Universitatea
Ştefan cel Mare
Suceava

Universitatea Ștefan cel Mare Suceava

Facultatea de Inginerie Electrică

și Știința Calculatoarelor

LUCRARE DE LICENȚĂ

Sistem înglobat bazat pe microcontroler pentru controlul rotației în funcție de intensitatea luminii

Coordonator proiect:

S.l. dr. ing. Zagan Ionel

Student:

Pascal Vasile-Marian

SUCEAVA

2022

Rezumat

În lucrarea de față sunt expuse etapele și tehnologiile utilizate pentru realizarea unui sistem înglobat bazat pe microcontroler pentru controlul rotației în funcție de intensitatea luminii ambientale preluată de senzorii de intensitate a luminii de pe machetă, aceasta fiind așezată pe un rulment cu un motor pas cu pas controlat automat sau manual, dintr-o interfață grafică.

Proiectul înglobează o parte hardware, concepută prin intermediul unui kit de dezvoltare STM32 F429I-Discovery alături de o serie de echipamente electronice și de senzori dar și o parte software creată în limbajul de programare C++. Configurația pinilor este făcută cu ajutorul software-ului specializat STM32 CubeMX, iar interfața este proiectată în software-ul specializat TouchGFX. Acest proiect are idealul de a simula un sistem inteligent de control pe un singur ax al rotației unei machete în funcție de intensitatea luminii solare.

Utilizatorului i se va oferi posibilitatea de a observa parametrii de intensitate ai luminii printr-un grafic și de a seta controlul sistemului de pe automat pe manual, având capacitatea de a schimba rotația machetei în funcție de preferințe prin intermediul interfeței grafice.

CUPRINS

1. INTRODUCERE

1.1 Contextul proiectului

Un sistem inteligent este un sistem care recepționează datele primite din mediul înconjurător în care se află și efectuează pașii necesari pentru a își maximiza șansele de a își îndeplini cu succes scopul pentru care a fost creat.

1.2 Tema proiectului

Tema proiectului constă în realizarea unui concept de sistem înglobat bazat pe microcontroler pentru controlul automat al rotației unei machete în funcție de intensitatea luminii ambientale. Macheta este așezată pe un rulment controlat de un motor pas cu pas al cărui driver se va ocupa de poziționare cu ajutorul senzorilor de intensitate a luminii și al rulmentului.

Funcționalitatea sistemului se poate schimba în funcție de nevoi, aceasta putând fi controlată din interfața grafică prezentă pe LCD-ul încorporat pe placa de dezvoltare STM32 F429I-Discovery. În cazul în care utilizatorul va alege controlul manual al rotației machetei, acesta va primi accesul total pentru modificarea poziției de rotație. Aplicația oferă o experiență plăcută prin interfața grafică intuitivă. Pe lângă butoanele de control al modului de poziționare, aceasta afișează valorile parametrilor intensității luminoase citite din mediu de către senzorii de intensitate a luminii și un grafic dinamic cu diferența dintre valorile citite de senzori.

2. TEHNOLOGII FOLOSITE

2.1 Limbajul de programare C++

2.1.1 Introducere

C++ este un limbaj de programare pentru folosință generală, creat de Bjarne Stroustrup ca o îmbunătățire a limbajului de programare C, mai exact "C cu clase". Limbajul s-a extins în mod foarte semnificativ de-a lungul timpului, iar limbajul C++ are caracteristici orientate pe obiecte, generale și funcționale, pe lângă facilitățile de manipulare a memoriei. Aproape întotdeauna este pus în practică drept un limbaj compilat, iar mulți furnizori oferă compilatoare C. C++ a fost creat cu o orientare către programarea de sisteme și către software încorporat, cu resurse limitate, cu performanțe, eficiență și de asemenea, flexibilitate de utilizare ca puncte forte ale proiectării sale. C++ s-a dovedit a fi util și în multe alte contexte, inclusiv aplicațiile desktop, jocurile video, serverele (ex. căutarea pe internet sau bazele de date) și aplicațiile cu performanțe critice (de exemplu, comutatoarele telefonice sau sondele spațiale). Bjarne Stroustrup dorea un limbaj mai eficient și mai flexibil, asemănător cu C, care să ofere, de asemenea mai multe caracteristici de programare în nivel înalt pentru organizarea mai bună a programelor.

2.1.2 Filosofie

De-a lungul vieții limbajului de programare C++, îmbunătățirea și evoluția sa a fost condusă de un set de principii:

- Trebuie să fie determinat de probleme cât mai reale, iar caracteristicile sale trebuie să fie cât mai utile în programele din lumea reală.
- Fiecare caracteristică ar trebui să fie implementabilă, cu o modalitate rezonabil de evidentă de a executa acest lucru.
- Programatorii ar trebui să fie liberi să își aleagă propriul stil de programare, iar acest stil ar trebui să fie pe deplin susținut de C++.
- Acceptarea unei particularități utile este mult mai importantă decât prevenirea oricărei posibile utilizări greșite a C++.

- C++ ar trebui să ofere facilități pentru organizarea programelor în fragmente de sine stătătoare, bine definite, și să ofere facilități pentru combinarea părților dezvoltate separat.
- Să nu accepte încălcări implicite ale sistemului de tipuri, dar să accepte încălcări explicite, adică cele cerute în mod explicit de către programator.
- Tipurile create de utilizator trebuie să aibă același suport și aceleași performanțe ca și tipurile încorporate.
- Caracteristicile neutilizate nu trebuie să aibă un impact negativ asupra executabilelor create (de exemplu, prin scăderea performanțelor).
- Nu ar trebui să se găsească un limbaj sub C++, cu excepția limbajului de asamblare.
- C++ ar trebui să îndeplinească funcții împreună cu alte limbaje de programare existente, în loc să promoveze mediul de programare propriu incompatibil și separat.
- În cazul în care intenția programatorului este necunoscută, trebuie să i se permită acestuia să o precizeze prin asigurarea controlului manual.

2.1.3 Librăria standard

Standardul C++ este format din doar două părți: limbajul de bază împreună cu biblioteca standard a limbajului. Programatorii C++ se așteaptă ca biblioteca standard să se regăsească în implementarea majoră a limbajului C++; ea include tipuri de agregate (vectori, matrici liste, hărți, seturi, cozi, stive, tuple), algoritmi (random_shuffle, find, for_each, binary_search), facilități de intrare/ieșire (iostream - pentru citirea și scrierea de la consolă și fișiere), bibliotecă de sistem de fișiere, pointeri inteligenți folosiți la gestionarea automată a memoriei, bibliotecă multi-threading, suport pentru atomics, utilități de timp (măsurare, obținerea orei curente etc.), un sistem pentru conversia raportării erorilor care nu utilizează excepții C++, generator pentru numere aleatoare și o versiune puțin schimbată a bibliotecii standard din C (o face să fie conformă cu sistemul de tipuri existent în C++).

O semnificativă parte a bibliotecii C++ se bazează pe Standard Template Library (STL). Instrumentele utile oferite de STL includ containere drept colecții de obiecte precum

liste și vectori, iteratori ce oferă acces la array-uri și algoritmi pentru operații cum ar fi sortarea și căutarea.

Majoritatea compilatoarelor C++ oferă o implementare ce se potrivește cu standardele bibliotecii standard C++.

2.1.4 Orientări de bază C++

Orientările de bază pentru C++ sunt o inițiativă condusă de Bjarne Stroustrup, inventatorul C++ și Herb Sutter, convocatorul și președintele Grupului de lucru ISO pentru C++, pentru a ajuta programatorii să scrie "C++ modern" prin utilizarea celor mai bune practici pentru standardele de limbaj C++11 și mai noi, precum și pentru a ajuta dezvoltatorii de compilatoare și instrumente de verificare statică să creeze reguli pentru detectarea practicilor de programare greșite.

Scopul principal este de a scrie în mod eficient și consecvent un cod C++ sigur din punct de vedere al tipurilor și resurselor. Orientările de bază au fost anunțate în cadrul discursului de deschidere la CPPCon 2015.

Liniile directoare sunt însoțite de Guideline Support Library (GSL), o bibliotecă de tipuri și funcții numai de antet pentru implementarea liniilor directoare de bază și a instrumentelor de verificare statică pentru aplicarea regulilor.

2.1.5 Compatibilitate

Pentru a oferi o mai mare libertate furnizorilor de compilatoare, comitetul de standardizare C++ a decis să nu dicteze punerea în aplicare a manipulării numelor, a gestionării excepțiilor și a altor proprietăți specifice implementării. Unul din dezavantajele acestei decizii este că se preconizează că codul obiect produs de compilatoare diferite va fi incompatibil. Cu toate acestea, au mai fost încercări pentru standardizarea compilatoarelor pentru anumite tehnologii sau sisteme de operare deși acestea par să fie abandonate în mare parte în prezent.

C++ este deseori considerat a fi un supraansamblu al C, dar acest lucru nu este strict adevărat. Majoritatea codurilor C pot fi ușor făcute să fie compilate corect în C++, dar există câteva diferențe care fac ca unele coduri C valide să fie invalide sau să se comporte diferit în C++. De exemplu, C permite conversia implicită de la *void** la alte tipuri de pointer, dar C++ nu permite acest lucru (din motive de siguranță a tipurilor).

De asemenea, C++ definește multe cuvinte cheie noi, cum ar fi *new* și *class*, care pot fi utilizate ca identificatori (de exemplu, nume de variabile) într-un program C.

