UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Protocol de comunicare pentru microcontrolere

propusă de

Andrei Teodor Timofte

Sesiunea: *iulie, 2020*

Coordonator științific

Lect. Dr. Cosmin-Nicolae Vârlan

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAȘI

FACULTATEA DE INFORMATICĂ

Protocol de comunicare pentru microcontrolere

Andrei Teodor Timofte

Sesiunea: *iunie, 2020*

Coordonator științific

Lect. Dr. Cosmin-Nicolae Vârlan

Avizat,

Îndrumător Lucrare de Licență

Titlul, Numele și prenumele _____

Data_____ Semnătura _____

DECLARAȚIE privind originalitatea conținutului lucrării de licență

Subsemntatul(a) ……

domiciliul în ……..

născut(ă) la data de ……..…., identificat prin CNP ....., absolvent(a) al(a) Universității „Alexandru Ioan Cuza" din Iași, Facultatea de ……. specializarea ……, promoția ……., declar pe propria răspundere, cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art.143 al. 4 si 5 referitoare la plagiat, că lucrarea de licență cu titlul:

_____

_____

_____elaborată sub îndrumarea dl. / d-na _____, pe care urmează să o susțină în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului său într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice in vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diploma sau de disertație şî în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă

rodul cercetării pe care am întreprins-o.

Dată azi,…… Semnătură student……

DECLARAȚIE DECONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „*Protocol de comunicare pentru microcontrolere*”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.)careînsoțescaceastălucraresăfieutilizateîncadrulFacultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași, *data*

Absolvent *Andrei Teodor Timofte*

_____

ACORD PRIVIND PROPRIETATEA DREPTULUI DE AUTOR

Facultatea de Informatică este de acord ca drepturile de autor asupra programelor-calculator, în format executabil și sursă, să aparțină autorului prezentei lucrări, *Andrei Teodor Timofte*.

Încheierea acestui acord este necesară din următoarele motive:

Iași, *data*

Decan *Adrian Iftene*

_____

(semnătura în original)

Absolvent *Andrei Teodor Timofte*

_____

(semnătura în original)

Cuprins

Introducere

Ideea lucrării

Protocol de comunicare pentru microcontrolere este o lucrare practică ce are ca scop implementarea unei biblioteci specifice mediului de lucru Arduino prin care două (sau mai multe) plăcuţe de acest tip pot comunica între ele mesaje, la nivel hardware, prin intermediul firelor. Scopul prezentei lucrări este de a oferi un mod sigur de comunicare serială prin pinii digitali puşi la dispoziţie de către plăcuţă, din punctul de vedere al integrităţii mesajelor, folosindu-se idei din stiva TCP/IP.

Ideea acestei teme a plecat de la necesitatea existenţei unei biblioteci care să asigure integritatea datelor transmise între două plăcuţe Arduino cât şi posibilitatea de a interconecta şi comunica simultan între mai mult de două plăcuţe de acest tip . Acesta bibliotecă, ce împrumută idei teoretice din protocolul TCP/IP, îşi poate găsi cu uşurinţă, o utilizare în domeniul *IoT*, un domeniu tehnologic aflat în plină expansiune în ultimul deceniu.

Pentru aceasta, am creat o bibliotecă Arduino, denumită **TwoArduinoSerialCom**. Această bibliotecă permite interconectarea a două plăcuţe Arduino prin metode (funcţii) stabilite la nivel software. Paradigma de programare folosită în dezvoltarea acestei biblioteci este cea de Programare Orientată pe Obiecte, limbajul de dezvoltare fiind o submulţime de funcţii ale bibliotecii C standard. **TwoArduinoSerialCom** suportă două categorii de comunicare între plăcuţe de tip Arduino: comunicarea fără confirmare (asemănătoare protocolului UDP din stiva TCP/IP) şi comunicarea cu confirmare (asemănătoare protocolului TCP din stiva TCP/IP). Un element comun al acestor două protocoale este fragmentarea mesajului iniţial în pachete de dimensiuni prestabilite, din motive de performanţă dar şi calcularea unei sume de control a pachetului transmis pentru identificarea integrităţii acestuia.

Comunicare fără confirmare

Comunicarea fără confirmare sau comunicarea fără conexiune presupune trimiterea mesajului fragmentat în blocuri, de la expeditor către destinatar, fără a se asigura că există o confirmare de primire din partea destinatarului, a pachetelor de date transmise. În acest mod, expeditorul nu este dependent în niciun fel de destinatar, viteza de transfer a datelor fiind una

ridicată deoarece nu există retransmitere de pachete, ca în cazul comunicării cu confirmare. La destinatar, pachetele se vor uni, astfel formând mesajul trimis de către expeditor. Acest tip de comunicare conţine un mechanism de validare a corectitudinii datelor, implementat prin intermediul unei sume de control ce face parte din fiecare pachet tranzacţionat. Astfel, destinatarul va putea şti dacă pachetul primit conţine date valide sau nu, neputând însă sa le corecteze.

Comunicare cu confirmare

Comunicarea cu confirmare sau comunicarea cu o conexiune stabilă presupune în primul rând o metodă de interconectare între interlocutori. Prin aceasta se asigură faptul că expeditorul şi destinatarul se „cunosc" unul pe celălalt şi sunt pregătiţi să înceapă comunicarea de date prin intermediul porturilor seriale. Ulterior interconectării, se poate începe comunicarea de mesaje între cele două părţi. Similar protocolului de comunicare fără confirmare, mesajul este împărţit în pachete de dimensiune fixă, un pachet conţinând şi o sumă de control. O primă diferenţă sunt biţii redundanţi ce fac parte din pachet, pentru corectarea mesajului la expeditor, în cazul în care datele transmise sunt corupte din cauza erorilor la nivel fizic / hardware. Prin acest mecanism se asigura integritatea datelor transferate.

O altă diferenţa între acest tip de comunicare şi cel enunţat în subcapitolul anterior este confirmarea primirii cu succes a pachetului. În caz contrar, destinatarul va cere retrimiterea pachetului de către expeditor până când toate pachetele, implicit tot mesajul compus din pachete, au fost transmise cu succes. În acest mod, se asigură o comunicare fără erori între cele două părţi.

Alte abordări ale acestei probleme

O abordare similară a acestei probleme reprezintă biblioteca **SerialTransfer** ce poate fi găsită pe GitHub[1]. Aceasta permite fragmentarea datelor în pachete de dimensiune variabilă, detectare de erori prin algoritmul Cyclic Redundancy Check – 8, transmitere de tipuri de date precum bytes, int, floats şi chiar structuri de date, compatibilitate cu porturile

[1] https://github.com/PowerBroker2/SerialTransfer

hardware UART[2] şi cele software prestabilite de către plăcuţele Arduino. Biblioteca conţine şi un mecanism de tratare şi expunere, către utilizator, a erorilor apărute în procesul de transmitere. De asemenea, suportă orice rată BAUD de transfer a datelor. Rata BAUD reprezintă numărul maxim de biţi ce pot fi transferaţi într-o secundă, în cadrul unui canal de comunicaţii.

În capitolul Anexe, subcapitolul Anexă 1, voi arăta un exemplu de utilizarea a bibliotecii **SerialTransfer**, conform prezentării din secţiunea README.md de pe GitHub[1,], pentru a putea compara apoi cu biblioteca **TwoArduinoSerialCom**. Spre deosebire de această bibliotecă, lucrarea de licenţă prezentă conţine mecanisme suplimentare inspirate din protocolul TCP/IP, care oferă o comunicare orientată pe conexiune stabilă între cele două părţi. În subcapitolul următor vom vedea mai detaliat diferenţele dintre cele două biblioteci.

Contribuţii

În această secţiune vă voi prezenta prin ce se diferenţiază bibliotecă mea faţă de cea prezentată în subcapitolul anterior. Metodele publice ale acestei biblioteci le puteţi găsi în capitolul Anexe, subcapitolul Anexă 2. Biblioteca **TwoArduinoSerialCom** conţine o serie de metode pe care le-am implementat, prin intermediul cărora se realizează transmiterea de date către o altă plăcuţă Arduino ce rulează aceeaşi bibliotecă.

O primă diferenţă între cele două biblioteci este faptul că **TwoArduinoSerialCom** suportă transferul datelor prin oricare două porturi (RX, TX), şi nu doar porturile 0 şi 1 ale plăcuţei Arduino, care sunt în mod implicit pentru comunicarea serială. Deoarece sunt doar două porturi, biblioteca **SerialTransfer** permite implicarea în cadrul aceluiaşi proiect a doar două plăcuţe. Prin mărirea numărului de porturi ce suportă comunicarea serială, biblioteca **TwoArduinoSerialCom** face posibilă integrarea în cadrul aceluiaşi proiect a mai multe plăcuţe ce pot intercomunica, pe rând, două câte două.

O a doua diferenţă este aceea că biblioteca **TwoArduinoSerialCom** asignează fiecărui plăci Arduino un număr unic de identificare denumit în cadrul sistemului: UAID[3]. Astfel, utilizatorul are opţiunea să folosească un sistem de adresare, în caz că doreşte acest lucru. Spre exemplu, dacă întreg sistemul conţine mai multe plăcuţe Arduino interconectate şi există un server / router ce va centraliza şi redirecţiona datele, metodele de scriere şi citire suportă

[2] Universal asynchronous receiver-transmiter
[3] Unique Arduino Identifier

un argument prin care se specifică numărul unic de identificare al plăcuţei destinatare / expeditoare.

În cazul în care utilizatorul a optat pentru modul de comunicare cu confirmare, alte diferenţe între cele două biblioteci sunt existenţa mecanismelor de detectare şi corectare dar şi de retrimitere a pachetelor corupte. Cele două mecanisme conlucrează şi încercă să corecteze un pachet, dacă acesta conţine maximum doi biţi greşiţi. În cazul în care se identifică mai mulţi biţi greşiţi, mecanismul va cere retransmiterea pachetului care a fost identificat ca şi corupt. De asemenea, înaintea începerii comunicării, plăcuţele îşi vor semnala una alteia prezenţa în cadrul sistemului, printr-un mecanism de conectarea. Rolul acestuia este evitarea trimiterii de date către un destinatar care nu este pregătit să le primească. De asemenea există şi un mecanism de încheiere a conexiunii, ce funcţionează similar.

Concluzie

În concluzie, contribuţiile acestei biblioteci faţă de cea prezentată în subcapitolul precedent sunt următoarele:

- mecanism de interconectare între două plăcuţe pe baza unui UAID
- disponibilitatea tuturor porturilor pentru comunicare de mesaje ceea ce permite conectarea în cadrul aceluiaşi proiect / *sketch-uri* a mai multe plăcuţe Arduino
- creare unui id unic pentru fiecare plăcuţă Arduino
- transmiterea mesajelor în două moduri: cu şi fără confirmare
- segmentarea mesajului în blocuri de dimensiune fixă
- detectarea şi corectarea mesajelor la destinaţie în caz ca au apărut erori în procesul de transmitere
- mecanism de reordonare a pachetelor primite
- mecanism de încetare a conexiunii

În capitolul Anexe, subcapitolul Anexă 3, puteţi găsi un exemplu funcţional de cod client – server.

Structura licenţei

Această lucrare de licenţă conţine trei capitole, Abordări teoretice, Descrierea Soluţiei şi Concluziile lucrării. De asemenea, fiecare subcapitol parte a unui capitol, va conţine la final o secţiune de concluzii, în care se vor sumariza principalele subiecte atinse.

În primul capitol al acestei licenţe veţi găsi explicaţii ale părţilor teoretice din spatele bibliotecii **TwoArduinoSerialCom,** precum o scurtă introducere despre mediul de dezvoltare şi plăcuţa Arduino, algoritmul de detectare de erori, algoritmul de detectate şi corectare de erori, protocolul de stabilire a conexiunii şi de încetarea a acesteia, toate implementate în lucrarea de faţă.

În al doilea capitol se atinge subiectul descrierii soluţiei găsite. Aici voi detalia, explica şi arăta, structurile de date utilizate, o parte din algoritmii folosiţi, structura pachetului de date transmis către destinatar cât şi o mică secţiune în care voi vorbi despre problemele întâlnite de mine de-a lungul procesului de dezvoltare al acestei biblioteci.

Capitolul al treilea conţine concluziile aceste lucrări: un sumar al tuturor celorlalte capitole anterioare şi câteva statistici din punctul de vedere al memoriei şi al vitezei utilizate de către biblioteca.

1 Abordare teoretică

În acest capitol vom discuta despre aspectele teoretice ale acestei lucrări de licenţă. Deoarece această bibliotecă împrumută idei din protocolul TCP/IP, idei prin care se realizează diferite mecanisme de conectare între plăcuţe, de detectare şi corectare de erori, şi de retransmitere de pachete, în acest capitol voi explica fundamentele teoretice ce stau la baza acestora şi cu ajutorul cărora se realizează transferul datelor fără erori de la expeditor la destinatar.

1.1 Arduino – hardware şi software

Ce este o plăcuţă Arduino şi la ce ne foloseşte? Arduino este o companie *open-source* ce poate fi privită din două perspective: hardware şi software. La nivel hardware, compania Arduino produce multiple modele de plăcuţe Arduino ce sunt compuse dintr-un microcontroler, de obicei de tip Atmel AVR de 8, 16 sau 32 de biţi. Plăcuţa are ataşaţi diferiţi pini analogici şi digitali ce permit conectarea acesteia la alte plăcuţe hardware, denumite *shielduri*. Modul de comunicare între *shielduri* şi plăcuţa Arduino mamă se efectuează, de obicei, prin conexiune serială. Scopul principal este de a controla şi programa diferiţi senzori şi dispozitive electronice similare ce sunt conectate la plăcuţa Arduino. Aici intervine partea de software, compania Arduino oferind suport în acest sens programatorilor. Aceasta a creat un mediu integrat de dezvoltare (*IDE*) prin care se poate scrie cod, asemănător limbajului C şi C++, cod ce conţine diferite metode şi *API-uri* ce uşurează accesul la pinii plăcuţei. Se poate spune că Arduino este un limbaj de programare, dar este mai mult o formulare improprie, deoarece acesta conţine o mulţime de funcţii împrumutate din limbajul C / C++. *IDE-ul* deţine un compilator de C/C++ (avr-g++) şi poate încărca codul scris în memoria plăcuţei Arduino. Programele pentru plăcuţă Arduino suportă orice limbaj de programare atâta timp cât există un compilator ce va produce codul maşină binar pentru microcontrolerul Atmel AVR. Comunitatea Arduino oferă o serie de biblioteci *open*-source ce sunt incluse în (*IDE*) şi care oferă diferite funcţionalităţi precum biblioteca din această lucrare. Datorită sprijinului oferit de programatori prin crearea diverselor biblioteci, dezvoltarea proiectelor electronice cu Arduino este una simplă şi fără prea multe bătăi de cap, chiar şi pentru programatorii începători.

