Universitatea POLITEHNICA București
Facultatea Automatică și Calculatoare
Departamentul Automatică și Informatică Industrială

# LUCRARE DE LICENȚĂ

# Actualizarea firmware-ului prin comunicație radio (Over-the-Air Update)

Coordonator
*Conf. Dr. Ing. Maximilian Nicolae*

Absolvent
*Dimoiu Mihai – Lorin*

**Anul absolvirii (2018)**

# Cuprins

# 1. Introducere

## 1.1. Generalități și motivație

În primele zile ale tehnologiei IoT, actualizarea dispozitivelor la distanță a cauzat adesea perturbări intermitente și de degradare a performanțelor. Pe măsură ce platformele IoT au ajuns la maturitate, s-a adoptat și un nou mod de actualizare de la distanță și fiabilitatea dispozitivelor conectate care nu au mai prezentat nicio întrerupere.

Actualizările firmware-ului prin radio se referă la practica actualizării de la distanță a codului, setări de configurație sau actualizarea cheilor de criptare pe un dispozitiv integrat. Hardware-ul integrat trebuie să fie construit în așa măsură încât această funcționalitate să fie suportată.

Actualizarea firmware-ului prin radio este una dintre caracteristicile principale ale producătorilor de IoT care își promovează produsele. Dar ce este ascuns în spatele acestor actualizări și care sunt avantajele și dezavantajele OTA?

Actualizarea firmware-ului descrie un concept general de furnizare a noului software responsabil pentru controlul hardware pus la dispoziție, folosind interfața aerului, ceea ce înseamnă una dintre interfețele de comunicație radio disponibile.

În ceea ce privește orice alt mecanism pentru actualizările de firmware existente, există multe funcții similare care sunt utilizate de alte domenii și uneori se amestecă ceea ce duce la neînțelegeri și așteptări greșite, de obicei din partea clientului.

Motivația a venit din necesitatea de a actualiza un nod ce se află deseori în locații care sunt la distanță sau greu accesibile, spre exemplu un senzor care se află pe tavan la 20 de metrii înălțime, sau un dispozitiv poziționat în pământ. Sistemele inteligente fac posibilă programarea prin aer a acestor dispozitive, fără a mai fi nevoie de acces fizic la dispozitiv, economisind bani și timp dacă nodul trebuie reprogramat.

În această lucrare se regăsesc combinate domeniul de transmisii radio și domeniul software și, de asemenea, domeniul hardware.

Mecanismul OTA are nevoie de software-ul existent si de hardware-ul dispozitivului țintă pentru a putea facilita actualizarea. Noul software este transferat către dispozitivul țintă, instalat, și pus în funcțiune. Deseori este necesară repornirea dispozitivului pentru ca noua actualizare sa prindă efect, această acțiune este făcută automat în cele mai multe cazuri.

Un aspect foarte important al programării radio este acela ca un nod central, de exemplu un server, poate trimite actualizarea la toți utilizatorii din raza de acțiune. Un utilizator poate refuza noile actualizări, defecta sau chiar altera actualizarea. Utilizatorul care refuza programarea poate fi exclus automat de pe canal.

Recent, cu noile concepte ale rețelelor de senzori wireless si ale internetului obiectelor, unde noile rețele constau in sute sau chiar mii de noduri, actualizarea OTA a luat o direcție nouă. Internetul obiectelor introduce multe provocări în implementări. Se văd în casele noastre un număr

crescut de dispozitive conectate la un router și PC-urile conectate la multiple dispozitive inteligente care închid ușa de la distanță, deschid jaluzelele, controlează lumina, temperatura și umiditatea. Pentru ca aceste dispozitive să rămână securizate, actualizarea OTA trebuie să fie integrată în dispozitivele IoT.

Pentru prima dată programarea OTA este aplicată pentru benzile de frecventa nelicențiate, numite si ISM – Industrial, Scientific and Medical.

Plasarea beacon-urilor în locuri greu accesibile, pentru a produce informații despre un produs ce va fi afișat pe un dispozitiv inteligent, ca de exemplu smartphone, atunci când acesta intră în raza de acțiune a beacon-ului. Numărul beacon-urilor face imposibilă programarea lor individuala după ce au fost plasate. Actualizarea firmware-ului devine un factor important în salvarea timpului.

Aceste benzi sunt :

- 868 – 868.6 MHz, în Europa, 1 canal.
- 900 – 928 MHz, în America de Nord, până la 10 canale.
- 2400 – 2483.5 MHz, în întreaga lume, până la 16 canale.

De asemenea in programarea OTA se aplică un consum de energie mic și viteze de transmitere reduse, folosind protocoale precum **802.15.4**, **Wi – Fi**, **Bluetooth**, **ZigBee**, etc.

## 1.2. OTA vs. Non OTA

În general, termenul OTA implică folosirea mecanismelor wireless pentru a distribui date sau pachete cu actualizări pentru firmware sau actualizări software pentru un dispozitiv mobil. Acest lucru îi facilitează utilizatorului să nu mai meargă la un centru service, pentru schimbarea software-ului, actualizarea lui, sau schimbarea de parametrii.

Metodele Non OTA sunt reprezentate de faptul că utilizatorul trebuie să meargă la service și să ceară ajutor, sau folosirea unui PC și a unui cablu pentru a conecta dispozitivul pentru a schimba setările, adăugarea software-ului, etc.

Oricum, actualizările OTA nu sunt un topic nou, această arie a fost cercetată și există deja diferite forme de implementare pe piață. De exemplu, actualizările de firmware a telefoanelor inteligente sau actualizările de aplicații de pe acestea. Diferența majoră între aceste forme de actualizare prin radio este în actualizarea sistemelor integrate (SoC), unde restricțiile sunt foarte mari, resursele sunt critice și operează în aplicații critice.

Dacă prin diferitele metode care au fost implementate, nu este gestionat corect procesul de actualizare a software-ului, și securizat, în cele mai multe cazuri pot duce la pierderi financiare și pot avea consecințe grave.

Procesele tehnologice aduc numeroase aplicații în număr din ce in ce mai mare de sisteme. Producătorii de sisteme ar dori să aducă timpul de ieșire pe piață cât mai mic, fără a ignora cerințele privind siguranța și funcționalitatea. Aceasta este mai mult o provocare de inginerie și este specifică implementării, dar totuși un software portabil este foarte apreciat.

Prin urmare, una din soluțiile portabile de actualizare a firmware-ului pentru o resursă constrânsă a sistemelor integrate este un domeniu interesant de cercetare.

## 1.3. Considerente de securitate

În orice moment al unui dispozitiv conectat la internet, el devine exploatabil. În ce mod actualizările prin radio sunt trimise pot introducere multe riscuri, malware, căderi de sistem, defecțiuni fizice în timpul căderii sistemului și exploatarea datelor personale.

Așadar aceste actualizări prin radio sunt cruciale din punct de vedere al securității canalului prin care sunt primite actualizările de către dispozitivul țintă și trebuie evitate cât mai mult posibil breșele de securitate.

Unul din motivele sistemelor conectate sunt vulnerabilitățile deoarece ele nu primesc actualizări regulate.

Dispozitivele conectate OTA au nevoie de actualizări, dar orice dată trimisă sau recepționată de către dispozitiv trebuie să includă și componente de securitate pentru protecția utilizatorilor.

## 1.4. Soluții de securitate pentru actualizarea OTA

Dispozitivele OTA au nevoie de securitate punct la punct și trebuie luat în considerare acest lucru pentru toate dispozitivele noi fabricate. În general, o soluție completă de securitate pentru dispozitivele conectate și a actualizărilor OTA ar trebui să includă identitate, autentificare, criptare, infrastructura securizată, mai multe firewall-uri, etc.

Certificatele digitale joaca un rol foarte important în securitatea prin radio. Certificatele folosite ar trebui sa asigure furnizarea identității pentru fiecare dispozitiv criptat pentru datele în tranzit. Certificatele autentifica utilizatorii sau dispozitivul pentru a stabili dacă sunt eligibili pentru a primi actualizarea.

De asemenea pot introduce un semn pentru a verifica dacă mesajul vine dintr-o sursă corectă si nu a fost alterat pentru a dispozitivul.

Acesta este doar începutul. Peste 200 de miliarde de dispozitive din toată lumea vor avea capabilitatea de se actualiza prin radio până în anul 2020, potrivit Intel [1].

Securitatea ar trebui să fie construită în momentul fabricației pentru a proteja milioane de utilizatori în viitor.

Un protocol de securitate, protocolul criptografic sau protocolul de criptare este un protocol abstract sau concret care efectuează o funcție legată de securitate și aplică metode criptografice, adesea ca secvențe ale primitivelor criptografice. Un protocol descrie modalitatea de utilizare a unor algoritmi.

# 2. Studiul actual al soluțiilor în domeniu

Este importantă documentarea in metoda de actualizare radio suportată de către producători și utilitarele de dezvoltare când se selectează componentele dorite pentru viitoarea aplicație IoT ce va fi dezvoltată. Când un dispozitiv cu memorie Flash este evaluat, actualizarea prin radio oferă o abordare economica, deoarece nu este necesar o memorie flash externă, ceea ce implica costuri ridicate.

Unele dispozitive nu acceptă actualizarea bootloader-ului prin radio, deoarece acesta a fost protejat, sau a fost scris in silicon de către producător. De aceea această lucrare vine în ajutorul utilizatorilor care vor să își actualizeze complet dispozitivele.

Este foarte important să înțelegem aceste constrângeri în alegerea dispozitivelor pentru aplicația dorită.

Când se fac actualizări prin radio, datele trimise prin radio sunt susceptibile de a fi interceptate. De aceea este important ca autentificarea să fie făcută și să fie asigurat că dispozitivul țintă este singurul care interpretează datele. Datele trimise prin radio trebuie criptate pentru ca alte dispozitive radio de comunicare să nu poată decoda datele.

Versiunea 4.2 a BLE a implementat mecanisme de securitate care cresc protecția împotriva atacurilor de tip “man – in – the – middle” și a pasivității ascultării pentru a reduce furtul de IP și preluarea dispozitivului de către atacator.

Dacă se folosește o versiune mai veche a BLE, este bine ca să se adauge manual un strat de protecție a IP-ului dumneavoastră pentru a evita pasivitatea ascultării.

Cerințele produselor din marketul IoT se schimbă mai rapid decât cerințele produselor tradiționale. O tendință viitoare pentru producătorii de automobile este furnizarea de actualizări de firmware prin aer ca și serviciu. Deoarece acesta controlează funcționalitatea unui vehicul, securitatea este foarte importantă. Este necesară extinderea procedurii de scriere în memorie pentru a verifica dacă firmware-ul a fost scris corect în ECU. Procedura de verificare implică doar funcții de hash foarte simple și este astfel potrivită pentru resursele limitate ale unui vehicul [2].

Un vehicul obișnuit conține o rețea de sisteme embedded formată din 40 – 60 de unități. Pe măsură ce se dezvoltă funcționalități din ce în ce mai avansate, există nevoia de a actualiza software-ul care rulează pe aceste dispozitive embedded. Actualizările sunt lansate pentru a îmbunătății funcționalitatea existentă sau pentru a remedia erorile descoperite după lansarea în producție a software-ului. Există mai multe avantaje cu această abordare. În primul rând, aceasta implică inconveniente minime pentru clienți, deoarece nu este necesar ca clientul să aducă vehiculul la o stație de service pentru actualizarea firmware-ului. În al doilea rând, permite actualizări mai rapide. Odată ce este lansat firmware-ul, acesta poate fi descărcat și instalat în vehicul.

Vehiculele inteligente moderne au unități electronice de control care conțin firmware care permite diferite funcții în vehicul. Versiunile noi de firmware sunt dezvoltate în mod constant pentru a elimina erorile și pentru a îmbunătății funcționalitatea. [3]

Cel mai bun exemplu deocamdată al actualizării OTA este cel al companiei Tesla [4], prin care mașinile Tesla pot primi actualizări software Over the Air și nu mai este nevoie pentru a rechema în proprietarul împreuna cu mașina în service pentru a face această actualizare.

Alt exemplu foarte bun este actualizarea software a telefoanelor inteligente de către producători precum Google, Apple, Huawei, etc.

Să ne amintim că IoT nu este o "jucărie" la modă, dar, da, va face unele ridicări grele ca schimbător de joc pe termen lung pentru afaceri si societăți. Pentru noi, aceasta a fost revelația făcută de Tesla : își poate fixa mașinile "prin aer", în timp ce stau în garajele sau căile de acces ale proprietarilor sau în parcarea de la locul de muncă, în aproape măsura în același mod în care telefoanele inteligente primesc actualizări software.

În aproape toate situațiile, principala misiune a internetului obiectelor este să faciliteze îndepărtarea activității fără valoare adăugată din cursul vieții de zi cu zi, fie la serviciu, fie privat. În cazul Tesla, acest rol este clar. Mai degrabă decât să aibă sarcina obositoare a unei călătorii neplanificate la dealer, proprietarii Tesla pot să-și petreacă toată ziua cu alte activități în timp ce mașina se "repară" singură. Cu alte cuvinte, există o valoare monetară imediată, iar tehnologia se extinde diferențiind marca.

Un alt lucru, chiar mai pasiv, pe care IoT este obligat sa îl introducă în viața de zi cu zi, vine sub forma de mașini care pot simți în mod pre-emptiv când uleiul trebuie schimbat și nu datorită trecerii timpului sau a kilometrilor, ci prin modul în care mașina a fost utilizată in diferite condiții pentru a-și îndeplini funcția în interiorul motorului, și anume vâscozitatea reală și temperatura de vârf.

Încărcarea autoturismelor noastre cu toți acești senzori, procesoare și cel mai important, un mijloc de comunicare este primul pas spre conceptele mult discutate despre vehicul – vehicul și vehicul – infrastructura, ambele concepte fiind precursoare pentru viitorul "fără șofer". Ceea ce a făcut Tesla este că autovehiculele pot accepta și transmite informații unei mașini externe.

Conform [5] IoT a luat naștere de-a lungul ultimilor ani și a avut o desfășurare substanțială pentru mai multe standarde, arhitecturi și platforme. Acest firmware trebuie să fie într-o continuă dezvoltare pentru a elimina erorile și pentru a îmbunătăți funcționalitatea dispozitivului. Este de preferat un sistem de actualizare a firmware-ului, pentru că permite actualizarea rapidă și încurajează abilitatea pentru dezvoltare. IoT-ul a apărut ca o paradigmă de comunicare a diverselor obiecte integrate și interconectate cu microcontroller-ele, care le permit să comunice și să interacționeze cu utilizatorii pentru a atinge obiective comune.

IoT-ul preia încet lumea unde s-a prezis ca până în 2020 până la 200 de miliarde de dispozitive conectate. Cu implementări la scară largă, menținerea acestor dispozitive reprezintă un lucru greu a operațiunilor. De asemenea, implementările uriașe ale dispozitivelor IoT, cele mai multe preocupări se concentrează pe confidențialitate și securitate. Pentru a reduce din efortul de întreținere pe teren și pentru a îndeplini aceste sarcini de la distanță și în siguranță, poate fi adoptată aplicația actualizării prin radio [6].

Vehiculele viitoare vor fi conectate fără fir la vehiculele din apropierea acestora, de asemenea la infrastructura rutieră și la internet, devenind o parte a internetului. Cu noile caracteristici de confort, funcțiile de siguranță și o serie de servicii noi pentru vehicule, vor fi integrate în toate vehiculele inteligente în viitorul apropiat. O metodă rapidă, fiabilă și sigură de

diagnosticare și reconfigurare a vehiculului, precum și instalarea de un nou software pe unitățile electronice de control (ECU). Actualizările software sunt extrem de benefice pentru clienți pentru a repara defecțiuni software anterioare, activarea unor noi funcții, etc [7].

# 3. Definirea problemei

## 3.1. Cerințe și complexitate

Câteva cerințe extrem de importante pentru ca produsul final să funcționeze în parametrii, sunt următoarele: securitate, robustețe, atomicitate, protejat de eșecuri. Vom vorbi în continuare despre fiecare în parte.

- Securitate : este factorul cel mai important, din punct de vedere financiar. Poate duce prin brețele de securitate la pierderi de date, coruperea dispozitivelor fizice, pierderi financiare foarte mari și instabilitatea sistemelor.
- Robustețe : trebuie ca sistemul să fie capabil să facă față erorilor în timpul executării și de a face față intrării eronate. Tehnicile formale, cum ar fi testul "fuzz", sunt esențiale pentru a demonstra robustețea deoarece acest timp de testare implica intrări nevalide sau neașteptate.
- Atomicitate : implică indivizibilitate și ireductibilitate, deci o actualizare prin radio a software-ului sistemului trebuie ca să fie instalată în întregime sau deloc realizată.
- Protejată de eșecuri (fail-safe) : este un mecanism de inginerie în caz de eșec sau o practică de proiectare care, în cazul unui tip specific de defecțiune, răspunde în mod inerent într-un mod care nu va produce niciun prejudiciu unui echipament, mediului în care dispozitivul se află sau a oamenilor. Ar trebui dacă există erori de sistem, acesta să cunoască ultima dată când a funcționat bine și să se întoarcă acolo.

În ceea ce privește complexitatea în actualizările radio, trebuie luat în calcul mai multe aspecte precum autentificarea, securitatea, roll-back și monitorizarea. În cele ce urmează vom discuta despre aceste aspecte pe rând.

- Autentificarea : Este această actualizare legitimă? Autentificarea este actul de confirmare a adevărului unui atribut al unei singure bucăți de date revendicate de o entitate. Spre deosebire de identificare, care se referă la actul de a declara sau a indica astfel o afirmație care atestă identitate unei persoane sau a unui lucru, autentificarea este procesul de confirmare a identității.
- Securitatea : Primesc ceea ce tu trimiți? Securitatea este libertatea sau rezistența față de eventualele pagube. Beneficiarii de securitate pot fi dispozitivele integrate, telefoanele inteligente, mașinile inteligente, și orice alt fenomen vulnerabil la schimbările nedorite prin mediul său.
- Roll-back : Descrie procesul de întoarcere a unui produs hardware sau a unui program software înapoi la o versiune anterioară după ce a întâmpinat probleme cu o versiune ulterioară.

- Monitorizare : Acest aspect introduce căi critice în dezvoltarea ulterioară a sistemelor, deoarece monitorizarea și gestionarea performanțelor se străduiește să detecteze și să diagnosticheze problemele complexe legate de performanța sistemelor, pentru a menține nivelul așteptat al serviciilor.

Exista două seturi de măsurarea performanței care sunt monitorizate îndeaproape :

- Primul set de performanță definește performanța experimentală de către utilizatorii finali ai aplicației. Un exemplu de performanța este timpul mediu de răspuns în sarcină maximă. Componentele setului includ timpul de încărcare și de răspuns.
- Cel de-al doilea set de metrici de performanță măsoară resursele de calcul utilizate de aplicație pentru sarcină, indicând dacă există o capacitate adecvată pentru a suporta sarcina, precum și posibile localizări ale unui blocaj de performanță. Măsurarea acestor cantități stabilește o bază de performanță empirică pentru aplicație. Linia de bază poate fi apoi utilizată pentru a detecta modificările în performanță. Modificările în performanță pot fi corelate cu evenimente externe și ulterior utilizate pentru a anticipa schimbările viitoare în performanța aplicațiilor.

## 3.2. Firmware

Firmware-ul este o clasă specifică de software care oferă controlul la nivel scăzut al hardware-ului specific al dispozitivului. Acesta este responsabil pentru funcționarea generală a întregului sistem. Este adesea stocat pe memoria flash de pe chip. Firmware-ul poate oferi fie un mediu de operare standard pentru software-ul mult mai complex al dispozitivul de pe care rulează, fie funcționează executând toate funcțiile de control, monitorizare și manipulare a datelor. Câteva exemple de dispozitive care conțin firmware sunt sistemele embedded, aparatele de consum, calculatoarele, perifericele calculatoarelor, etc. Aproape toate dispozitivele electronice mai noi conțin firmware.