Unele incompatibilități au fost eliminate prin revizuirea din 1999 a standardului C (C99), care acceptă acum caracteristicile C++, cum ar fi comentariile de linie (//) și declarațiile amestecate cu codul. Pe de altă parte, C99 a introdus o serie de caracteristici noi pe care C++ nu le suporta și care erau incompatibile sau redundante în C++, cum ar fi array-urile cu lungime variabilă, tipurile native de numere complexe (totuși, clasa *std::complex* din biblioteca standard C++ oferă o funcționalitate similară, deși nu este compatibilă cu codul), inițializatori desemnați, literali compuși și cuvântul cheie *restrict*. Unele dintre caracteristicile introduse de C99 au fost incluse în versiunea ulterioară a standardului C++, C++11 (dintre cele care nu erau redundante). Cu toate acestea, standardul C++11 introduce noi incompatibilități, cum ar fi nepermiterea asignării unui literal de șir de caractere la un pointer de caractere, care rămâne valabil în C.

Pentru a amesteca codul C și C++, orice declarație sau definiție de funcție care urmează să fie apelată din/folosită atât în C, cât și în C++, trebuie declarată cu legătură C prin plasarea ei într-un bloc *extern "C" {/*...*/}*. O astfel de funcție nu se poate baza pe caracteristici care depind de manipularea numelor (adică supraîncărcarea funcțiilor).

2.1.6 Limbaj

Limbajul C++ are două caracteristici (componente) principale: o mapare cât mai directă a proprietăților hardware ce sunt furnizate de subsetul limbajului C și abstracțiuni fără costuri suplimentare întemeiate pe aceste mapări. Limbajul C++ este descris de Stroustrup ca fiind "un limbaj de programare ușor de abstractizat [conceput] pentru construirea și utilizarea abstracțiilor eficiente și elegante"; și "oferirea atât a accesului la hardware, cât și a abstractizării reprezintă baza limbajului C++. A face acest lucru în mod eficient este ceea ce îl distinge de alte limbaje".

C++ moștenește cea mai mare parte a sintaxei C. În următoarea figură (Figura 2.1) este versiunea lui Bjarne Stroustrup a programului "Hello world", care folosește facilitatea stream a bibliotecii standard C++ pentru a scrie un mesaj pe ieșirea standard:

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
    std::cout << "Pascal Marian\n";
}
```

Figura 2.1 Hello world în C++

C++ suportă, ca și în C, patru moduri pentru gestionarea memoriei: obiecte cu durată de stocare statică, a firelor de execuție, automată și dinamică.

Obiecte cu durată cu stocare statică

Obiectele cu durată de stocare statică sunt generate înainte de intrarea în *main()* și sunt șterse în ordinea inversă a generării, după ce *main()* se încheie. Ordinea generării nu este specificată de standarde pentru a permite o oarecare libertate în ceea ce privește tipul de organizare a implementărilor. În mod mai formal, acest tip de obiectele au o durată de viață ce "trebuie să dureze pe toată durata programului".

Obiecte cu durată de stocare a firelor de execuție

Acest tip de variabile sunt asemeni cu obiectele cu durată de stocare statică. O diferență principală constă în faptul că timpul de creare se efectuează înaintea creării firului de execuție, distrugerea facându-se după ce firul de execuție a fost unit.

Obiecte cu durată de stocare automată

Tipurile de variabile cele mai comune în C++: variabilele locale din interiorul unei funcții și variabilele temporare. Caracteristica variabilelor automate este reprezentată de o durată de viață limitată la domeniul de aplicare al variabilei. Variabilele sunt create și potențial inițializate în momentul declarării și sunt șterse în ordinea inversă a creării atunci când se părăsește domeniul de aplicare. Acest lucru este implementat prin alocare pe stivă.

Variabilele locale sunt create pe măsură ce punctul de executare trece de punctul de declarare. În cazul în care variabila deține un inițializator sau un constructor, acesta este folosit pentru a determina starea de inițializare a obiectului creat. Variabilele locale sunt șterse în acel moment când funcția în care au fost declarate se termină. La sfârșitul duratei de

viață a obiectului sunt apelați destructori C++ pentru variabilele locale, acceptând o disciplină de gestionare automată.

Variabilele membre sunt create în momentul creării obiectului părinte. Membrii tabloului sunt inițializați în ordine de la 0 până la ultimul membru al tabloului. Atunci când obiectul părinte este distrus, variabilele membre sunt șterse în ordinea inversă creării. De exemplu, când obiectul părinte este un "obiect automat", acesta va fi șters atunci când se termină domeniul său de aplicare, ceea ce pornește ștergerea totală a membrilor săi.

Obiecte cu durată de stocare dinamică

Aceste obiecte au o durată de viață dinamică și pot fi generate direct folosind *new* și distruse cu *delete*. C++ suportă, de asemenea, *malloc* și *free,* din C, dar acestea nu sunt compatibile cu *new* și *delete*. Utilizarea lui new returnează o adresă către memoria alocată. Orientările de bază din C++ recomandă să nu se utilizeze direct *new* pentru crearea de obiecte dinamice, ci să se folosească pointeri inteligenți prin *make_unique<T>* pentru proprietatea unică și *make_shared<T>* pentru proprietatea multiplă cu referințe.

Obiecte

C++ aduce caracteristici de programare orientată pe obiecte în C. Acesta oferă conceptul de clase cu caracteristici prezente în mod obișnuit în limbajele OOP și în câteva limbaje non-OOP: abstractizare, polimorfism, moștenire, încapsulare,.

O trăsătură distinctă a claselor C++ este suportul pentru destructori deterministici ceea ce alte limbaje nu oferă.

Încapsulare

Încapsularea reprezintă “ascunderea” unor informațiilor pentru asigurarea că structuri de date și operatori sunt folosiți conform utilizării dorite de dezvoltator. C++ oferă posibilitatea de definirea claselor și funcțiilor ca mecanisme principale pentru încapsulare. Membrii din cadrul unei clase pot fi declarați ca fiind publici, protejați sau privați, pentru impunerea explicită a încapsulărării. Membrii publici al unei clasei poate fi accesat de către orice funcție din program. Membrii privați sunt accesibili numai funcțiilor membre ale clasei respective și funcțiilor și claselor cărora clasa le-a acordat explicit acces de („prieteni”). Membrii protejați sunt accesibili membrilor claselor care moștenesc clasa de bază, pe lângă clasa însăși și orice prieteni.

Moștenirea

Moștenirea permite unui tip de date să dobândească proprietăți ale altor tipuri de date. În funcție de clasa de bază, moștenirea poate fi privată, publică sau protejată. Specificatorul acesta de acces deduce dacă clasele neafiliate și clasele derivate ar putea să acceseze membrii moșteniți ai clasei de bază, fie publici sau protejați. Doar moștenirea publică reprezintă ceea ce se înțelege prin „moștenire”. Formele private și protejate de moștenire sunt mult mai puțin utilizate.

În cazul în care specificatorul de acces lipsește, o „clasă” moștenește direct modul privat, iar o „structură” moștenește modul public. Moștenirea virtuală înseamnă că clasele de bază pot fi declarate ca fiind „virtuale”. Moștenirea virtuală este modul de asigurare de existența unei singure instanțe a unei clase de bază în graficul de moștenire, astfel se evită probleme de ambiguitate întâlnite la moștenirea multiplă.

Operatori și supraîncărcarea operatorilor

C++ oferă peste 35 de operatori, care acoperă aritmetica de bază, manipularea de biți, indirecția, comparațiile, operațiile logice și altele. Marea majoritate a operatorilor pot fi supraîncărcați pentru tipuri definite de programator. Operatorii care nu pot fi supraîncărcați sunt accesul la membri (. și .*) și operatorul condițional. Acest set mare de operatori supraîncărcabili este important pentru a face ca tipurile definite de programator în C++ să se asemene cu tipurile integrate.

Operatorii supraîncărcabili sunt o parte foarte importantă a tehnicilor avansate de programare în C++ precum pointerii inteligenți. Supraîncărcarea unui operator nu schimbă numărul de operanzi pe care operatorul îi utilizează și nici precedența calculelor ce implică operatorul. Operatorii "&&" și "||" supraîncărcați își pierd caracteristica de evaluare în scurtcircuit.

Polimorfism

Polimorfismul oferă o interfață comună pentru mai multe tipuri de implementări și pentru ca acțiunea obiectelor create să fie diferită în funcție de circumstanțe.

C++ acceptă tipuri de polimorfisme statice (care sunt rezolvate la compilare) și dinamice (care sunt rezolvate la execuție). La compilare, polimorfismul nu acceptă luarea anumitor decizii în timpul execuției, în timp ce la execuție, polimorfismul implică, de obicei, o penalizare semnificativă de performanță.

Supraîncărcarea funcțiilor permite programatorului declararea a mai multor funcții ce au același nume, însă argumente distincte (polimorfism ad-hoc). Distingerea funcțiilor se face prin tipurile sau numărul de parametri conținuți. În funcție de contextul de utilizare, același nume de funcție poate face referire la funcții distincte.

Șabloane

Șabloanele în C++ acceptă un mecanism de scriere a codului polimorfic generic, adică polimorfism parametric. Prin intermediul modelului de șablon recurent, este posibilă implementarea unei forme de polimorfism static care imită îndeaproape sintaxa de suprascriere a funcțiilor virtuale.

Referințele la un tip de clasă de bază în C++ și indicatorii de variabile se pot referi la obiecte din orice clase derivate ale acelui tip. Asta permite ca array-urile și alte tipuri de containere să păstreze pointeri la obiecte de tipuri diferite. Acest lucru permite polimorfismul dinamic, unde obiectele la care se face referire se pot comporta diferit, în funcție de tipurile lor, fie reale fie derivate.

Funcții membre virtuale

Atunci când o funcție dintr-o clasă derivată suprascrie o funcție dintr-o clasă de bază, tipul obiectului determină funcția care trebuie apelată. O anumită funcție este suprascrisă în momentul când nu există nicio distincție în ceea ce privește tipul sau numărul de parametri între definiții ale acelei funcții. În momentul compilării, este posibil să nu fie posibilă determinarea tipului obiectului și, deci, funcția corectă care trebuie apelată, având în vedere doar un pointer la clasa de bază; decizia este, prin urmare, amânată până la momentul execuției. Acest lucru se numește "dispecerizare dinamică".

Funcțiile sau metodele membre virtuale permit apelarea celei mai specifice implementări a funcției, în funcție de tipul real al obiectului în momentul execuției. În implementările C++, acest lucru se realizează în mod obișnuit cu ajutorul tabelelor de funcții virtuale.

Tratarea excepțiilor

Gestionarea excepțiilor este utilizată pentru a comunica existența unei probleme sau erori de execuție de la locul în care a fost detectată până la locul în care problema poate fi tratată. Aceasta permite ca acest lucru să se facă într-un mod uniform și separat de codul principal, detectând în același timp toate erorile.

În cazul în care apare o eroare, se aruncă (se ridică) o excepție, care este apoi captată de cel mai apropiat gestionar de excepții adecvat. Excepția determină ieșirea din domeniul de aplicare curent și, de asemenea, din fiecare domeniu de aplicare exterior (propagare) până când se găsește un gestionar adecvat, apelând la rândul său destructori ai oricăror obiecte din aceste domenii de aplicare ieșite. În același timp, o excepție este prezentată sub forma unui obiect care conține date despre problema detectată (Figura 2.2).