1.1.1 Specificaţii tehnice

În continuare, voi vorbi despre specificaţiile tehnice ale plăcuţei de dezvoltare Arduino UNO. Cum am menţionat mai sus, plăcuţa dispune de un microcontroler Atmel AVR de tip Atmega328p, 14 pini digitali de intrare / ieşire, 6 pini analogici, port USB, şi un buton de repornire. Viteza de executarea a microcontrolerului este de 16MHz, suficientă pentru a efectua sarcini complexe în contextul programării componentelor electronice. Tensiunea de funcţionare a plăcuţei este de 5V, compatibilă cu majoritatea *shieldurilor* aflate pe piaţă.

Memoria plăcuţelor Arduino este împărţită în trei categorii: memoria flash, memoria SRAM[4] şi memorie EEPROM[5]. Arduino UNO dispune de 32 KB de memorie flash din care 0.5 KB sunt ocupaţi de programul de iniţializare a programelor utilizatorilor (*bootloader*-ul). În această memorie se stochează programul utilizatorului, denumit şi *sketch*, dar şi orice iniţializare de date. Memoria SRAM[4] este de 2 KB şi este o memorie volatilă. Aceasta este, din punctul meu de vedere, cel mai important tip de memorie al plăcuţei Arduino, deoarece programatorul este limitat la o cantitate de memorie mică pe care trebuie să o gestioneze corect, în caz contrat aceasta poate fi depăşită şi codul poate avea un comportament impredictibil. În acest tip de memorie se află salvate datele statice, memoria de tip *Heap* şi de tip *Stack*. Memoria EEPROM[5] este un tip de memorie nevolatilă de 1 KB, care poate fi scrisă şi citită din codul utilizatorului. Chiar dacă aceasta are o viteză de scriere / citire mai mică decât memoria SRAM, aceasta poate fi uneori foarte folositoare.

1.1.2 Comunicarea serială

Plăcuţa Arduino UNO deţine doi pini digitali de tip RX / TX, special pentru comunicarea serială de mesaje între plăcuţe Arduino prin intermediul firelor de lungime scurtă. Aceasta se realizează prin intermediul unei componente hardware UART[6], integrate în arhitectura microcontrolerului, prin intermediul căreia se obţine trimiterea, recepţionarea şi interpretarea biţilor (0 şi 1) de la un microcontroler la altul. Transmiterea datelor se realizează asincron, neexistând niciun mecanism de sincronizare sau de validare a datelor primite de către destinatar. Viteza de transmitere este un parametru configurabil asupra căruia trebuie să cadă de acord cele două microcontrolere.

[4] Static Random Access Memory
[5] Electrically Erasable Programmable Read-Only Memory
[6] Universal Asynchronous Receiver Transmitter

1.1.3 Concluzie

În concluzie, privind specificaţiile tehnice, plăcuţa Arduino UNO se potriveşte pentru dezvoltarea diverselor componente electronice deoarece are o viteză mare de reacţie, în ciuda faptului că are o putere mică de calcul şi deţine o memorie limitată (doar 2 KB de memorie SRAM[4]), ce trebuie gestionată cu atenţie de către programator. De asemenea microcontrolerul plăcuţei suportă comunicarea serială, subiect principal al acestei lucrări.

1.2 Procedeul de conectare *Three Way Handshake*

Procedeul de conectare *Three Way Handshake* este un procedeu ce face parte din protocolul TCP (Transmission Control Protocol). Acesta presupune o interschimbare de pachete de comunicare, rolul fiind de a stabili un contract comun stabil şi de încredere între două părţi (client – server) ce doresc să comunice. Utilizarea acestuia în cadrul bibliotecii **TwoArduinoSerialCom** a fost necesară pentru a interconecta două plăcuţe Arduino în modul de comunicare cu confirmare, astfel încât apoi să se poată începe comunicarea propriu-zisă. Implementarea mea a acestui protocol se găseşte în două metode publice ale bibliotecii: connect() şi listen() şi este realizată pe baza a două *flaguri*: SYN (Synchronize Sequence Number), ACK (acknowledged).

Paşii de conectare a unui client la server sunt următorii:

1. Serverul apeleaza metoda listen(), prin care ascultă şi aşteaptă ca clientul să se conecteze.
2. Clientul apelează metoda connect(), prin care doreşte să stabilească o conexiune, şi va trimite către server un pachet de conectare de tip SYN, ce conţine *flag*-ul SYN = 1, valoarea iniţială a secvenţei, seq_client, este numărul ce identifică unic plăcuţa Arduino client, şi *flag*-ul ACK = 0.
3. Serverul primeşte pachetul de conectare şi va trimite înapoi un nou pachet de tip SYN-ACK, unde SYN = 1, valoarea secvenţei, seq_server, este numărul ce identifică unic plăcuţa Arduino server, şi ACK = seq_client + 1;
4. Clientul primeşte pachetul de conectare şi va trimite înapoi un pachet de tip ACK, unde SYN = 0, seq_client = ACK şi ACK = seq_server + 1

Prin intermediul acestor 4 paşi, se realizează o conexiune între client şi server. Putem observa că *flagurile* ce vor fi trimise sunt incrementate pe baza celor din pachetul primit la

pasul anterior. La fiecare pas, clientul cât şi serverul vor face verificările necesare astfel încât procedeul să fie respectat. Dacă unul dintre *flaguri* nu respectă regulile convenţiei, înseamnă ca a apărut o eroare în procedeul de conectare, metoda listen() având un mecanism de resetare prin care aşteaptă conectarea clienţilor în continuare. Metoda connect() nu dispune de un astfel de mecanism, ea întorcând o eroare în acest caz. Mai jos, în Figura 1, găsim o schemă a paşilor descrişi mai sus.

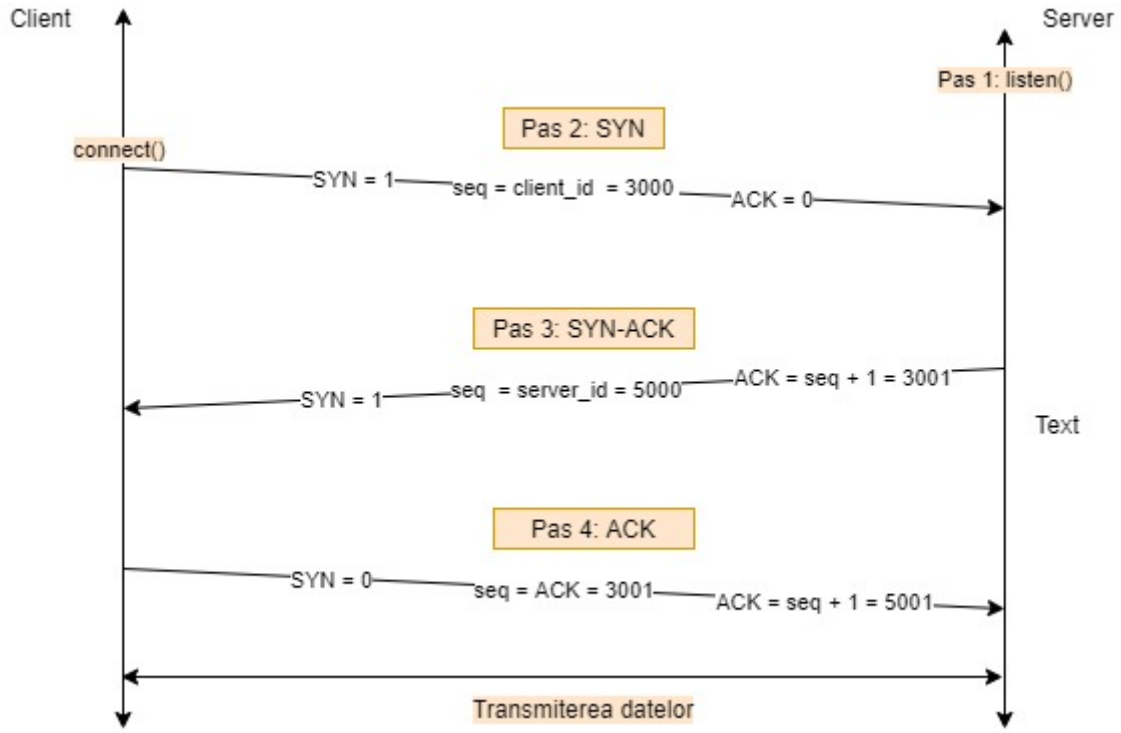


Figura 1 - Schema procedeului THW

1.3 Procedeul de încheiere a conexiunii

Similar procedeului de stabilire a conexiunii între client şi server, procedeul de încheiere a conectării între cele două părţi are loc prin transmitere de pachete, în 4 paşi, cu ajutorul a 2 *flaguri,* FIN (Finish) şi ACK (acknowledged). Paşii sunt similari cu cei de mai sus, utilizându-se *flagul* FIN în de SYN. Metodele publice ce trebuie să fie apelate sunt: clientClose() şi serverClose(). O schemă simplificată a procedeului se găseşte în Figura 2.

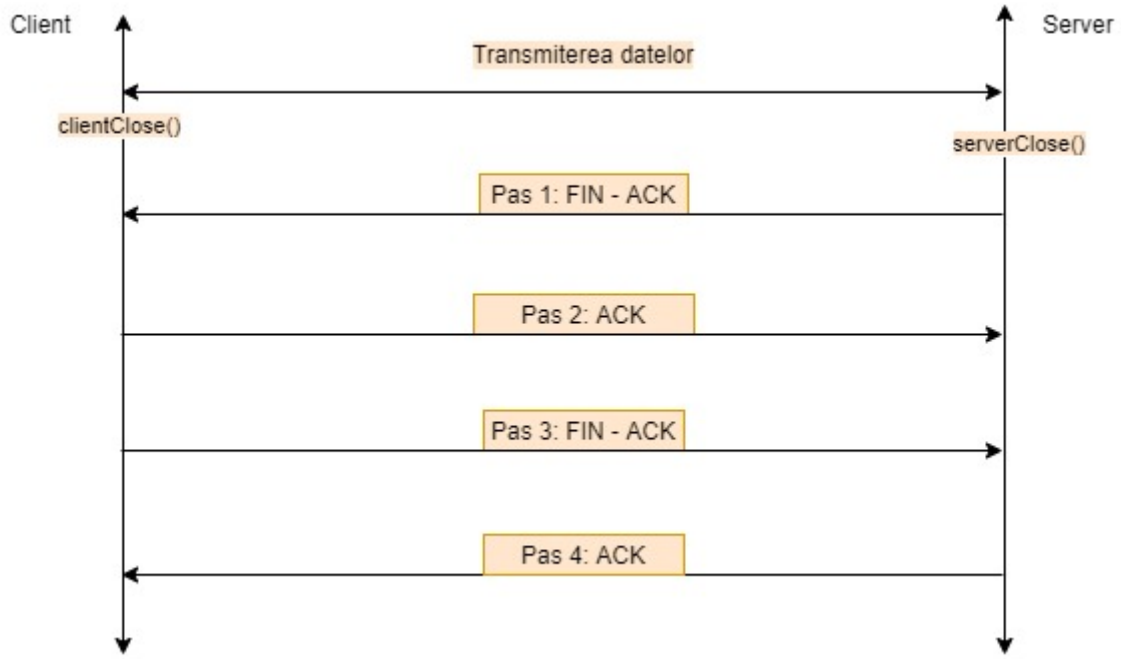


Figura 2 - Schema procedeului de încheiere a comunicării

1.4 Algoritmul lui Fletcher - Suma de control a lui Fletcher

Algoritmul lui Fletcher face parte din categoria algoritmilor pentru detectarea erorilor în cadrul comunicării seriale a mesajelor. În cursul transmiterii mesajelor de la expeditor la destinatar, acestea sunt transformate în secvenţe de biţi şi transportaţi unul câte unul. Modificare unui singur bit în timpul acestui proces din 0 în 1 sau invers, reprezintă o eroare de bit şi este inclus în categoria erorilor de transmitere. Astfel tot pachetul transmis este semnalat ca şi un pachet cu eroare, consecinţele fiind pierderea sensului mesajului şi propagarea de erori mai departe în sistem. Cauzele care pot provoca erorile de bit sunt de două feluri: software şi hardware. În cazul erorilor software, în subcapitolul anterior am vorbit despre memorie şi importanţa gestionării corecte a acesteia. Erorile în transmiterea biţilor pot apărea chiar înainte ca trimiterea datelor să aibă efectiv loc, acestea neputând fi „prinse” de programator. O cauză des întâlnită este depăşirea memoriei SRAM prin alocări dinamice excesive în memoria Heap şi depăşirea acesteia, ce automat conduce la un comportament impredictibil al programului şi la apariţia de erori. În cazul erorilor hardware, aici vom intra în domeniul telecomunicaţiilor unde este prezent efectul de zgomot, când semnalul electric transmis este alterat din motive externe, provocate de om sau de alte mecanisme ce induc modificări ale câmpului electromagnetic. Transferul biţilor este un concept teoretic, aceştia în practică transformându-se în semnale de intensităţi diferite care sunt trimise prin fir către

cealaltă plăcuţă. În cazul Arduino UNO, care funcţionează cu o intensitatea a curentului electric de 5V, bitul 0 este transformat într-un semnal de intensitate 0V, iar bitul 1 într-un semnal de intensitate 5V. Apariţia unei erori înseamnă modificarea intensităţilor de 0V sau 5V, în timpul transportului curentului electric, în altă intensitate pe care mai apoi protocolul de comunicare serială va încerca să-l interpreteze în 0V sau 5V, în acest mod existând posibilitatea apariţiei unei erori de bit.

1.4.1 Tipuri de erori şi detectarea acestora

Există trei tipuri de erori: erori singulare de bit, erori multiple de biţi şi erori în masă de biţi (trei sau mai mulţi biţi). Ultimul tip de eroare este cel mai dificil de detectat şi corectat, fiind unul des întâlnit în comunicarea serială. De asemenea, există diferiţi algoritmi de detecţie a erorilor: verificarea parităţii biţilor, verificarea redundanţei ciclice (CRC), şi sume de control.

Verificarea parităţii biţilor presupune adăugarea unui bit suplimentar la mesaj. Acest bit suplimentar este calculat astfel: se numără biţii de 1 (sau 0, depinde de abordare) din mesaj iar apoi se vede dacă aceştia sunt în număr par sau nu. Dacă sunt în număr par, bitul suplimentar va fi 1, altfel va fi 0. Acelaşi proces este repetat la destinatar, iar dacă bitul suplimentar are o valoare diferită, atunci se poate spune că a apărut o eroare. Acest algoritm este unul ce nu ar trebui folosit în practică, nefiind unul sigur, din motive lesne de înţeles.