Firmware-ul este păstrat în dispozitive de memorie nevolatile, ca de exemplu ROM, EEPROM, sau memorie flash. Pentru a schimba firmware-ul unui dispozitiv, se poate face mai rar sau deloc în timpul vieții. Unele dispozitive de memorie unde au amplasat firmware-ul sunt memorii permanente și nu se pot modifica după fabricație. Motivele obișnuite pentru actualizarea firmware-ului includ adăugarea de funcții noi și remedierea defectelor sau a erorilor. Această modificare a firmware-ului necesită înlocuirea fizică a circuitelor ROM integrate sau a memoriei flash pentru a fi reprogramată printr-o procedură specială.

Abilitatea de a actualiza firmware-ul prin aer este tema centrală a acestei lucrări.

## 3.3. Bootloader

Bootloader-ul este o aplicație a cărui scop principal este de a permite actualizarea unui software de sistem fără utilizarea unui hardware specializat, cum ar fi un programator JTAG – **J**oint **T**est **A**ction **G**roup, sau a unei interfețe SWD – **S**erial **W**ire **D**ebug. Bootloader-ul

gestionează imaginile sistemului. Acestea pot comunica pe o varietate de protocoale, cum ar fi modulul UART – **U**niversal **A**synchronous **R**eceiver **T**ransmitter, CAN, $I^2C$, Ethernet, USB, Bluetooth, Wi – Fi, etc. Sistemele cu bootloader au cel puțin 2 imagini coexistente pe același microcontroller, și trebuie să includă codul de verificare dacă este în desfășurare o nouă de actualizare a software-ului.

# 4. Arhitectura hardware

Proiectarea constă în dezvoltarea unui firmware, numit bootloader, care va fi responsabil cu pornirea și verificarea sistemului, iar apoi va trebui să realizeze conexiune radio cu gateway-ul pentru a manipula datele primite prin radio de la acesta și sa le interpreteze corespunzător.

Pentru a realiza acest proiect s-au folosit 2 plăci de dezvoltare CC2650 de la producătorul Texas Instruments. Aceste plăci au diagrama bloc după cum urmează :

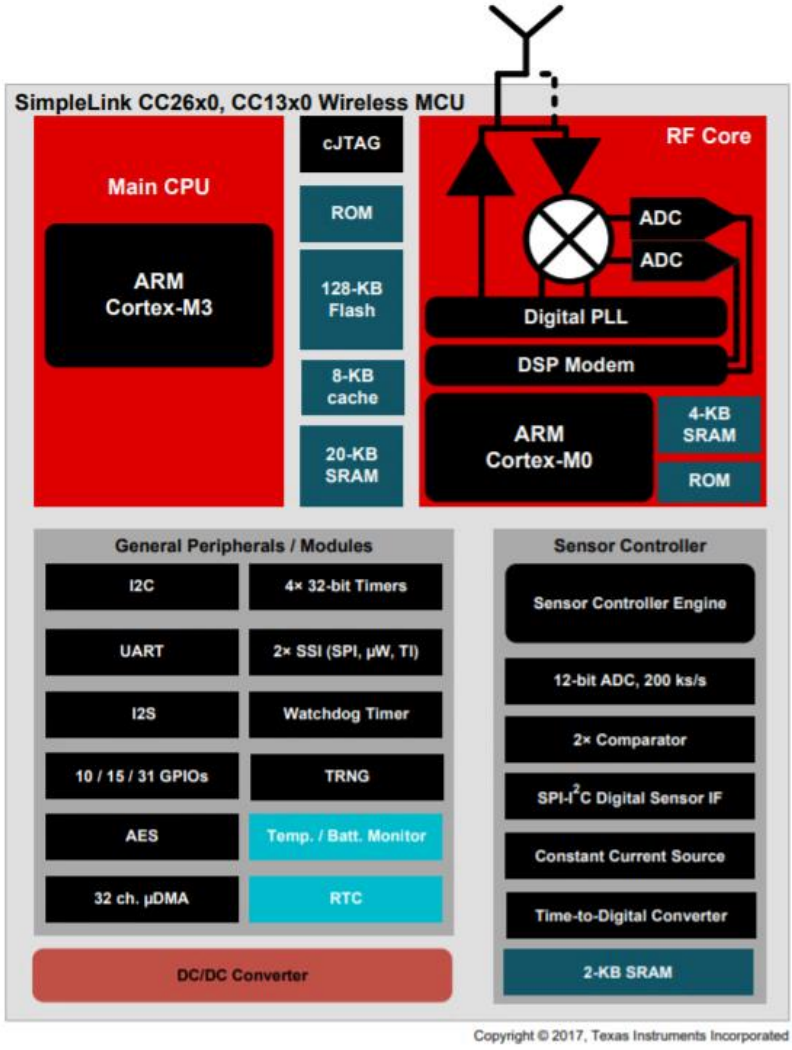


*Fig. 4.1 Diagrama bloc a dispozitivului CC2650 [8]*

Am ales această placă de dezvoltare deoarece oferă o multitudine de posibilități de a lucra cu ea. Comparând cu alte plăci ce se găsesc pe piață, cum ar fi Nordic nRF52832, ESP8266, NXP KW40z, Microchip RN4020. Comparativ cu aceste plăci din exemplul anterior, placa pe care am ales-o mi s-a părut fiind cea mai accesibilă, bucurându-mă de o documentație vastă.

Luând în considerație specificațiile din figura 4.1, am decis că este cea mai potrivită alegere pentru lucrarea mea.

## 4.1. Placa CC2650 LaunchPad Texas Instruments

Plăcile CC2650 LaunchPad sunt construite în jurul nucleului de la ARM ® Cortex – M3 [21] care aduce performanță ridicată, având arhitectura pe 32biți. Acest procesor aduce performanțe ridicate, un cost scăzut al platformei care îndeplinesc cerințele minimale ale implementării memoriei, număr de pini reduși și o putere de consum foarte mică. Următoarele caracteristici aparțin acestui nucleu:

- Arhitectură pe 32biți optimizată pentru o amprentă redusă.
- Performanță excelentă de procesare combinată cu manipularea rapidă a întreruperilor
- Combinarea instrucțiunilor ARM Thumb® - 2 și a instrucțiunilor pe 32biți într-un set ce livrează o performanță ridicată așteptată pe plaja de 32biți pentru câțiva KBytes de memorie pentru clasa de aplicații.
    - Pe un singur ciclu se execută mai multe instrucțiuni și împărțire hardware
    - Manipulare atomică a biților (bit – banding), livrare maximă a memoriei folosită
    - Acces nealiniat la date, face eficientă împachetarea datelor în memorie
- Executare rapidă a codului permite încetinirea vitezei procesorului sau creșterea timpului de sleep.
- Arhitectură Harvard caracterizată prin magistrale separate pentru date și respectiv magistrală pentru instrucțiuni.
- Determinist, performanță ridicată pentru manipularea întreruperile pentru aplicații critice de timp real .
- Sistem de depanare (debug) îmbunătățit cu capabilități extinse de întrerupere și trasare
- Migrarea de la familia de procesoare ARM7 ™ pentru o performanță mai bună și eficiență energetică.
- Optimizat pentru un singur ciclu în folosirea memoriei flash.
- Energie consumată Ultra – Low integrată în modurile de sleep.
- Operează la 48 MHz.

Diagrama bloc a procesorului Cortex – M3 arată nucleul principal. Acest procesor este construit pe un nucleu de performanță ridicată cu 3 stagii de pipeline, având arhitectură Harvard, deci fiind ideal pentru cererile de aplicații embedded. Procesorul oferă o eficiență din punct de vedere energetic foarte bună printr-un set de instrucțiuni eficiente și un design optimizat la o scară largă, care oferă de asemenea o procesare de ultimă generație. Setul de instrucțiuni include o plajă de multiplicări SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) și înmulțiri cu capacități acumulate, saturație aritmetică și diviziune hardware dedicată.

Pentru a facilita un preț cât mai avantajos la cumpărare, acest procesor Cortex – M3 implementează componente de sistem integrate pe o suprafață cât mai redusă și în felul acesta se reduce zona de ocupare procesorului, îmbunătățind și în același timp manevrabilitatea întreruperilor și capacităților de debug a sistemului. Cortex -M3 implementează o versiune a setului de instrucțiuni Thumb, bazat pe tehnologia Thumb – 2, astfel se asigură densitate mai mare a codului și cerințele de memorie pentru program sunt reduse. Procesorul Cortex – M3 oferă prin setul de instrucțiuni o performanță uimitoare așteptată de la o arhitectură modernă pe 32biți, cu densitatea mare a codului de 8biți, respectiv, de 16biți.

Procesorul Cortex – M3 integrează foarte îndeaproape un controller vectorial de întrerupere imbricată (NVIC – **N**ested **V**ector **I**nterrupt **C**ontroller) pentru a asigura execuția rapidă a rutinelor de rezolvare a întreruperilor, reducând drastic întârzierea întreruperii(ISR – **I**nterrupt **S**ervice **R**outines).

Prin construcția hardware se stivuiesc regiștrii și capacitatea de a suspenda operațiile de încărcare și de stocare multiplă ceea ce reduce și mai mult întârzierea latenței. Optimizarea pipe-line-ului, de asemenea, reduce, într-un mod semnificativ costurile aferente, atunci când se trece de la un ISR la altul. Pentru a optimiza design-ul de consum redus, se integrează NVIC-ul cu modurile de sleep, incluzând modul deep – sleep, care permite întregului dispozitiv să se închidă rapid.

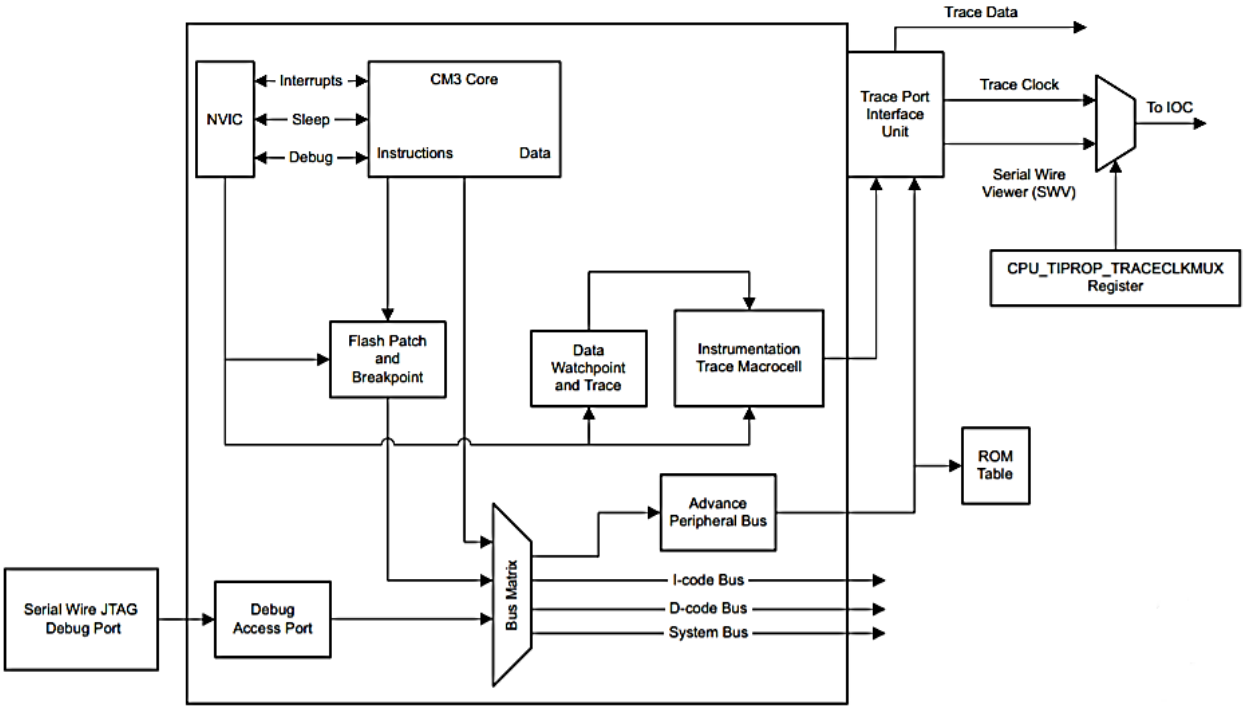


*Fig. 4.1.1 Vizualizarea de ansamblu a diagramei bloc a procesorului Cortex – M3 [9]*

Interfațarea de nivel înalt a procesorului Cortex – M3 utilizând tehnologia AMBA ® pentru a oferi acces la memoria de mare viteză și un timp de întârziere mai mic. Nucleul procesorului suportă accesul la date nealiniate și implementează manipularea atomică a biților, care permite comenzi mai rapide la periferice, sisteme spinlock-uri și manipularea într-un mod sigur a datelor booleene.

## 4.2. Regiștrii procesorului Cortex – M3

Regiștri de bază nu sunt mapați în memorie și sunt accesați prin numele regiștrilor, astfel încât adresa de bază nu este aplicabilă și nu există niciun offset.

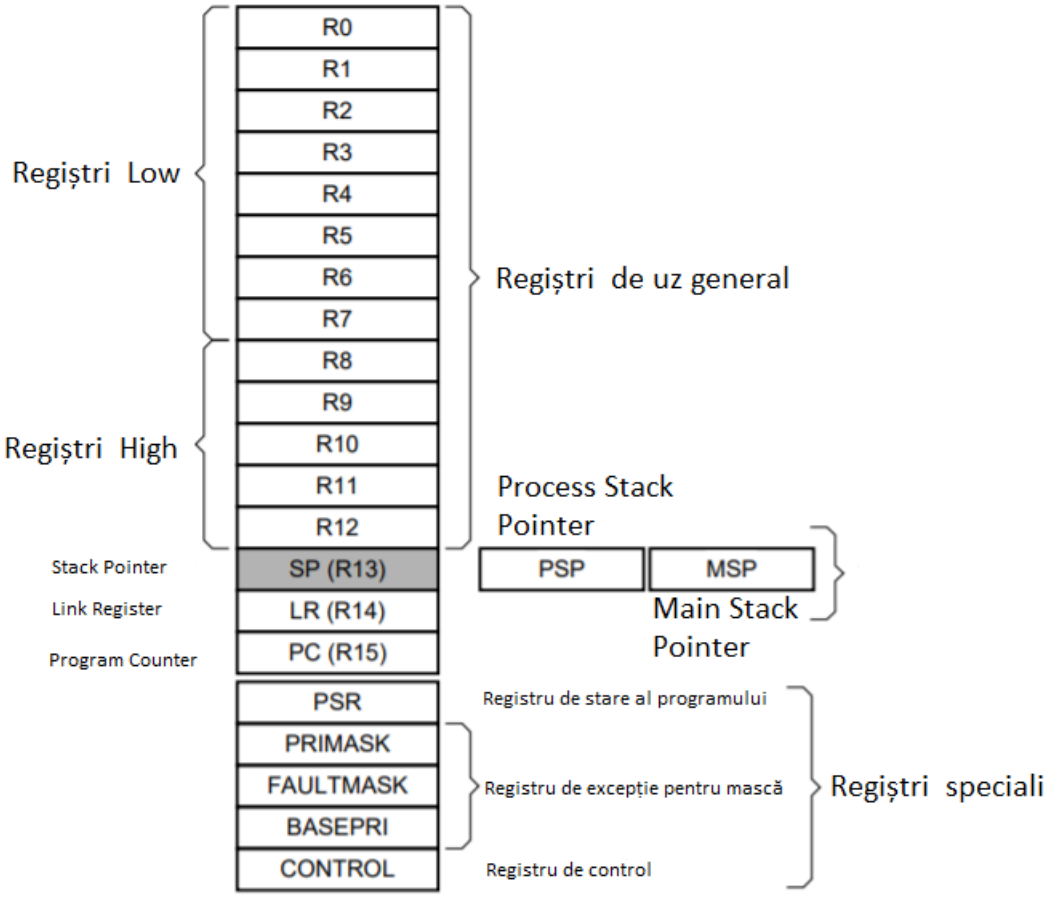


*Fig. 4.2.1 Setul de regiștri ai procesorului Cortex – M3 [9]*

## 4.3. Modurile de funcționare ale procesorului

Procesorul sistemului are 3 moduri diferite de funcționare, și anume : run, sleep și deep sleep. Fiecare mod este folosit pentru a închide ceasurile interne din CPU-ul sistemului, în plus față de ceasurile perifericelor care pot fi blocate în conformitate cu modul actual al CPU-ului din sistem. Modul de deep sleep este în unele cazuri, unul dintre multele cerințe pentru scăderea tensiunii și a puterii.

Sistemul de alimentare al dispozitivului CC2650 LaunchPad este unul complex și este controlat de către hardware.

## 4.4. Managementul tactului de ceas (Clock)

Placa de dezvoltare CC2650 LaunchPad are un ceas flexibil multiplexat unde sistemul de ceas poate fi derivat din mai multe surse. Următoarea figură arată acest lucru.

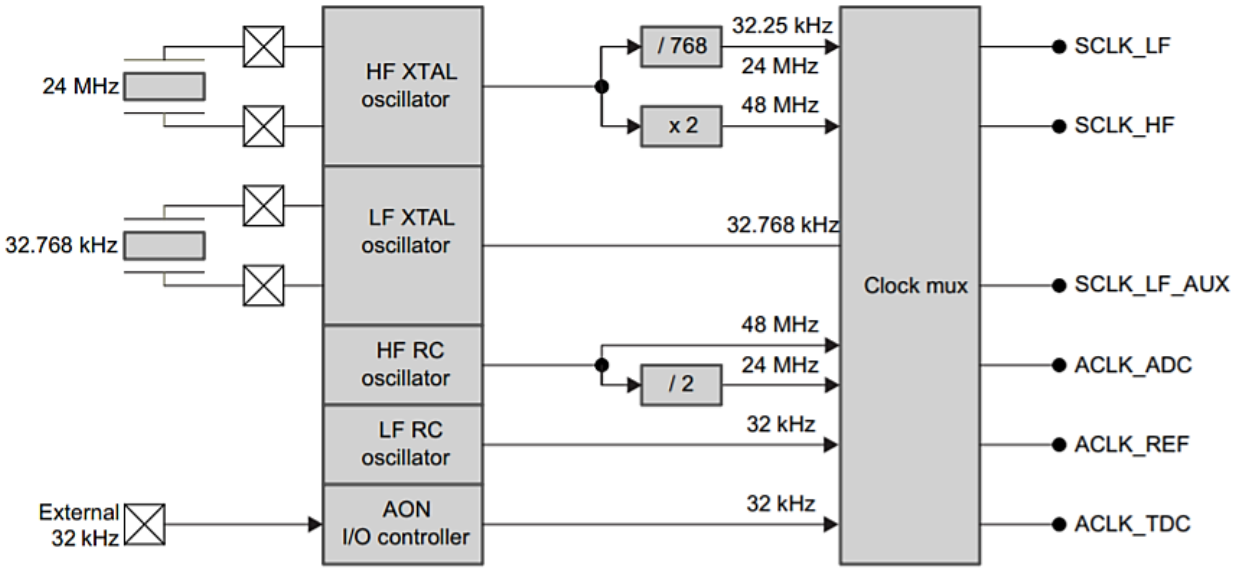


*Fig. 4.2.1 Sursele tactului de ceas [9]*

## 4.5. Perifericele procesorului Cortex – M3

Perifericele sistemului implementate sunt următoarele :

- Timer de sistem (SysTick) : Furnizează un mecanism simplu și flexibil de control.
- NVIC
  - Facilitează manipularea întreruperilor și a excepțiilor cu o întârziere mică.
  - Lucrează cu controllerul sistemului pentru a controla managementul de putere.
  - Implementează un sistem de control al regiștrilor.
- Bloc de control al sistemului Cortex – M3 : Furnizează informații despre implementările de sistem și controlul sistemului, incluzând configurarea, controlul și raportarea excepțiilor sistemului.