```cpp
#include <iostream>
#include <vector>
#include <stdexcept>

int main() {
    try {
        std::vector<int> vec{3, 4, 3, 3, 1};
        int i{vec.at(4)}; // Aruncă o excepție, std::out_of_range (indexarea pentru vec este de la 0-3 nu de la 1-4)
    }
    // Un gestionar de excepții, prinde std::out_of_range, care este aruncat de vec.at(4)
    catch (std::out_of_range &e) {
        std::cerr << "Accesarea unui element inexistent: " << e.what() << '\n';
    }
    // Pentru a prinde orice alte excepții din biblioteca standard (acestea derivă din std::exception)
    catch (std::exception &e) {
        std::cerr << "Excepție aruncată: " << e.what() << '\n';
    }
    // Prindeți orice excepție nerecunoscută (adică cele care nu derivă din std::exception)
    catch (....) {
        std::cerr << "Eroare fatală!\n";
    }
}
```

Figura 2.2 Tratarea excepțiilor în C++

De asemenea, este posibil să se arunce excepții în mod intenționat, folosind cuvântul cheie *throw*; aceste excepții sunt tratate în mod obișnuit. În unele cazuri, excepțiile nu pot fi utilizate din motive tehnice. Un astfel de exemplu este o componentă critică a unui sistem încorporat, în care fiecare operațiune trebuie să fie garantată ca fiind finalizată într-un anumit interval de timp. Acest lucru nu poate fi determinat cu ajutorul excepțiilor, deoarece nu există instrumente care să determine timpul maxim necesar pentru tratarea unei excepții. Spre deosebire de tratarea semnalelor, în care funcția de tratare este apelată din punctul de eșec, tratarea excepțiilor iese din domeniul curent înainte de intrarea în blocul *catch*, care poate fi localizat în funcția curentă sau în oricare dintre apelurile de funcție anterioare aflate în prezent pe stivă. [1]

2.2 TouchGFX

2.2.1 Descriere

TouchGFX reprezintă un cadru software grafic avansat, optimizat pentru microcontrolerele STM32. Fiind avantajat de arhitectură și caracteristicile grafice ale STM32, TouchGFX accelerează revoluția tehnologiei HMI-of-things prin crearea de interfețe grafice de utilizator extraordinare, fiind asemănătoare cu cele de pe un smartphone.

Cadrul TouchGFX conține TouchGFX Designer, un instrument PC de construcție grafică simplu pentru utilizare, fiind bazat pe motorul TouchGFX, avand un nucleu grafic optimizat și puternic. TouchGFX ușurează dezvoltarea de interfețe grafice, îmbinând generarea automată de cod și simulatorul WYSIWYG. Acesta înglobează toate etapele prin iterații rapide peste prototipuri finalizate, de la schițele simple de proiectare până la produse finale exclusive.

TouchGFX Designer este utilizabil drept instrument software independent și permite o evaluare grafică rapidă și ușoară având o dovadă de concept. Cadrul TouchGFX este distribuit în pachetele STM32Cube MCU cu tot cu TouchGFX Designer. Acesta este perfect compatibil cu instrumentul de generare de cod și inițializare STM32CubeMX pentru dezvoltarea fără nicio problemă a aplicației grafice într-un mediu de proiect unicat. TouchGFX este alcătuit din trei părți principale - două instrumente și un cadru.

- TouchGFX Designer: Un constructor de interfață grafică ușor de utilizat în TouchGFX care permite crearea aspectului vizual al aplicației TouchGFX.
- TouchGFX Generator: Un plugin STM32CubeMX în care utilizatorul poate configura și genera un strat de abstractizare (AL) TouchGFX personalizat pentru hardware-ul său bazat pe STM32.
- TouchGFX Engine: Cadrul TouchGFX C++ care conduce aplicația UI. Gestionează actualizările ecranului, evenimentele utilizatorului și sincronizarea.

Tehnologia avansată TouchGFX este optimizată pentru microcontrolerele STM32, oferind o performanță maximă cu o încărcare minimă a procesorului și o utilizare minimă a memoriei. [2]

2.2.2 Caracteristici

Structură:

- Creare simplă a mai multor ferestre și a tranzițiilor asociate.

Widget-uri:

- Gamă largă de widget-uri personalizabile, cum sunt containerele de glisare sau progresul ciclului, pentru crearea fără efort ale interfețelor grafice.

Skin-uri:

- Gamă de skin-uri grafice ușor de utilizat ce permit crearea de prototipuri constante fără a fi nevoie de un designer grafic.
- Nicio restricție în utilizarea de grafică personalizată.

Interacțiuni:

- Interacțiuni dinamice pentru crearea de aplicații ușor de utilizat.

Container personalizat:

- Crearea unui control personalizat reutilizabil al aplicației.
- Dezvoltare ușoară a platformei cu aspect și aspect unificat.

Manipulare text:

- Fonturi și tipografii specificate și gestionate simplu.
- Serviciu complet și ușor utilizabil pentru traducere.
- Suport complet pentru diferite alfabete și scripturi, cum ar fi alfabetul latin, chirilic, arab, chineză și japoneză.

Generarea de coduri:

- TouchGFXDesigner menține și generează un cod C++ foarte performant.
- Cod generat de instrument separat în întregime de codul utilizatorului.

- Extensiile de cod de tipuri multiple sunt posibile pentru animații unice și pentru interconexiuni de sistem.
- Suport pentru medii de dezvoltare integrate, cum sunt IAR Embedded Workbench, Arm Keil și IDE-uri bazate pe compilatorul GCC.

2.2.3 Grafică încorporată

Cuvântul încorporat înseamnă lucruri diferite pentru persoane diferite. Pentru unii, un sistem încorporat înseamnă un microcontroler vechi de 8 biți foarte fiabil, fără sistem de operare și practic fără RAM, ROM sau GPIO. Pentru alții ar putea însemna un telefon inteligent din zilele noastre, cu gigabytes de tot.

Pentru TouchGFX, sistemele integrate înseamnă orice sistem bazat pe un microcontroler STM32, iar grafica înseamnă aplicații interactive cu o interfață utilizator care rulează la 60 de cadre pe secundă.

Pentru a realiza aplicații grafice pe astfel de platforme, avem în vedere patru componente principale direct implicate. Desigur, într-un sistem încorporat pot fi prezente mult mai multe componente, dar acestea sunt mai puțin legate de afișarea graficii (Figura 2.3).

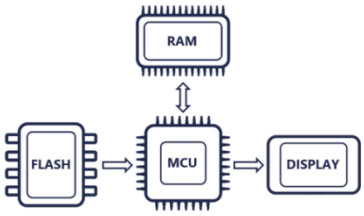


Figura 2.3 Componente principale

Pe scurt, TouchGFX, folosind MCU (Microcontroller Unit), creează și actualizează o imagine în RAM, asamblând părți din flash. Imaginea asamblată este transferată pe ecran. Acest proces se repetă ori de câte ori este posibil și necesar. MCU-ul face toată munca grea în acest proces. Acesta citește imaginile din memoria flash și le scrie în memoria RAM. Calculează culorile rezultate la îmbinarea unui text roșu semitransparent cu o imagine și le

salvează în RAM. Redă și stochează toți pixelii unui cerc în RAM. Transferă imaginea din RAM pe ecran. Memoria RAM este locul în care este stocată imaginea calculată rezultată. Memoria RAM este citită și scrisă de MCU atunci când actualizează grafica și este citită din nou atunci când imaginea rezultată este transferată pe ecran.În multe cazuri, imaginea rezultată este stocată în memoria RAM internă a MCU. În cazurile în care nu este fezabil ca imaginea rezultată să fie stocată în memoria RAM internă, se poate adăuga la configurație o memorie RAM externă.

Memoria Flash este locul în care sunt plasate toate datele statice. Imagini, fonturi și texte. Flash-ul este citit de MCU, iar conținutul este scris sau combinat cu memoria RAM. De cele mai multe ori se adaugă o memorie flash externă la configurație, deoarece memoria flash internă este rareori suficient de mare pentru a conține toate resursele grafice. Pentru aplicații foarte simple, flash-ul intern este suficient. Ideal ar fi ca memoria flash să fie mapată, astfel încât pixelii să poată fi citiți direct din flash și scriși în RAM. În caz contrar, atunci când memoria flash nu este mapată, conținutul acesteia poate fi transferat în memoria RAM și apoi citit de acolo. Afișajul este locul în care imaginea este afișată efectiv în fața ochilor unei persoane. Imaginea calculată (framebuffer) stocată în memoria RAM este trimisă de MCU către afișaj la intervale regulate.

Un framebuffer este o bucată de memorie care este actualizată de către motorul grafic pentru a conține următoarea imagine care urmează să fie afișată pe ecran. Framebuffer-ul este o parte contiguă de memorie RAM de o anumită dimensiune. Un framebuffer are o lățime și o înălțime asociate. Prin urmare, ne gândim de obicei la un framebuffer ca la o parte bidimensională a memoriei, indexabilă prin coordonatele x, y. Conținutul framebuffer-ului este ceea ce este transferat în cele din urmă către și văzut pe ecranul fizic. Prin urmare, este foarte frecvent ca lățimea și înălțimea pixelilor din framebuffer și de pe afișaj să fie identice.

2.2.4 Motorul grafic

Principala responsabilitate a motorului grafic TouchGFX este de a desena grafică pe ecranul unui dispozitiv încorporat. Această secțiune va oferi o imagine de ansamblu asupra tipului de motor grafic TouchGFX și va oferi câteva informații de fond despre motivul pentru care este așa. Motoarele grafice pot fi împărțite în două categorii principale.

Motoarele grafice în mod imediat oferă un API care permite unei aplicații să deseneze direct lucruri pe afișaj. Este responsabilitatea aplicației să se asigure că operațiile de desenare corecte sunt invocate la momentul potrivit.

Motoarele grafice în mod reținut permit utilizatorului să manipuleze un model abstract al componentelor afișate. Motorul se ocupă de traducerea acestui model de componente în operațiile corecte de desenare grafică la momentul potrivit.