Sumele de control sunt o metodă ce poate fi folosită în practică deoarece are rezultate foarte bune. Există diferiţi algoritmi ce folosesc acest concept, asemănător cu algoritmul de verificare a parităţii biţilor. Un astfel de algoritm este cel al lui Fletcher: Sumele de control ale lui Fletcher. Unul dintre proprietăţile acestui algoritm, a cărui implementare este în Tabela 1, este viteza de calcul a sumelor de control, vitală în transmiterea mesajelor. Algoritmul este unul simplu, pe care îl voi descrie mai jos: acesta va conţine două valori - sume de 8 biţi fiecare, ce vor fi calculate şi adăugate la mesajul iniţial. Prima sumă va reprezenta suma tuturor byţilor mesajului, modulo 255 (pentru a respecta lungimea maximă de 8 biţi, $2^8 - 1 = 255$). A doua sumă se va calcula simultan cu prima şi va fi suma tuturor valorilor pe care o va avea prima sumă, modulo 255.

```
int checkSum1 = 0, checkSum2 = 0, dataLength = strlen(data);

for (int index = 0; index < dataLength; index++) {
    checkSum1 = (checkSum1 + data[index]) % 255;
    checkSum2 = (checkSum2 + checkSum1) % 255;
}
```

```
checkSum1 %= 255;
data[dataLength + 1] = checkSum1;

checkSum2 %= 255;
data[dataLength + 2] = checkSum2;
```

Tabela 1 - Algoritmul lui Fletcher în C / C++

Noul mesaj cu cele două sume adăugate la cel inițial este transmis către destinatar, unde algoritmul lui Fletcher va fi din nou utilizat pentru a calcula din nou sumele de control. Dacă sumele de control de la destinatar coincid cu cele primit de la expeditor, înseamnă că transmiterea mesajului a avut loc cu succes, altfel avem o eroare ce trebuie corectată. Despre această problemă voi relata în subcapitolul următor.

1.4.2 Concluzie

Această bibliotecă utilizează algoritmul de detectare de erori al lui Fletcher în ambele moduri, atât cel de comunicare fără confirmare cât şi cel cu confirmare. Deoarece algoritmul este unul *light* şi viteza de calcul a sumelor este redusă, mi s-a părut o idee bună utilizarea acestuia în detrimentul algoritmului CRC care are rezultate similare cu acesta, dar cu o viteză de calcul mai ridicată. Mai multe detalii despre implementarea acestui algoritm le veţi putea găsi în capitolul următor: Descrierea soluţiei.

1.5 Coduri Hamming - Algoritmul de detecţie şi corectare de erori prin distanţa Hamming

Algoritmul de detecţie şi corectare de erori prin distanţă Hamming este un algoritm prin care se pot detecta (ca şi în cazul algoritmului lui Fletcher) dar şi corecta erorile apărute în procesul de transmitere al unui mesaj. Codul lui Hamming poate ajuta la detectarea a până la doi biţi greşiţi şi corectarea acestora. Ideea principală a acestui algoritm este de a insera în mesajul inițial, pe poziţii prestabilite, biţi de paritate denumiţi în cadrul acestui algoritm şi biţi redundanţi. Prin intermediul acestora se vor detecta, şi ulterior corecta eventualele erori apărute în procesul de transmitere.

Modalitatea de calcul a acestor biţi redundanţi dar şi raţionamentul matematic din spatele algoritmului, aplicate în această lucrare de licenţă, constituie subiectul acestui subcapitol. Acestea vor fi explicate şi exemplificate în următoarele rânduri.

1.5.1 Determinarea numărului de biţi redundanţi şi a poziţiei acestora

Să presupunem că avem un cuvânt iniţial, format dintr-o înşiruire de biţi 0 şi 1 din mulţimea $\{0, 1\}$, de lungime k. Primul pas în aplicarea algoritmului lui Hamming este determinarea numărului de biţi redundanţi m, ce vor fi adăugaţi cuvântului iniţial. Lungimea noului cuvânt obţinut este $n = k + m$. O singură eroarea apărută în noul cuvânt poate fi pe orice poziţie $1...n$. Două erori apărute pot fi pe oricare două poziţii $1...n$, numărul total de posibilităţi de poziţii al acestora fiind combinări de n luate câte 2, $\binom{n}{2}$. Generalizând, dacă avem între 1 şi e erori în total în noul cuvânt, numărul de configurări ale cuvântului nou cu erori posibile este $\sum_{i=1}^{e}\binom{n}{i}$. Numărul total de cuvinte noi ce pot fi construite cu m biţi redundanţi sunt 2^m, unde 2 reprezintă cardinalul mulţimii $\{0, 1\}$. Dacă vom considera că o poziţie din cele 2^m este corectă, rezultă relaţia cunoscută sub numele de **Marginea lui Hamming**, prin care vom calcula numărul de biţi de redundanţă ce trebuie adăugaţi la cuvântul iniţial.

$$2^{\mathrm{m}} - 1 \geq \sum_{i=1}^{e}\binom{n}{i}$$

Tabela 2 - Relaţia Marginea lui Hamming

Pentru calculul dinamic în cod al numărului m, vom utiliza o aproximare a termenului al doilea al inegalităţii de mai sus. Numărul minim de configurări al noului cuvânt este combinări de n luate câte 1, aşadar relaţia de mai sus se poate rescrie ca:

$$2^m - 1 \geq \sum_{i=1}^{e}\binom{n}{i} \geq \binom{n}{1} \rightarrow 2^m - 1 \geq \binom{n}{1} \rightarrow 2^m - 1 \geq n \rightarrow 2^m - 1 \geq m + k$$

Tabela 3 - Relaţia Marginea lui Hamming simplificată

Aşadar, pentru a găsi numărul de biţi redundanţi m vom folosi relaţia $2^m - 1 \geq k + m$, unde k este lungimea cuvântului iniţial.

În cazul aceste biblioteci, lungimea mesajului este de 16 byţi, rezultând lungimea cuvântului de 128 biţi. Numărul de biţi redundanţi va fi, conform relaţiei simplificate din Tabela 3 egal cu $m = 8$. Aceştia vor fi plasaţi pe poziţiile ce reprezintă puteri ale lui 2 (de

exemplu, 1, 2, 4, 8, 16, etc.) şi vor fi denumiţi biţii $m_1, m_{2,}, m_4, m_8, m_{16}$ etc. Lungimea noului mesaj va fi de 136 biţi, 17 byţi. (128 biţi ai mesajului iniţial + 8 biţi redundanţi).

1.5.2 Calculul valorilor biţilor redundanţi

Biţii de redundanţă sunt biţi de paritate şi se calculează pe baza biţilor din cuvântul iniţial. După inserarea acestora, se va crea un cuvânt nou care va avea forma următoare a biţilor: $m_1\, m_2\, k_3\, m_4\, k_5\, k_6\, k_7\, m_8\, k_9$. Putem vedea cum biţii redundanţi au fost inseraţi printre biţii k_i ai cuvântului iniţial k, pe poziţiile puteri ale lui 2.

Un exemplu al calculului valorii biţilor redundanţi este următorul:

- m_1 va fi bitul de paritate al biţilor ce în reprezentarea lor binară au 1 pe prima poziţie: (3, 5, 7, 9, 11, etc).
- m_2 va fi bitul de paritate al biţilor ce în reprezentarea lor binară au 1 pe a doua poziţie: (3, 6, 7, 10, 11, 12, etc).
- m_4 va fi bitul de paritate al biţilor ce în reprezentarea lor binară au 1 pe a treia poziţie: (5-7, 12-15, 20-23, etc).

În acest mod, fiecare bit k_i din cuvântul iniţial face parte din măcar un bit de redundanţă m_i, corectarea oricărui bit din cuvântul iniţial fiind astfel posibil.

1.5.3 Detectarea şi corectarea biţilor greşiţi

Pentru a detecta şi corecta biţii greşiţi, mecanismul ce se execută la destinatar va trebui să efectueze operaţiile în sens inverse celor de mai sus. Un prim pas este înlăturarea biţii redundanţi din mesajul primit, astfel obţinând mesajul original trimis de expeditor. Pe baza acestuia, urmează recalcularea biţilor redundanţi şi compararea lor cu biţii redundanţi primiţi de la expeditor care au fost înlăturaţi la pasul anterior. Dacă există o egalitate perfectă, înseamnă ca mesajul transmis nu conţine nicio eroare de bit. În cazul în care se identifică biţi redundanţi cu o valoare diferită faţă de biţii redundanţi primiţi de la expeditor, mecanismul va detecta o eroare de bit şi o va corecta în modul următor: se formează un număr în baza 2 din biţii redundanţi, iar valoarea transformată în baza 10 reprezintă poziţia bitului ce a cauzat eroarea. Acest lucru se întâmplă datorită legăturilor matematice create prin selecţia specială a biţilor mesajului care intră în calculul unui bit de redundanţă.

Spre exemplu dacă biţii redundanţi calculaţi la destinatar sunt 1, 1, 0, 1, rezultă că eroarea se află pe poziţia 13 ($1101_{(2)} = 13_{(10)}$). Odată ce ştim poziţia bitului greşit, trebuie doar să-l inversăm, aplicând operatorul Not, astfel corectându-l.

1.5.4 Concluzie

Această bibliotecă foloseşte algoritmul de detecţie şi corecţie a lui Hamming pentru corectarea mesajelor transmise de la o plăcuţă la alta. Algoritmul se aplică pe un cuvânt iniţial de lungime 128 de biţi şi este capabil să corecteze până la doi biţi de eroare. După corectarea a maxim doi biţi de eroare, se aplică algoritmul lui Fletcher, iar dacă sumele de control coincid cu cele trimise de expeditor, atunci pachetul primit este unul corect şi dublu verificat. În caz contrar, destinatarul nu poate corecta pachetul, cerând expeditorului retransmiterea acestuia.

Algoritmul lui Hamming de detecţie şi corecţie a mesajelor este un algoritm simplu de implementat cu o complexitate scăzută, rapiditatea acestuia fiind un factor de decizie important pentru care am optat să-l implementez în această bibliotecă. Detaliile despre implementarea acestuia le vom discuta în capitolul următor.

1.6 Exemplificarea detectării şi corectării unui mesaj transmis

În cadrul acestui capitol voi exemplifica detectarea şi corectarea unui mesaj ce este transmis de protocolul de comunicare. Pentru aceasta, vom lua în considerare utilizarea protocolului de comunicare cu confirmare, deoarece acesta foloseşte atât algoritmul de detecţie al lui Fletcher – **Suma de control a lui Fletcher** cât şi **Algorimtul de detecţie şi corectare de erori prin distanţă Hamming**. Să presupunem că expeditorul vrea să transmită către destinatar următorul mesaj de tip string: **„ABC”**.

Primul pas este calcularea sumelor de control cu ajutorul algoritmului lui Fletcher, descris în Tabela 1. Astfel, vom obţine **checkSum1 = 198** ((‚A’ + ‚B’ + ‚C’) % 255 = (65 + 66 + 67) % 255 = 198) şi **checkSum2 = 139** ((65 + (65+66) + (65+66+67)) % 255 = 394 % 255 = 139). Aceste sume vor fi transmise odată cu mesajul către destinatar.

Al doilea pas este calcularea numărului de biţi redundanţi ce vor fi adăugaţi la reprezentarea binară a mesajului **„ABC”.** Pe baza formulei din Tabela 3, deoarece lungimea

mesajului este 3 byţi, 24 biţi, numărul *m* de biţi redundanţi va fi egal cu 5. ($2^5 - 1 \geq 5 + 24$). Aşadar, lungimea noului mesaj va fi de 29 de biţi.

Reprezentarea binară a mesajului „ABC” fără biţi redundanţi:

01000001 – 01000010 – 01000011

Reprezentarea binară a mesajului „ABC” cu biţi redundanţi adăugaţi pe poziţiile puteri ale lui 2:

m1 m2 0 *m4* 1 0 0 *m8* – 0 0 0 1 0 1 0 *m16* – 0 0 0 1 0 0 1 0 – 0 0 0 1 1

Calculul valorilor biţilor redundanţi a fost explicat în subcapitolul anterior şi este realizat astfel:

- *m1* – paritatea biţilor de pe poziţiile 3, 5, 7, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 care sunt în ordine: 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1 -> 3 valori de 1 deci rezultă că *m1* = 1;
- *m2* – paritatea biţilor de pe poziţiile 3, 6, 7, 10, 11, 14, 15, 18, 19, 22, 23, 26, 27 care sunt în ordine: 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0 -> 3 valori de 1 deci rezultă că *m2* = 1;
- *m4* – paritatea biţilor de pe poziţiile 5, 6, 7, 12, 13, 14, 15, 20, 21, 22, 23, 28, 29, care sunt în ordine: 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1 -> 6 valori de 1 deci rezultă că *m4* = 0;
- *m8* – paritatea biţilor de pe poziţiile 9, 10, 11, 12, 13, 14, 15, 23, 24, 25, 26, 27, 28, 29 care sunt în ordine: 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1 -> 4 valori de 1 deci rezultă că *m8* = 0
- *m16* – paritatea biţilor de pe poziţiile 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29 care sunt în ordine: 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1 -> 4 valori de 1 deci rezultă că *m16* = 0;

Prin urmare, stringul care va fi transmis către destinatar va avea reprezentarea binară următoare: **11**00**1**00**0** – 00010**1**0**0** – 0010010 – 00011.

Pentru a arăta modul de detectare şi corectare a mesajului la destinatar, vom presupune că în cadrul transmisiei mesajului a apărut o eroare singulară de bit la poziţia 11, mesajul devenind astfel **„CBC”** din **„ABC”**. Aşadar bitului 11 i se va modifica valoarea din 0 în 1.

Prin urmare, destinatarul primeşte următoarea reprezentare binară: **11**001**0**0 – 00*1*1010**0** – 0010010 – 00011.

Următorul pas reprezintă recalcularea biţilor de redundanţă la destinatar pe baza celorlalţi biţi, proces asemănător cu cel de la expeditor, şi compararea acestora cu biţii redundanţi primiţi. Deoarece valoarea bitului 11 s-a schimbat din 0 în 1, rezultă că următorii biţi redundanţi nu vor mai fi la fel ca cei primiţi de la expeditor, deoarece în calculul lor intră şi bitul cu numărul 11: *m1*, *m2*, *m8*. Aşadar, deoarece exista măcar un bit de redundanţă diferit, înseamnă că avem o eroare în cadrul transmiterii de mesaj. Corectarea acesteia se realizează însumând indecşii biţilor redundanţi greşiţi (1 + 2 + 8), rezultând că eroarea a survenit la bitul de pe poziţia 11. Corectarea acesteia se face aplicând operatorul NOT pe acest bit.

Suplimentar, aplicând algoritmul de detecţie al lui Fletcher – **Suma de control a lui Fletcher,** vom observa că cele două sume de control sunt diferite, 200 respectiv 145, în loc de 198 şi 139. Astfel, avem un mecanism suplimentar de detecţie de erori în cadrul transmiterii mesajelor.