## 4.6. SysTick

Procesorul Cortex – M3 include un sistem de timer integrat, care oferă un counter simplu de 24biți, clear – on – write, descrescător, cu un mecanism flexibil de control. Counter-ul poate fi utilizat în mai multe moduri diferite. De exemplu counter-ul poate fi :

- Un timer RTOS care se declanșează la o rată programabilă (de exemplu, la 100 Hz) și invocă o rutină.
- O alarmă de mare viteză folosind ceasul de tact al sistemului.
- O alarmă cu rata variabilă sau un timer de semnal – durata este dependentă de ceasul de referință folosit și domeniul dinamic al counter-ului.

Timer-ul constă din trei regiștri și anume :

- SysTick control și registrul de stare (STCSR) : Un counter de stare și de control pentru a – și configura ceasul, activa counter-ul, a activa întreruperea sistemului SysTick și a determina starea counter-ului.
- Registrul de valoare a reîncărcării (STRVR) : Valoarea de reîncărcare a counter-ului, folosită pentru a furniza valoarea de înfășurare a counter-ului.
- Registrul de valori curente (STCVR) : Valoarea curentă a counter-ului

Când timer-ul este activat, începe contorizarea pentru fiecare puls de ceas de la valoarea reîncărcată la 0, reîncarcă valoarea în registrul STRVR pe următorul front al ceasului, apoi decrementează pe căderea frontului. Ștergerea registrului STRVR dezactivează counter-ul de pe următoarea înfășurare. Când counter-ul ajunge la 0, bitul de stare COUNTFLAG este setat. Apoi acest bit de stare COUNTFLAG se șterge după citire.

Scrierea în registrul STCVR șterge registrul și bitul de stare COUNTFLAG. Scrierea nu declanșează logica expresiei SysTick. Pe citire, valoarea curentă este valoarea registrului la momentul accesării registrului. Counter-ul SysTick rulează de pe ceasul sistemului. Dacă acest semnal de ceas este oprit pentru modul de economisire a energiei, counter-ul SysTick se oprește. Asigurarea că în software se folosesc cuvinte aliniate pentru accesarea regiștrilor SysTick.

## 4.7. NVIC

Această secțiune descrie modulul NVIC și suportă următoarele caracteristici :

- 34 de linii de întrerupere
- Prioritate programabilă pe 8 nivele de prioritate pentru fiecare întrerupere. Un număr mai mare în prioritate corespunde cu o prioritate mai mică la execuție. Deci nivelul 0 are cea mai mare prioritate la întrerupere
- Latență mică pentru tratarea excepțiilor și pentru manipularea întreruperilor
- Detectarea pulsului și a nivelului semnalelor de întrerupere
- Reprogramarea dinamică a nivelelor de întreruperi
- Gruparea în câmpuri prioritate a valorilor prioritare și sub prioritare
- Încascadarea întreruperilor
- Întreruperi nemascabile externe (NMI – **N**on **M**askable **I**nterrupt)

Procesorul stivuiește automat starea lui la intrarea în tratarea întreruperii și restabilește starea lui la ieșirea din aceasta, fără instrucțiuni generale, oferind o abordare excepțională a întârzierii.

O întrerupere rămâne în așteptare până când una din condiții se îndeplinește :

- Procesorul intră în rutina de tratarea întreruperii, se va schimbă starea întreruperii din așteptare în activ. Atunci :
    - Pentru o întrerupere de nivel sensibil, când procesorul revine din rutina de tratare a întreruperii, NVIC-ul eșantionează semnalul de întrerupere. Dacă semnalul este afirmat, starea întreruperii se schimbă în așteptare, ceea ce ar putea duce la reintroducerea imediată în rutina de tratare a întreruperii. În caz contrar, starea întreruperii se schimbă în stare inactivă.
    - Pentru o întrerupere de tip puls, modului NVIC continuă să monitorizeze semnalul de întrerupere și daca acesta de tip puls, starea întreruperii se schimbă din așteptare în activă. În acest caz, când procesorul se întoarce din rutina de tratare a întreruperii, starea întreruperii se schimbă în așteptare, ceea ce poate determina ca procesorul să reintre în rutina de tratare a întreruperii. Daca semnalul de puls al întreruperii nu mai pulsează, atunci când procesorul iese din rutina de tratare a întreruperii, starea întreruperii se schimbă în inactiv.
- Software-ul scrie bitul de înregistrare corespunzător al întreruperii :
    - Pentru o întrerupere de tip sensibilă la nivel, dacă semnalul de întrerupere este încă prezent, starea întreruperii nu se schimbă. În caz contrar, starea întreruperii se schimbă în stare de inactivitate.
    - Pentru o întrerupere de tip puls, starea întreruperii se modifică la starea inactiv dacă starea în care este era de așteptare sau la inactiv dacă stare era activă sau în așteptare.

După ce s-a scris să se șteargă o întrerupere, se poate ca modului NVIC să îi ia mai multe cicluri de procesare pentru a o detecta deșertarea ei din cauza memoriei tampon de scriere. Astfel dacă întreruperea se face ca o ultimă acțiune într-un manipulator de întrerupere, este posibil ca manipulatorul să finalizeze în timp ce modulul NVIC detectează întreruperea așa cum este încă afirmat, determinând reluarea erorilor pentru mecanism de preluare a întreruperilor. Această situație poate fi evitată prin ștergerea sursei de întrerupere la începutul întreruperii, fie prin efectuarea unei citiri de la aceeași adresă după scrierea pentru ștergerea sursei de întreruperi si golirea buffer-ului de scriere.

Procesorul ARM ® Cortex ® - M3 și controllerul de întrerupere vectorială imbricată (NVIC) prioritizată și gestionează toate excepțiile. Starea procesorului este stocată automat în stivă și la sfârșitul execuției rutinei de tratare a întreruperii, starea procesorului este refăcută. Vectorul este preluat în paralel, permițând astfel intrarea eficientă a întreruperii. Procesorul suportă încascadarea întreruperilor, ceea ce permite efectuarea performantă a întreruperilor fără suprapunerea stării de salvare și refacere a procesorului.

Stările de excepție :

- Inactiv : Excepția nu este activă și nici în așteptare.
- Așteptare : Excepția așteaptă sa fie tratată de către procesor. O cerere de întrerupere de la un periferic sau din software poate să își schimbe stare corespunzător întreruperii.
- Activ : O excepție este tratată de către procesor dar nu este finalizată. Manipulatorul de excepții poate întrerupe execuția ei din cauza altei întreruperi cu prioritate mai mare. În acest caz ambele întreruperi sunt în stare activă.

- Activă sau în așteptare : Excepția este tratată de către procesor, și este o excepție în așteptare de la aceeași sursă.

Tipurile de excepții :

- Reset : Resetarea este invocată la pornire rece sau pornire caldă. Modelul de excepție tratează resetarea ca formă specială de excepție. Când se declanșează resetarea, funcționarea procesorului se oprește, în orice moment al unei instrucțiuni.
- Excepție hardware : O excepție hardware este o excepție care apare din cauza unei erori în timpul procesării excepțiilor sau datorită faptului că o excepție nu poate fi gestionată de niciun alt mecanism de excepție. Excepțiile hardware au o prioritate fixată de -1, ceea ce înseamnă că acestea au o prioritate mai mare decât orice altă excepție.
- Excepție de magistrală : O excepție de magistrală este o excepție care apare din cauza unei erori legate de memorie pentru o instrucțiune, ca de exemplu o eroare prefetch. Această excepție poate fi activata sau dezactivată.
- Excepție de folosire : O eroare de utilizare este o excepție care apare din cauza unei defecțiuni legate de executarea instrucțiunilor, cum ar fi următoarele
    - O instrucțiune nedefinită
    - Un acces nealiniat ilegal
    - O stare invalidă pe o instrucțiune de execuție
    - O eroare pe returul unei excepții
- Întreruperi (IRQ) : O întrerupere sau IRQ este o excepție semnalizată de o aplicație periferică sau generată din software și adusă la NVIC (prioritizată). Toate întreruperile sunt asincrone la executarea instrucțiunilor. În sistem, perifericele folosesc sistemul de întreruperi pentru a comunica cu procesorul Cortex – M3.

  Manipulările excepțiilor se fac prin ISR-uri.

Intrarea în excepții se face când există o excepție în așteptare cu o prioritate mai mare și fie procesorul este în modul thread, fie noua excepție are o prioritate mai mare decât excepția tratată, in acest caz noua excepție face pre – emtivitate la excepția curentă.

Pre – emtivitate : Când procesorul execută un handler de excepție, o altă excepție poate preveni procedura de tratare a excepție dacă prioritatea sa este mai mare decât prioritatea excepției tratate. Când o excepție preemtează o altă excepție, excepțiile se numesc excepții imbricate.

## 4.8. Debug

Integrarea configurabilă a debug-ului pentru procesorul Cortex – M3, care implementează o soluție hardware – debug completă prin soluții de SWD sau JTAG. SW furnizează un sistem foarte vizibil a procesorului și a memoriei prin tradiționalul port JTAG.

Pentru urmărirea sistemului, procesorul integrează o macro celulă a urmăririi instrumentației (ITM – **I**nstrumentation **T**race **M**acrocell), pe lângă punctele de supraveghere a datelor. Pentru a furniza un profil simplu și rentabil a evenimentelor de urmărire a sistemului, un

SWV (**S**erial **W**ire **V**iewer) poate exporta un flux de mesaje generate de către software, printr-un singur pin al dispozitivului.

Subsistemul de debug al plăcii CC2650 care implementează 2 standarde IEEE pentru scopuri de debug și pentru testare :

- Standardul IEEE 1149.1 : Standardul pentru portul de acces a testării arhitecturii de scanare frontală. Acest standard este cunoscut sub acronimul JTAG.
- Clasa 4 IEEE 1149.7 : Standardul pentru portul de acces cu test redus și funcționalitatea îmbunătățită a portului de acces și a arhitecturii de scanare de frontieră. Acest lucru este cunoscut sub acronimul cJTAG (**c**ompact JTAG). Acest standard serializează tranzacțiile IEEE 1149.1 folosind o varietate de formate de compresie pentru a reduce numărul de pini necesari implementării unui port de debug JTAG.

Subsistemul de debug implementează de asemenea un firewall pentru accesul neautorizat la porturile de debug sau test. Figura următoare arata subsistemul de debug.

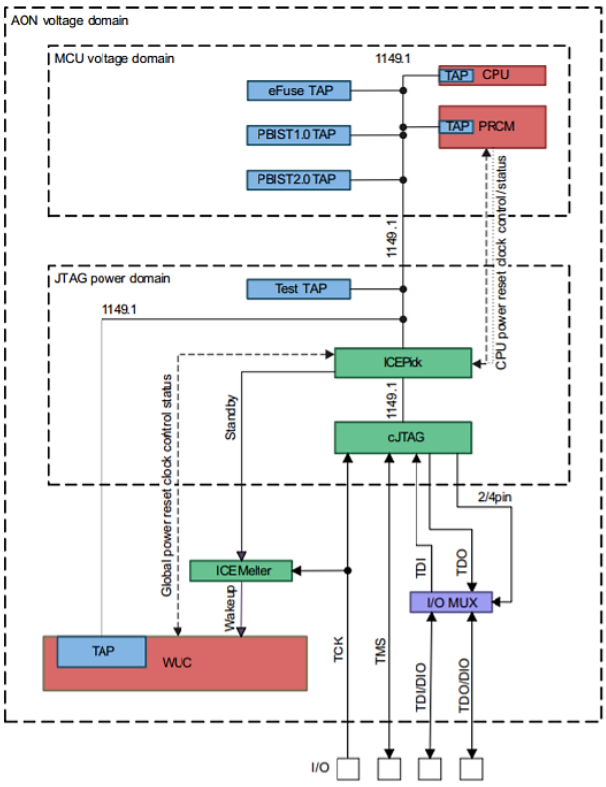


*Fig. 4.8.1 Subsistemul de debug [9]*

Standardul IEEE 1149.1 folosește următoarele semnale pentru a suporta operațiile :

- TCK (**T**est **Cl**ock) : Acest semnal sincronizează automatul de stare intern.
- TMS (**T**est **M**ode **S**elect) :Acest semnal este eșantionat la frontul de creștere a semnalului TCK pentru a determina starea următoare.
- TDI (**T**est **D**ata **I**n) : Acest semnal reprezintă datele care se shiftează în test sau programarea logică a dispozitivului. TDI este eșantionat pe frontul de creștere a lui TCK când automatul de stare intern este în starea corectă.

- TDO (**T**est **D**ata **O**ut) : Acest semnal reprezintă datele care se shiftează afara din testare sau programarea logica a dispozitivului și este valid pe frontul descrescător a lui TCK când automatul de stare intern este în starea corectă.

Nu exista pin dedicat I/O pentru TRST. Subsistemul de debug se resetează când restul sistemului se resetează sau la pornire.

Fiecare stare are două ieșiri, așadar toate tranzacțiile pot fi controlate de un singur semnal TMS eșantionat cu frecvența semnalului TCK.

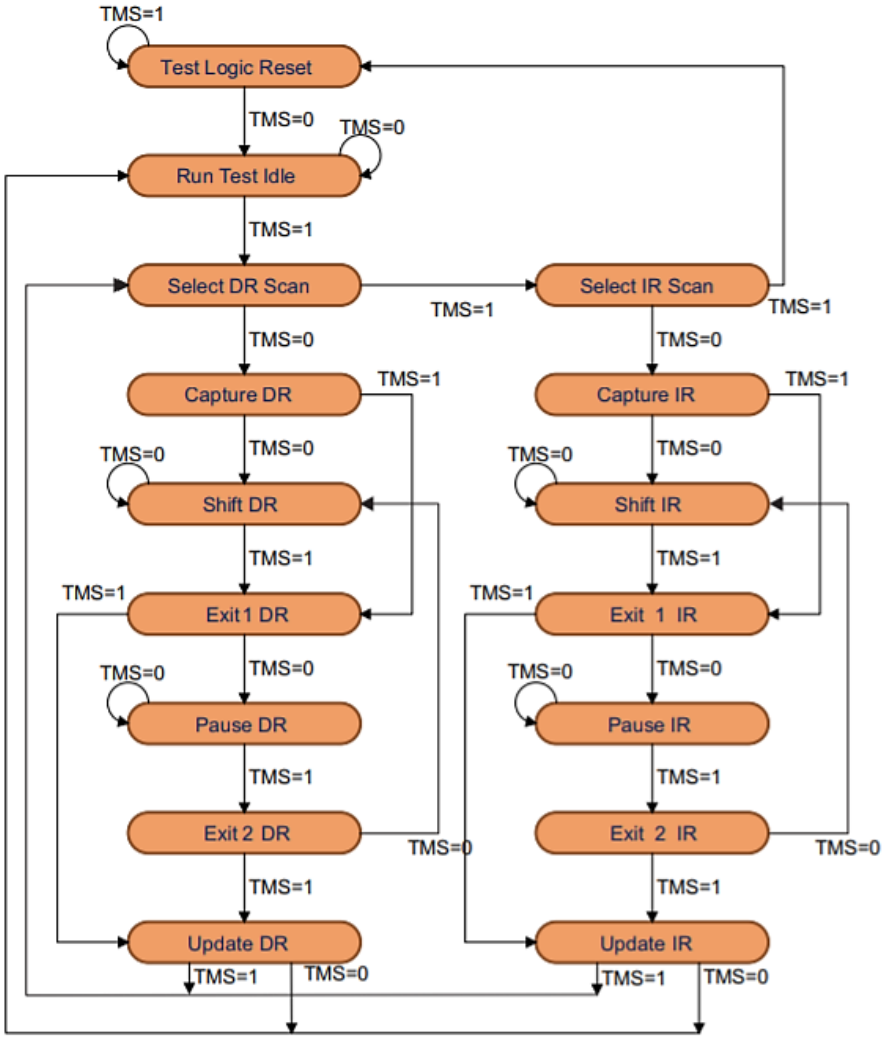


*Fig. 4.8.2 Automatul de stare al JTAG – ului [9]*

Dispozitivul CC2650 LaunchPad implementează un mecanism numit HIB (**H**alt **i**n **B**oot) pentru a asigura că emulatorul extern poate prelua controlul dispozitivului înainte ca acesta să execute orice cod. Când acest mecanism detectează activitate de debug, codul de pornire se oprește și așteaptă o instrucțiune de întrerupere (WFI) la sfârșitul execuției ei înainte de a sări înapoi la codul de executat din memoria flash.

Detecția activității pe pinul TCK care alimentează JTAG-ul, sunt condițiile de HIB la următoarea pornire. Dacă energia pe JTAG este oprita și intrarea în testul de resetare logică (TLR) se va întâmpla daca, condițiile HIB sunt șterse. Condițiile HIB nu sunt șterse dacă registrul AON_WUC : SHUTDOWN.EN este scris pe 1.

Ieșirea din HIB, se face prin emulatorul extern conectat la dispozitiv și suspendă, apoi reia execuția CPU-ului prin DAP. După ce se reia, programul continuă cu codul aplicație scris în memoria flash.

## 4.9. Power and Clock Management (PCRM)

PCRM-ul la dispozitivul CC2650 este foarte flexibil pentru a facilita consumul redus de energie pentru aplicații. Următoarea figură descrie în detaliu pentru clock și controlul puterii în adiție cu acoperirea funcțiilor de reset.

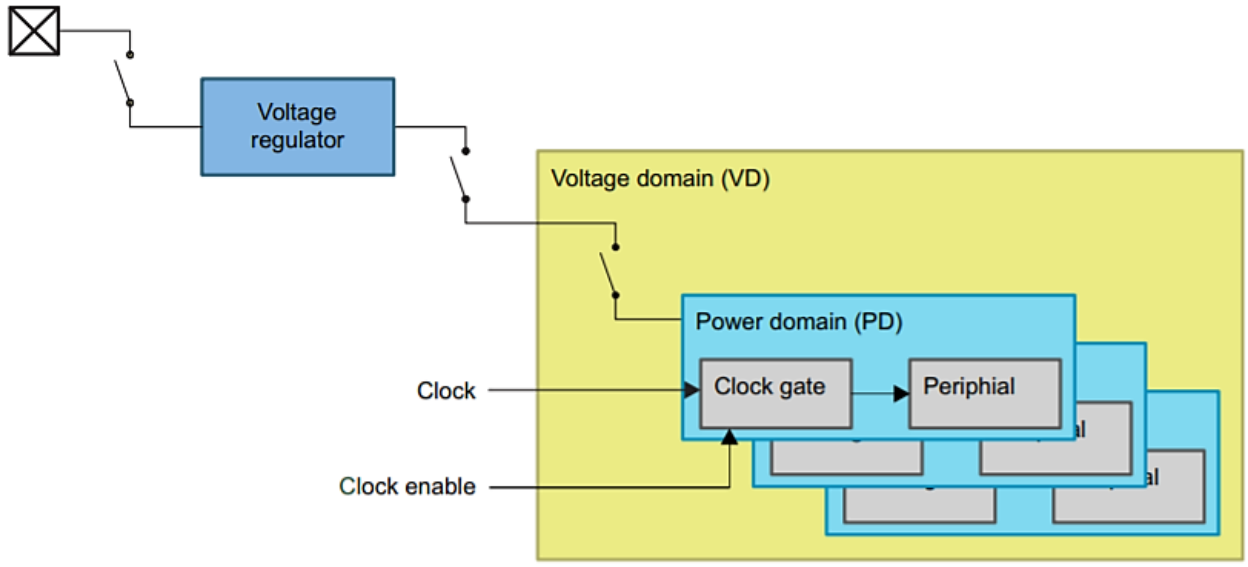


*Fig. 4.9.1 Ierarhia caracteristicilor pentru economisirea energiei [9]*

Figura 4.9.1 arată ierarhia caracteristicilor pentru economisirea energiei in dispozitivul CC2650 LaunchPad. Consumul redus de energie și timpul ciclurilor pentru modul de economisire energie este invers proporțional. Modul de economisire a energiei cu cel mai scăzut consum de energie are nevoie de la cel mai lung timp de inițializare, de asemenea, trezirea dispozitivului înapoi în modul activ.