TouchGFX respectă principiile grafice de tip *mod reținut*. Pe scurt, acest lucru înseamnă că TouchGFX oferă un model care poate fi manipulat de către utilizator, iar TouchGFX se ocupă apoi de traducerea acestui model într-un set optimizat de apeluri de metode de redare. Beneficiile faptului că TouchGFX este de tip *mod reținut* sunt multe. Cele principale sunt:

- Ușurința de utilizare: Un motor grafic de tip *retained* este ușor de utilizat. Utilizatorul se adresează configurației componentelor de pe ecran prin invocarea metodelor modelului intern și nu se gândește la operațiile de desenare propriu-zise.

- Performanță: TouchGFX analizează modelul scenei și optimizează apelurile de desen necesare pentru a realiza modelul pe ecran. Aceasta include faptul că nu desenează în mod deliberat componentele ascunse, desenând și transferând numai părțile modificate ale componentelor, gestionând framebufferii și multe altele.

- Gestionarea stării: TouchGFX ține evidența părții din modelul scenei care este activă. Acest lucru, la rândul său, facilitează utilizatorului optimizarea conținutului modelului de scenă.

Principalul dezavantaj al aderării TouchGFX la schema grafică cu *mod reținut* este:

- Consumul de memorie: Reprezentarea modelului de scenă ocupă o anumită cantitate de memorie. TouchGFX atinge nivelurile sale de performanță, de obicei redarea a 60 de cadre pe secundă, prin analiza modelului scenei și optimizarea redării corespunzătoare efectuate. S-au depus eforturi mari pentru a reduce cantitatea de memorie utilizată de modelul de scenă al TouchGFX. În aplicațiile tipice, cantitatea de memorie pentru acest model este cu mult sub un kilooctet. [3]

2.3 CubeMX

2.3.1 Prezentare generală

STM32Cube este o decizie originală a STMicroelectronics pentru a face viața dezvoltatorilor mai ușoară prin reducerea timpului, efortului și costurilor de dezvoltare. STM32Cube conține întregul portofoliu de dispozitive STM32, bazate pe nuclee Arm Cortex pe 32 de biți. STM32CubeMX este un instrument grafic de configurare software ce permite generarea de coduri C/C++, cod de inițializare cu ajutorul unor asistenți grafici.

O platformă software încorporată cuprinzătoare, livrată per serie (cum ar fi STM32CubeF2 pentru seria STM32F2 și STM32CubeF4 pentru seria STM32F4)

- STM32Cube HAL, software-ul încorporat al stratului de abstractizare STM32 care asigură o portabilitate maximă în întregul portofoliu STM32.
- API-uri de nivel scăzut (LL) care oferă un strat rapid și ușor orientat către experți, care este mai apropiat de hardware decât HAL. API-urile LL sunt disponibile numai pentru un set de periferice.
- Un set consistent de compact middleware, cum sunt USB, RTOS, TCP/IP.
- Toate utilitățile software încorporate, livrate cu un set complet de exemple.

2.3.2 Principii

Clienții trebuie să identifice rapid MCU-ul care îndeplinește cel mai bine cerințele lor (nucleu arhitectura nucleului, caracteristicile, dimensiunea memoriei, performanța...). În timp ce principalele preocupări ale proiectanților de plăci sunt de a optimiza configurația pinilor microcontrolerului pentru configurația plăcii lor și de a îndeplini cerințele aplicației (alegerea modurilor de funcționare a perifericelor), dezvoltatorii de sisteme integrate sunt mai interesați de dezvoltarea de noi aplicații pentru un anumit dispozitiv țintă și să migreze proiectele existente către microcontrolere diferite.

Timpul necesar pentru a migra la noi platforme și pentru a actualiza codul C la noile drivere de firmware adaugă întârzieri inutile la proiect. STM32CubeMX a fost dezvoltat în

cadrul STM32Cube al cărui scop este de a satisface cerințele cheie ale clienților pentru a maximiza reutilizarea software-ului și a minimiza timpul de creare a sistemului țintă:

- Reutilizarea software-ului și portabilitatea proiectării aplicațiilor sunt realizate prin intermediul soluției de firmware STM32Cube care propune un API comun pentru nivelul de abstractizare hardware în întregul portofoliu STM32.
- Timpul de migrare optimizat se realizează datorită cunoștințelor încorporate în STM32CubeMX privind microcontrolerele STM32, perifericele și middleware-ul (stive de protocoale de comunicare LwIP și USB, sistemul de fișiere FatFs pentru sisteme integrate mici, FreeRTOS).

Interfața grafică STM32CubeMX îndeplinește următoarele funcții:

- Configurarea rapidă și ușoară a pinilor MCU, a ceasurilor și a modurilor de funcționare pentru perifericele și middleware selectate.
- Generarea unui raport de configurare a pinilor pentru proiectanții de plăci.
- Generarea unui proiect complet cu toate bibliotecile și codul C de inițializare necesare pentru a configura dispozitivul în modul de operare definit de utilizator. Proiectul poate fi deschis direct în mediul de dezvoltare a aplicațiilor selectat (pentru o selecție de IDE-uri acceptate) pentru a continua dezvoltarea aplicației.

În timpul procesului de configurare, STM32CubeMX detectează conflictele și setările invalide și le evidențiază prin pictograme semnificative și sfaturi utile pentru instrumente. [4]

2.3.3 Caracteristici

STM32CubeMX vine cu următoarele caracteristici:

Managementul proiectelor

STM32CubeMX permite utilizatorului să creeze, să salveze și să încarce proiecte salvate anterior:

- Atunci când STM32CubeMX este lansat, utilizatorul poate alege să creeze un proiect nou sau să să încarce un proiect salvat anterior.

- Salvarea proiectului salvează setările utilizatorului și configurația efectuată în cadrul proiectului proiect într-un fișier .ioc care va fi utilizat atunci când proiectul va fi încărcat în STM32CubeMX din nou.

STM32CubeMX permite, de asemenea, utilizatorului să importe proiecte salvate anterior în proiecte noi.

Proiectele STM32CubeMX vin în două variante:

- Numai configurația MCU: fișierul .ioc este salvat într-un dosar de proiect dedicat.
- Configurație MCU cu generare de cod C: în acest caz, fișierele .ioc sunt salvate într-un dosar de proiect dedicat, împreună cu codul sursă C generat. Poate exista doar un singur fișier .ioc pe proiect.

Crearea ușoară a unui proiect pornind de la un MCU, o placă sau un exemplu

Fereastra de proiect nou permite utilizatorului să creeze un proiect prin selectarea unui microcontroler, o placă sau un exemplu de proiect din portofoliul STM32. Sunt disponibile diferite opțiuni de filtrare pentru a ușura selectarea MCU și a plăcii.

Configurare ușoară a pinout-ului

- Din vizualizarea Pinout, utilizatorul poate selecta perifericele dintr-o listă și poate configura modurile de periferice necesare pentru aplicație. STM32CubeMX atribuie și configurează pinii în mod corespunzător.
- Pentru utilizatorii mai avansați, este posibilă și maparea directă a unei funcții periferice pe un pin fizic, utilizând vizualizarea Pinout. Semnalele pot fi blocate pe pini pentru a împiedica rezolvarea conflictelor de către STM32CubeMX să mute semnalul pe un alt pin.
- Configurația Pinout poate fi exportată ca fișier .csv.

Generarea completă a proiectului

Generarea proiectului include pinout, firmware și cod C de inițializare a middleware-ului pentru un set de IDE-uri. Se bazează pe bibliotecile de software încorporate STM32Cube. Se pot efectua următoarele acțiuni:

- Pornind de la pinout-ul definit anterior, utilizatorul poate continua cu configurarea middleware-ului, a ceasurilor, a serviciilor (RNG, CRC, etc...) și parametrii periferici.

- STM32CubeMX generează inițializarea corespunzătoare în Cod C. Rezultatul este un director de proiect care include fișierul main.c generat și fișierul C pentru configurare și inițializare, plus o copie a fișierelor de antet necesare HAL (Nivelui Abstract Hardware) și bibliotecile middleware necesare, precum și fișiere specifice pentru IDE-ul selectat.

- STM32CubeMX poate genera fișiere de utilizator prin utilizarea de fișiere freemarker .ftl definite de utilizator.

- Din meniul Project settings (Setări proiect), utilizatorul poate selecta lanțul de instrumente de dezvoltare (IDE) pentru care trebuie generat codul C. STM32CubeMX se asigură că fișierele de proiect relevante pentru IDE sunt adăugate în dosarul de proiect, astfel încât proiectul să poată fi importat direct ca proiect nou în cadrul unui IDE terț (IAR™ EWARM, Keil™ MDK-ARM).

Calculul consumului de energie

Pornind de la selectarea unui număr de referință al microcontrolerului și a unui tip de baterie, utilizatorul poate defini o secvență de etape reprezentând ciclul de viață al aplicației și parametrii (alegerea frecvențelor, perifericele activate, durata etapelor). Puterea STM32CubeMX Consumption Calculator returnează consumul de energie și durata de viață a bateriei corespunzătoare estimări. [5]

3. COMPONENTE UTILZATE PENTRU REALIZAREA PROIECTULUI

3.1 STM32F429I-DISC1

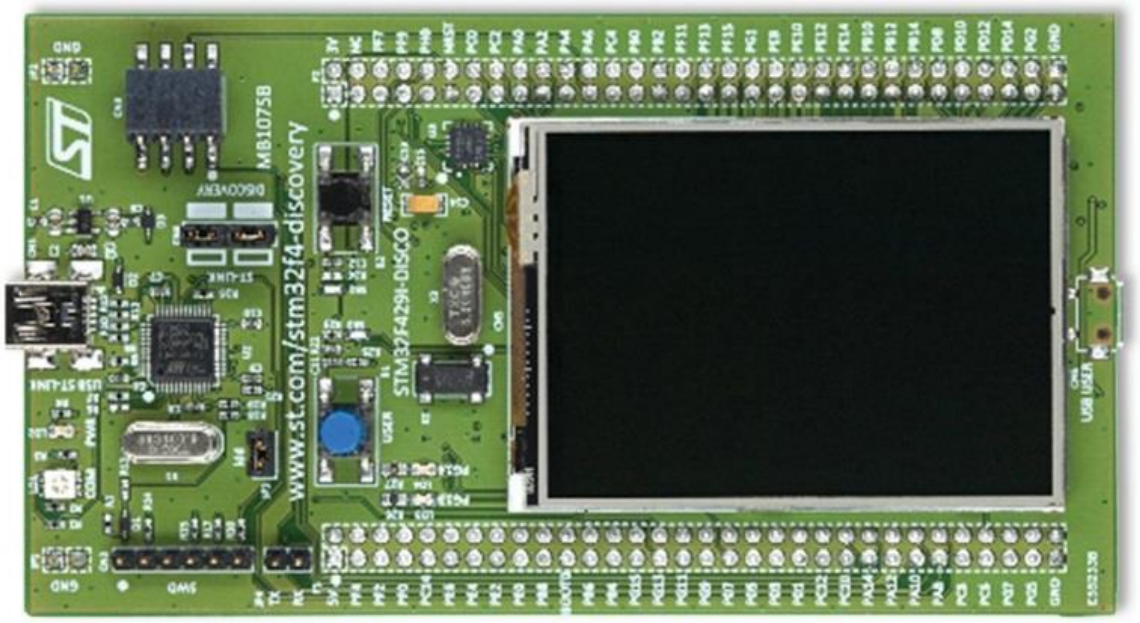


Figura 3.1 Placa de dezvoltare STM32F429I

3.1.1 Introducere

Kitul 32F429IDISCOVERY permite utilizatorilor să dezvolte cu ușurință aplicații cu STM32F429 de înaltă performanță bazat pe nucleul Arm® Cortex®-M4. Acesta include un instrument de depanare încorporat ST-LINK/V2-B, un LCD TFT QVGA de 2,4", o memorie SDRAM externă de 64 Mbit, un giroscop ST MEMS, un conector USB OTG Micro-AB, LED-uri și butoane. Schema plăcii poate fi vizualizată în Figura 3.2.

3.1.2 Caracteristici

- Microcontroler STM32F429I cu 2 Mbytes de memorie Flash, și 256 Kbytes de RAM într-un pachet LQFP144