1.7 Biblioteca Arduino SoftwareSerial

Într-un capitol anterior am vorbit despre cum se realizează comunicarea serială la nivel hardware şi faptul că o plăcuţă Arduino are în mod implicit doar două porturi ce o permit, portul 0 şi 1. Biblioteca **SoftwareSerial** este una *open-source*, ce permite comunicarea serială prin orice port al unei plăcuţe Arduino, simulând prin software transferul de biţi ce se efectuează în mod normal la nivel hardware. Biblioteca este foarte folositoare atunci când vrei să interconectezi mai multe plăcuţe Arduino, acest subiect fiind unul dintre cele principale ale acestei lucrări de licenţă, motiv pentru care am şi ales să o intregrez în acest proiect. În continuare, vă voi arăta cum am folosit metodele bibliotecii în acest proiect.

După includerea bibliotecii în proiect, în clasa principală am creat un pointer de tip **SoftwareSerial**, deoarece doresc să iniţializez în mod dinamic instanţa clasei, pe baza datelor clientului. În constructorul clasei principale, care primeşte prin parametri porturile la care se doreşte să se realizeze transferul datelor, şi viteza de transfer a acestora (rata BAUD), am iniţializat pointerul creat cu o instanţă a clasei **SoftwareSerial,** iar apoi am „pornit-o” prin apelul metodei begin(rataBAUD). În Tabela 4, găsiţi fragmente de cod de iniţializare a bibliotecii:

```
SoftwareSerial *softwareSerial;
softwareSerial = new SoftwareSerial(rxPort, txPort);
softwareSerial->begin(beginSpeed);
```

Tabela 4 - Iniţializarea bibliotecii SoftwareSerial

Pentru a putea primi date prin intermediul bibliotecii, trebuie apelată metoda listen(). Rolul acesteia este de a schimba portul la care se ascultă, în cazul folosirii mai multor porturi de tip **SoftwareSerial,** deoarece o constrângere a bibliotecii este faptul că doar un singur port poate primi date la un moment dat**.** Acum că avem portul stabilit pentru a primi date, trebuie să vedem dacă sunt date în *bufferul* serial de primire al clasei. Pentru aceasta, vom apela metoda available(), care va returna numărul de caractere ce „aşteaptă" să fie citite. Dacă numărul de caractere este mai mare ca zero, putem apela metoda de citire, read(), care returnează un caracter citit. Pentru a scrie, se apelează funcţia write() ce primeşte ca parametru un pointer către o zonă de memorie unde se află un string. În Tabela 5 vedeţi metodele prin care se utilizează această bibliotecă.

```
softwareSerial->listen();

if (softwareSerial->available() > 0){
  char c = softwareSerial->read();
}

char writeBuffer[] = „Write Buffer Text".
softwareSerial.write(writeBuffer);
```

Tabela 5 – Utilizarea bibliotecii SoftwareSerial

1.7.1 Concluzie

Utilizarea bibliotecii *open-source* Arduino **SoftwareSerial**, mi-a uşurat şi permis multiplicarea porturilor prin care poate face comunicarea serială între mai multe plăcuţe Arduino. Datorită acesteia dar şi prin mecanismul implementat în biblioteca acestei lucrări, care permite identificarea plăcuţelor printr-un număr unic UAID, se poate comunica în cadrul aceluiaşi program între mai mult de două plăcuţe, după modelul client – server – client. Biblioteca este foarte uşor de utilizat, după cum aţi văzut metodele descrise mai sus.

2 Descrierea soluţiei

În acest capitol voi prezenta clasele principale împreună cu funcţiile pe care le-am implementat în biblioteca **TwoArduinoSerialCom**, prin intermediul cărora este posibilă comunicarea serială între două plăcuţe Arduino. Paradigma de programare utilizată în dezvoltarea acestei biblioteci este Programarea Orientată pe Obiecte care presupune lucrul cu clase şi obiecte din limbajul de programare C++. Principiile de bază ale Programării Orientate pe Obiecte folosite în această lucrare sunt încapsularea datelor, polimorfismul şi moştenirea. În primul subcapitol vom vorbi despre rolul fiecărui modul existent şi structura acestora. Al doilea subcapitol va fi despre gestionarea memoriei şi statistici ale bibliotecii, în timp ce în ultimul capitol voi prezenta diagrama arhitecturală UML a bibliotecii.

2.1 Modulele bibliotecii

2.1.1 Clasa ArduinoSerialCom

Deoarece biblioteca suportă două moduri de comunicare, comunicarea serială fără comunicare şi comunicare serială cu comunicare, codul pentru implementarea acestora deţine câteva funcţionalităţi comune pe care am decis să le implementez într-o clasă comună, de bază, numită **ArduinoSerialCom**. Membrii acestei clase sunt de tip *public* şi *protected* şi vor fi moşteniţi de către clasele **UdpProtocol** şi **TcpProtocol**. Metodele de tip *public* vor putea fi apelate în mod direct de către utilizatorii bibliotecii.

Membrii principali de tip *protected* ale acestei clase sunt:

- **Connection connection** – clasa Connection este una creată în cadrul bibliotecii şi se ocupă de starea conexiunii între expeditor şi destinatar;
- **Error error** – Clasa Error este una creată în cadrul bibliotecii şi se ocupă de tratarea erorilor din acest sistem;
- **HardwareSerial *hardwareSerial** – această clasă este una ce face partea din familia claselor Arduino prin care biblioteca va avea acces direct la un pointer de tip HardwareSerial ce va fi iniţializat în mod dinamic de către utilizator;

- **SoftwareSerial *softwareSerial** – acest pointer este de tip SoftwareSerial, clasă ce face parte din biblioteca **SoftwareSerial**, şi va fi iniţializat în mod direct de către utilizator;
- **bool SHOW_LOGS** – prin această variabilă booleană, utilizatorul poate stabili dacă vrea să vadă la consola serială logurile făcute de bibliotecă, în timpul transmiterii datelor. valoarea implicită este *false*;
- **bool ASYNC_MODE**– prin această variabilă booleană, utilizatorul poate stabili dacă vrea să utilizeze funcţia read în mod blocant sau non-blocant. valoarea implicită este *false*;
- **void softwareSerial_readBytes(char *data, int length)** – această metodă este una internă prin care se vor citi datele de lungime **length** de la portul serial stabilit şi se vor salva în pointer **data**;
- **void computeChecksum(char *data, int &checkSum1, int &checkSum2)** – prin această metodă se vor calcula cele două sume de control ale pachetului de date transmis atât în modul de comunicare fără confirmare cât şi cu confirmare;
- **bool hasPacketErrors(char *data, int _checkSum1, int _checkSum2)** – prin această metodă se va identifica la destinatar dacă pachetul conţine eroare prin compararea sumelor de control calculate la destinatar şi cele primite de la expeditor;
- **static void setUniqueArduinoIDToEEPROM(char *UAID) –** prin această metodă se va salva în memoria nevolatilă a plăcuţei Arduino un număr unic de identificare denumit de către mine în cadrul sistemului **UAID** (Unique Arduino Identifier)**;**
- **char specialChr[2] = "\f";**

Membrii principali de tip *public* ale acestei clase şi care sunt disponibile şi utilizatorului, sunt:

- **void initializePorts(int rxPort, int txPort) -** prin această metodă se iniţializează de către utilizator**,** porturile de comunicare serială ale obiectului **SoftwareSerial;**
- **void initializeSerial(HardwareSerial &Serial, int beginSpeed, int timeout), void initializeSerial(HardwareSerial &Serial, int beginSpeed); –** aceasta

este metoda publică care configurează în cadrul bibliotecii portul serial la care se vor afişa eventualele erori şi instanţiază obiectul serial**.** Argumentul **beginSpeed** reprezintă rata BAUD cu care vrem să se efectueze comunicarea serială. Această metodă vine în patru versiuni cu ajutorul utilizării conceptului de supraîncărcare. Valoarea în mod implicit pentru parametrul funcţiei ce lipseşte este: **timeout** = 1000;

- **void printLastError() –** această metodă va afişa la portul serial stabilit de către utilizator prin metoda **initializeSerial**, eroarea de tip **Error** ce reprezintă o eroare în cadrul procesului de transmitere a informaţiei;
- **const char *getConnectionStatus() -** prin această metodă publică, utilizatorul poate afla care este starea conexiunii sistemului. Funcţia va returna o reprezentarea string de tip **Connection**;
- **static int getUniqueArduinoIDFromEEEPROM() –** această metodă va returna numărul unic UAID al plăcuţei Arduino. În cazul în care plăcuţă nu are salvată în memoria EEPROM un UAID, funcţia va sesiza acest lucru şi va genera un UAID random de patru cifre pe care îl va transmite ca parametru metodei **setUniqueArduinoIDToEEPROM.**

2.1.2 Clasa Connection

În această clasă se găsesc constantele ce definesc tipul unei conexiuni în cazul comunicării cu sau fără confirmare. În Tabela 6, se află valorile pe care le poate avea o conexiune. În mod implicit, tipul conexiunii este **DISCONNECTED**. Scopul acestei clase este de a marca starea unei conexiuni, clasa având un membru *private* de tip **connectionStatus**, prin intermediul căruia s-a implementat un mecanism de restricţionare a utilizării funcţiilor. Spre exemplu, în modul de comunicare cu confirmare, scrierea sau citirea nu se pot realiza decât dacă tipul conexiunii este **CONNECTED**. Acest lucru presupune în prealabil apelarea metodei de conectare, care dacă se realizează cu succes, modifică starea conexiunii din **DISCONNECTED** în **CONNECTED.**

```
enum connectionStatus {
    ERROR,          // eroare în sistem, ex: în mecanismul de conectare
    DISCONNECTED,
    FINISHED,
    CONNECTED,
  };
```

Tabela 6 - Stările conexiunii

2.1.3 Clasa Error

Clasa **Error** se ocupă de erorile ce pot apărea în această bibliotecă. Aceasta conţine două constante de tip enumerabil: **errorMessages**, în care se găsesc toate mesajele de eroare ce pot fi afişate către utilizator, şi **errorCodes**, care sunt nişte coduri de eroare în strânsă legătură cu tipul conexiunii. Spre exemplu, dacă utilizatorul doreşte să scrie un mesaj prin funcţia **write**, dar anterior nu a fost apelată metoda de conectare (starea conexiunii este de tip **DISCONNECTED)**, funcţia **write** va returna o valoare int de tip **errorCodes** (**Error::DISCONNECTED** = -2). Valorile celor două constante enumerabile sunt:

```
enum errorMessages {
    CONNECT_PROTOCOL_NOT_CONNECTED_ERROR,
    CONNECT_PROTOCOL_PROTOCOL_HAS_FINISHED,

    WRITE_PROTOCOL_NOT_CONNECTED_ERROR,
    READ_PROTOCOL_NOT_CONNECTED_ERROR,
    CLOSE_PROTOCOL_NOT_CONNECTED_ERROR,

    LISTEN_INTERNAL_ERROR,
    CONNECT_INTERNAL_ERROR,
    WRITE_INTERNAL_ERROR,
    READ_INTERNAL_ERROR,
    CLIENT_CLOSE_INTERNAL_ERROR,
    SERVER_CLOSE_INTERNAL_ERROR,
  };

enum errorCodes {
    ERROR = -3,
    DISCONNECTED = -2,
    FINISHED = -1,
    CONNECTED = 1,
    CLOSED_CONN = 1,
};
```

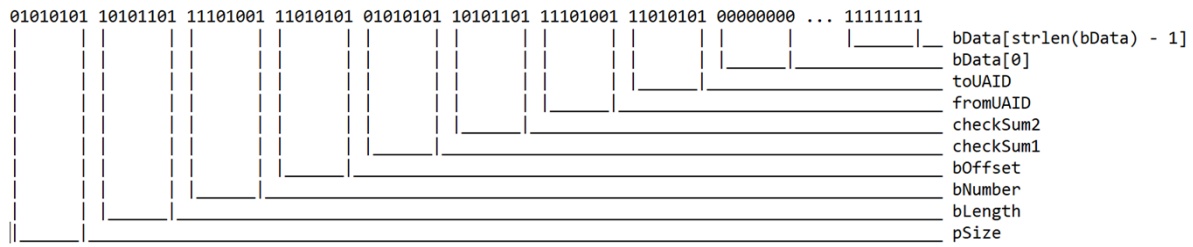
Metodele de tip *public* ale clasei sunt:

- **void setError(errorMessages _error);**
- **char const* getError()** – returnează o valoare string a valorilor de tip **errorMessages**.

2.1.4 Clasa UdpPacket

Aşa cum am menţionat în primul capitol al acestei lucrări, transmiterea mesajului de la destinatar către expeditor se realizează prin împărţirea mesajului iniţial în blocuri de câte 16 byţi. Un astfel de bloc va reprezenta corpul unui pachet ce va fi trimis către destinatar. Clasa **UdpPacket** defineşte structura unui pachet din cadrul protocolului de comunicare fără confirmare. Dimensiunea totală a unui astfel de pachet este de 49 de byţi, din care antetul

pachetului reprezintă 33 de byţi, iar corpul acestuia 16 byţi. În antetul pachetului se află informaţii despre acesta precum lungimea totală (49 byţi), lungimea corpului pachetului (16 byţi), numărul total de pachete ce vor fi transmise, indexul pachetului curent ce este transmis, cele două sume de control calculate pe baza corpului, UAID-ul expeditorului şi UAID-ul destinatarului. În Tabela 7 , puteţi vedea schema unui pachet de tip UdpPacket cât şi membrii acestei clase.




```
int pSize; // dimensiunea pachetului în byţi
int bLength; // lungimea bData
int bNumber; // numărul total de pachete
int bOffset; // indexul pachetului curent
int checkSum1;
int checkSum2;
int fromUAID; // UAID-ul expeditorului
int toUAID; // UAID-ul destinatarului
char *bData; // bloc de 16 byţi din mesajul iniţial

enum blockInformation {
  BLOCK_SIZE = 49,
  BLOCK_HEADER_SIZE = 33,
  BLOCK_BODY_SIZE = 16
};
```

Tabela 7 - Schema pachetului UdpPacket

2.1.5 Clasa UdpProtocol

Clasa **UdpProtocol,** ce moşteneşte clasa **ArduinoSerialCom**, implementează protocolul de comunicare fără confirmare, unde stabilirea conexiunii nu este necesară înainte de a se începe transmiterea pachetelor. Metodele de tip *public* ale acestei clase sunt:

- **int write(char *dataToSend, int toUAID);**
  int write(char *dataToSend, int fromUAID, int toUAID) – metoda principală de scriere a mesajelor către destinatar.
- **int read(char *dataToReceive, int &fromUAID);**
  int read(char *dataToReceive, int &fromUAID, int &toUAID) – metoda principală de citire a mesajelor trimise de către expeditor. Această funcţie are două comportamente în funcţie de variabila booleană **ASYNC_MODE**:

blocantă şi non-blocantă. În cazul în care **ASYNC_MODE** este *true*, funcţia va fi non-blocantă şi va returna imediat 0, dacă pe portul serial stabilit nu este primită nicio informaţie, altfel va returna numărul de caractere de citit.