## 4.10. Versatile Instruction Memory System

Principalele instrucții de memorii sunt încapsulate în module de sistem de memorii cu instrucțiuni versatile, care includ următoarele memorii :

- 128KBytes Flash
- 8KBytes RAM Cache sau GPRAM – **G**eneral **P**urpose **R**andom **A**ccess **M**emory
- 115KBytes Boot ROM – **R**ead **O**nly **M**emory

Modulul de sistem de memorii cu instrucțiuni versatile, trimite accesul CPU-ului și accesul la magistrala de sistem către memoriile adresate. Modulul VIMS arbitrează și accesul între CPU și magistrala de sistem.

Modulul VIMS funcționează la tactul de ceas de 48MHz.

Memoria flash este programabilă de către utilizator din software, din interfața de depanare, și din bootloader. Blocul RAM poate fi folosit ca și memorie cache pentru blocul flash, sau ca RAM de uz general.

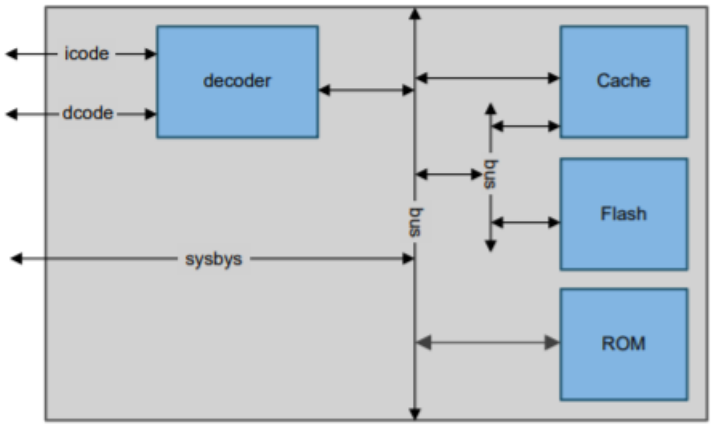


*Fig. 4.10.1 Vederea de ansamblu a modului VIMS [9]*

Configurația modulului VIMS este împărțita în 3 moduri și anume :

- GPRAM
- Cache
- Off

În cele ce urmează vom prezenta fiecare mod în parte, ilustrând figura 4.10.1.

- Modul GPRAM : Blocul RAM funcționează ca un RAM de uz general. Blocul flash nu are suport cache și toate accesele la memoria flash sunt direcționate direct către blocul flash.

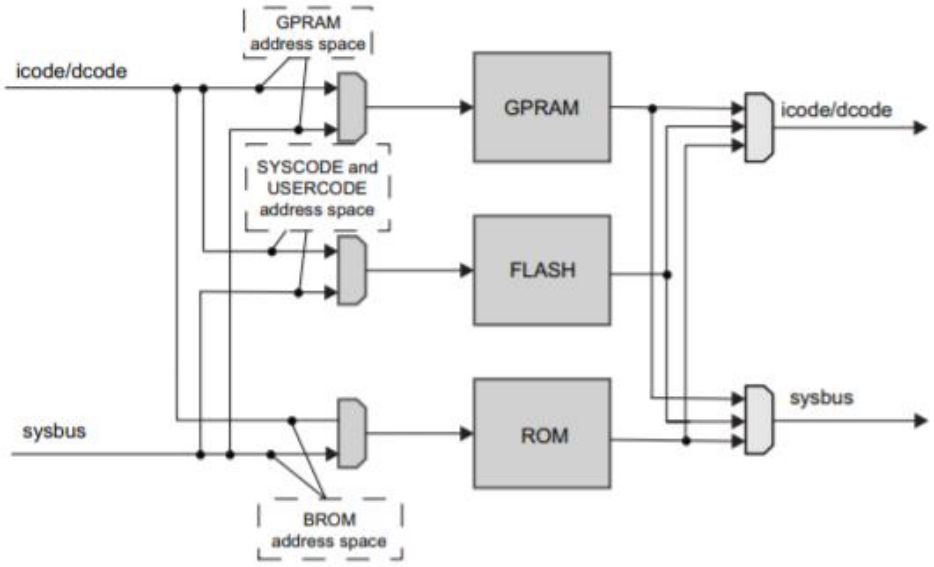


*Fig. 4.10.2 Vederea de ansamblu în modul GPRAM [9]*

- Modul Off : Blocul RAM este dezactivat și nu se poate face accesul de către CPU sau de la magistrala sistemului. Memoria flash nu are suport cache și accesul la flash este direcționat direct către acesta.

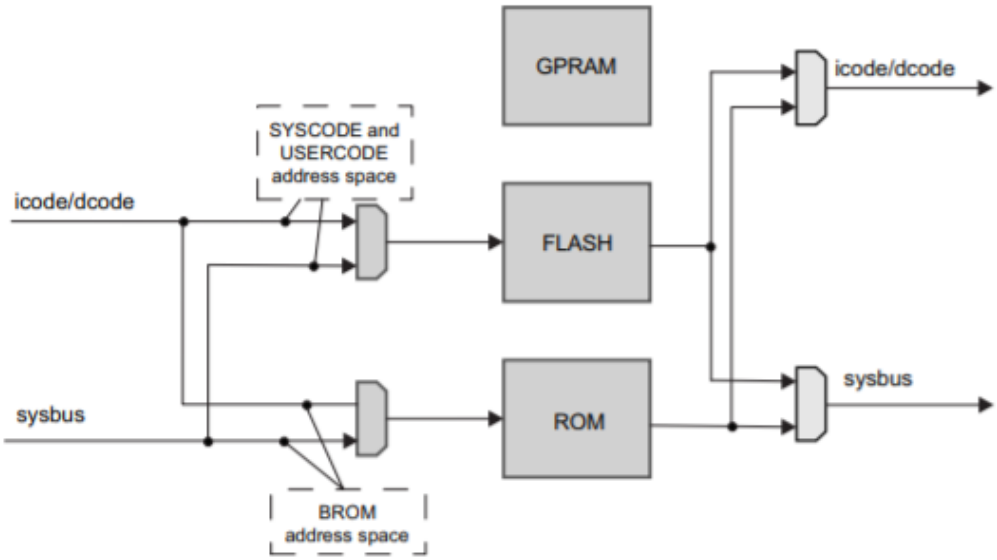


*Fig. 4.10.3 Vederea de ansamblu în modul Off [9]*

- Modul Cache : Blocul RAM funcționează un cache de înlocuire cu 4 căi, pentru blocul flash. Spațiu GPRAM nu este disponibil în acest mod. Suportul pentru cache este disponibil numai pentru accesul CPU-ului la spațiul de adrese SYSCODE. Magistrala sistem accesează blocul flash si CPU-ul accesează adresele din flash USERCODE care sunt direcționate către blocul flash.

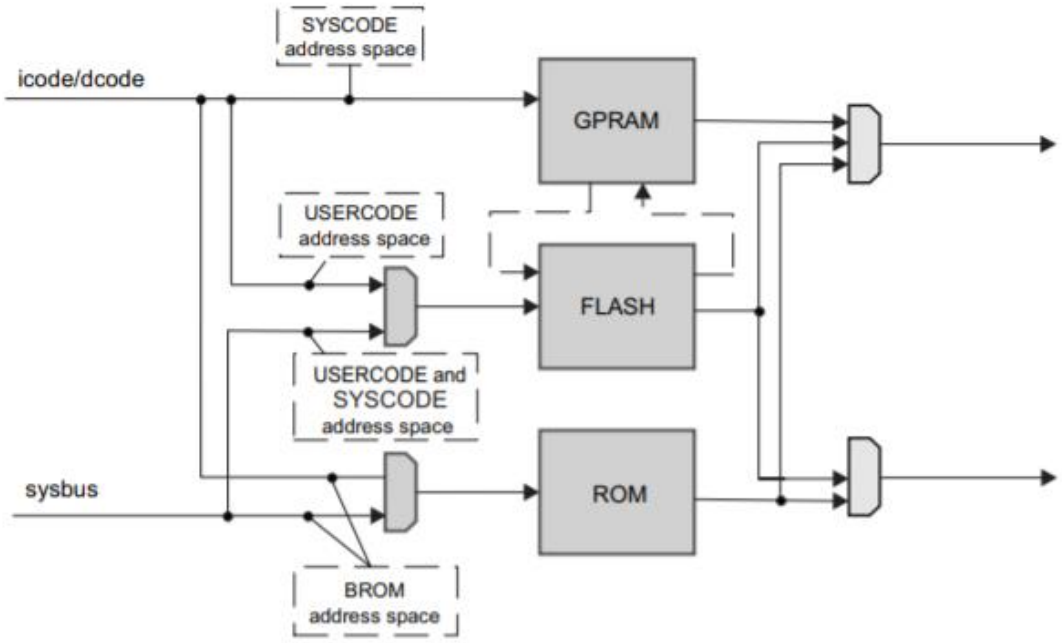


*Fig. 4.10.3 Vederea de ansamblu în modul Cache [9]*

În modul cache, conform figura 4.10.3, toate accesările de către CPU către adresele de spațiu SYSCODE sunt direcționate prima dată către cache. Cache-ul analizează adresele de intrare și eticheta interna a RAM-ului pentru a determina ce tip de acces cache este: cache miss sau cache hit.

În cazul a cache miss, accesul este trimis mai departe către blocul flash. Răspunsul de la blocul flash este direcționat înapoi către cache și apoi este actualizat cache-ul.

În cazul a cache hit, datele sunt extrase direct din memoria RAM cache.

Cache-ul conține de asemenea o memorie tampon deoarece dimensiunea cuvântului de memorie din cache-ul RAM-ului este de 64biți. Obiectivul folosirii unei memorii tampon este de a preveni re-extragerea părților de 32biți din date care au fost deja extrase dar nu și folosite în accesul precedent. Memoria tampon este curățată ca o parte a schemei individuale. Memoria tampon din modului VIMS conține 2 memorii în linie cu dimensiunea cuvântului de 64biți.

## 4.11. Memoria flash

Fiecare microcontroller are o memorie integrată pe chip pe care se stochează programul. Secțiunea principală din memoria flash este divizată în 4 regiuni și anume :

- Regiunea de bootloader : În această regiune se va stoca bootloader-ul prin se va încărca aplicația de utilizator. Această regiune ocupă spațiul de 32KBytes (32768Bytes), începând cu adresa 0x0000 până la adresa 0x8000.

- Regiunea codului de execuție : În această regiune se va stoca programul care va fi încărcat prin radio de către bootloader. Această regiune poate ocupa până la 71.91406KBytes (73640Bytes)
- Regiunea liberă : Această regiunea este spațiul liber al utilizatorul pe care îl poate folosi la voia proprie. Dacă regiunea de cod pentru a treia imagine ocupa spațiul de 71.91406KBytes, această regiune va fi 0 și nu va mai putea fi folosită de către utilizator.
- Ultima regiune este rezervată configurărilor microcontroller-ului. Această regiune este de 88bytes.

Deoarece bootloader-ul se ocupă de programarea aplicației în memoria flash, este foarte important să înțelegem organizația de memorie flash.

Pe procesoarele ARM® Cortex, versiunile 6 și 7 bazate pe sisteme integrate (SoC), spațiul liniar de adrese este de 4GB.

Memoria flash este spartă în secțiuni. Cea mai mică secțiune de flash este numita și pagină. Paginile pot fi grupate împreună pentru a forma structuri mai mari, numite și sectoare. Sectoarele se combină pentru a forma blocuri.

Fiecare microcontroller are moduri diferite în implementarea și manipularea acestor blocuri. Microprocesorul selectat pentru această lucrare permite scrierea la nivel de cuvânt, acest cuvânt este de 32biți ca și arhitectura microprocesorului și protecția la scriere la nivel de sector.

În cele mai multe cazuri, cea mai mică secțiune de flash care poate fi șters este un sector, totuși, granularitatea acestui microcontroller este la nivel de pagină.

Memoria flash este un mediu electronic de stocare non – volatil pe chip, care poate fi programată electronic. Dispozitivele embedded, cum ar fi cele utilizate în timpul acestei lucrări, au memorie Flash On – Chip, care stochează bootloader-ul și firmware-ul aplicației.

Stocarea datelor în memoria flash este destul de diferită de memoria RAM statică (SRAM) și are nuanțe proprii. Deoarece memoria flash este o memorie nevolatilă, aceasta reține datele chiar și după ce sursa de alimentare a fost scoasă. De obicei, este necesară o operație de ștergere înainte ca scrierea să poată fi perforată. Pe baza tipului de memorie flash, blocul de ștergere variază. De exemplu, în memoria flash de tip NAND, datele pot fi scrise în blocuri, în timp ce memoria flash de tip NOR, datele mărimii unui cuvânt mașină pot fi scrise într-o locație ștearsă sau pot fi citite ulterior.

Înainte de actualizarea memoriei flash, memoria cache VIMS și buffer-elor care trebuie invalidate și golite pentru a preveni datele vechi sau instrucțiunile sa fie preluate din acestea după ce se face actualizarea memoriei flash. Prin urmare, modul VIMS trebuie setat la modul GPRAM – **G**eneral **P**urpose **R**andom **A**ccess **M**emory, sau oprit înainte de programare, iar ambele tampoane trebuie dezactivate.

Memoria flash este organizată în pagini a câte 4KBytes fiecare, care pot fi șterse individual. Un cuvânt de 32 – biți poate fi programat individual, schimbând biții din 1 în 0. În plus, folosirea unei memorii tampon (buffer) asigură abilitatea de programare a 32 de cuvinte continue în memoria flash în jumătate din timpul programării individuale.

Ștergerea unui bloc determină resetarea tuturor biților din bloc la valoarea 1. Sectoarele de 8KBytes pot fi protejate individual. Protecția permite ca sectoarele să fie marcate ca fiind numai

pentru citire sau execuție, oferind astfel niveluri diferite de protecție a codului. Sectoarele doar pentru citire nu pot fi șterse sau programate, ceea ce protejează modificarea conținutului acestor blocuri.

Executarea sectoarelor care pot fi doar executate nu pot fi șterse sau programate, și pot fi doar citire de către mecanismul de preluare a instrucțiunilor al controllerului. Memoria flash lucrează în principal la tactul de ceas al sistemului de 48MHz.

În timpul programării memoriei flash, scriere sau ștergere, aceasta nu trebuie să fie citită. Dacă instrucțiunea executată are nevoie de folosirea memoriei flash, aceasta trebuie pusă în SRAM și executată din SRAM cât timp operațiile cu memoria flash sunt în desfășurare.

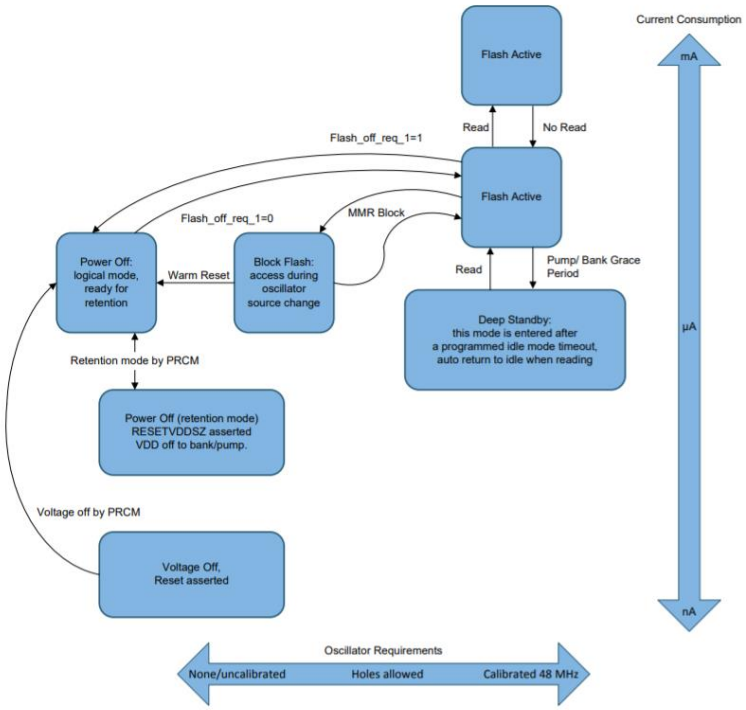


*Fig. 4.11.1 Managementul energetic al memoriei flash [9]*

Cerințele de management a energiei sunt detaliate în cele ce urmează, conform fig. 4.11.1. :

- Tensiunea oprită : Modulul logic $V_{DD}$ este oprit. Bancurile sunt ținute în modul deep sleep. Acest mod necesită o resetare și o configurare software pentru a deveni activă.
- Oprire : Acest mod este asemănător cu modul de tensiune oprită. Singura diferență este că modul reține toți regiștri. Din acest mod, modulul poate deveni activ fără nicio configurare software.
- Deep standby : Circuitele interne sunt parțial oprite. Nu este nevoie de nicio configurare pentru a deveni active, dar există o întârziere din cauza creșterii tensiunii la cea nominală.
- Inactiv : Folosește caracteristici avansate de reducerea energiei în bancuri pentru a salva energia când nicio citire nu este în desfășurare. În acest mod, schimbarea între citirea activă și inactivă este făcută fără nicio întârziere.
- Citire : Flash-ul este constant activ fără nicio reducere de energie.

Metode de schimbarea modurilor :

- Setarea între modul de tensiune oprită sau oprire poate fi făcuta din sistemul de management al energiei.
- Setarea modului Deep standy poate fi activat cu :
    - PRCM – **P**ower **R**eset and **C**lock **M**anagement.
    - Prin scrierea unui registru în MMR.
    - Prin începerea secvenței de citire din memoria flash.
- Schimbarea între modul inactiv și cel de citire este făcut automat când citirea s-a terminat. Schimbarea între aceste 2 moduri poate fi dezactivată prin setarea unui registru intern.
- Schimbarea între modul inactiv și oricare alt mod este făcută prin setarea unui registru cu o valoare asociată unui mod. După un anumit timp fără citire, modul selectat intră în funcțiune. Ultimul pas este atins de către sistemul de management al chipului, si acest mod poate fi tensiunea oprită sau oprire.

## 4.12. SRAM

Placa de dezvoltare CC2650 LaunchPad este prevăzută cu o memorie SRAM de un singur ciclu de 20KBytes cu reținere totală în orice mod de operare, exceptând oprirea. Oricum, retenția poate fi configurată în pagini de 4KBytes, acest lucru nu reduce semnificativ consumul de energie. Este recomandat sa se rețină întregul SRAM în tot timpul.

Deoarece modificările de citire și modificările de citire sunt operații foarte costisitoare de timp, ARM a introdus tehnologia bit – banding în procesoarele Cortex – M3. Cu această tehnologie, fac ca procesoarele pentru anumite regiuni din harta de memorie SRAM și spațiul periferic să poată folosi alias-uri de adresă pentru a accesa biți individuali într-o operație atomică.

Datele pot fi transferate, de asemenea, către și din SRAM folosind controller-ul **m**icro **d**irect **m**emory **a**ccess (µDMA). Procesorul responsabil pentru radio Cortex – M0 are de asemenea acces la memoria RAM a sistemului.