- LCD TFT QVGA DE 2,4" QVGA
- USB OTG cu conector Micro-AB

- I3G4250D, senzor de mișcare MEMS ST cu senzor de mișcare cu ieșire digitală pe 3 axe (giroscop)
- Șase LED-uri:
  - LD1 (roșu/verde) pentru comunicația USB
  - LD2 (roșu) pentru pornirea la 3,3 V
  - Două LED-uri de utilizator: LD3 (verde), LD4 (roșu)
  - Două LED-uri USB OTG: LD5 (verde) VBUS și LD6 (roșu) OC (supracurent)
- Două butoane (utilizator și resetare)
- 64-Mbit SDRAM
- Antet de extensie pentru I/O-uri LQFP144 pentru o conexiune rapidă la placa de prototipuri și o probare ușoară
- ST-LINK/V2-B încorporat
- Funcții USB:
  - Port de depanare
  - Port COM virtual
  - Memorie de masă
- Alimentarea cu energie a plăcii: prin intermediul magistralei USB sau de la o tensiune de alimentare externă de 5 V
- Software gratuit cuprinzător, inclusiv o varietate de exemple, parte a STM32CubeF4 MCU Package sau STSW-STM32138, pentru utilizarea bibliotecilor standard moștenite

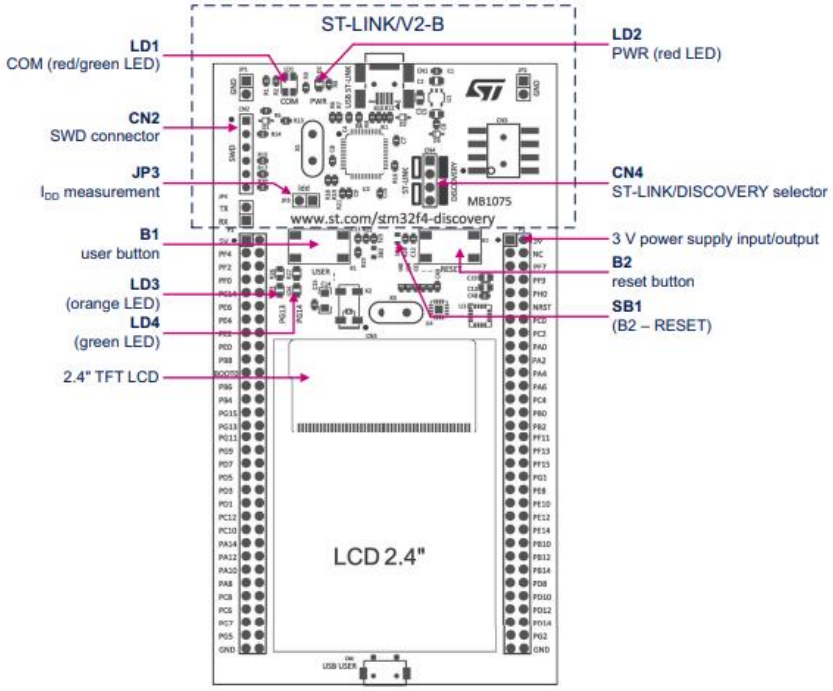


Figura 3.2 TOP Layer-ul plăcii [6]

3.1.3 Alimentarea și selectarea puterii

Alimentarea cu energie electrică este asigurată fie de PC-ul gazdă prin intermediul cablului USB, fie de o sursă de alimentare externă de 5 V. Diodele D2 și D3 protejează pinii de 5 V și 3 V de sursele de alimentare externe:

- 5 V și 3 V pot fi utilizate ca surse de alimentare de ieșire atunci când o altă placă de aplicații este conectată la pinii P1 și P2.

În acest caz, pinii de 5 V și 3 V livrează o sursă de alimentare de 5 V sau 3 V, iar consumul de energie al pinilor de alimentare trebuie să fie mai mic de 100 mA.

- 5 V și 3 V care pot fi utilizate drept surse de alimentare de intrare, de exemplu, atunci când conectorii USB nu sunt conectați la PC.

În acest caz, placa STM32F429 Discovery trebuie să fie alimentată de o sursă de alimentare sau de un echipament auxiliar care trebuie să fie Safety Extra Low Voltage (SELV) cu capacitate de alimentare limitată.

3.1.4 Memorie

SDRAM-ul de 64 de biți este o memorie CMOS de mare viteză, cu acces aleatoriu dinamic, proiectat pentru a funcționa în sisteme de memorie de 3,3 V care conțin 67 108 864 biți. Aceasta este configurat intern ca o memorie DRAM cu patru bancuri cu o interfață sincronă.

Fiecare bancă de 16.777.216 biți este organizată astfel: 4.096 de rânduri pe 256 de coloane pe 16 biți. SDRAM-ul de 64 de biți include modurile de reîmprospătare automată, de economisire a energiei și de dezactivare. Toate semnalele sunt înregistrate pe frontul pozitiv al semnalului de ceas, CLK. STM32F429I citește și scrie date la 80 MHz. [6]

3.1.5 Procesorul ARM Cortex-M4

Procesorul Arm Cortex-M4 este un procesor încorporat foarte eficient. Procesorul este dezvoltat pentru a se adresa tuturor piețelor de control pentru semnale digitale ce solicită o combinație ușor de utilizat și eficientă de capacități de control și de procesare a semnalelor. Îmbinarea dintre funcționalitățile de procesare a semnalelor de eficiență înaltă și cu avantajele consumului redus de energie, ale costului redus și ale ușurinței de utilizare ale procesoarelor Cortex-M satisface numeroase piețe. Printre acestea se numără piețele de control al gestionării energiei, ale motoarelor auto și de automatizare industrială.

Procesorul Cortex-M4 este un procesor de înaltă performanță pe 32 de biți proiectat pentru piața de microcontrolere. Acesta oferă beneficii semnificative dezvoltatorilor, printre care:

- performanță de procesare remarcabilă combinată cu gestionarea rapidă a întreruperilor
- depanare îmbunătățită a sistemului, cu capacități extinse de breakpoint și urmărire
- nucleu de procesor, sistem și memorii eficiente
- un consum de energie ultra-redus cu mod de veghe integrat și un mod de veghe profundă opțional

- robustețe de securitate a platformei, cu unitatea de protecție a memoriei (MPU) integrată opțională.

Procesorul Cortex-M4 este construit pe un nucleu de procesor de înaltă performanță, cu o arhitectură Harvard pipeline în 3 etape, ceea ce îl face ideal pentru aplicațiile încorporate solicitante.

Procesorul oferă o eficiență energetică excepțională printr-un set de instrucțiuni eficiente și un design optimizat pe scară largă, oferind hardware de procesare de vârf, inclusiv calculul opțional în virgulă mobilă cu precizie unică conform IEEE754, o gamă de capacități de multiplicare cu un singur ciclu, de multiplicare cu acumulare aritmetică de saturație și diviziune hardware dedicată.

Pentru a facilita proiectarea dispozitivelor sensibile la costuri, procesorul Cortex-M4 implementează componente de sistem strâns cuplate care reduc suprafața procesorului, îmbunătățind în același timp semnificativ gestionarea întreruperilor și capacitățile de depanare a sistemului. Setul de instrucțiuni Cortex-M4 oferă performanța excepțională așteptată de la o arhitectură modernă pe 32 de biți, cu densitatea mare de cod a microcontrolerelor pe 8 și 16 biți.

Procesorul Cortex-M4 integrează îndeaproape un controler de întreruperi configurabil Nested Vectored Interrupt Controller (NVIC), pentru a oferi performanțe de întrerupere de top în industrie. NVIC include un NMI (Non Maskable Interrupt) care poate oferi până la 256 de niveluri de prioritate a întreruperilor. Integrarea strânsă a nucleului procesorului și a NVIC asigură o execuție rapidă a serviciului de întreruperi. Caracteristici esențiale:

- integrarea strânsă a perifericelor sistemului reduce suprafața și costurile de dezvoltare
- FPU de o singură precizie conform IEEE754 opțional
- capacitate de codare pentru actualizări ale sistemului ROM
- optimizarea controlului energiei componentelor sistemului
- moduri de veghe integrate pentru un consum redus de energie
- execuția rapidă a codului permite un ceas de procesor mai lent sau mărește timpul de funcționare în modul sleep

- diviziune hardware și multiplicare rapidă orientată spre procesarea semnalelor digitale, cu multiplicare de acumulare
- aritmetică de saturație pentru procesarea semnalelor
- gestionarea deterministă și de înaltă performanță a întreruperilor pentru aplicațiile critice din punct de vedere al timpului
- unitate opțională de protecție a memoriei (MPU) pentru aplicații critice de siguranță
- capabilități extinse de depanare și urmărire definite de implementare:
    - Serial Wire Debug și Serial Wire Trace reduc numărul de pini necesari pentru depanare, urmărire și profilare a codului. [7]

3.2 Motor pas cu pas 28BYJ-48

Figura 3.3 Motor pas cu pas 28BYJ-48

Controlarea unui motor pas cu pas se execută cu ajutorul unor serii de bobine electromagnetice, arbore central al acestora conținând o serie de magneți montați pe el, iar alimentarea alternativă crează câmpuri magnetice care atrag sau resping magneții așezați pe arbore cauzând rotația motorului.

Motorul este de tip unipolar și este constituit din patru bobine și șase fire. Alimentarea bobinelor se realizează prin racordurile centrale care sunt îmbinate între ele. Unipolaritatea pentru motoarele pas cu pas înseamnă că puterea intră întotdeauna în același pol. [8]

Specificații tehnice:

- 5V reprezintă tensiunea pentru funcționarea corespunzătoare
- Motorul are 4 faze
- Variația de viteză este de 1/64 pași
- 100Hz este frecvența
- 34.3 mNm este cuplul minim de tracțiune

3.3 Senzorul de iluminare TEMT6000

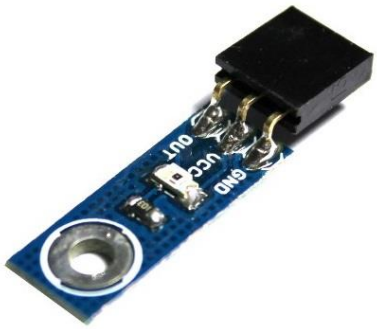


Figura 3.4 Senzorul de luminozitate TEMT6000

3.3.1 Introducere

Senzorii de lumină au tot felul de utilizări practice în epoca modernă, mai ales în dispozitivele cu luminozitate automată a ecranelor și în camerele digitale pentru a ajusta expunerea.

Fiind un fototransistor, acest senzor se comportă ca orice alt tranzistor NPN - cu cât lumina care intră pe bază este mai mare, cu atât mai mult curent poate trece de la colector la emițător. Numai lumina care se încadrează în spectrul vizibil (390-700 nm) va altera baza. Acest senzor poate gestiona tensiuni atât de la dispozitive de 5V, cât și de la dispozitive de 3,3V.

Pentru a face ca efectuarea măsurătorilor de lumină să fie cât mai ușoară, acest senzor a fost proiectat într-un circuit de divizor de tensiune. TEMT600 acționează ca una dintre rezistențele din divizor și, pe măsură ce lumina care îl lovește se modifică, la fel se întâmplă și cu tensiunea de pe pinul SIG. [9]

Pentru a citi această tensiune, am conectat pur și simplu pinul OUT de pe TEMT6000 la un pin de conversie analogică/digitală de pe microcontrolerul STM32F429I-DISC1.

3.3.2 Specificații tehnice

- Tensiunea colector-emitor: 6 V
- Tensiunea colector-emitor: 1,5 V
- Curent maxim: 20 mA
- Mod de alimentare: 3,3 V până la 5 V
- Lungime de undă: 390-700 nm
- Unghi de detecție: 60 grade
- Dimensiuni: 24 x 8 mm

4. IMPLEMENTARE

4.1 Configurarea pinilor

Pentru controlul motorului pas cu pas 28BYJ-48 a fost nevoie să configurez patru pini (PE2, PE3, PE4, PE5) folosind CubeMX (Figura 4.1). Aceștia sunt de tipul GPIO_Output (Scop General Intrare/Ieșire) și trimit impulsuri electrice controlate către driverul motorului.

Pentru preluarea valorilor analogice citite ciclic din mediu de către senzorii de intensitate a luminii am folosit câte un pin pentru fiecare din cei doi senzori (PA0 și PC1). Acești pini sunt de tipul ADC_Input (Convertor Analog-Digital) și au acces direct la memoria microcontrolerului prin setarea DMA-ului (Figura 4.2).

Figura 4.1 Configurația pinilor