În continuare, voi detalia mecanismul de scriere şi citire a mesajelor. Funcţia **write** primeşte ca argument un pointer de tip char, reprezentând mesajul ce trebuie trimis, şi două argumente, id-ul plăcuţei Arduino de unde se trimite mesajul şi id-ul plăcuţei Arduino către care este destinat mesajul. Funcţia returnează o valoarea de tip **int,** ce reprezintă, în caz că scrierea a fost realizată cu succes, numărul de byţi scrişi, altfel un cod de tip **Error,** descris în subcapitolul anterior, **Clasa Error.**

Implementarea funcţiei presupune segmentarea mesajului din **dataToSend**, în bucăţi de câte 16 byţi, şi popularea unui obiect de tip UdpPacket cu informaţiile necesare. Spre exemplu dacă funcţia se apelează în acest mod: `write(„Ana are mere si Ion pere”, 1234, 5678);` deoarece lungimea mesajului strlen(„Ana are mere si Ion pere”) = 24 byţi, se vor transmite succesiv două pachete, unul de 16 byţi şi altul de 8 byţi, de la expeditor către destinatar. Forma pachetelor va fi următoarea:

```
UdpPaacket packet;

// Pachetul 0
packet.pSize = 49; // dimensiunea pachetului în byţi
packet.bLength = 16; // lungimea bData
packet.bNumber = 2; // numărul total de pachete
packet.bOffset = 0; // indexul pachetului curent
packet.checkSum1 = 232;
packet.checkSum2 = 21;
packet.fromUAID = 1234; // UAID-ul expeditorului
packet.toUAID = 5678; // UAID-ul destinatarului
char *bData = „Ana are mere si ; // bloc de 16 byţi din mesajul iniţial

// Pachetul 1
packet.pSize = 49; // dimensiunea pachetului in byţi
packet.bLength = 8; // lungimea bData
packet.bNumber = 2; // numărul total de pachete
packet.bOffset = 1; // indexul pachetului curent
packet.checkSum1 = 153;
packet.checkSum2 = 89;
packet.fromUAID = 1234; // UAID-ul expeditorului
packet.toUAID = 5678; // UAID-ul destinatarului
char *bData = „Ion pere ; // bloc de 16 byţi din mesajul iniţial
```

După popularea unui astfel de pachet, acesta va fi formatat şi trimis prin portul serial către destinatar. Formatarea presupune crearea unui string prin concatenarea elementelor pachetului **packet** separate printr-un caracter special. Caracterul special ales de mine este caracterul de control `\f`, care nu reprezintă un caracter text (ce poate fi scris de la tastatură).

Aşadar în urma concatenării elementelor, stringul ce va fi transmis arată în felul următor. Pentru lizibilitate, am înlocuit aici caracterul de control `\f,` cu caracterul text , (virgulă).

```
49,16,2,0,232,21,1234,5678,Ana are mere si  // Pachetul 0 formatat

49,8,2,1,153,89,1234,5678,,,,,,,,,,,,,,,Ion pere // Pachetul 1 formatat
```

În cazul în care lungimea blocului extras din **dataToSend**, este mai mică de 16 byţi, atunci diferenţa va fi suplimentată prin adăugare de caractere de control, astfel încât lungimea totală să fie persistentă la toate pachetele. Am decis să procedez în acest fel deoarece funcţia **read** va extrage din stringul formatat datele necesare pe baza caracterului special şi va popula un obiect de tip UdpPacket, acest proces nefiind unul complex de implementat. Aşadar, funcţia read se va ocupa de primirea pachetelor formatate în acest mod, şi de reasamblarea lor.

În acest fel, transferul de date are loc între destinatar şi expeditor, prin împărţirea pachetelor în blocuri de dimensiune fixă, oferind astfel viteză de transmitere şi siguranţa integrităţii datelor.

2.1.6 Clasa TcpConnectionPacket

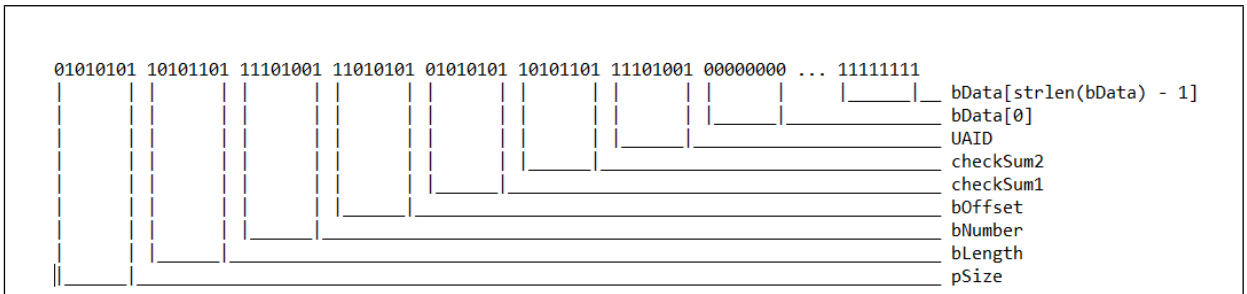
Clasa **TcpConnectionPacket** este similară cu clasele **UdpPacket** şi **TcpPacket**. Aceasta conţine membri necesari procesului de conectare între plăcuţele Arduino, descris în primul capitol al aceste lucrări, **Abordare teoretică**. Totodată, această clasă este folosită pentru trimiterea / primirea confirmării faptului că pachetul de tip **TcpPacket** a fost primit cu succes. Membrii clasei sunt:

```
int syn;
int seq;  // /ACK:0, SYN:1 CON:2 FIN:3 ERR:9
int ack;

enum blockInformation {
  BLOCK_SIZE = 12,
};
```

2.1.7 Clasa TcpPacket

Clasa **TcpPacket** se aseamănă ca rol şi funcţionalitate cu clasa **UdpPacket** şi este folosită pentru modul de comunicare cu confirmare. Diferenţele între cele două clase sunt lipsa campului f**romUAID**, deoarece acesta este deja cunoscut de către destinatar în urma procesului de conectare, dar şi mărirea lungimii corpului pachetului cu 1 byte, din cauza adăugării celor 8 biţi redundanţi ai algoritmului de detecţie şi corectare de erori prin distanţa Hamming.

Schema clasei **TcpPacket** cât şi membrii acesteia se găsesc în Tabela 8:




```
int pSize; // dimensiunea pachetului în byţi
int bLength; // lungimea bData
int bNumber; // numărul total de pachete
int bOffset; // indexul pachetului curent
int checkSum1;
int checkSum2;
int UAID; // UAID-ul destinatarului
char *bData; // bloc de 16 byţi din mesajul iniţial

enum blockInformation {
  BLOCK_SIZE = 46,
  BLOCK_HEADER_SIZE = 29,
  BLOCK_BODY_SIZE = 17
};
```

Tabela 8 - Schema pachetului TcpPacket

2.1.8 Clasa TcpProtocol

Clasa **TcpProtocol,** ce moşteneşte proprietăţile clase **ArduinoSerialCom**, implementează modul de comunicare cu confirmarea, ce reprezintă cea mai mare parte a funcţionalităţii acestei biblioteci. Principalele mecanisme implementate aici sunt cel de interconectare între plăcuţe în trei paşi, scriere şi citire a pachetelor cu confirmare de primire, dar şi cel de încheiere a conexiunii între cele două părţi. Voi începe cu structura clasei, apoi voi exemplifica şi câteva idei din implementările metodelor acestei clase.

Metodele de tip *public* ale acestei clase sunt:

- **int listen()** – metodă folosită de către plăcuţa Arduino, ce va juca rol de server în acest protocol de comunicare**;** valoarea returnată va fi UAID-ul plăcuţei care s-a conectat;
- **int connect()** – metodă folosită de către plăcuţă Arduino, ce va juca rol de client, pentru a se conecta către alt Arduino ce va juca rol de server**;**
- **int write(char *dataToSend, int UAID);**
- **int read(char *dataToReceive, int &UAID);**

- **int clientClose() –** metodă folosită de către client pentru a notifica serverul că conexiunea va lua sfârşit**;**
- **int serverClose() –** metodă folosită de server pentru a asculta notificarea clientului de sfârşire a conexiunii**;**

Membrii principali de tip *protected* ai acestei clase sunt:

- **TcpPacket packetRead;**
- **TcpPacket packetWrite;**
- **TcpConnection packetConnectionRead;**
- **TcpConnection packetConnectionWrite;**
- **char **orderedPackets –** o matrice ce va fi folosită pentru salvarea mesajelor extrase din pachete, şi ordonarea acestora**;**
- **char *dataSendEncodedString –** stringul rezultat în urma codificării prin algoritmul lui Hamming**;**
- **void encodeWithHammingDistanceCode(char *dataSendString);**
- **void decodeWithHammingDistanceCode(char* dataSendEncodedString);**
- **void baseTwoToChar(int *bits, int length, char *str);**
- **void charToBaseTwo(char *str, int *bits);**
- **bool isPowerOfTwo(int number);**

Procesul de creare a pachetelor şi trimitere a acestora de la destinatar către expeditor este identic ca în cazul implementării descrise în subcapitolul **Clasa UdpProtocol,** singurele diferenţe fiind codificarea câmpului bData, reordonarea pachetelor trimise la destinatar şi mecanismul de confirmare de primire a pachetelor cu succes.

Funcţia **write** primeşte ca argument un pointer de tip char, reprezentând mesajul ce trebuie trimis, şi id-ul plăcuţei Arduino către care este destinat mesajul. Funcţia returnează o valoarea de tip **int,** ce reprezintă, în caz că scrierea a fost realizată cu succes, numărul de byţi scrişi, altfel un cod de tip **Error,** descris în subcapitolul anterior, **Clasa Error.** În această, funcţie, este implementat mecanismul de trimite cu confirmare a pachetelor. Acesta funcţionează în felul următor: se trimite un pachet către destinatar iar destinatarul în caz că a reuşit să-l decodeze cu succes prin algoritmul cu distanţe al lui Hamming, trimite înapoi un pachet de tip conexiune ce va conţine numărul pachetului decodat plus unu. Astfel,

expeditorul va putea şti dacă destinatarul a primit pachetul corect, şi va putea să transmită următorul pachet, sau în caz contrar, îl va trimite din nou pe acelaşi.

În continuare, în Tabela 9, vă voi arăta fragmente din codul ce reprezintă mecanismul de trimitere cu confirmare a pachetelor.

```
SENT_ACK = false;
while (SENT_ACK == false) {
    softwareSerial->write(packet, strlen(packet));

    waitRead();
    softwareSerial_readBytes(bDataConnection, bDataConnectionLength);
    formatReceiveConnectionData(bDataConnection);

    if (packetConnectionRead.ack == packetWrite.bOffset + 1
    && packetConnectionRead.syn == 1) { // dacă am primit confirmarea

        SENT_ACK = true; // trecem la pachetul următor, altfel rămânem în
bucla while şi trimitem din nou acelaşi pachet.
        free(packetWrite.bData);

      }
}
```

Tabela 9 - Mecanismul de confirmare al mesajelor transmise

Funcţia **read**, pe lângă decodarea pachetului cu ajutorul algoritmului cu distanţe al lui Hamming, mai are şi rolul de a reordona pachetele. Pentru aceasta, am folosit o matrice **orderedPackets**, în care voi insera fiecare string **bData** din pachet pe poziţia **bOffset**. Astfel, evităm folosirea unui algoritm de sortarea, aceasta realizându-se în mod natural. Matricea **orderedPackets** se alocă în mod dinamic în momentul primirii primului pachet, în funcţie de lungimea acestuia. În Tabela 10, se află codul de alocare dinamică şi de salvare a datelor în matrice.

```
if (packetRead.bOffset == 0) {
    orderedPackets = (char **)malloc(sizeof(char *) * packetRead.bNumber);
    for (int index = 0; index < packetRead.bNumber; index++) {
      orderedPackets[index]    =    (char    *)malloc(sizeof(char    *)    *
packetRead.bLength + 1);
      memset(orderedPackets[index], '\0', sizeof(char) * packetRead.bLength
+ 1);
    }
  }

strcpy(orderedPackets[packetRead.bOffset], packetRead.bData);
```

Tabela 10 - Alocare dinamică şi folosirea matricei orderedPackets

2.2 Gestionarea memoriei

Gestionarea memoriei în cadrul oricărei biblioteci Arduino are un rol foarte important, deoarece aşa cum am prezentat în primul capitol al acestei lucrări, capacitatea memoria utilizabilă SRAM este foarte redusă, de doar 2KB. Din cauza acestei limitări, biblioteca Arduino trebuie foarte bine optimizată din punctul de vedere al memoriei, deoarece cei 2KB sunt împărţiţi între alocări statice de date, alocările dinamice (*Heap*) dar şi alocările variabilelor locale la nivelul funcţiilor (*Stack*). Majoritatea problemelor de memorie apar atunci când memoriile utilizate de *Stack* şi *Heap* încep să se suprapună, astfel apărând conflicte şi eventual erori în comportamentul programului. În cazul alocărilor dinamice de memorie şi a dealocării acesteia când nu mai avem nevoie de ea, acel spaţiu utilizat nu poate fi refolosit de către *Stack*, şi uneori, datorită fragmentării prea mari a acesteia nu poate fi refolosit nici de *Heap*. Pentru a rezolva această problemă de suprapunere, există diferite metode prin care se poate reduce considerabil memoria SRAM:

- ştergerea fragmentelor de cod nefolosite: funcţii, variabile, bucăţi de cod care nu vor fi niciodată executate;
- eliminarea codului duplicat şi utilizarea funcţiilor generale care să îndeplinească multiple funcţionalităţi;
- folosirea *macroului* F() prin care mutăm în memoria Flash PROGMEM, variabilele statice din memoria SRAM pentru că nu le vom rescrie niciodată, aşadar reprezintă o risipă de memorie SRAM.
- alocările vectorilor să nu ocupe mai multă memorie decât este necesar
- înlocuirea variabilelor supradimensionate cu cele de dimensiune potrivită (de exemplu variabilele **int** care ocupă 4 byţi să fie înlocuite cu **byte** sau **uint8_t** ce ocupă 1 byte), acolo unde este cazul.

În cadrul dezvoltării acestei biblioteci, deoarece am folosit algoritmi care au avut nevoie de memorie, de exemplu **Algoritmul de detecţie şi corectare de erori prin distanţa Hamming**, am încercat pe cât de mult posibil să respect indicaţiile de mai sus şi astfel să obţin un cod cât mai optimizat. În marea majoritate a codului, am folosit alocarea dinamică a unei resurse cu ajutorul metodelor **malloc** şi **memset**, iar după ce am terminat utilizarea acelei resurse, am eliberat-o prin metoda **free.**

În Figura 3 de la pagina 37 , se află un exemplu de memorie utilizată de un program ce implementează biblioteca TwoArduinoSerialCom în modul de comunicare cu confirmare şi care joacă rol de server, având două instanţe obiect ale clasei TcpProtocol, adică doi clienţi ce se vor conecta. Observăm că sketchul nostru împreună cu biblioteca utilizează 1082 byţi din 2048 byţi disponibili, însă, în acest *sketch* a fost declarată o variabilă `char dataToReceive[300];` care ocupă 300 byţi.