## 4.13. µDMA – Micro Direct Memory Access

Procesorul plăcii CC2650 LaunchPad include un controller numit și µDMA. Controller-ul dispune de o metodă de transmitere a blocurilor de date de la periferice către blocurile de memorie și invers, eliberând procesorul de această sarcina, astfel concentrându-se pe sarcini mai importante.

Acest controller are canale dedicate pentru fiecare modul suportat de pe chip-ul CC2650 LaunchPad, care poate fi programat să efectueze automat aceste transferuri între periferice si blocurile de memorie, perifericul fiind gata să transfere mai multe date.

Controllerul µDMA este un controller DMA flexibil și foarte configurabil, proiectat să funcționeze eficient cu nucleul procesorului Cortex – M3 din microcontroller. Controllerul accepta mai multe dimensiuni pentru date și scheme de incrementare a adreselor, multiple niveluri de

prioritate în rândul canalelor DMA și mai multe moduri de transfer pentru a permite transferuri sofisticate de date.

Fiecare funcție periferică are un canal dedicat pe controllerul µDMA care poate fi configurat independent de celelalte. Controllerul µDMA implementează metode de configurare folosind structuri de control al canalelor ținute în memoria sistemului. În timp ce sunt acceptate moduri simpliste de transfer, există posibilitatea creării unor moduri sofisticate de sarcini în memorie care să permită controlul µDMA să efectueze transferuri de dimensiuni arbitrare către și din locații arbitrare, ca parte a unei singure cereri de transfer. De asemenea, controllerul µDMA suportă utilizarea unui buffer ping – pong pentru a permite streaming-ul constant de date către un periferic sau de la un periferic.

Fiecare canal are o dimensiune configurabilă. Această mărime configurabilă este numărul de elemente transferate în rafală înainte ca controllerul µDMA să solicite prioritatea canalului. Utilizând această dimensiune configurabilă, este posibil să fie controlat exact câte elemente sunt transferate către sau de la un periferic de fiecare dată când se face o cerere de serviciu µDMA.

## 4.14. UART – Universal Asynchronous Receiver Transmitter

Electronica încorporată este despre interconectarea circuitelor, a procesoarelor și a altor circuite pentru a crea un sistem simbiotic. Așadar pentru ca aceste circuite să poată schimba informații între ele, ele trebuie să împărtășească un protocol de comunicație comun. Interfața serială dă drumul la fluxul de date cu un singur bit la un moment dat de timp.

Reguli pentru comunicația serială : Protocolul serial are numeroase reguli încorporate în mecanisme care ajută la siguranță, robustețe și prevenția erorilor în transferul de date. Aceste mecanisme pe care le primim în schimbul unui semnal de ceas extern sunt :

- Biți de date.
- Biți de sincronizare.
- Biți de paritate.
- Rata de biți exprimată de obicei în biți/secundă.

Partea cea mai critică este asigurarea că ambele dispozitive funcționează pe aceeași configurație de protocol pe magistrală serial.

Controllerul de pe placa de dezvoltare CC2650 include un modul de UART care dispune de următoarele caracteristici :

- Comunicație full duplex.
- Generator programabil pentru rata de biți, cu viteze până la 3Mbps.
- Buffere de tip FIFO separate de transmitere 32x8 și de recepție 32x12 pentru a reduce încărcarea serviciului de întreruperi.
- Dimensiunea programabilă a buffer-ului FIFO.
- Comunicare standard asincronă pentru biții de start, stop și paritate.
- Generarea caracterului de linie nouă și detecția acestuia.
- Caracteristici total programabile ale interfeței seriale :

    - 5, 6, 7 sau 8 biți de date.
    - Detecție de paritate și generarea ei.
    - Unul sau doi biți de stop.
- Suport pentru funcțiile moderne de tip CTS și RTS.
- Transferuri eficiente de date folosind controllerul de acces direct la memorie (µDMA).
    - Canal separat pentru fiecare trimitere și recepție.
- Controlul de flux hardware programabil.

Pentru a interfața 2 dispozitive utilizând modulul UART, este necesar conectarea a 3 fire. Aceste 3 fire sunt :

- RX → TX
- TX ← RX
- GND ↔ GND

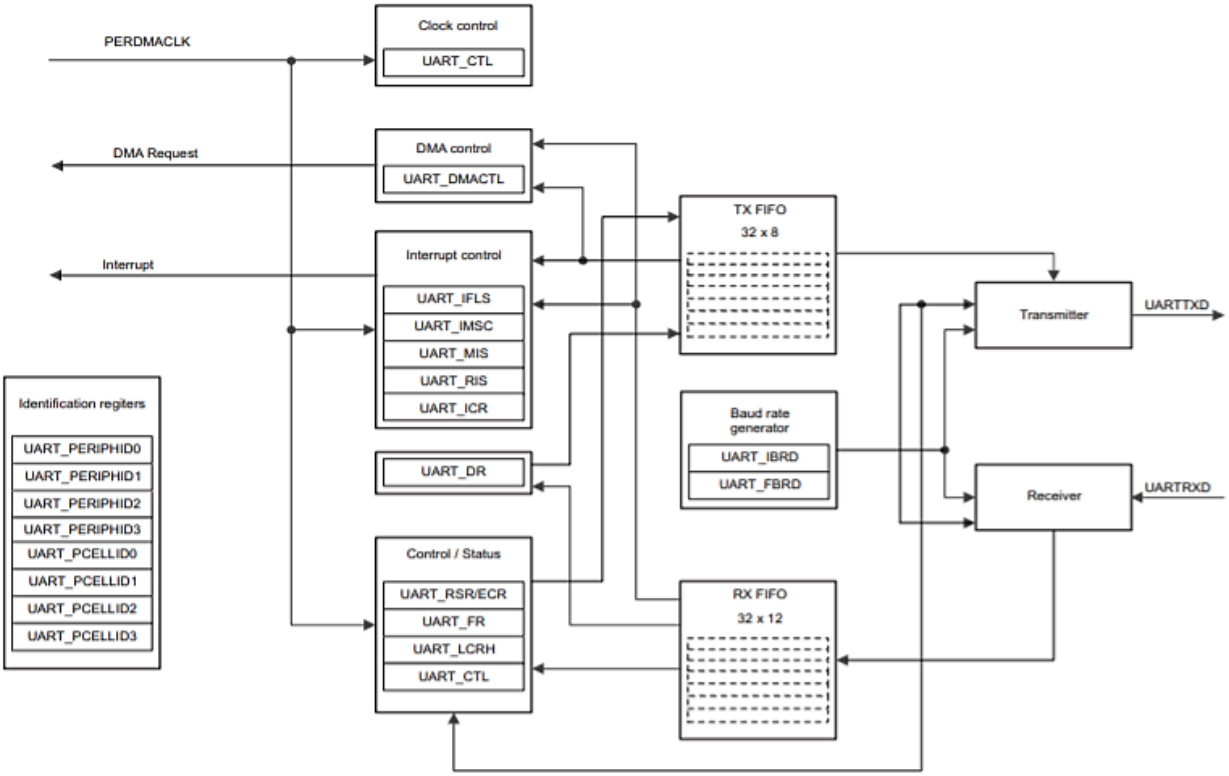


*Fig. 4.14.1 Diagrama bloc a modulului UART [9]*

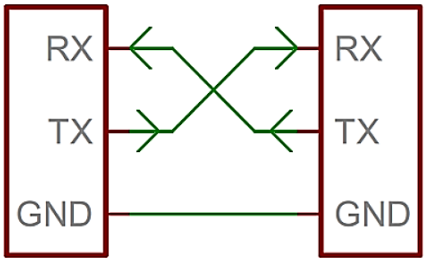


*Fig. 4.14.2 Interfațarea a două dispozitive prin UART [10]*

Pentru această lucrare s-a ales următoarea configurație :

- Lungimea datelor de 8biți.
- Un singur bit de stop.
- Viteza de comunicație (baud – rate) de 9600 biți/secundă.

Comunicația serială prin interfața UART este folosită pentru a comunica între nucleul RF și procesorul principal, și de asemenea pentru a comunica cu calculatorul.

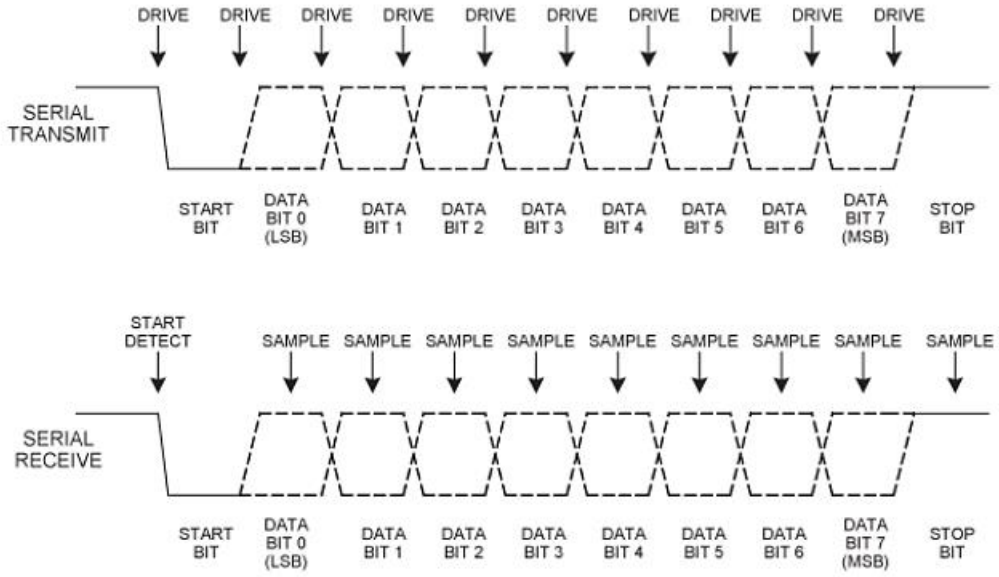


*Fig. 4.14.3 Secvența de trimitere (sus) și cea de recepție (jos) pentru UART [11]*

## 4.15. Radio

Nucleul de radio frecvență conține un procesor ARM Cortex – M0 care se interfațează cu circuitele de radio frecvență analogice și banda de bază, gestionează datele către și de la partea sistemului și asamblează biții într-o structură de pachete dată. Nucleul RF oferă un API (**A**pplication **P**rogram **I**nterface) la procesorul sistemului Cortex – M3. Nucleul RF se poate ocupa în mod automat de aspectele critice de timp ale protocoalelor 802.15.4 RF4CE, ZigBee și Bluetooth Low Energy.

Nucleul RF are un bloc de memorie SRAM dedicată de 4KBytes și funcționează aproape în totalitate, separat de ROM.

## 4.16. Descrierea modului radio și vederea de ansamblu mutare

Core-ul RF poate primi cereri de nivel înalt de la procesorul sistemului CPU și efectuează toate tranzacțiile necesare pentru a le îndeplini. Aceste solicitări sunt orientate în primul rând către transmiterea și recepția informațiilor prin canalul radio, dar pot include și sarcini suplimentare de întreținere, cum ar fi caracteristicile de testate, de depanare sau de calibrare.

Ca și un framework general, tranzacțiile dintre CPU-ul sistemului și core-ul RF funcționează în felul următor:

- Core-ul RF poate să acceseze date și parametrii de configurare din RAM-ul sistemului. Acest acces scade amprenta memoriei core-ului RF, evitând traficul inutil între diferite părți ale sistemului și reduce consumul global de energie.
- În mod similar, core-ul RF poate decoda și scrie înapoi conținutul pachetului recepționat, împreună cu informațiile despre stare, în RAM-ul sistemului.
- Pentru scopurile protocolului de confidențialitate și de autentificare, core-ul RF poate accesa și subsistemul de securitate.
- Core-ul RF recunoaște comenzi complexe din partea procesorului central Cortex – M3 (ca de exemplu CCA, RX, cu recunoaștere automată, etc.) și le subdivizează în comenzi fără intervenția suplimentară a procesorului central.

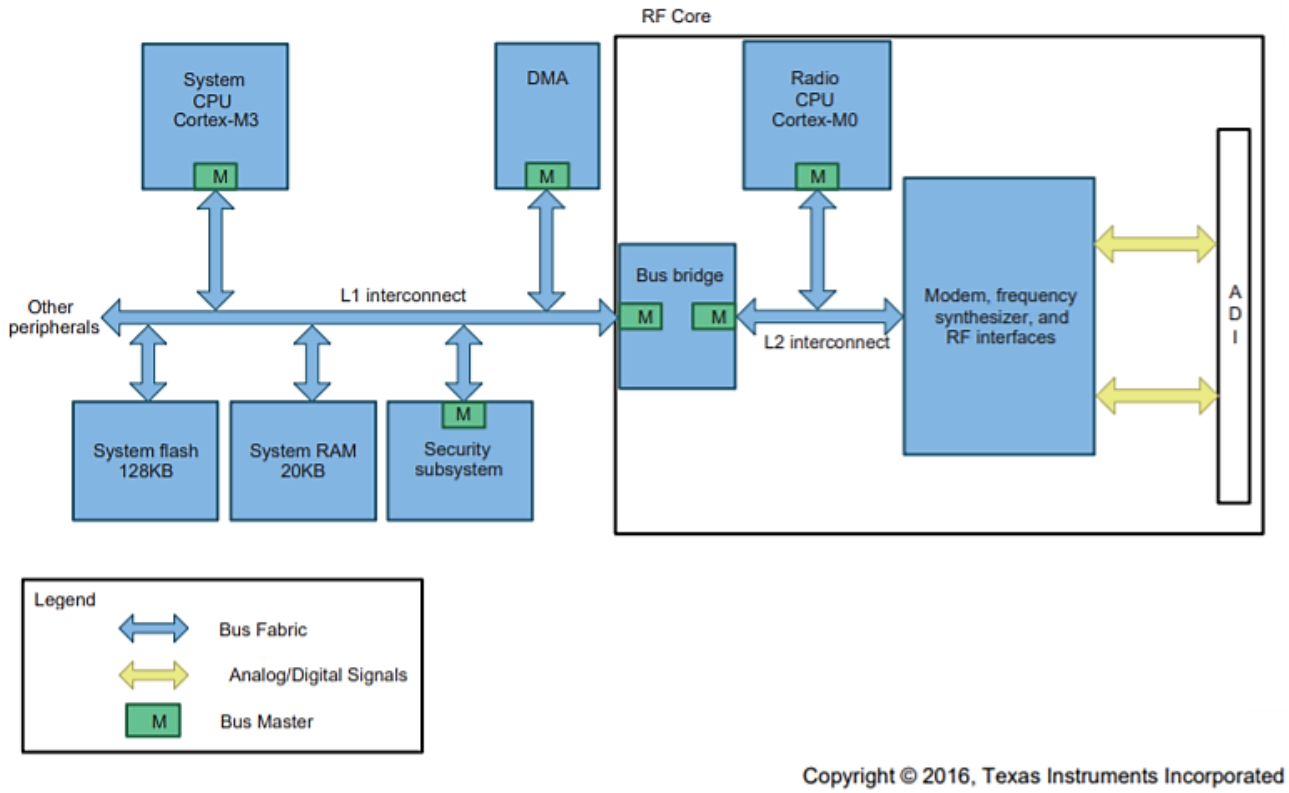


*Fig. 4.16.1 Structura core-ului RF cu dependențe externe [9]*

În figura 4.16.1 avem următoarele funcții :

- System Side :
    - Procesorul sistemului : Procesorul principal care rulează aplicațiile, împreună cu stiva de protocoale de nivel înalt și eventual câteva caracteristici MAC pentru protocoale. CPU-ul execută cod din ROM și memoria flash.
    - RAM-ul sistemului : Conține informațiile pachetelor de trimite (payload) și diferiți parametrii sau configurări pentru tranzacțiile date.
    - Subsistemul de securitate : Cuprinde diferite elemente pentru a furniza confidențialitatea și autentificarea protocolului.
    - DMA : Într-un mod opțional, DMA-ul este însărcinat pentru a transfera informații de la RAM-ul modulului radio la RAM-ul sistemului și vice versa. Dacă nu este utilizat accesul direct către CPU.
- Radio Side :
    - Radio CPU : Procesorul principal este responsabil cu RF. Acceptă comenzi de nivel înalt de la procesorul sistemului și le planifică în diferite părți ale core-ului RF.
    - Modem, sintetizator de frecvențe, interfețe RF : Acesta este nucleul radioului, transformând biții în semnale modulate și invers.

## 4.17. Radio doorbell

Modulul radio doorbell este cel care realizează comunicația între procesorul sistemului și procesorul radio, cunoscut de asemenea ca și CPE (**C**ommand and **P**acket **E**ngine). Doorbell-ul radio conține un set dedicat de regiștri, parametri în oricare dintre memoriile RAM ale dispozitivului și un set de întreruperi pentru amândouă procesoare.

În plus, parametrii și mesajul util sunt transferați prin intermediul memoriei RAM sau a memoriei radio. Dacă există parametrii sau payload-uri în memoria RAM a sistemului, sistemul trebuie să rămână alimentat, atât timp totul este în RAM-ul radio, apoi procesorul sistemului poate să intre în modul de economisire a energiei.

În timpul funcționării, procesorul radio actualizează parametrii și payload-ul în RAM și generează întreruperi. CPU-ul sistemului poate masca întreruperile, astfel încât să rămână în modul inactiv sau intră în modul sleep până la terminarea întregii operații radio.

Deoarece CPU-ul sistemului și CPU-ul radio au o zonă de RAM partajată, trebuie asigurat că nu pot apărea condiții de concurență. Acest lucru poate fi realizat în software prin reguli bine stabilite în stratul de abstractizare hardware (HAL – **H**ardware **A**bstraction **L**ayer).

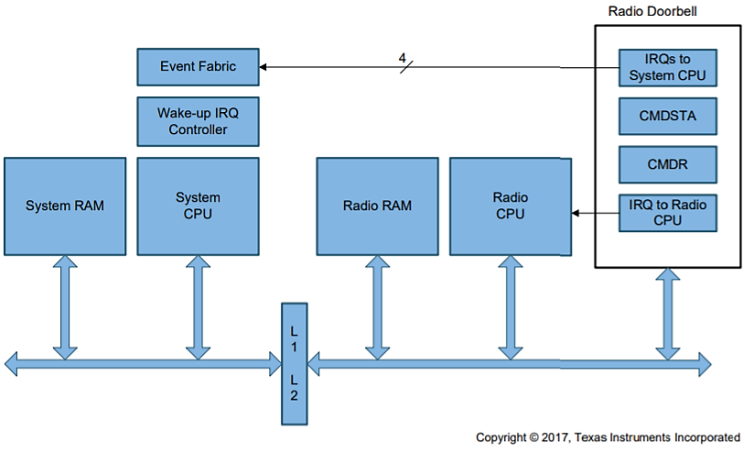


*Fig. 4.17.1 Suport hardware pentru HAL [9]*

Core-ul RF ascunde complexitatea operațiunilor radio prin furnizarea unui HAL unificat procesorului sistemului.

### 4.18. Comenzile și regiștrii de stare și evenimente

Comenzile sunt trimise la radio prin intermediul regiștrilor CMDR, în timp ce registrul CMDSTA este în modul read – only care asigură feedback-ul stării radioului. Registrul CMDR poate fi scris numai în timp ce se citește 0. În caz contrar, scrierea este ignorată. Atunci când registrul CMDR este 0 și este scrisă o valoare diferită de 0, radio-ul CPU este notificat și registrul CMDSTA devine 0. După aceasta, valoarea scrisă poate fi citită din registrul CMDR până când procesorul radio procesează comanda, la care registrul indică înapoi 0.

Când comanda respectivă a fost procesată de către procesorul radio, registrul CMDSTA conține o stare non – zero, care este furnizată atunci când registrul CMDR revine la 0. În același timp, apare o întrerupere RFCMDACK. Acest tip de întrerupere este, de asemenea, mapată în registrul RFACKIFG, care trebuie sa fie șters atunci când întreruperea a fost procesată.