```
/*****************************************************************//**
*
* \descriere RoutineInit
* - pornește DMA-ul pentru ADC-ul 1 și ADC-ul 3 pentru citirea valorilor analogice
* și conversia lor la digital
* - pornește Timer-ul 1 și Timer-ul 3 pentru citire ciclică la fiecare tick de ceas
*
*********************************************************************/
void RoutineInit()
{
    HAL_ADC_Start_DMA(&hadc1, &oAdcConvertedValuePA0, NUMBER_OF_VALUES);
    HAL_ADC_Start_DMA(&hadc3, &oAdcConvertedValuePC1, NUMBER_OF_VALUES);

    HAL_TIM_Base_Start_IT(&htim2);
    HAL_TIM_Base_Start_IT(&htim3);
}
```

Figura 4.2 Pornire DMA pentru conversia analogică/digitală

Funcția RoutineInit este apelată în task-ul creat folosind sistemul de operare în timp real "FreeRTOS" (Figura 4.3).

```
/* USER CODE BEGIN Header_StartTaskAnalogInput */
/**
* @brief Function implementing the taskAnalogInput thread.
* @param argument: Not used
* @retval None
*/
/* USER CODE END Header_StartTaskAnalogInput */
void StartTaskAnalogInput(void *argument)
{
  /* USER CODE BEGIN StartTaskAnalogInput */

  RoutineInit();

  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END StartTaskAnalogInput */
}
```

Figura 4.3 Apelare funcție RoutineInit() în task

4.2 Crearea interfeței

Interfața (Figura 4.4) este creată cu ajutorul software-ului specializat TouchGFX și este constituită din două ecrane principale, primul ecran afișând valorile citite din mediu de către senzorii de intensitate a luminii și un grafic dinamic ce reprezintă diferența dintre valorile citite de senzori. Al doilea ecran este ecranul de control al motorului pas cu pas de unde putem seta modul de control și putem roti motorul în direcția dorită interacționând cu butoanele pentru stânga (L) și dreapta (R).

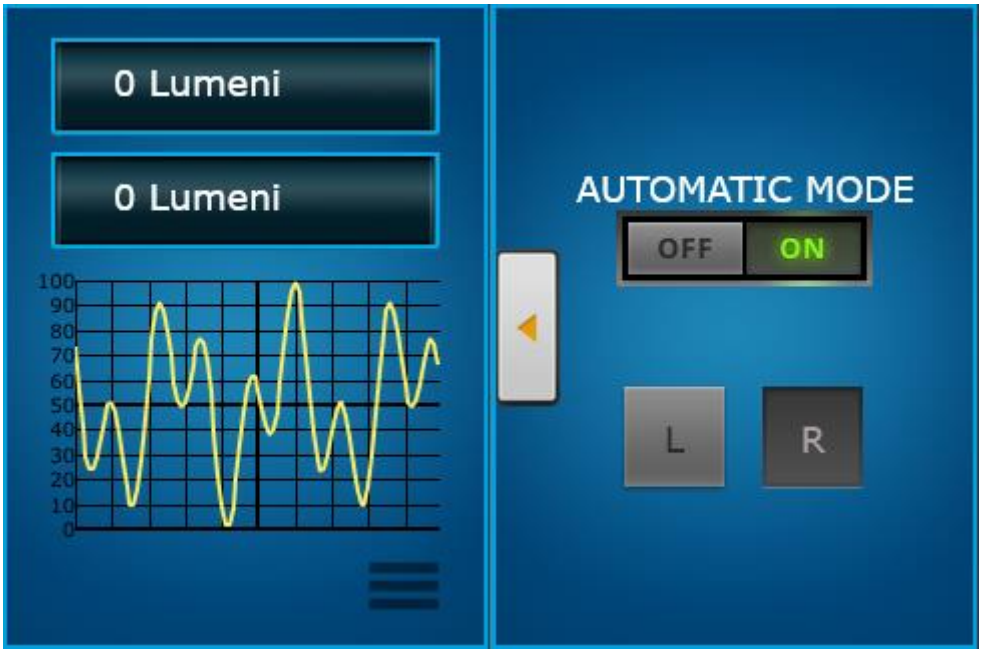


Figura 4.4 Interfață

Comunicația dintre microcontroler și interfață pentru afișarea datelor preluate din mediu de către senzorii de intensitate a luminii este realizată în fișierul „MainScreenView.cpp” cu ajutorul a două funcții care se execută ciclic în timp real. Funcțiile sunt prezentate în figura de mai jos (Figura 4.5).

