* End Critical Section */
        // OS_ExitCritical();

        /* Wait for context Change */
        //__WFI();
    }

    OS_ExitCritical();
}

/*
 * Release the lock
 */
void Mutex::Release()
{
    UART::LogTask(UART::VERBOSE, Scheduler::pCurrentTaskM->taskIdM, "Release");

    /* Start Critical */
    OS_EnterCritical();

    /* Only the owner can release the mutex */
    if(pOwnerM != Scheduler::pCurrentTaskM)
    {
        UART::LogTask(UART::ERROR, Scheduler::pCurrentTaskM->taskIdM, "Mutex
Release : Not the owner!!!");
    }

    if(stateM == Locked)
    {
        UART::LogTask(UART::VERBOSE, Scheduler::pCurrentTaskM->taskIdM, "State
Locked");

        /* Check for waiting queue */
        if(waitingQueueM.Empty())
        {
            stateM = Available;
        }
        else
        {
            /* Wake the first waiting task. Actually,
             * the scheduler will decide if the task will be able
             * to run on the next timer event. Here we change
             * the availability of the task. */
            waitingQueueM[0]->stateM = Task::READY;

            /* Now we have the new owner */
            pOwnerM = waitingQueueM[0];
```

```
            /* Remove the task from waiting list */
            waitingQueueM.Remove(0);

            /* Add task to ready tasks list */
            Scheduler::CheckWaitingTasks();

            /* Give the chance to run - IF priority is considered here. */
            //SetPendSvInterrupt();
        }
    }

    /* End Critical */
    OS_ExitCritical();
}

/*
 * Should not be ever called
 */
Mutex::~Mutex()
{
    OS_EnterCritical();

    if(!waitingQueueM.Empty())
    {
        for(int i = 0; i < waitingQueueM.Count(); ++i)
        {
            waitingQueueM[i]->stateM = Task::READY;
        }
        Scheduler::CheckWaitingTasks();
        waitingQueueM.Clear();
    }

    OS_ExitCritical();
}
```