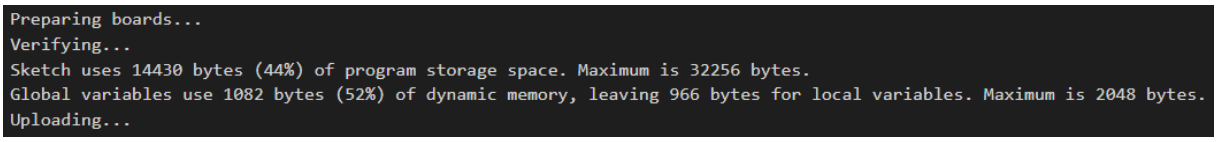


Figura 3 - Memoria utilizată de un *sketch* TwoArduinoSerialCom

În concluzie, dimensiunea bibliotecii ocupă în medie, în jur de 40-50% din memoria SRAM a plăcuţei Arduino, ceea ce din punctul meu de vedere este un procentaj bun, lăsând spaţiu de flexibilitate şi utilizatorului pentru implementarea funcţionalităţilor proprii.

2.3 Arhitectura proiectului – Diagrama UML

În Figura 4 găsim principalele clase ale acestei biblioteci: **UdpProtocol** şi **TcpProtocol** ce moştenesc proprietăţile clasei de bază **ArduinoSerialCom**.

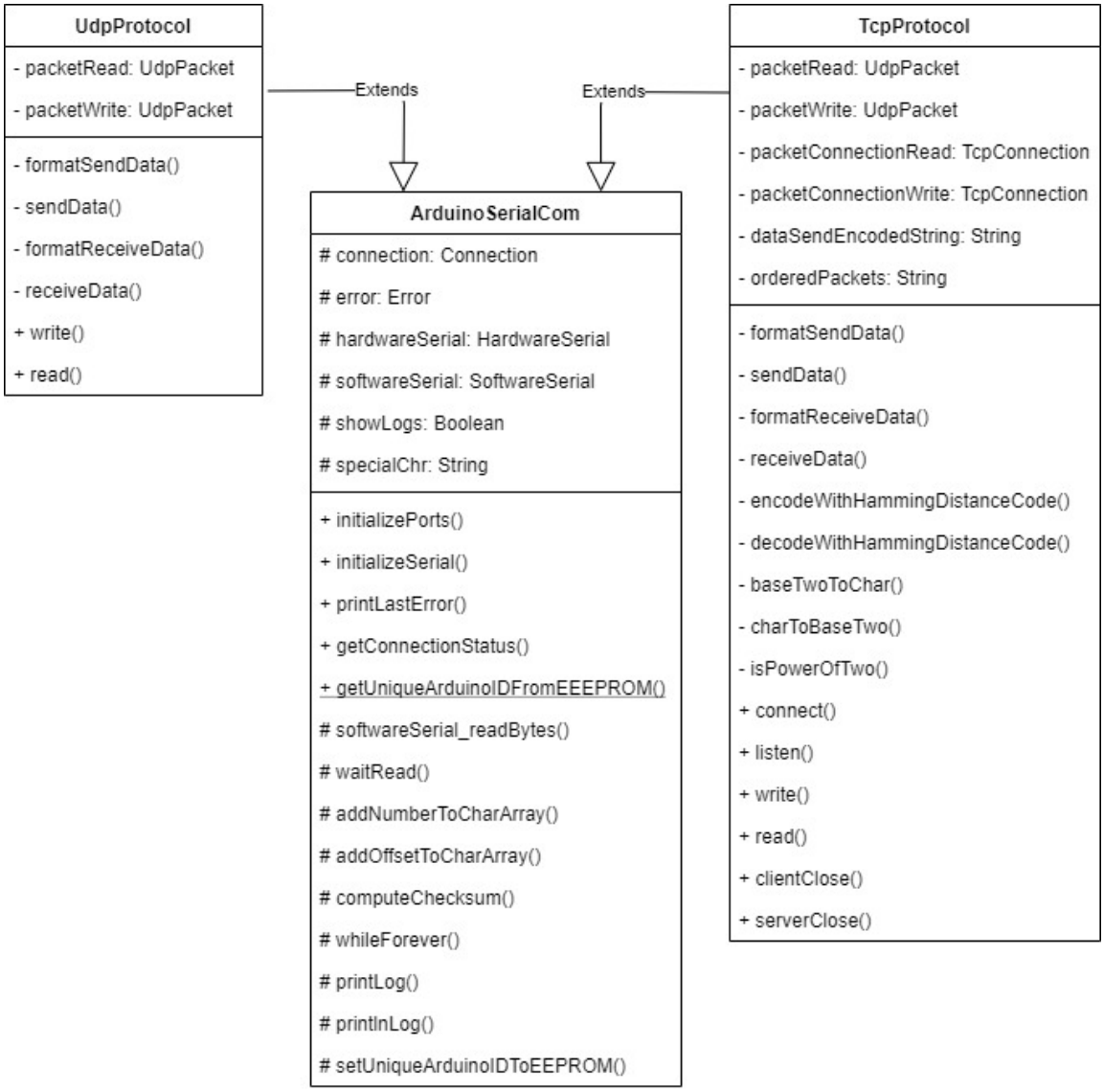
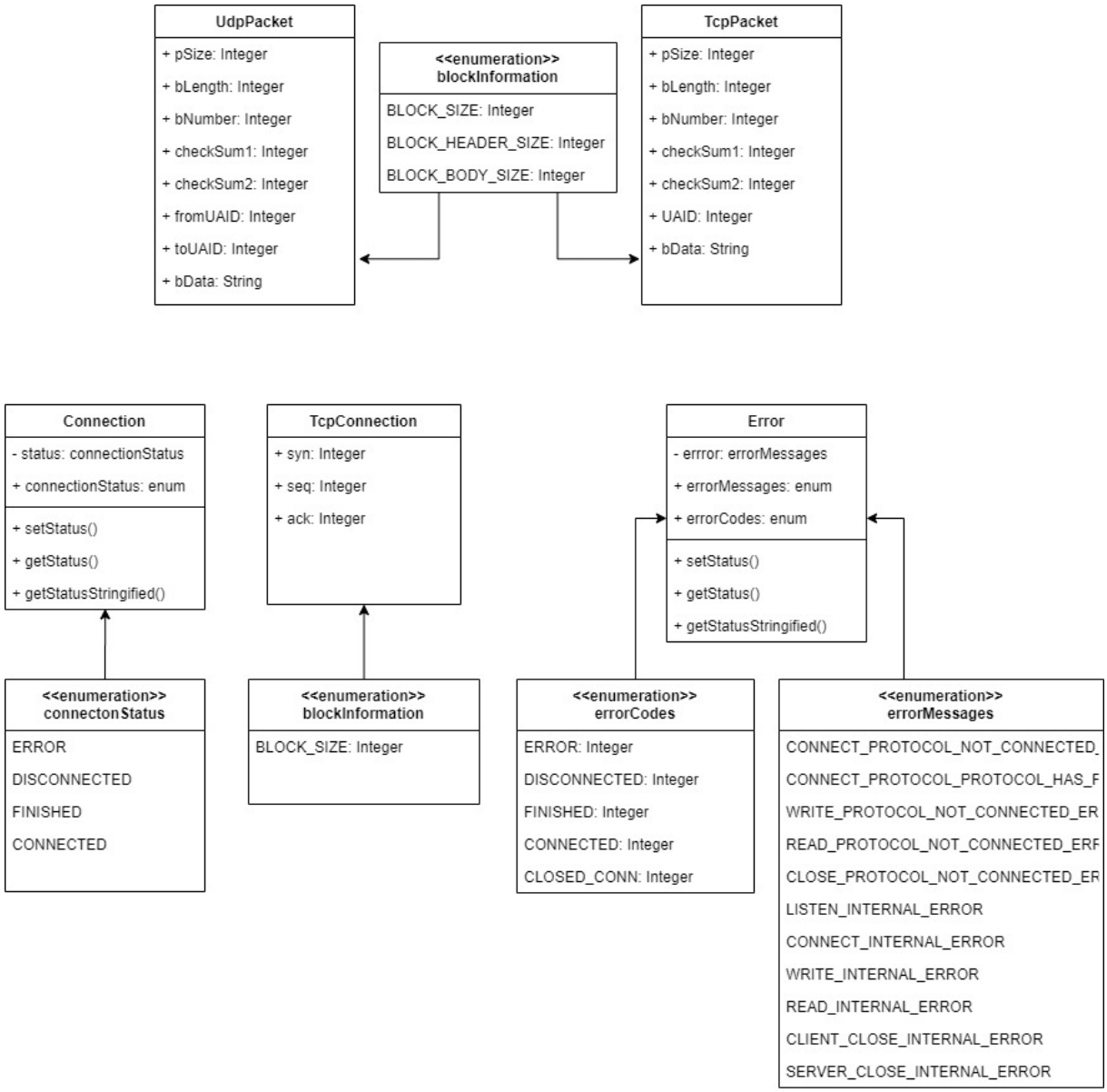


Figura 4 – Schema UML a claselor principale

În se află structura celorlalte clase din bibliotecă: **Udpacket**, **TcpPacket**, **Connection**, **Error**, **TcpConnection**. Aceste clase sunt utilizate în cadrul claselor din Figura 4.


UdpPacket
+ pSize: Integer
+ bLength: Integer
+ bNumber: Integer
+ checkSum1: Integer
+ checkSum2: Integer
+ fromUAID: Integer
+ toUAID: Integer
+ bData: String
<<enumeration>>
blockInformation
BLOCK_SIZE: Integer
BLOCK_HEADER_SIZE: Integer
BLOCK_BODY_SIZE: Integer
TcpPacket
+ pSize: Integer
+ bLength: Integer
+ bNumber: Integer
+ checkSum1: Integer
+ checkSum2: Integer
+ UAID: Integer
+ bData: String
Connection
- status: connectionStatus
+ connectionStatus: enum
+ setStatus()
+ getStatus()
+ getStatusStringified()
TcpConnection
+ syn: Integer
+ seq: Integer
+ ack: Integer
Error
- errror: errorMessages
+ errorMessages: enum
+ errorCodes: enum
+ setStatus()
+ getStatus()
+ getStatusStringified()
<<enumeration>>
connectonStatus
ERROR
DISCONNECTED
FINISHED
CONNECTED
<<enumeration>>
blockInformation
BLOCK_SIZE: Integer
<<enumeration>>
errorCodes
ERROR: Integer
DISCONNECTED: Integer
FINISHED: Integer
CONNECTED: Integer
CLOSED_CONN: Integer
<<enumeration>>
errorMessages
CONNECT_PROTOCOL_NOT_CONNECTED_
CONNECT_PROTOCOL_PROTOCOL_HAS_F
WRITE_PROTOCOL_NOT_CONNECTED_ER
READ_PROTOCOL_NOT_CONNECTED_ERF
CLOSE_PROTOCOL_NOT_CONNECTED_ER
LISTEN_INTERNAL_ERROR
CONNECT_INTERNAL_ERROR
WRITE_INTERNAL_ERROR
READ_INTERNAL_ERROR
CLIENT_CLOSE_INTERNAL_ERROR
SERVER_CLOSE_INTERNAL_ERROR

2.4 Medii de testare ale bibliotecii TwoArduinoSerialCom

Biblioteca **TwoArduinoSerialCom** este una ce implementează comunicarea serială între plăcuţe de tip Arduino la nivel hardware, prin intermediul firelor. Datorită acestui fapt, a fost necesar să testez biblioteca în diferite medii de testare, cu diferite cabluri, astfel încât să pot vedea dacă apar probleme de transmisie între cele două plăcuţe. În realizarea dezvoltării bibliotecii, pentru a comunica între două plăcuţe Arduino, am utilizat două cabluri scurte de 9 cm (specifice Arduino), unul pentru scrierea / transmiterea datelor şi unul pentru citirea / primirea acestora. Experimentul de testare a constat în utilizarea a 4 cabluri diferite, prin lungime şi material: fire specifice Arduino de 9 cm şi 20 cm din cupru liţat, 2 fire dintr-un cablu USB de 60 cm din cupru liţat şi 2 fire torsadate dintr-un cablu de internet tip CAT5 de lungime 600 cm din aluminiu cuprat. Codul conţine o buclă infinită unde se transmite un mesaj constant de lungime 72 de caractere, cod ce a fost rulat pe rând pe toate cele 4 tipuri de cablu,: „Salut, eu sunt Andrei si sunt student la Facultatea de Informatica Iasi!”; primirea aceluiaşi mesaj înapoi de la destinatar şi măsurarea timpului între transmitere şi primire. Bucla infinită ne permite sa vedem cu exactitate cum fluctuează timpii de-a lungul a mai multe transmisii succesive.

Codul prin care am realizat acest test se găseşte în Tabela 11, şi utilizează modul de comunicare cu confirmare, **TcpProtocol**, viteza de transmisie a datelor fiind 9600 BAUD:

```
float startTime = 0, endTime = 0;

void loop() {
    if (tcpProtocol.connect() < 0) {
      tcpProtocol.printLastError();
    }

    while (1) {
      startTime = millis();
      if ((length = tcpProtocol.write(dataToSend, thisUAID)) < 0) {
        tcpProtocol.printLastError();
      }

      if ((length = tcpProtocol.read(dataToReceive, destinationUAID)) < 0){
        tcpProtocol.printLastError();
      }
      endTime = millis();
      Serial.println("DONE");
      Serial.println(dataToReceive);
      Serial.println(endTime - startTime);
    }

    if (tcpProtocol.clientClose() < 0) {
      tcpProtocol.printLastError();
    };
}
```

Tabela 11 – Codul Arduino expeditor

În cealaltă parte, la destinatar codul arată similar (buclă infinită, citire string, scriere string) dar fără măsurarea timpului. Rezultatele obţinute sunt următoarele:

1) Fire de 9 cm din cupru liţat, lungime totală circuit 9 cm x 2 = 18 cm, timpul de la transmitere până la primire variază între 1493 – 1494 milisecunde, ca în Figura 5.