```
/*************************************************************//**
*
* \descriere analogUpdatePA0
* - transmite datele preluate de senzorul de lumină conectat la pinul PA0
* către interfață
*
*****************************************************************/
void MainScreenView::analogUpdatePA0(uint32_t value)
{
    memset(&textLightSensorPA0Buffer, 0, TEXTLIGHTSENSORPA0_SIZE);
    Unicode::snprintf(textLightSensorPA0Buffer, sizeof(textLightSensorPA0Buffer), "%d", value);
    textLightSensorPA0.invalidate();
}

/*************************************************************//**
*
* \descriere analogUpdatePA0
* - transmite datele preluate de senzorul de lumină conectat la pinul PC1
* către interfață
*
*****************************************************************/
void MainScreenView::analogUpdatePC1(uint32_t value)
{
    memset(&textLightSensorPC1Buffer, 0, TEXTLIGHTSENSORPC1_SIZE);
    Unicode::snprintf(textLightSensorPC1Buffer, sizeof(textLightSensorPC1Buffer), "%d", value);
    textLightSensorPC1.invalidate();
}
```

Figura 4.5 Transmitere date către interfață

Graficul se încarcă dinamic la fiecare tic de ceas cu puncte reprezentând diferența dintre valorile citite de către cei doi senzori care preiau informația legată de intensitatea luminii ambientale (Figura 4.6).

```
/*************************************************************//**
*
* \descriere handleTickEvent
* - reîncarcă graficul cu puncte reprezentând diferența dintre valorile
* citite de senzori
*
*****************************************************************/
void MainScreenView::handleTickEvent()
{
    if(analogValuePC1 > analogValuePA0)
        dGraphSensors.addDataPoint(analogValuePC1 - analogValuePA0);
    else
        dGraphSensors.addDataPoint(analogValuePA0 - analogValuePC1);
}
```

Figura 4.6 Încărcare grafic cu date

Rotirea motorului pas cu pas înspre stânga sau dreapta din interfață se face prin apăsarea butoanelor din ecranul de control, doar dacă modul automat este setat pe “OFF”. La apăsarea acestora, se execută cod C++, prezentat în figura de mai jos (Figura 4.6).

```
/*************************************************************//**
*
* \descriere moveStepperToRight
* - poziționează motorul pas cu pas înspre dreapta cu 32 pași (o rotație completă are 2048 pași)
* - aplică un curent pe fiecare pin la care este conectat motorul pas cu pas, alternativ
*
*****************************************************************/
void Model::moveStepperToRight()
{
    for(int i = 0; i < NUMBER_OF_STEPS; i++)
    {
        HAL_GPIO_WritePin(PHASE1_GPIO_Port, PHASE1_Pin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(PHASE2_GPIO_Port, PHASE2_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE3_GPIO_Port, PHASE3_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE4_GPIO_Port, PHASE4_Pin, GPIO_PIN_RESET);

        HAL_Delay(DELAY_VALUE);

        HAL_GPIO_WritePin(PHASE1_GPIO_Port, PHASE1_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE2_GPIO_Port, PHASE2_Pin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(PHASE3_GPIO_Port, PHASE3_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE4_GPIO_Port, PHASE4_Pin, GPIO_PIN_RESET);

        HAL_Delay(DELAY_VALUE);

        HAL_GPIO_WritePin(PHASE1_GPIO_Port, PHASE1_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE2_GPIO_Port, PHASE2_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE3_GPIO_Port, PHASE3_Pin, GPIO_PIN_SET);
        HAL_GPIO_WritePin(PHASE4_GPIO_Port, PHASE4_Pin, GPIO_PIN_RESET);

        HAL_Delay(DELAY_VALUE);

        HAL_GPIO_WritePin(PHASE1_GPIO_Port, PHASE1_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE2_GPIO_Port, PHASE2_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE3_GPIO_Port, PHASE3_Pin, GPIO_PIN_RESET);
        HAL_GPIO_WritePin(PHASE4_GPIO_Port, PHASE4_Pin, GPIO_PIN_SET);

        HAL_Delay(DELAY_VALUE);
    }
    HAL_Delay(750);
}
```

Figura 4.7 Cod C++ pentru rotirea motorului spre dreapta

4.3 Modul automat

Modul automat este reprezentat de un algoritm pentru rotirea motorului pas cu pas și implicit al machetei în funcție de direcția din care intensitatea luminii are o valoare mai mare.

Senzorii de intensitate a luminii preiau datele din mediu, iar când valoarea intensității luminii citită de către unul dintre senzori este mai mare, motorul se rotește în direcția dinspre care intensitatea luminii este mai mare și se va opri atunci când valorile citite de senzori sunt aproximativ egale, asta însemnând că panoul solar este îndreptat spre lumină.

Algoritmul implementat poate fi observat în figura de mai jos (Figura 4.7).

```
/*********************************************************************//**
*
* \descriere analogUpdatePA0
* - actualizează valoarea analogică citită de senzorul de lumină conectat la pinul PA0
* - face o medie aritmetică de 10 valori (smoothing)
* - poziționează motorul pas cu pas înspre dreapta dacă valoarea citită de la senzorul de lumină
* conectat la pinul PA0 e mai mare decât valoarea citită de la senzorul de lumină
* conectat la pinul PC1
* - oprește rotația dacă valoarea e mai mare decât o anumită valoare "ANALOGIC_MAX_VALUE"
*
**********************************************************************/
void Model::analogUpdatePA0()
{
    uint32_t sumPA0 = 0;

    for(int i = 0; i < NUMBER_OF_VALUES; i++)
    {
        sumPA0 += oAdcConvertedValuePA0[i];
    }

    modelListener->analogUpdatePA0(sumPA0 / NUMBER_OF_VALUES);
    analogValuePA0 = sumPA0 / NUMBER_OF_VALUES;

    if((isControlModeAuto == true) &&
    (analogValuePA0 - analogValuePC1) > ANALOGIC_DIFFERENCE &&
    (analogValuePA0 < ANALOGIC_MAX_VALUE))
    {
        moveStepperToRight();
    }
}


/*********************************************************************//**
*
* \descriere analogUpdatePC1
* - actualizează valoarea analogică citită de senzorul de lumină conectat la pinul PC1
* - face o medie aritmetică de 10 valori
* - poziționează motorul pas cu pas înspre stânga dacă valoarea citită de la senzorul de lumină
* conectat la pinul PC1 e mai mare decât valoarea citită de la senzorul de lumină
* conectat la pinul PA0
* - oprește rotația dacă valoarea e mai mare decât o anumită valoare "ANALOGIC_MAX_VALUE"
*
**********************************************************************/
void Model::analogUpdatePC1()
{
    uint32_t sumPC1 = 0;

    for(int i = 0; i < NUMBER_OF_VALUES; i++)
    {
        sumPC1 += oAdcConvertedValuePC1[i];
    }

    modelListener->analogUpdatePC1(sumPC1 / NUMBER_OF_VALUES);
    analogValuePC1 = sumPC1 / NUMBER_OF_VALUES;