# 5. Sistem de operare în timp real – TI RTOS

Un sistem de operare în timp real [12] este un sistem de operare intenționat să servească la aplicații de timp real care procesează datele de intrare, de obicei fără nicio întârziere. Cerințele de timp de procesare care includ orice întârziere a sistemelor de operare, sunt măsurate în 10-a parte din secundă sau chiar incrementări mai scurte de timp. Un sistem de timp real este strâns legat de timp, având constrângeri de timp fixe și bine definite. Procesarea trebuie efectuată în limitele definite de sistem sau sistemul va eșua să îndeplinească condiția de sistem de operare în timp real. Sistemele de operare în timp real operează prin evenimente sau împărtășind timpul procesor. Sistemele bazate pe evenimente schimbă sarcini pe bază de prioritatea acestora, în timp ce sistemele de împărtășire de timp schimbă task-urile pe baza unei întreruperi dată de ceas.

Caracteristica cheie a unui sistem de timp real este gradul de consecvență în ceea ce privește durata de timp necesară pentru acceptarea și finalizarea unui task al aplicației.

Un sistem de operare în timp real implementat în hardware, are un jitter mai mic decât sistemele de operare de timp real implementate software. Acestea sunt mai permisive decât sistemele de timp real implementate hardware. Un sistem de operare în timp real care poate îndeplini un termen limită este un sistem în timp real software, dacă acesta atinge un termen limită determinist, acesta este un sistem de operare în timp real implementat hardware.

Un sistem de operare în timp real are un algoritm avansat de a planifica execuția task-urilor. Flexibilitatea planificatorului permite o orchestrare mai largă a priorităților din sistem. Dar un sistem de operare în timp real este dedicat unui set restrâns de aplicații.

Doi factori cheie într-un sistem de operare în timp real sunt întârzierile minime a întreruperilor și întârzierile minime de comutare între task-uri. Un sistem de operare în timp real este evaluat mai mult pentru cât de repede sau previzibil poate să răspundă decât pentru cantitatea de efort pe care o poate depune într-o anumită cuantă de timp.

Cele mai comune design-uri pentru sistemele de operare în timp real [13] sunt :

- Conduse de evenimente (event – driven) : Schimbă task-urile doar când un eveniment de prioritate mai mare are loc. Acest tip de sistem de operare se numește pre – emptiv, sau planificare pe priorități.
- Împărtășirea timpului de execuție : Acest tip de design schimbă task-urile regulat, pe baza unei întreruperi de ceas, sau pe baza unui eveniment. Acest tip de sistem de operare se numește round robin.

Design-ul de împărtășirea timpului schimbă task-urile mult mai des decât ar fi necesar, ceea ce face un multi-tasking mai neted, dând iluzia că un procesor execută mai multe aplicații în paralel.

Design-urile pentru planificator au de obicei 3 stări, și anume :

- În desfășurare : se execută pe procesor.
- Pregătită : task-ul este gata să intre pe procesor și să fie executat.
- Blocat : în așteptarea unui eveniment. De exemplu un eveniment I/O.

Majoritatea task-urilor sunt în stare blocată sau pregătite să fie executate. În general, un singur task poate rula pe procesor la un moment dat. Numărul de task-uri din coada de așteptare poate varia foarte mult, în funcție de numărul de task-uri de care are nevoie sistemul de operare să îndeplinească și tipul de planificator ales. Pe un sistem de operare în timp real, având un planificator pre – emptiv mai simplu, dar tot odată multi-tasking, trebuie să renunțe la timpul acordat pe procesor alor task-uri, ceea ce poate face ca coada de așteptare să aibă un număr mai mare de task-uri care sunt gata să fie executate.

Trebuie să avem în evidență să nu inhibăm pre – empțiunea în timpul căutării. Secțiunile critice mai mari ar trebui să fie împărțite în bucăți mai mici. Dacă o întrerupere se produce care face un task de prioritate mai mare să fie gata în adăugarea unui task de prioritate mai mică, task-ul cu prioritate mai mare poate fi adăugat mai rapid și să fie executat imediat înaintea task-ului de prioritate mai mică.

Timpul critic de răspuns, numit și timpul flyback, este timpul pentru a ceda un task gata și restaurarea task-ului cu prioritatea cea mai mare. Într-un sistem de operare în timp real bine conceput, pregătirea unui nou task va dura de la trei până la douăzeci de instrucțiuni pentru fiecare intrare în coada de așteptare, iar restaurarea task-ului cu cea mai mare prioritate poate dura de la cinci până la treizeci de instrucțiuni.

În sistemele de operare mai avansate, task-urile de timp real împart resursele cu multe alte task-uri care nu sunt de timp real, și lista poate fi arbitrar de lungă pentru task-uri care sunt gata. În astfel de sisteme, o listă a planificatorului de acest tip implementată prin liste înlănțuite este inadecvată.

Algoritmi folosiți în sistemele de operare în timp real :

- Planificator cooperativ.
- Planificator pre – emptiv.
- Planificator round robin.
- Planificator de rată monotonă.
- Planificator pre – emptiv cu priorități fixe, implementat cu un feliator de timp (Time Slicing).
- Cel mai apropiat termen limită.
- Grafice stocastice cu traversare multi – thread.

Comunicarea inter task și împărtășirea resurselor disponibile se face prin :

- Dezactivarea sau mascarea temporară a întreruperilor.
- Semafoare : pot fi binare sau incrementale. Acest tip blochează resursele sau le deblochează în funcție de ce este nevoie. Task-urile celelalte trebuie să aștepte pentru a debloca resursele să se poată executa. Un semafor binar este echivalentul unui mutex – Mutual Exclusive.
- Transmiterea mesajelor printr-o schemă.

Alocarea memoriei este mult mai critică într-un sistem de operare în timp real decât în alte sisteme de operare obișnuite. În primul rând pentru stabilitatea sistemului, nu pot fi scurgeri de memorie, adică memorie care se alocă și este neutilizată, dar niciodată eliberată. Dispozitivul ar trebui să funcționeze pe o perioadă nelimitată, fără a mai avea nevoie de repornire. Din acest punct de vedere, alocarea dinamică a memoriei este fragmentată, ceea ce poate crește timpul de acces la ea. Când este posibil, toate alocările necesare de memorie sunt specificate static la momentul compilării programului.

Un alt motiv important pentru a se evita alocarea dinamică a memoriei, este fragmentarea memoriei. Cu alocarea și eliberarea frecventă a bucăților mici de memorie, poate să apară o situație atunci când memoria este segmentată în mai multe regiuni, în acest caz în care sistemul de operare în timp real nu poate să aloce un bloc mare de memorie continuă.

În al doilea rând, viteza de alocare este și ea extrem de importantă. O schemă standard de alocare a memoriei va scana o listă înlănțuită de dimensiune necunoscută pentru a găsi un bloc de memorie adecvat, care este inacceptabil într-un sistem de operare de timp real, deoarece alocarea memoriei trebuie să aibă loc într-o anumită cuantă de timp.

Deoarece discurile mecanice au un timp mult mai mare de răspuns și imprevizibil, schimbarea de fișiere pe disc nu este folosită din aceleași motive ca alocarea RAM.

Algoritmul simplu de blocuri de mărime fixă, funcționează destul de bine pentru sistemele embedded datorită amprentei mici.

Sistemul de operare în timp real de la producătorul Texas Instruments [14] este un ecosistem de instrumente embedded creat pentru utilizarea într-o gamă largă de procesoare embedded. Acest sistem include o componentă de sistem de operare în timp real numită "Kernel TI – RTOS" cunoscut anterior ca și "SYS / BIOS" care a evoluat din vechiul "DSP / BIOS" [15], împreună cu componentele suplimentare care suportă drivere de dispozitiv, sisteme de fișiere, instrumente și comunicații inter – procesor cum ari fi DSP / BIOS Link.

TI – RTOS permite o dezvoltare rapidă, eliminând necesitatea ca dezvoltatorii să scrie și să mențină software-ul, cum ar fi planificatorii, stive de protocoale și drivere. Acesta combină un kernel multi – tasking în timp real cu componente middleware suplimentare, inclusiv stive de USB și TCP / IP, un sistem de fișiere FAT și drivere de dispozitive, permițând dezvoltatorilor să se concentreze pe diferențierea aplicațiilor lor. TI – RTOS oferă o platformă software integrată consecventă pentru dispozitivele de microcontrolere de la TI, facilitând astfel transferarea aplicațiilor moștenite la cele mai mici dispozitive.

TI – RTOS oferă o gamă largă de servicii de sistem unei aplicații embedded, cum ar fi multi-tasking-ul pre – emptiv, analiza în timp real, și gestionarea memoriei. Deoarece TI – RTOS poate fi utilizat într-o largă varietate cu constrângeri de procesare și constrângeri de memorie. Acest sistem de operare de timp real fost proiectat să fie foarte configurabil.

Sistemul de operare în timp real TI 1.00 a fost lansat inițial în iulie 2012 pentru microprocesoarele TI. Varianta 2.00 a Ti a fost finalizat în aprilie 2014 procesul de redenumire și a integrat kernel-ul TI – RTOS și alte componente sub umbrela software.

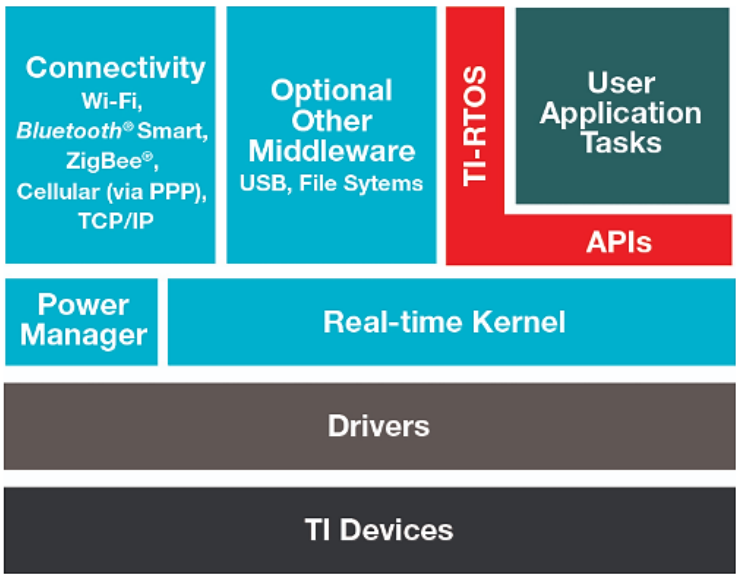


*Fig. 5.1 Sistemul de operare în timp real – Texas Instruments [14}*

TI – RTOS Kernel oferă suport pentru multe tipuri diferite de fire de execuție într-un sistem embedded, și anume :

- Întreruperi hardware (HWI) : firele de execuție de suport inițiate sunt întrerupere hardware.
- Întreruperi software (SWI) : aceste întreruperi sunt structurate pentru a fi similare cu întreruperile hardware, dar permit ca procesarea să fie amânată până după terminarea unei întreruperi hardware.
- Task : un fir de execuție discret care poate executa sau bloca în timp ce așteaptă pentru a apărea un eveniment.
- Inactiv : firul de execuție cu prioritatea cea mai mică care rulează doar atunci când niciun alt fir de execuție nu este gata sa fie executat.

Kernel – ul TI – RTOS este prevăzut cu unelte pentru a configura hărți de memorie a sistemului embedded și de asemenea pentru a permite buffer – elor să fie alocate și dealocate în timp ce sistemul rulează. Managerul de memorie folosit în timpul execuției este configurabil, astfel încât fragmentarea memoriei poate fi minimizată, dacă este necesar acest lucru.

De asemenea Kernel – ul TI – RTOS oferă module care îi permit să furnizeze informații despre modul de execuție al sistemului. În plus, mediul de dezvoltare integrat Code Composer Studio [19] poate prelua aceste date înregistrate și le poate afișa grafic pentru dezvoltator.

# 6. Programarea modulului RF

## 6.1. Întreruperile core-ului RF

Cele 4 linii de întreruperi ale core-ului RF către procesorul central Cortex – M3 sunt enunțate în cele ce urmează :

- RF_CPE0 – Întreruperea cu numărul 9
- RF_CPE1 – Întreruperea cu numărul 2
- RF_HW – Întreruperea cu numărul 10
- RF_CMD_ACK – Întreruperea cu numărul 11

Procesorul radio dispune de 32 de surse de întreruperi software care generează întreruperile RFCPE0 și RFCPE1 în procesorul central al dispozitivului. Un registru de semnalare a întreruperii poate indica când o întrerupere software s-a produs, iar întreruperile pot fi de asemenea activate individual. În plus, întreruperea RFCMDACK este ridicată automat atunci când registrul CMDSTA este actualizat.

Unele întreruperi definite în software au o semnificație comună în toate comenzile. Detaliile fiecărei întreruperi sunt definite pentru fiecare protocol care utilizează o anumită întrerupere. Unele întreruperi sunt utilizate într-un singur protocol, în timp ce altele sunt utilizate în multe protocoale. Întreruperile sunt descrierea registrului RFCPEIFG.

## 6.2. Comenzile RF și Pachetul de întreruperi

Cele două întreruperi la nivelul de sistem RF_CPE0 și RF_CPE1 pot fi produse dintr-un număr de întreruperi de nivel scăzut produse de către CPE. Fiecare din aceste întreruperi pot fi mapate către RF_CPE0 sau RF_CPE1 folosind registrul RFCPEISL. În plus, generarea întreruperilor la nivelul sistemului pot fi activate sau dezactivate utilizând registrul RFCPEIEN.

Pentru cazul unui eveniment care va declanșa o întrerupere, bitul corespunzător din registrul RFCPEIFG este setat pe 1. Ori de câte ori un bit în RFCPEIFG și bitul corespunzător RFCPEIEN sunt amândoi 1, întreruperea la nivelul sistemului este declanșată. Acest lucru corespunde la rutina de tratare a întreruperii care trebuie să șteargă biții din RFCPEIFG care corespund cu întreruperile care au fost procesate.

Ștergerea biților în RFCPEIFG se face prin scrierea 0 la acei biți, în timp ce biți de 1 rămân neschimbați.

## 6.3. Timer-ul radio

Radio-ul are propriul timer dedicat, modulul RAT (**Ra**dio **T**imer). RAT-ul este un timer de 32 biți de care rulează independent la 4MHz. RAT-ul are 8 canale cu funcționalitate de comparare și captare. Cinci din aceste canale sunt rezervate pentru procesorul radio, în timp ce celelalte trei canale sunt disponibile pentru utilizarea de către ARM Cortex – M3. Canalele disponibile sunt numerotate 5, 6 și 7.

RAT-ul poate funcționa numai în timp ce core-ul RF este alimentat. RAT-ul trebuie pornit din comanda CMD_START_RAT sau CMD_SYNC_START_RAT. RAT-ul trebuie să ruleze pentru a executa o comandă de operare radio cu pornire întârziată sau orice comandă de operare radio care rulează receptorul sau emițătorul. Când RAT-ul rulează, valoarea curentă a temporizatorului poate fi citită din registrul RATCNT.

## 6.4. Comenzi

CPU-ul radio permite utilizatorului să ruleze un set de primite sau comenzi de nivel înalt pentru procesorul central. După ce o comandă a fost emisă prin intermediul registrului CMDR, CPU-ul radio-ului îl examinează și decide o cale de execuție.

Există trei clase de comenzi :

- Comenzi directe
- Comenzi imediate
- Comenzi de operare radio

## 6.5. Fluxul datelor

Există două moduri de bază pentru transmiterea datelor sau recepționate pe calea aerului :

- Direct
- Printr-o coadă

Pe calea de transmitere directă a datelor trebuie ca să fie adăugate ca parte a parametrii comenzilor, direct sau printr-un pointer. Formatul exact depinde de comanda rulată. În mod normal, există un câmp de lungime și un buffer de date pentru TX (transmitere) și o lungime maximă, lungimea primită dacă această lungime este variabilă și tamponul de primire pentru RX (recepție).

Pentru unele operații, numărul de pachete primite sau transmise nu pot fi cunoscute în avans. Pentru astfel de operații, se utilizează un concept de coadă. O operație poate utiliza una sau mai multe cozi, de exemplu o coadă RX și o coadă TX pentru o operație combinată RX / TX sau mai multe cozi în funcție de informațiile din pachetele primite. Orice operație care utilizează cozi utilizează un sistem comun pentru menținerea acestora. O comandă de operare radio declară ce metode de date sunt folosite.

O intrare de date trebuie plasată într-o coadă. Cozile sunt setate ca făcând parte din structura comenzii radio. Orice comandă care folosește o coadă trebuie să conțină un pointer către intrarea datelor din structură.

## 6.6. Planificarea comenzilor

Procesorul sistemului este responsabil pentru programarea comenzilor după cum este necesar. Când se folosesc moduri de economisire a energiei, procesorul sistemului trebuie să se trezească pentru o scurtă perioadă de timp înainte de începerea următoarei operații, folosind modului de ceas de timp real.

O comandă de operare radio poate fi programată cu un start întârziat. Dacă o comandă este pornită cu o întârziere, procesorul radio trece în modul inactiv până când începe comanda. Comanda de operare radio este considerată a fi difuzată în timpul acestei întârzieri și nicio altă comandă de operare radio nu poate fi programată decât dacă comanda în așteptare este întreruptă sau oprită.

Procesorul sistemului poate programa comenzi de operare radio, utilizând următorul pointer. Acest pointer poate indica următoarea comandă care trebuie efectuată în lanț, iar prin această metodă se pot efectua operații complexe. În anumite condiții cum ar fi o eroare sau expirarea unui timer, următoarea comandă nu este pornită. În schimb, operațiunea se încheie sau un număr de comenzi pot fi omise. Dacă o nouă comandă este planificată în timp ce o altă comandă rulează, CPU-ul sistemului trebuie să aștepte ca comanda anterioară sau lanțul de comenzi să se termine. Comenzile IEEE 802.15.4 au excepții pentru această regulă.

Atunci când o comandă de operare radio este terminată, CPU-ul radioului ridică o întrerupere COMMAND_DONE către CPU-ul sistemului. Dacă un număr de comenzi sunt

înlănțuite după cum s-a explicat anterior, întreruperea COMMAND_DONE este suspendată după fiecare comandă, în timp ce întreruperea LAST_COMMAND_DONE este suspendată după ultima comandă din lanț. Pentru o comandă neînlănțuită, întreruperea LAST_COMMAND_DONE este ridicată după ce se execută. Când LAST_COMMAND_DONE este ridicată, COMMAND_DONE este întotdeauna ridicată în același timp. Înainte de a ridica întreruperea COMMAND_DONE, CPU-ul radioului actualizează câmpul de stare al structurii de comandă la o stare care indică faptul că comanda a fost terminată. CPU-ul radioului nu accesează structura de comandă după ridicarea întreruperii COMMAND_DONE.

## 6.7. Comanda structurii de date

Structurile de date sunt enumerate în cele ce urmează. Octetul index este compensarea de la pointer la acea structură. Câmpurile cu mai mulți octeți sunt de tipul little – endian și jumătate de cuvânt mașină, adică pe 16biți, sau pe alinierea cuvântului 32biți, așa cum este dată de dimensiunea câmpului. Pentru numerotarea biților, 0 este LSB. Coloana R / W este utilizată după cum urmează:

- R : CPU-ul sistemului poate citi un rezultatul întors. CPU-ul radioului nu citește câmpul.
- W : CPU-ul sistemului scrie o valoare. CPU-ul radioului o citește dar nu o poate modifica.
- R / W : CPU-ul sistemului scrie o valoare inițială. CPU-ul radioului poate să o modifice.

Structurile de date sunt o derivare din alte structuri de date, câmpurile din structura părinte nu se poate repeta dar coloana octetului index reflectă prezența lor.