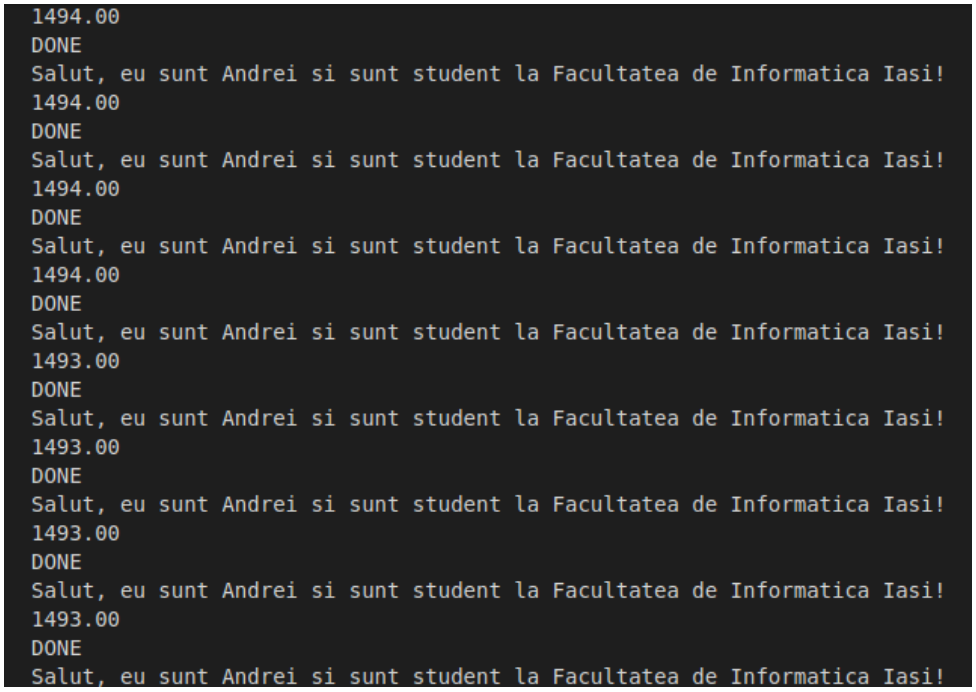


Figura 5 – Test fire 18 cm cupru liţat

2) Fire de 20 cm din cupru liţat, lungime totală circuit 20 cm x 2 = 40 cm, timpul de la transmitere până la primire variază între 1494 – 1495 milisecunde, ca în Figura 6.

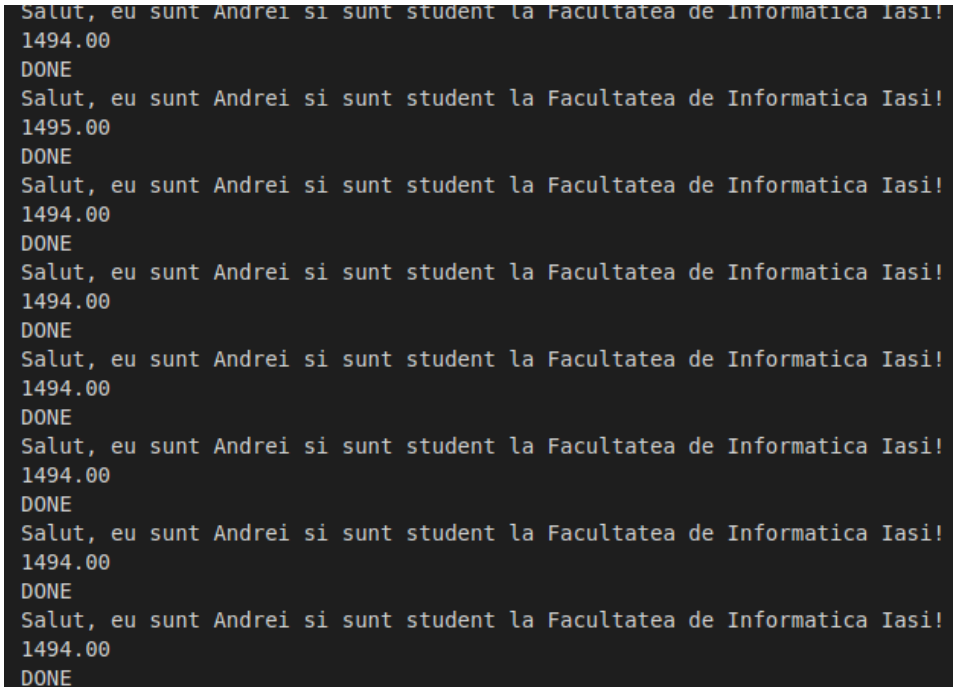


Figura 6 – Test fire 40 cm cupru liţat

3) Fire de 60 cm dintr-un cablu USB, lungime totală circuit 60 cm x 2 = 120 cm, timpul de la transmitere până la primire variază între 1496 – 1498 milisecunde, ca în Figura 7.

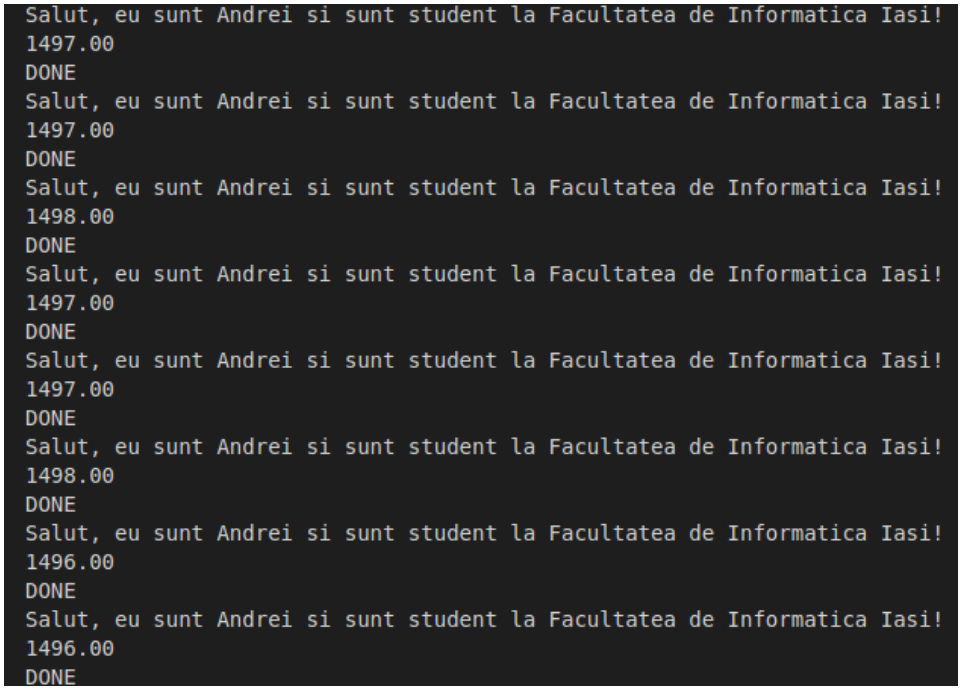


Figura 7 – Test fire cablu USB 120 cm cupru liţat

4) Fire de 600 cm dintr-un cablu de internet tip CAT5, lungime totală circuit 600 cm x 2 = 1200 cm, timpul de la transmitere până la primire variază între 1496 – 1498 milisecunde, ca în Figura 8.

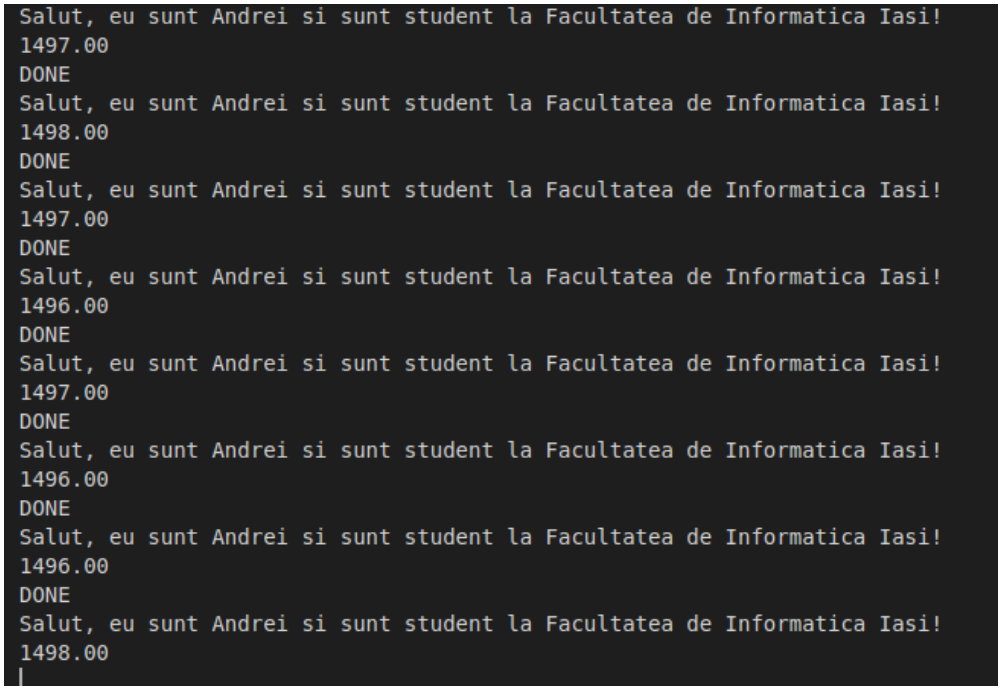


Figura 8 – Test fire torsadate cablu internet tip CAT5 1200 cm, aluminiu cuprat

În urma celor 4 experimente efectuate, am putut observa, cum era şi normal, o creştere a timpului de răspuns în funcţie de lungimea totală a firelor, singurul element variabil în experiment. Această creştere este una foarte mică, deşi lungimea firelor diferă destul de mult. Se poate observa că timpii sunt similari pentru experimentul 3) şi 4), deşi lungimea firelor din cadrul experimentului 4) este de 10 ori mai mare. Acest lucru se datorează materialului diferit (aluminiu cuprat) şi mecanismului de torsadare a firelor cablului de internet CAT5, cablu ce este construit să transmită date la o viteză de până la 100 Mbps.

Deoarece testele efectuate mai sus nu au produs diferenţe majore de timpi, am testat biblioteca şi la diferite viteze BAUD. Astfel, în Tabela 12 putem vedea diferite viteze BAUD: 9600, 19200, 38400, 57600, 74800; şi 3 tipuri de cablu, aceleaşi ca în testele de mai sus. Testul a constat în trimiterea de 100 de ori a unui mesaj de lungime 100 de caractere, primirea aceluiaşi mesaj înapoi, şi compararea celor două valori. În urma tuturor testelor, procentajul de corectitudine al mesajului primit a fost de 100%. Timpii din tabela de mai jos reprezintă o medie a timpilor în cele 100 de rulări.

Rata Baud	**9600**	**19200**	**38400**	**57600**	**74880**
18 cm	1493.4	1463.6	1450.9	1446.7	1445.7
120 cm	1496.5	1465.4	1452.8	1447.6	1438.9
1200 cm	1496.6	1465.6	1453	1447.8	1438.9

Tabela 12 - Tabel de comparaţie viteze BAUD – string 100 caractere

În Tabela 13 putem vedea aceleaşi teste de mai sus doar ca pe un string de lungime 255 de caractere. Cum era şi normal, timpul de transmisie – primire a crescut seminificativ, deoarece numărul operaţiilor şi verificărilor pentru integritatea mesajelor a crescut şi el.

Rata Baud	**9600**	**19200**	**38400**	**57600**	**74880**
18 cm	5104.1	5028.4	4999.9	4990.5	4988.1
120 cm	5114.3	5035.7	5002.3	4932.4	4963.4
1200 cm	5114.9	5036.2	5004.9	4933.4	4964.5

Tabela 13 - Tabel de comparaţie viteze BAUD – string 255 caractere

Timpii din tabela de mai sus sunt exprimaţi in milisecunde. Observăm o distribuţie normală a timpilor raportat la lungimea cablurilor. Rata BAUD maxim testată a fost de doar 74 KB / s (74800) deoarece în transmiterea datelor am utilizat bibliotecă **SoftwareSerial** ce are limitările ei din punctul de vedere al ratei BAUD, maximul fiind de 115 KB / s. Cu toate astea, nu am reuşit să transmit la această viteză maxima de 115 KB / s, din motive necunosute şi cred că independente de biblioteca mea.

3 Concluziile lucrării

Lucrarea **Protocol de comunicare pentru microcontrolere** denumită şi **TwoArduinoSerialCom** reprezintă o bibliotecă Arduino ce extinde posibilităţile de comunicare serială între plăcuţe de acest tip datorită implementărilor realizate de către mine. Principalele noutăţi ale bibliotecii sunt utilizarea în cadrul aceluiaşi proiect a mai multe plăcuţe Arduino, datorită creşterii numărului de porturi prin care se poate realiza comunicarea serială; integritatea datelor transmise prin implementarea mecanismelor de fragmentare a acestora în blocuri de dimensiune prestabilită şi prin detectarea şi corectarea erorilor; sistem de adresare între plăcuţe prin implementarea mecanismului de asignare a unui identificator unic de patru cifre (1000 – 9999) pentru fiecare plăcuţă; mecanism de interconectare plăcuţe ceea ce face ca comunicarea de mesaje să fie realizată bidirecţional.

Biblioteca **TwoArduinoSerialCom** se axează în principal pe transmiterea cu succes a datelor, fără erori. Viteza de lucru a acestei biblioteci nu a reprezentat pentru mine o caracteristică principală. La finalul proiectului, am putut observa că viteza este în parametri normali, transmisia datelor realizându-se aproape instant, sub o secundă. Desigur, viteza de transmisie este în mod direct influenţată de lungimea datelor, în acest caz pentru un string de lungime mare, de exemplu peste 300 de caractere, timpul de transmitere fiind de ordinul secundelor (4-5 secunde).

În opinia mea, această bibliotecă şi-a atins scopul iniţial, acela de a împrumuta unele idei din stiva TCP / IP şi de a le implementa în cadrul programării cu microprocesoare, idei ce se potrivesc acestui tip de programare, pentru a asigura comunicarea fără erori între plăcuţe de acest tip. Ca în cazul oricărui proiect, cred că mai există loc de optimizări, atât la nivel de memorie cât şi al vitezei, eu unul fiind mulţumit cu optimizările actuale realizate.

Dezvoltarea acestui proiect a fost realizată împreună cu un sistem de versionare a codului. Modul de lucru utilizat a implicat utilizarea *branchurilor* pentru fiecare *feature* în parte şi apoi alipirea acestuia în *branchul* de bază. (*master*). În capitolul Anexe, subcapitolul Anexă 3 se află conţinutul fişierului README.md al proiectului, ce se găseşte pe pagina de Github https://github.com/AndreiTimofte96/TwoArduinoSerialCom.