    if((isControlModeAuto == true) &&
    (analogValuePC1 - analogValuePA0) > ANALOGIC_DIFFERENCE &&
    (analogValuePC1 < ANALOGIC_MAX_VALUE))
    {
        moveStepperToLeft();
    }
}
```

Figura 4.8 Algoritm mod automat

4.4 Schema electrică

Shema electrică poate fi vizualizată mai jos (Figura 4.8).

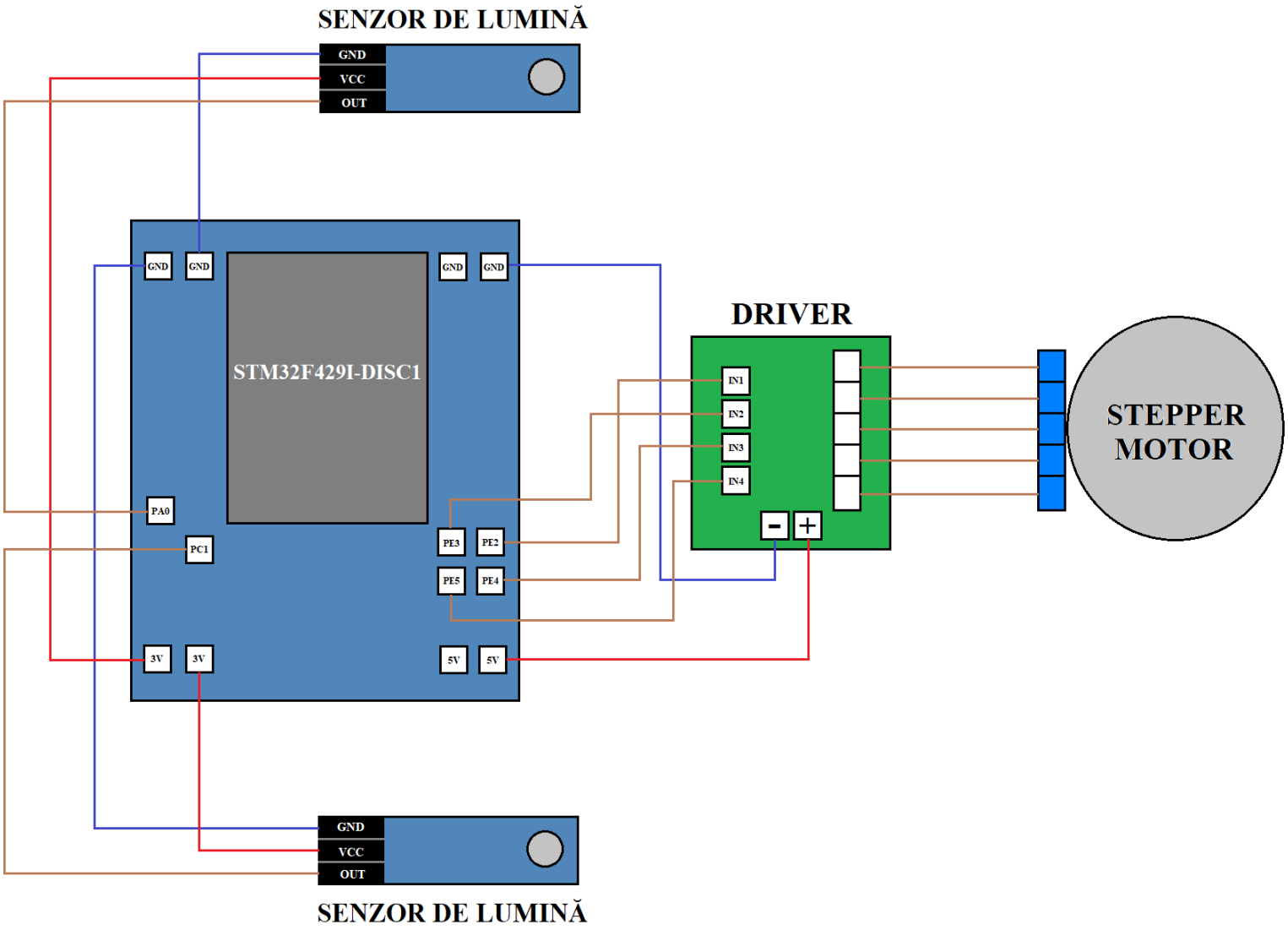


Figura 4.8 Schema electrică

4.5 Măsurători

Măsurătorile au fost realizate cu ajutorul unui osciloscop Analog Discovery 2 Digilent conectat la un FPGA Spartan 7. Capturile au fost realizate în programul WaveForms. În figura de mai jos (Figura 4.9), se pot observa valorile semnalelor primite de către cei doi

senzori de intensitate a luminii atunci când asupra lor este aplicată o lumină provenită de la lanterna telefonului, din diferite poziții.

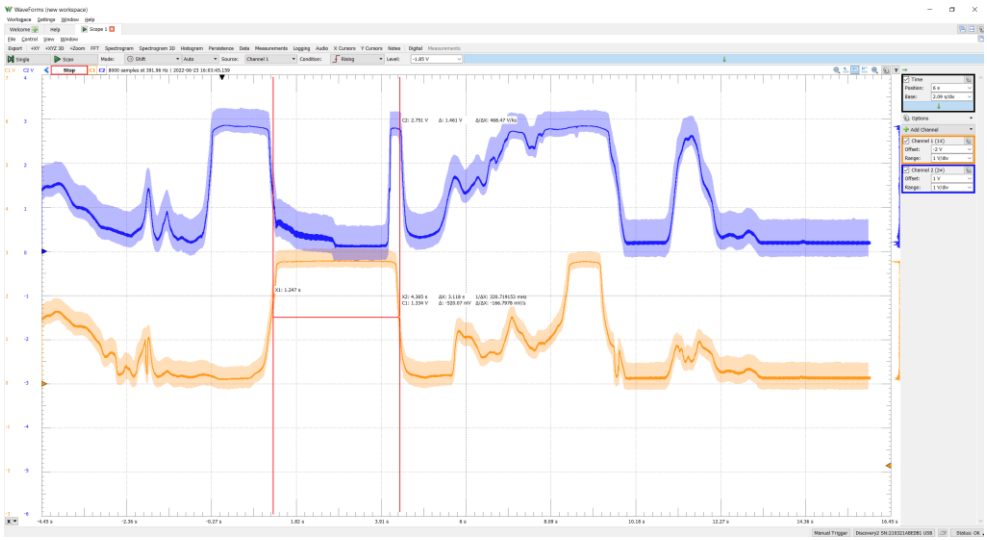
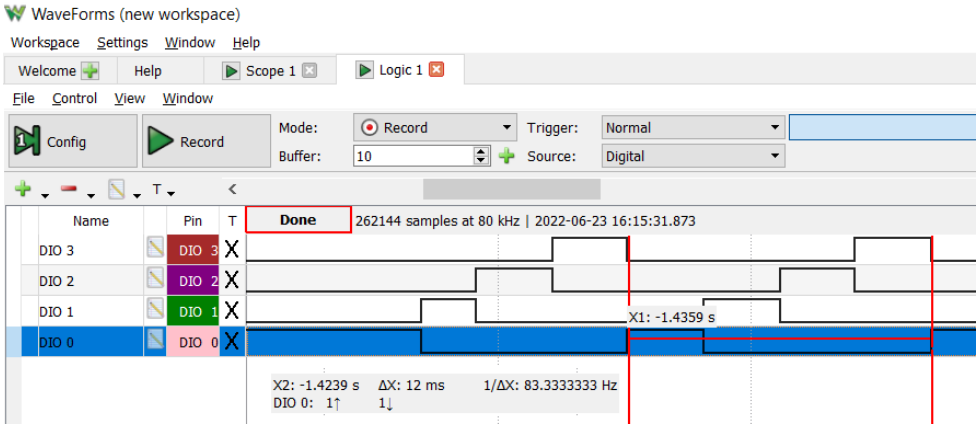


Figura 4.9 Valori ale semnalelor primite de senzorii de intensitate a luminii

În figura de mai jos (Figura 4.10) putem observa semnalele dreptunghiulare ale celor patru faze pentru rotația motorului pas cu pas în ambele direcții. Perioada semnalelor este de 12 milisecunde.



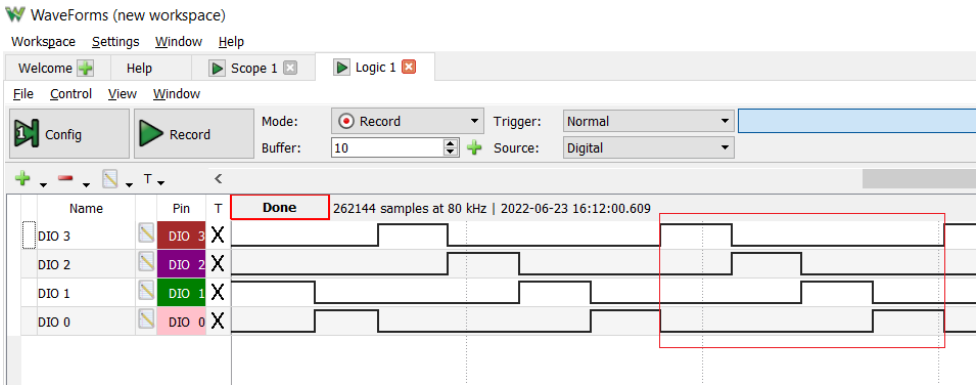


Figura 4.10 Semnalele dreptunghiulare pentru rotația motorului pas cu pas

5. CONCLUZII

Realizarea acestui proiect nu ar fi putut fi înfăptuită fără utilizarea cunoștințelor teoretice cât și practice dobândite de-a lungul celor 4 ani de studiu în cadrul Facultății de Inginerie Electrică și Știința Calculatoarelor din cadrul Universității "Ștefan cel Mare" din Suceava. Câteva din cunoștințele dobândite:

- Programare orientată pe obiecte
- Electrotehnică
- Componentele unui microcontroler
- Abilități de scriere modulară a codului
- Preluarea, analizarea și manipularea semnalelor din mediu cu ajutorul senzorilor

Prin dezvoltarea părților hardware și software ale acestui proiect, am pus în aplicare marea majoritate a cunoștințelor dobândite în facultate. Am reușit astfel să dezvolt un sistem inteligent prin care un motor pas cu pas rotește o machetă așezată pe un rulment în funcție de intensitatea luminii preluată de la senzorii de lumină atașați pe acoperișul acesteia.

6. BIBLIOGRAFIE

[1] Stroustrup Bjarne, The C++ Programming Language, 2013

[2] ST Website, TouchGFX Designer software tool for easy GUI creation and code generation on STM32 microcontrollers, https://www.st.com/en/development-tools/touchgfxdesigner.html#documentation, 2021

[3] TouchGFX Website, TouchGFX Documentation, https://support.touchgfx.com/4.18/docs/introduction/what-is-touchgfx, 2022

[4] ST Website, Product overview for CubeMx, https://www.st.com/en/development-tools/stm32cubemx.html, 2021

[5] ST Website, STM32CubeMX for STM32 configuration and initialization C code generation, https://www.st.com/en/development-tools/stm32cubemx.html#documentation, 2021

[6] ST Website, Getting started with STM32F429 Discovery software development tools, https://www.st.com/en/evaluation-tools/32f429idiscovery.html#documentation, 2021

[7] Developer Arm Website, Cortex -M4 Devices Generic User Guide Generic User Guide, https://developer.arm.com/documentation/dui0553/b/?lang=en, 2021

[8] Instructables, http://www.instructables.com/id/BYJ48-Stepper-Motor, 2018

[9] Sparkfun, TEMT6000 Ambient Light Sensor Hookup Guide, https://learn.sparkfun.com/tutorials/temt6000-ambient-light-sensor-hookup-guide/all, 2021

7. ANEXE