Singurul câmp obligatoriu pentru toate comenzile este numărul de comandă, care este un număr pe 16biți trimis ca 2 octeți ai structurii de comandă.

Unele comenzi imediate au câmpuri suplimentare, care sunt definite pentru fiecare comandă. Comenzile de operare radio au câmpuri obligatorii suplimentare. Toate câmpurile de comandă marcate ca “rezervate” trebuie să fie scrise 0.

# 7. Arhitectura software

Componentele necesare pentru dezvoltarea sistemului, capabil să actualizeze firmware-ul prin OTA este divizat în mai multe module pentru înțelegerea lor.

Fiecare proiect pentru sisteme embedded va avea propriile cerințe pentru proiectarea bootloader-ului, cu toate acestea există puține cerințe care sunt comune tuturor bootloader-elor. Acestea pot fi grupate în mai multe cerințe fundamentale care sunt comune tuturor bootloader-elor. Acestea sunt :

| Cerințe | Sub cerințe |
|---|---|
| Abilitatea de a schimba sau de a selecta modul de operare (Aplicație sau bootloader) | Bootloader-ul are inteligența de a găsi existența unei aplicații valide în memoria flash și transferul de control la aplicație pentru funcționarea normală. |
| | Bootloader-ul trebuie să facă distincția între funcționarea normală și mesajul de actualizare a firmware-ului. |
| Interfețe de comunicare (USB, UART, Wi – Fi, etc.) | Firmware-ul aplicației să poată fi transferat pe oricare dintre interfețele seriale, dar pentru această lucrare vom folosi interfața de comunicare UART și radio. |
| Formatul de fișier al firmware-ului (binar, hex, etc.) | Imaginea aplicației trebuie trimisă în fișier binar. Beneficiul folosiri unui fișier binar rezultă din faptul că acesta are o dimensiune semnificativ mai mică decât formatul Intel Hex sau a alor formate de fișiere și prin urmare va salva lățimea de bandă a rețelei atunci când este descărcată prin radio. Deși formatul de fișier Intel hex este text, trebuie efectuată o analiză suplimentară. |
| Sistemul flash (citire, scriere, ștergere) | Bootloader-ul trebuie să fie capabil să poată șterge secțiuni din memoria flash. Secțiunile aplicației vor fi împărțite în 2 regiuni : regiunea aplicației și regiunea pentru utilizator. În regiunea pentru utilizator se pot păstrarea copii de rezervă a firmware-ului aplicației actuale oferă posibilitatea de a reveni. Cu toate acestea, revocarea va fi complet transparentă și nu va necesita instrucțiuni explicite. |
| | Bootloader-ul scrie imaginea aplicației în memoria flash. |
| | Bootloader-ul trebuie să fie amplasat în primele sectoare ale memoriei flash. |

| | |
|---|---|
| SRAM (citire, scriere, ștergere) | Memoria SRAM integrată în chip ar trebui să fie partajată între bootloader și aplicație. |
| Verificarea aplicației | Bootloader-ul trebuie să fie capabil să calculeze CRC-ul aplicației. |
| Securitate | Bootloader-ul se va proteja singur de accidentalele modificări din partea aplicației. |

Tabelul 7.1. Cerințele bootloader-ului

## 7.1. Modele comportamentale ale bootloader-ului

Un bootloader nu este mult diferit de alte aplicații. Dar, aceasta este prima aplicație care se execută la pornirea sistemului, și transferă controlul sistemului către aplicație. Bootloader-ul are capacitatea de a șterge și de a programa o nouă aplicație. Pe un sistem tipic embedded, resursele sunt limitate, în special memoria pe chip (memoria flash și memoria SRAM), iar prin urmare, proiectarea bootloader-ului trebuie să fie foarte judicioasă în a decide caracteristicile bootloader-ului. Ar trebui să utilizeze cantitatea minimă de spațiu de memorie flash care va fi disponibilă pentru codul aplicației.

Din punct de vedere istoric, există două modele comportamentale care descriu modul în care se poate comporta un bootloader. În primul model, procesul de încărcare a bootloader-ului este complet automatizat și autonom în cadrul sistemului. Un exemplu în acest sens ar fi un bootloader de pe cardurile SD (**S**ecure **D**igital).

Bootloader-ul va detecta automat noul firmware și va gestiona propriu mod de scriere. Comenzile de la o sursă externă nu ar fi necesare pentru efectuarea cu succes a procesului.

În cel de-al doilea model, sistemul nu se ocupă automat de procesul de bootloader în sine. În schimb, bootloader-ul este inițializat într-o stare de așteptare și așteaptă comenzi de la o sursă externă. Această sursă externă o aplicație software de pe un PC care comandă bootloader-ul în diferitele stări necesare pentru a scrie noua imagine pe sistem. Motivul principal al aplicației software externe de a comanda procesul este ca în marea majoritate a aplicațiilor nu există un card SD pentru a prelua imaginea software. În schimb o sursă externă cu imagine acționează ca un master în procesul de bootloader-ul. Aici aplicația pentru sistemele embedded și software-ul PC funcționează în arhitectură server-client.

## 7.2. Bootloader-ul integrat pe placa de dezvoltare

Unele procesoare, cum ar fi TI CC2650 LaunchPad, au un bootloader intern care va încărca codul dintr-o sursă externă dacă sunt setați pinii I / O corespunzători și anumite configurări software. Într-o lume ideală, s-ar seta doar un pin I / O pentru a încărca noul cod în sistem. Acest lucru ar fi declanșat când sistemul se va activa. Bootloader-ul va încărca automat datele în spațiul pentru codul aplicației. Noul cod este transferat către procesor folosind unele metode de comunicare, UART sau SPI. Codul bootloader-ului înăuntrul chipului citește și transferă codul în adresele specifice pentru aplicație. Acesta este cel mai simplu bootloader.

## 7.3. Bootloader personalizat (custom)

Un bootloader particular partajează spațiul de cod cu aplicația. Marele avantaj al bootloader-ul personalizat este flexibilitatea. Acesta este bootloader-ul ales pentru această lucrare. Pentru programarea memoriei flash sunt necesare 2 etape: ștergerea și programarea. Într-un mod tipic, memoria flash este împărțită în sectoare și majoritatea memoriilor flash au granularitatea pentru ștergerea la nivel de pagină și granularitatea scrierii la nivel de cuvânt. Aceasta înseamnă că pentru a scrie un cuvânt la aceeași adresă, trebuie să se șteargă o pagină întreagă. Un bootloader primitiv, va șterge zona de aplicație din memoria flash și apoi va programa noua imagine.

Această abordare are multiple probleme precum :

- Noua imagine a aplicației ar putea fi coruptă de la sursă.
- Imaginea aplicației ar putea fi coruptă din cauza erorilor de comunicare.
- Ștergere parțială.
- Procesul de programare ar putea fi întrerupt.

Aceste probleme vor conduce la defectarea sistemului. Sistemul va prezenta un comportament diferit bazat pe scenariul de eroare. Există mai multe abordări pentru a rezolva aceste probleme. De exemplu, o sumă de control ar putea detecta dacă imaginea este coruptă, o copie de abordare de scriere va asigura că vechea aplicație păstrată până când noua aplicație este scrisă corect în memoria flash.

## 7.4. Pornirea din SRAM

Uneori este necesară și actualizarea bootloader-ului. Un bootloader nu este altceva decât o aplicație cu anumite cerințe specifice. Pentru a actualiza bootloader-ul rezident, bootloader-ul vechi descarcă bootloader-ul nou în SRAM, apoi transmite controlul către noul bootloader și rescrie vechiul bootloader cu cel din SRAM. Odată ce noul bootloader este scris, funcționarea normală poate fi reluată. Acest tip de bootloader este mult mai complex.

## 7.5. Inițializarea de început

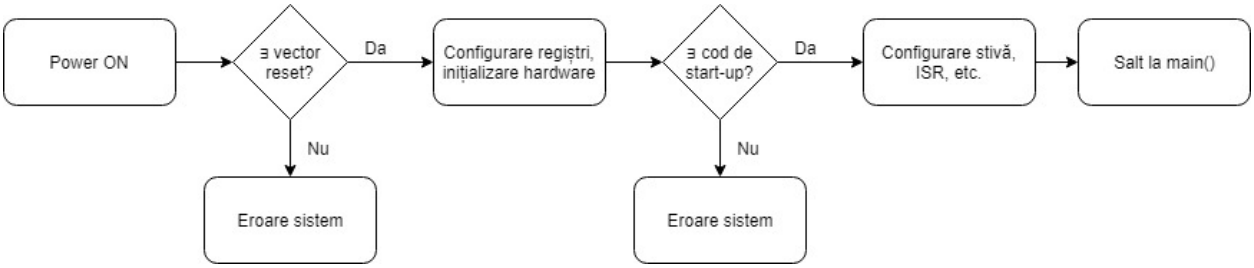


*Fig. 7.5.1 Secvența de pornire*

Procesul de pornire este prezentat în figura 7.5.1. Vectorul de reset este primul lucru care rulează la pornire. Este un cod specific pentru hardware. Efectuează funcții simple cum ar fi configurarea procesorului într-o stare stabilă predefinită prin configurarea regiștrilor, etc. Apoi se va trece la codul de pornire.

Codul de pornire este primul cod specific software-ului care rulează. Misiunea sa este de a configura în mod normal mediul software, astfel încât codul C sa poată să fie rulat. De exemplu, codul C presupune că există o regiune de memorie definită ca stivă și heap. Acestea sunt de obicei construite de software în loc de hardware. Prin urmare, această bucată de cod de pornire va defini liniile de întrerupere către stivă și pointerii către memoria heap. Variabilele care trebuie inițializate și anumite părți ale memoriei care au nevoie de ștergere se vor face aici. Practic, tot ceea ce este necesar pentru a muta lucrurile într-o stare cunoscută. La sfârșitul rutinei se va executa funcția main.

În acest moment există cel puțin două imagini software diferite care pot fi încărcate și executate de către procesor, bootloader-ul și aplicația, eventual și o imagine de aplicație de back-up. Pentru această lucrare, s-a decis că bootloader-ul va fi prima aplicație care va fi executată.

Presupunând că adresa de pornire a aplicației este întotdeauna fixată și dacă nu există actualizări, bootloader-ul va sări la adresa aplicației. Dacă nu există o aplicație la care să sară, sau este coruptă, execuția codului va rămâne în bootloader. Există câteva motive pentru care acest lucru se poate întâmpla :

- Bootloader-ul există în memoria flash, dar nu există nicio aplicație și ciclu de alimentare a fost reluat.
- Firmware-ul aplicației a fost șters ca parte a procesului de actualizare, dar înainte ca noua aplicație să poată fi scrisă, s-au produs unele erori sau cicluri de reset. Aceasta duce la o zonă a aplicației goală.

În astfel de cazuri, un bootloader primitiv va sări la adresa de pornire a aplicației fără a face nicio validare și sistemul ar putea să se blocheze pentru totdeauna într-o buclă infinită, fără a face nimic. Prin urmare, este imperativ să se verifice dacă indicatorul stack-ului aplicației există. Verificarea este specifică hardware-ului și este bazat pe SoC.

Acesta este testul cel mai de bază și funcționează în mare pare a scenariului, cu toate acestea există o problemă cu această abordare. Verificarea nu este validată dacă aplicația este într-adevăr corectă. Dacă imaginea aplicației a fost deteriorată de la sursă sau în timpul scrierii, atunci saltul la aplicație va duce la un comportament nedefinit. O soluție ușoară este de a efectua o validare sunt sumelor de control înainte de a sări la aplicație. Odată ce dezvoltarea aplicației s-a terminat, suma de control se calculează și se trimite împreună cu aplicația. Odată ce noua aplicație a fost programată în memorie, bootloader-ul folosește același algoritm folosit anterior pentru a genera suma de control a aplicației. Apoi se vor compara cele 2 sume de control. Dacă cele două sume de control nu se potrivesc, acest lucru indică că imaginea aplicației este coruptă, prin diferite metode. În acest caz, bootloader-ul nu va sări la imaginea aplicației și va rămâne în bootloader. Acest lucru îmbunătățește fiabilitatea sistemului.

După validarea cu sume de control, sistemul devine mai robust la defecțiuni, prin simpla stocare a sumei de control primite prin radio undeva în memoria flash non-volatilă.

## 7.6. Vectorul de reset și de întreruperi

Vectorul de reset este locația de memorie unde se află prima instrucțiune a aplicației. În timpul procesului de pornire, procesorul începe executarea programului de la adresa stocată în vectorul de reset. Pentru a face saltul de la bootloader la executarea codului aplicației s-au folosit instrucțiuni de asamblare.

Așadar, în următoarea secvență de cod, este exemplificat saltul la imaginea ce a fost scrisă prin intermediul bootloader-ului. Trebuie ca imaginea scrisă să fie mereu la aceeași adresă fixată. Am ales pentru această lucrare ca adresa să fie 0x10000.

```
void saltProgram()
{
    // Se va schimba constanta pentru a se potrivi cu vectorul
    // de start al imaginii
    //              vvvvvv
    asm(" MOV R0, #0x10000 ");     // .resetVecs sau .intvecs pentru aplicatie
                                   // sunt plasate la o adresa constanta #0xXXXX
    asm(" LDR SP, [R0, #0x0] ");  // Incarca pointerul stackului initial
    asm(" LDR R1, [R0, #0x4] ");  // Incarca adresa vectorului de reset
    asm(" BX R1 ");               // Sare la vectorul de reset al aplicatiei
}
```

Prin instrucțiunea MOV, se copiază valoarea imediată, adică adresa la care trebuie să sărim, 0x10000, în registrul de uz general R0. Apoi se va încărca cu offset imediat, în registrul Stack Pointer, valoarea din registrul R0, cu pre-indexare imediată de offset, de 0x0.

În cele ce urmează se va efectua o încărcare imediată în registrul de uz general, a valorii ce se află în registrul R0, dar de data aceasta cu un offset de 4 caractere. Un caracter reprezintă un byte.

În ultimul rând se va face ramificarea și execuția de la parametrul specificat, la noi fiind R1, unde s-a încărcat anterior vectorul de reset și se va începe execuția aplicației.

## 7.7. Formatul fișierului

Firmware-ul care urmează să fie recepționat poate fi în formate diferite, în funcție de toolchain-ul folosit, compilator, hardware-ul țintă. Unele dintre cele mai populare formate de fișiere sunt formatul Intel Hex, binar, ARM executable, etc. Toolchain-urile folosite de către TI generează fișiere Intel Hex [16] pe care le-am prelucrat in Python [17] pentru a le transforma în fișiere binare și să fie transmise, de asemenea TI mai generează fișiere obiect care au atât codul executabil cât și informații suplimentare despre debug.

Instrumentele ISP (**I**n **S**ystem **P**rogramming), cum ar fi JTAG, sunt suficient de inteligente pentru a elimina informațiile despre debug și pentru a încărca doar codul executabil pe dispozitivul țintă. Parsarea unui fișier hex este destul de ușoară. Este un text ASCII cu linii de text care urmează formatul de fișier Intel Hex [16]. Fiecare linie dintr-un fișier Intel Hex conține o înregistrare hex. Aceste înregistrări sunt alcătuite din numere hexazecimale care reprezintă codul limbii mașinii.

Două caractere hexazecimale din fișierul generat de către compilator, reprezintă un caracter ce va fi stocat în memoria flash a microcontroller-ului. Aceste fișiere Intel Hex sunt adesea folosite de programatori pentru a transfera programul care ar fi stocat în regiunea codului din memoria flash. Sfârșitul unei linii de cod hex se termină cu două caractere speciale. Acest proces este unul direct, cu toate că, în timp ce se primește un fișier hex pe interfața serială, bootloader-ul ar trebui sa pună în aplicare logica parsării.

Din motive de simplitate, bootloader-ul dezvoltat ca parte a acestei lucrări, se așteaptă să primească un fișier binar simplu.

Există două beneficii pentru această abordare :

- Fișierul binar simplu este semnificativ mai mic, chiar cu 60% mai mic decât fișierul hex, reducând astfel utilizarea traficului în rețea în timpul descărcării firmware-ului.
- Nu este nevoie să se parseze imaginea primită, reducând astfel întreruperile totale ale sistemului. În timpul procesului de actualizare a firmware-ului. Sistemul nu este utilizabil.

Conversia din fișierul hex în fișierul binar s-a făcut utilizând modulului [18] IntelHex din Python. Și anume se crea un obiect de tipul IntelHex și se salva sub formă binară. Următoarea secvență de cod exemplifică acest lucru.

```
from intelhex import IntelHex
ih = IntelHex("led1.hex")
ih.tofile("led1.bin", format = 'bin')
```

Secvența de cod din fișierul hex generat de către toolchain-ul TI:

```
:20568000034D40034DC2A0034E40034EC0034F40034403D09041E3541CC880AF131F14362
:2056A00042FFF000A52501005B4C0100014D01000200000002D0E0000405501000F80E0020FE
:0856C000B0560100C8000020F3
:205FA8000000800110008 2FFFDFF58003AFFBFF3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF98
:205FC800FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00FFFFFFFF00FFFFFFC500C5FF4A
:185FE8000000000FF00000000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFB2
:00000001FF
```

```
: 11 2222 33 44444444444444444444444444444444 55
```

Legenda codului este următoarea :

- : = Codul de start
- 11 = Numărul de octeți din înregistrare
- 2222 = Adresa unde se scriu
- 33 = Tipul de date
- 44...4 = Datele
- 55 = Suma de control
- Finalul liniei se termină cu 2 coduri speciale și anume :
    - Carrige Return – codul hex este 13
    - Line Feed – codul hex este 10

## 7.8. Vizualizator a fișierului Intel Hex

S-a dezvoltat un vizualizator pentru a vedea și interpreta fișierul hex mult mai ușor, și de asemenea pentru salvarea într-un fișier separat doar a datelor din formatul hex, alternativă a modului IntelHex din Python. Acest vizualizator putând fi implementat și pe un microcontroller.