Bibliografie

- Gilbert Held, (1996). Understanding Data Communications, (Comunicaţii de date). Bucureşti: Teora (1998).
- Biblioteca Arduino SerialTransfer: https://github.com/PowerBroker2/SerialTransfer
- Biblioteca Arduino SoftwareSerial: https://www.arduino.cc/en/Reference/SoftwareSerial
- INTERNET STANDARD, RFC: 793, TRANSMISSION CONTROL PROTOCOL, DARPA INTERNET PROGRAM, PROTOCOL SPECIFICATION, September 1981: https://tools.ietf.org/html/rfc793
- INTERNET STANDARD, RFC 768, J. Postel, ISI, 28 August 1980, User Datagram Protocol: https://tools.ietf.org/html/rfc768
- Stanford Computer Science Department Courses CS144 - User datagram protocol (UDP): https://www.scs.stanford.edu/09au-cs144/notes/l2-print.pdf

Anexe

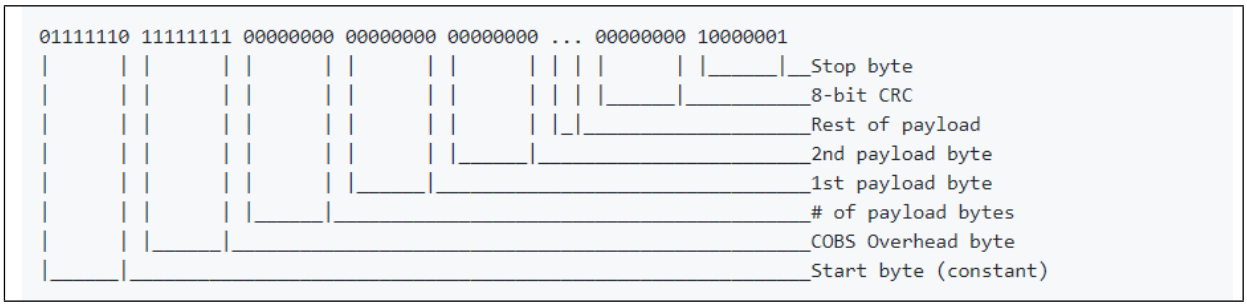
Anexă 1

Utilizarea bibliotecii SerialTransfer:

Includerea bibliotecii şi declararea unui obiect al clasei SerialTransfer.h:

```
#include "SerialTransfer.h"

SerialTransfer myTransfer;
```

Schema pachetului tranzacţionat:



Iniţializarea obiectului myTransfer şi a ratei BAUD:

```
Serial1.begin(115200);

myTransfer.begin(Serial1);
```

Transmiterea datelor de la expeditor:

```
myTransfer.txBuff[0] = 'h';
myTransfer.txBuff[1] = 'i';
myTransfer.txBuff[2] = '\n';

myTransfer.sendData(3);
```

Primirea datelor la destinatar:

```
if(myTransfer.available()){
  // see next step
}
else if(myTransfer.status < 0){
  Serial.print("ERROR: ");

  if(myTransfer.status == -1)
    Serial.println(F("CRC_ERROR"));

  else if(myTransfer.status == -2)
    Serial.println(F("PAYLOAD_ERROR"));

  else if(myTransfer.status == -3)
    Serial.println(F("STOP_BYTE_ERROR"));
}
```

Anexă 2

Metodele publice ale bibliotecii TwoArduinoCom

În această anexă vă voi prezenta pe scurt funcţiile principale ale acestei lucrări, pe care le-am implementat, disponibile pentru utilizatori. Pentru a include şi folosi biblioteca, trebuie mai întâi sa stabilim modul de comunicare dorit: comunicare fără confirmare sau cu confirmare. Mai jos găsim ambele moduri de includere:

```
#include “UdpProtocol.hpp” // comunicare fără confirmare

#include “TcpProtocol.hpp” // comunicare cu confirmare
```

Urmează iniţializarea bibliotecii care este la fel pentru ambele moduri de comunicare. Aici vom stabili şi portul serial pe care vrea utilizatorul să-l folosească, pentru că acesta nu este unic:

```
TcpProtocol myProtocol; // UdpProtocol myProtocol;

myProtocol.initializePorts(rxPort, txPort);

myProtocol.initializeSerial(Serial, 9600); //Serial, rata BAUD
```

În caz că utilizatorul a ales modul de comunicare cu confirmare, deoarece acesta funcţionează asemănător cu principiul client – server din protocolul TCP, se vor utiliza

următoarele funcţii: listen() şi connect(). Serverul va apela funcţia listen(), care „ascultă” clientul deja iniţializat să se conecteze. Clientul va apela funcţia connect(), pentru a se conecta la server. În cazul în care conexiune nu poate fi stabilită, serverul returnează un mesaj de eroare şi „ascultă” în continuare. Protocolul de conectare este asemănător cu procedeul *Three Way Handshake* prezent în cadrul protocolului TCP din TCP/IP. Funcţia printLastError() va afişa, la portul serial stabilit în iniţializarea biblioteci, ultima eroare detectată de către aceasta.

```
// Server
int UAID = myProtocol.listen();

// Client
if (myProtocol.connect() < 0) {
    myProtocol.printLastError();
}
```

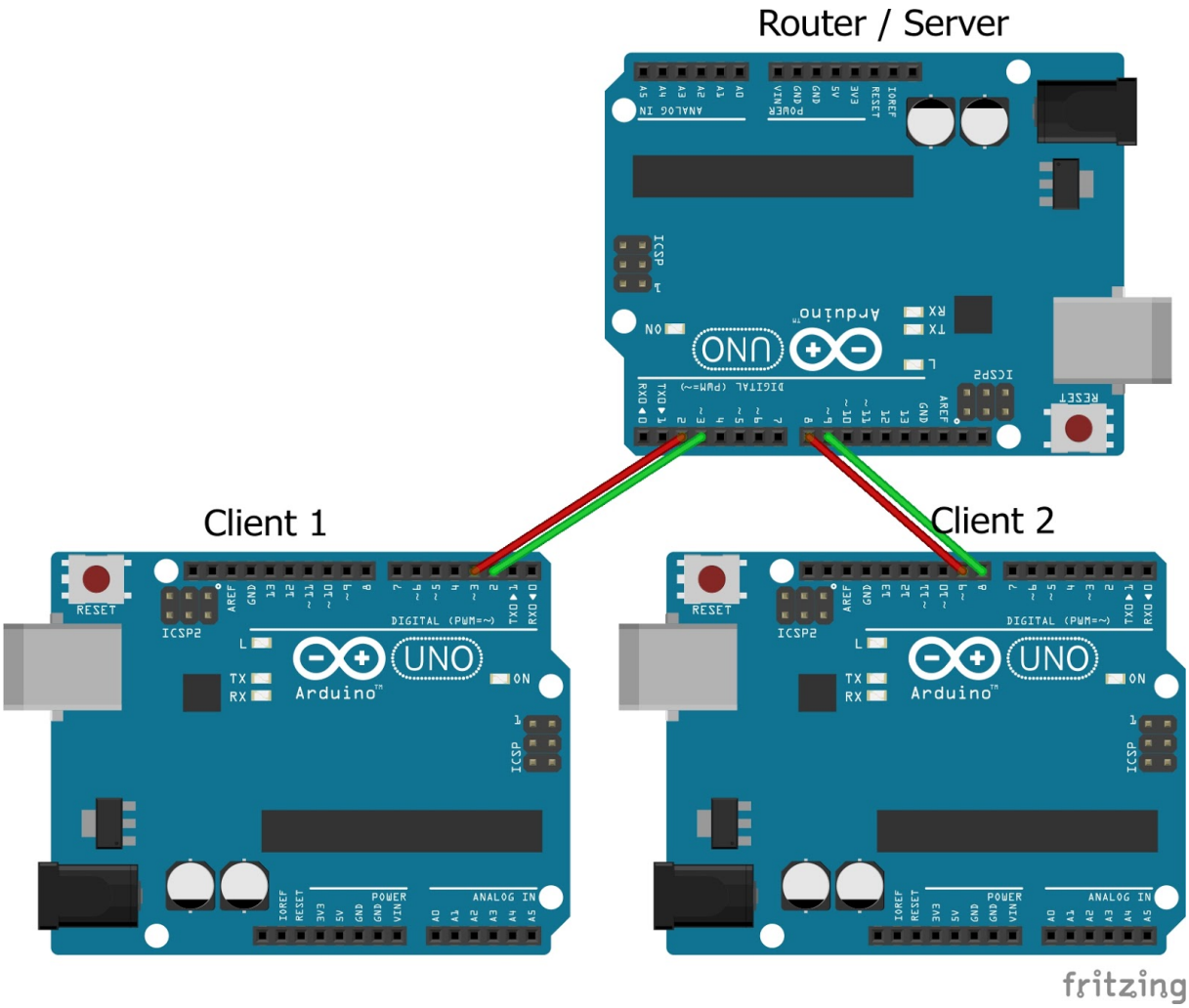
Funcţiile de citire şi scriere se numesc: read() şi write(). Acestea permit două argumente: primul este un pointer către o zonă de memorie unde se află mesajul ce va fi primit / trimis, iar al doilea parametru este UAID-ul plăcuţei Arduino de unde a fost trimis mesajul / către cine sa fie trimis. Acest ultim argument al funcţiei este opţional.

```
char data[] = „Data”;
int senderUAID, destinationUAID;

if ((length = tcpProtocol.read(dataToReceive, senderUAID)) < 0) {
      tcpProtocol.printLastError();
}

if (!tcpProtocol.write(userInput, &destinationUAID)) {
     tcpProtocol.printLastError();
}
```

Pentru încheierea conexiunii între client şi server, se vor apela următoarele funcţii ce au rolul de a semnala bibliotecii încheierea transmiterii de mesaje între cele două părţi. Mai multe detalii despre toate aceste funcţii se află în capitolul 2 al acestei lucrări, subcapitolul 2.1 Modulele bibliotecii.

```
// Server
myProtocol.serverClose();

//Client
myProtocol.clientClose();
```

Schema electronică a unui exemplu, ce implică trei plăcuţe Arduino: 2 clienţi şi un router / server.



Anexă 3

Conţinutul fişierului READ.md al paginii de Github a bibliotecii: https://github.com/AndreiTimofte96/TwoArduinoSerialCom/blob/master/README.md

TwoArduinoSerialCom

TwoArduinoSerialCom also named two-arduino-COM, is an Arduino Library to transfer serial data, ensuring data integrity, via Arduino Serial Communication UART device.

This library supports two modes of communication, following the client - server model: without confirmation, similar to the UDP protocol, and with confirmation that the message has been received successfully, similar to the TCP protocol.

Library features:

• interconnection mechanism between two Arduinos based on a UAID;

• availability of all ports for message communication which allows connecting multiple Arduino boards within the same sketch;

• creates a unique id for each Arduino board and saves it into EEPROM memory;

• sending messages in two ways: with and without confirmation;

• message segmentation into fixed-size blocks;

• detection and correction of messages at the destination in case of errors in the transmission process;

• mechanism for reordering the received packets;

• connection termination mechanism;

In both ways, the library initialization contains common elements:

1. **Include the library and create UdpProtocol / TcpProcol class instance:**

```
#include “UdpProtocol.hpp” // without confirmation
UdpProtocol myProtocol;
or
#include “TcpProtocol.hpp” //  with confirmation
TcpProtocol myProtocol;
```

2. **Initialize the library**

```
myProtocol.initializePorts(rxPort, txPort);

myProtocol.initializeSerial(Serial, 9600, 500); //Serial port, BAUD rate, Serial timeout
```

1. *Communication without confirmation*

```
if ((length = udpProtocol.write(text, UAID)) < 0) { // text, Arduino board destination UAID
      udpProtocol.printLastError(); // prints last error
    }

if (!udpProtocol.read(dataToReceive, fromUAID)) { // text, Ardunio board sender UAID
```

```
        udpProtocol.printLastError();
}
```

2. *Communication with confirmation*

First we need to interconnect the boards:

Server Side:

```
 if ((clientUAID = tcpProtocol1.listen()) < 0) {
   tcpProtocol1.printLastError();
 }
```

Client Side:

```
 if (tcpProtocol.connect() < 0) {
    tcpProtocol.printLastError();
 }
```

Then we can start serial transfer:

```
if ((length = tcpProtocol.write(text, destinationUAID)) < 0) {
    tcpProtocol.printLastError();
}


if ((length = tcpProtocol.read(dataToReceive, senderUAID)) < 0) {
    tcpProtocol.printLastError();
}
```

Closing connection server-side:

```
 if (tcpProtocol.serverClose() < 0) {
     tcpProtocol.printLastError();
   }  // else
   while (1);
```

Closing connection client-side:

```
if (tcpProtocol.clientClose() < 0) {
    tcpProtocol.printLastError();
  }  // else
  while (1);
```

Complete Client side Code with TcpProtocol class:

```
// UAID = 8808
#include "TcpProtocol.hpp"

TcpProtocol myProtocol;

void setup() {
  myProtocol.initializePorts(2, 3);                      // RX, TX
  myProtocol.initializeSerial(Serial, 9600, 500);  //Serial, baudRate, Serial.setTimeout
  myProtocol.useShowLogs(false); // the default is false
  myProtocol.useAsyncMode(false); // the default is false

}

char dataToSend[] = "Some text sent by client";
char dataToReceive[300];
```

```
int destinationUAID = 4987;
int senderUAID;
int length;

void loop() {
  if (myProtocol.connect() < 0) {
    myProtocol.printLastError();
  }

  if ((length = myProtocol.write(dataToSend, destinationUAID)) < 0) {
    myProtocol.printLastError();
  }

  if ((length = myProtocol.read(dataToReceive, senderUAID)) < 0) {
    myProtocol.printLastError();
  }

  Serial.print(senderUAID);
  Serial.println(":");
  Serial.println(dataToReceive);
  Serial.println(length);

  if (myProtocol.clientClose() < 0) {
    myProtocol.printLastError();
  }  // else
  while (1)
    ;
}
```

Complete Server side Code with TcpProtocol class:

```
// UAID = 4987
#include <TcpProtocol.hpp>

TcpProtocol myProtocol;

int serverUAID = TcpProtocol::getUniqueArduinoIDFromEEEPROM();

char dataToSend[] = "Some text sent by server";
char dataToReceive[300];

int destinationUAID;  //should be 8808;
int senderUAID;
int length;

void setup() {
  myProtocol.initializePorts(2, 3);                       // RX, TX
  myProtocol.initializeSerial(Serial, 9600, 500);  //Serial, baudRate, Serial.setTimeout
  myProtocol.useShowLogs(false); // the default is false
  myProtocol.useAsyncMode(false); // the default is false

}

void loop() {
  if ((destinationUAID = myProtocol.listen()) < 0) {
    myProtocol.printLastError();
  }

  if ((length = myProtocol.read(dataToReceive, senderUAID)) < 0) {
    myProtocol.printLastError();
  }
  Serial.print(senderUAID);
  Serial.println(":");
  Serial.println(dataToReceive);
  Serial.println(length);

  if ((length = myProtocol.write(dataToReceive, destinationUAID)) < 0) {
    myProtocol.printLastError();
  }
```

```
  if (myProtocol.serverClose() < 0) {
    myProtocol.printLastError();
  }

  while (1)
    ;
}
```