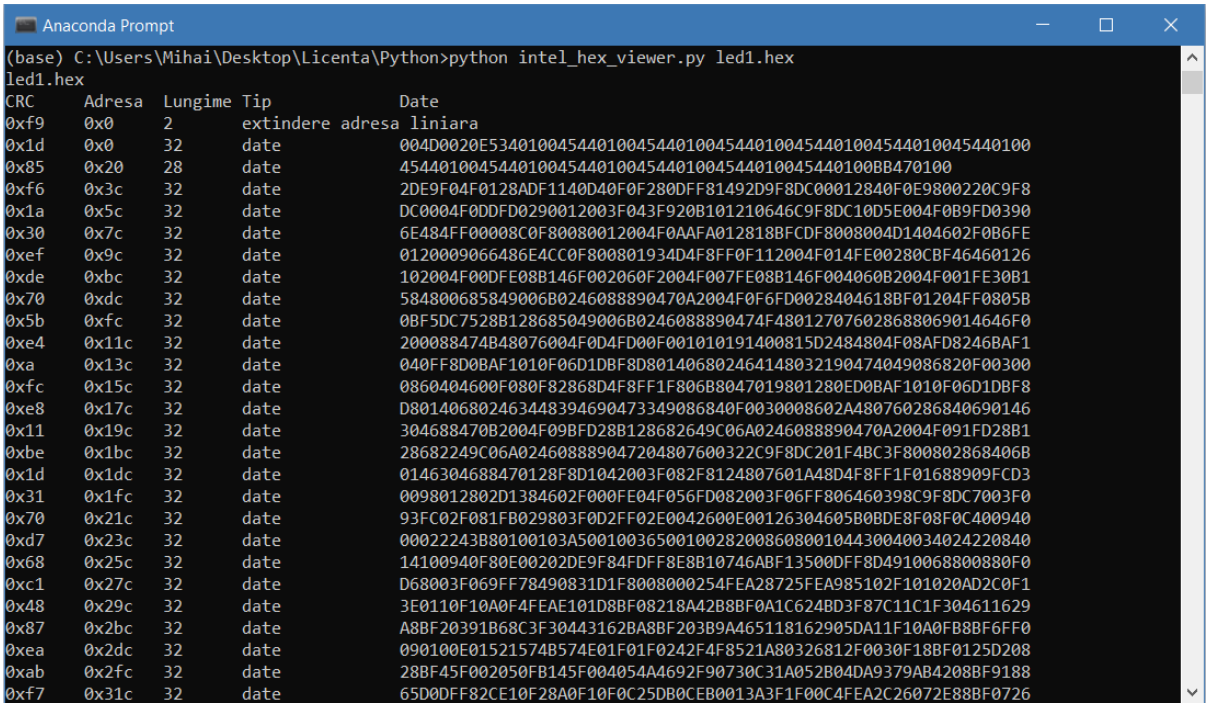


*Fig. 7.8.1 Captura de ecran a execuției vizualizatorului*

Pentru a fi implementată pe microcontroller, se va trimite toată linia din fișierul hex care conține înregistrarea și se va face o analiză după cum urmează:

- Se va face abstracție de primul caracter fiind un cod special care semnifică începutul înregistrării.
- Se va prelua de la caracterul cu indexul 1 și 2, combinându-le într-un singur caracter și se va stoca într-o variabilă ce va fi transmisă prin radio, acesta reprezentând numărul de octeți din înregistrare din câmpul de date.
- Se va prelua adresa din înregistrare de la caracterul cu indexul 3 până la caracterul cu indexul 6, care de asemenea se va face conversia în hex din caractere ascii.
- Se vor prelua datele din înregistrare începând de la caracterul cu indexul 9 până la 9 + 2 × numărul de octeți din înregistrare.
- Se va prelua CRC – ul din înregistrare de la caracterul cu indexul 9 + 2 × numărul de octeți din înregistrare până la 11 + 9 × numărul de octeți din înregistrare.

# 8. Detalii de implementare

## 8.1. Standul de testare

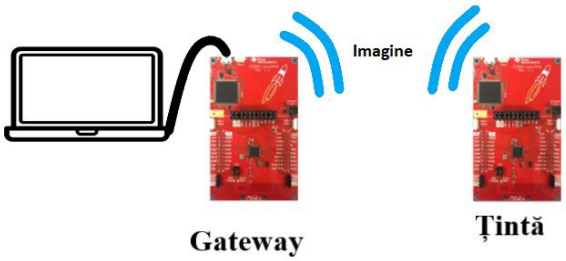


*Fig. 8.1.1 Arhitectura fizică*

Pentru arhitectura fizică s-au folosit 2 plăci de dezvoltare numite CC2650 LaunchPad de la producătorul Texas Instruments. O schiță pentru aceasta se poate vedea în figura 8.1.1

Cele 2 plăci sunt Downloader și Target sunt după cum urmează :

- Downloader-ul acționează ca un gateway, se poate folosi orice dispozitiv pentru a trimite date.
- Target-ul primește pachetele, interpretează fiecare comandă din pachete și acționează conform comenzii.

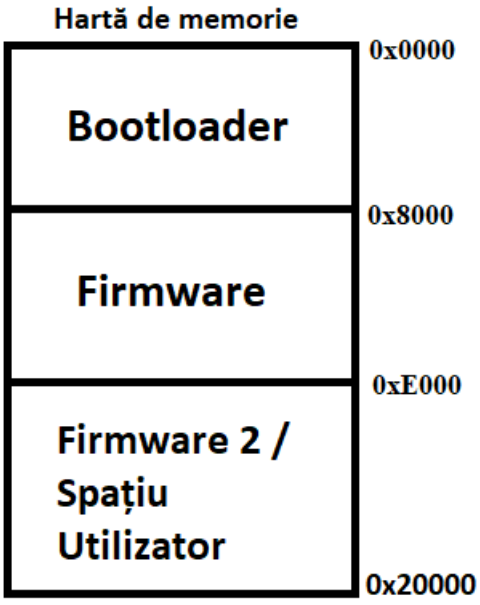


*Fig. 8.1.2 Harta de memorie*

Memoria non-volatila, flash, a dispozitivului țintă este împărțită in 2 sau mai multe secțiuni conform fig. 8.1.2. Dimensiunea totala a memoriei este de 128KBytes. Secțiunea de memorie a imaginii bootloader-ului, având dimensiunea de 32KBytes, care se este responsabil cu pornirea dispozitivului și se executa prima dată la pornirea dispozitivului. Verifică dacă există vreo actualizare pentru firmware-ul principal și primește noile actualizări prin radio, apoi scrie noua imaginea în secțiunea de memorie a firmware.

Secțiunea firmware, având dimensiunea de 24KBytes din totalul de memorie a hărții de memorie este codul aplicației care definește funcționalitatea sistemului. De asemenea pot exista secțiuni pentru stocarea datelor persistente de către utilizator, spațiul rămas este de 72KBytes.

Desigur că dimensiunea firmware-ului, spațiul rămas și adresele la care se pot scrie imaginile sunt configurabile în totalitate.

Pentru criptarea datelor se poate folosi modulul integrat în dispozitivul folosit de noi, și anume standardul avansat de securitate (AES). Acest modul de securitatea AES furnizează operații de criptare și decriptare a datelor bazat pe un chip hardware implementat cu o cheie binară. Modulul acceptă o cheie de 128 biți pentru criptare și utilizează algoritmul simetric, ceea ce înseamnă că cheile de criptare și decriptare sunt identice. Criptarea convertește datele textului simplu într-o formă neinteligibilă numită text de cifru. Decriptarea textului de cifru convertește datele criptate anterior înapoi în forma originală a textului simplu.

## 8.2. Actualizarea firmware-ului prin radio (proprietar)

Eficacitatea actualizării prin radio a fost dovedită în aplicații precum smartphone-uri care primesc actualizări noi și fixarea breșelor periodic. Actualizarea de firmware prin OTA la dispozitive embedded prin radio a adus această eficacitate de asemenea la internetul obiectelor.

Exista 3 motive pentru actualizarea dispozitivelor embedded prin OTA în contextul internetul obiectelor și anume :

- Aplicații largi și eterogene a dispozitivelor

Numărul de dispozitive și diferite, joacă un rol foarte important în rețelele distribuite. Standardizarea OTA interfațează reutilizarea arhitecturii pentru mai multe noduri.

- Schimbarea frecventă a cerințelor

Internetul obiectelor creste rapid schimbând market-ul cu multe cerințe noi ale produselor, și caracteristici noi sunt adăugate regulat. Amenințarea de securitate și a breșelor sunt cel mai mare factor care produce aceste schimbări regulate. Pentru a proteja împotriva atacurilor de noi viruși, firmware-ul dispozitivului poate încorpora fixuri și mai securitate îmbunătățită prin algoritmii actualizați prin radio. Actualizările radio pot schimba de asemenea și reconfigurările a resurselor hardware.

- Cererea critica a pieței

Sistemele IoT au un timp foarte scurt de design și este o cerere constantă de inovație și de plasare a ultimelor actualizări.

Procesul general de dezvoltare este de a face over – design a hardware-ului pentru a susține cerințe extinse a marketului pentru o perioadă îndelungată de timp.

OTA face posibilă plasare soluțiilor în mai multe etape. De exemplu, design-ul inițial pentru un termostat poate ieși mai devreme cu un senzor termal și viitoarele actualizări pot activa și citirea umidității cu ajutorul senzorului.

Această abordare este de proiectare hardware este un proces gândit pentru viitoarele lansări luate în considerare în faza de arhitectură dintr-o perspectiva hardware.

Trebuie luată în considerare presiunea lansării unei noi versiuni de firmware foarte rapid, din cauza cererii pieței. Această presiune se poate transforma în software instabil, sau produse care nu au fost verificate, testate și optimizate într-o măsură riguroasă.

De asemenea pentru noile actualizări foarte frecvente există riscul ca să nu fie acceptate de utilizatori foarte bine.

În ziua de astăzi standardul radio impune un consum de energie foarte scăzut și este suportat de majoritatea PC-urilor și a telefoanelor inteligente. Răspândit foarte mult, radio-ul se face preferat în aplicațiile IoT. În fiecare an, standardul radio evoluează, lansând noi versiuni pentru a încorpora cât mai multe facilități si caracteristici cerințelor aplicațiilor IoT.

În competiția pieței, fabricanții de silicon sunt într-o cursa continuă pentru a oferi soluții complete la noile versiuni de actualizare prin radio. În multe cazuri, fabricanții oferă aceste facilități în mai multe stagii. De asemenea schimbările foarte dese, fac ca timpul de testare să devină foarte scurt și poate introduce foarte multe defecte.

O arhitectură de bootloader robustă ar trebui să suporte aceste cazuri diferite să fie flexibilă îndeajuns astfel încât să se adapteze la schimbările de specificație a aplicațiilor.

## 8.3. Aplicațiile software

S-au dezvoltat aplicații în limbajul Python pentru trimiterea pe serială a datelor care apoi vor fi trimise prin radio.

Bootloader-ul și firmware-urile s-au dezvoltat în limbajul C, folosind mediul de dezvoltare integrat (IDE) Code Composer Studio [19]. Aceste imagini, bootloader-ul si cele imaginile de test vor fi compilate cu toolchain-ul pus la dispoziție de Texas Instruments, rezultând un fișier Intel Hex.

Pentru încărcarea fișierului binar s-au dezvoltat multiple versiuni de bootloader. Două dintre acestea sunt pentru încărcarea pe serială. Unul încarcă programul în rafală, iar celălalt primește pachete de dimensiuni reduse.

Citirea din fișier a mai multor octeți s-a făcut din fișierul binar, urmând ca apoi să fie împachetate și trimise către microcontroller utilizând interfața serială. Configurarea și trimiterea pe interfața serială a unui caracter și interpretarea acestuia ca și comandă iar apoi trimiterea unui pachet către dispozitivul unde se face actualizarea.

Pentru a trimite adresa unde se va salva noul firmware, a fost nevoie să fie spartă în 4 caractere a câte 8 biți, deoarece pe interfața serială un singur caracter la un moment dat de timp și reconstrucția adresei de memorie pe microcontroller utilizând lucrul cu biți.

Când se face scrierea în memoria flash trebuie să se verifice în prealabil dacă automatul de stare implementat de memoria flash este gata să scrie și nu s-au produs erori. Trebuie ținut cont și de sectoarele unde dorește să se scrie noua imagine, deoarece nu trebuie să fie suprapuse cu adresele din memorie care aparțin bootloader-ului.

Aplicația de bootloader este organizată pe 4 thread-uri, fiecare thread pentru un lucru specializat. Aceste thread-uri sunt :

- Thread-ul de așteptarea caracterelor pe interfața serială și punerea acestora într-un buffer de emitere.
- Thread-ul pentru închiderea modului de recepție radio și pornirea modului de emitere radio și trimiterea efectivă a caracterelor primite pe interfața serială.
- Thread-ul pentru ascultarea radio și copierea într-un buffer intern din memoria SRAM a ceea ce se primește.
- Thread-ul pentru interpretarea comenzilor din buffer-ul de recepție, scrierea în memoriei flash și confirmarea că s-a primit pachetul.

Bootloader-ul este gândit pentru comunicație full duplex. Acesta pornește modul de ascultare imediat după ce s-a terminat transmiterea. Comunicația full duplex este implementată prin ping-pong.

Pentru a seta modul de trimitere se face cererea către procesorul responsabil de comunicația radio, dacă acesta este în folosire de către alt thread, nu poate să fie folosit încă o dată. Trebuie setată frecvența și prioritatea comenzilor.

Frecvența 2440MHz, această frecvență trebuie setată pe ambele dispozitive, pentru a se înțelege.

Prioritatea de trimitere este normală, deoarece nu avem altă nevoie de radio în afara de a încărca o imagine în flash și pachetele nu trebuie să se facă aleator, este nevoie de secvențierea pachetelor. Se va prelua timpul de la core-ul RF pentru a adăuga gestiunea automată a energiei și pentru a radia mai puțin în jur.

Se va da comanda de trimitere după care se verifică dacă s-au produs erori la recepție pentru cea de a doua placă, de exemplu nu se primește confirmare de primire a pachetului, iar dacă nu s-a primit confirmarea, se va reîncerca trimiterea, altfel este semnalat prin pornirea led-urilor dacă s-a eșuat trimiterea aceluiași pachet de mai multe ori. După care se va bloca thread-ul curent și se va porni thread-ul de recepție pentru a închide modul de transmitere și a seta parametrii pentru recepție. Asemănător se va face cererea de acces către core-ul RF, se va seta frecvența și se va seta întreruperea pentru când se va primi un pachet, după care se blochează thread-ul.

Întreruperea care se produce la primirea unui pachet este o întrerupere software (SWI) și în această întrerupere se verifică evenimentul și masca evenimentului, pot exista mai mult evenimente care pot produce întreruperea dar pe noi ne interesează doar dacă a venit un pachet. Întreruperea se declanșează doar dacă dispozitivul țintă își recunoaște adresa în pachet. Vom copia

acest pachet din coada alocată, într-un buffer din memorie după care se deblochează thread-ul pentru interpretarea pachetului și scrierea în memorie a acestuia.

Pentru ca o întrerupere să se producă, este nevoie de același preambul și cuvânt de sincronizare pe ambele dispozitive, altfel un dispozitiv doar trimite iar celălalt nu va primi deoarece va crede că nu este pentru el.

Un cuvânt de sincronizare este pentru a sincroniza transmisiile de date, indicând sfârșitul informațiilor din antet (header) și începutul datelor. Acest cuvânt este o secvență cunoscută de ambele dispozitive care comunică pentru a identifica începutul unui cadru (frame) și se mai numește semnal de referință. Codurile prefixelor permit identificarea neechivocă a secvențelor de sincronizare și poate servi drept cod de sincronizare.

Acest cuvânt de sincronizare pe care l-am folosit este : 0xD391D391.

Comutarea între thread-uri se va face prin semafoare, deoarece thread-urile au priorități diferite iar dacă nu se vor folosi această metodă, se va executa mereu thread-ul cu cea mai mare prioritate fiind un sistem de operare în timp real cu planificator pre – emptiv. Pentru thread-ul care ascultă pe interfața serială. Acest thread a fost conceput pentru a simula una dintre plăci ca fiind un gate-way.

Verificarea că s-a scris o imagine nouă pe flash reprezintă prin saltul la aceasta. O imagine va aprinde și va stinge un led verde, iar cealaltă imagine va aprinde și va stinge un led verde și un led roșu.

# 9. Concluzii și dezvoltări viitoare

Abilitatea de a efectua actualizări de firmware, după ce un sistem a fost deja livrat, este foarte atractiv pentru producătorii de sisteme. A devenit mai relevantă în ultima vreme, datorită creșterii impresionante a dispozitivelor din domeniului IoT. O soluție eficientă și scalabilă este livrarea unui bootloader împreună cu produsul real, totuși aceasta este o sarcină dificilă din varii motive :

- În lumea IoT, unde consumul de energie este de o importanță extrem de mare, produsele bazate pe procesoarele ARM sunt foarte utilizate din motivul că au un consum extrem de redus. Modelul de licențiere a companiei ARM și a diferitelor familii de procesoare ARM a permis producătorilor de hardware să proiecteze o gamă foarte largă de produse bazate pe aceste procesoare ARM.
- Sistemele integrate cu o putere foarte mică au de asemenea și resurse foarte limitate.
- Diferite medii de implementare.

Din aceste motive, bootloader-ele sunt aplicații de mare interes, dar ideile generice din spatele fiecărui bootloader sunt identice, un model de design bun ar putea fi dezvoltat în timp ce se concentrează efortul pe partea de inginerie pentru a dezvolta bootloader-ul. Deși bootloader-ul nu este produsul final, dar are impact mare în partea produsului final și prin urmare, proiectarea bootloader-ului ar trebui să fie luată în considerare încă din stadiul incipient.

Potrivit directorului de la Intel, securitatea este al III-lea pilon al dispozitivelor, performanța și conectivitatea fiind celelalte două. O actualizare a software-ului nesecurizată ar putea deschide cutia Pandorei. Prin urmare, trebuie depuse eforturi pentru a actualiza software-ul într-un mod cât mai sigur posibil. În vederea unei securități mai bune, în viitor se intenționează să se îmbunătățească bootloader-ul curent prin integrarea caracteristicilor de securitate.

De asemenea, se intenționează să se crească viteza de transmitere a unei imagini noi, pentru a scădea timpul de așteptare, și implicit scăderea șanselor de interceptare a pachetelor. Desigur că acest lucru poate implica și trecerea la un sistem superior, cu caracteristici mai bune, ca de exemplu:

- Un procesor mai bun pentru a face schimbările între thread-uri mult mai rapid, eventual un procesor cu mai multe nuclee pentru a face paralelizare.
- Mai multă memorie, deoarece memoria actuală este de doar 128KBytes și nu este suficientă pentru aplicații complexe partajând memoria cu bootloader-ul și firmware-ul.
- Module de comunicație mai rapide, spre exemplu LAN, Bluetooth 5.0, Wi – Fi 5GHz
- Reducerea consumul de energie cât mai mult posibil.

Se dorește să se poată face translatarea imaginii într-o rețea, actualizând firmware-ul țintă trecând prin mai multe dispozitive.

# 10. Bibliografie


[1] A guide to internet of things [https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html] (06.11.2018)

[2] A framework for Self – Verification of Firmware Updates over the Air in Vehicle ECUs [https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4746641] (06.11.2018)

[3] Secure Firmware Updates over the Air in Intelligent Vehicles [https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4531926] (06.11.2018)

[4] Tesla's quick fix for its breaking system came from the ether [https://www.wired.com/story/tesla-model3-braking-software-update-consumer-reports/] (06.11.2018)

[5] Internet of Things: Over – the – Air Firmware Update in LightweightMesh Network Protocol for Smart Urban Development [https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7581459] (06.15.2018)

[6] Secure FoTA object for IoT [https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8110218] (06.12.2018)

[7] An efficient and Secure Automotive Wireless Software Update Framework [https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8141933] (06.13.2018)

[8] Texas Instruments CC2650 Datasheet [http://www.ti.com/lit/ds/symlink/cc2650.pdf] (06.13.2018)

[9] CC13x0, CC26x0 SimpleLink™ Wireless MCU - Technical Reference Manual [http://www.ti.com/lit/ug/swcu117h/swcu117h.pdf] (06.14.2018)

[10] Serial communication [https://learn.sparkfun.com/tutorials/serial-communication/all] (06.14.2018)

[11] UART – Function Description [https://www.amebaiot.com/en/uart/] (06.14.2018)

[12] RTOS Concepts [http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts] (06.15.2018)

[13] Tanenbaum, Andrew, Modern Operating Systems, Upper Saddle River, NJ: Pearson/Prentice Hall, p. 160

[14] Texas Instrumens – Real Time Operating System [http://processors.wiki.ti.com/index.php/TI-RTOS] (06.14.2018)

[15] Difference between DSP/BIOS and SYS/BIOS [http://processors.wiki.ti.com/index.php/Differences_Between_DSP/BIOS_and_SYS/BIOS] (06.15.2018)

[16] Intel Hex format [http://www.keil.com/support/docs/1584/] (06.15.2018)

[17] Python Official [https://www.python.org/]

[18] Python IntelHex [https://pypi.org/project/IntelHex/]

[19] Code Composer Studio [http://www.ti.com/tool/CCSTUDIO]

[20] Over-the-air firmware upgrades for Internet of Things devices [http://www.embedded-computing.com/embedded-computing-design/over-the-air-firmware-upgrades-for-internet-of-things-devices] (06.21.2018)

[21] ARM ® Cortex [https://www.arm.com/]