1. Proiectarea aplicației

1.1. Analiza platformei hardware pe care va fi executată aplicația

O platformă hardware este compusă din seturi de elemente hardware compatibile ce permit executarea de aplicații software. Fiecare astfel de platformă are definit propriul limbaj(limbajul mașină), iar programele ce rulează pe o astfel de platformă sunt special construite pentru fiecare categorie de procesor. Aceasta definește standardul în jurul căruia poate fi dezvoltat un sistem, este o bază de tehnologii pe care sunt construite alte tehnologii sau procese.

Din analiza temei propuse deducem că pentru îndeplinirea obiectivelor este nevoie de existența a două dispozitive fizice ce pot comunica, fiecare având un rol esențial în proces, unul ce va trimite instrucțiuni/comenzi, iar altul ce le poate intercepta și interpreta.

Au fost analizate toate opțiunile în vederea selecției sistemelor ce vor putea comunica, iar ca rezultat, au fost alese ca inițiator/emițător mulțimea dispozitivelor mobile ce folosesc ca sistem de operare platforma Android, alegere motivată de excelența portabilității unui program pentru o astfel de platformă și de infrastructura de dezvoltare a aplicațiilor destinate acestuia. În același timp, componenta pe post de receptor este reprezentată de microcontroller-ul ESP32, alegere motivată de independența cipului de periferice, având integrate module de Bluetooth și Wi-Fi.

Principalul motiv al alegerii tehnologiei Bluetooth în schimbul tehnologiilor cu funcționalități asemănătoare este acela că acesta oferă posibilitatea existenței unei conexiuni între dispozitive fără a mai exista intermediari care pot manipula pachetele primite, astfel este asigurată securitatea la nivel de transfer al datelor. Concomitent, selectarea Bluetooth-ului Low Energy în locul Bluetooth-ului „clasic” a fost motivată de faptul că deși cel clasic oferă posibilitatea de a transfera pachete mari de date într-o perioadă mai scurtă, acesta consumă mai multă energie față de succesorul său. Totodată, caracteristicile tehnologiei cu energie redusă sunt conforme cu cerințele proiectului.

1.2. Modulele proiectului

Un modul este o parte separabilă logic a unui program, în acest mod, proiectul este format din două programe software și o componentă hardware. Unul din programele software este destinat dispozitivelor mobile, iar cel de-al este destinat microcontroller-ului ESP32. Componenta hardware este reprezentată de microcontroller și ansamblul elementelor de circuit conectate la pinii de intrare/ieșire ai acestuia.

Prin aplicația mobilă se dorește accesarea componentei Bluetooth a dispozitivului, așa încât prin intermediul interfeței cu utilizatorul să se poată realiza descoperirea, asocierea și trimiterea comenzilor către dispozitivele-receptor din proximitate. Aceasta va împărțită în 3 module, primul(„components”) ce conține elementele pur vizuale ale aplicației, al doilea(„logic”) ce cuprinde implementările funcționalităților, iar al treilea(„screens”) realizează cuplarea modulelor.

Programul pentru microcontroller va aștepta cererile de asociere, va interpreta mesajele primite de la dispozitivele asociate, va lua măsuri în funcție de natura mesajului primit și va gestiona evenimentele de întrerupere apărute în perioada de execuție. Așadar, acesta va cuprinde logica de inițializare și configurare a componentei Bluetooth, dar și logica de configurare a pinilor de intrare/ieșire ai acestuia.

Pentru a contura comunicarea între dispozitivul mobil și microcontroller a fost creată următoarea diagramă de activități(Figura 1.1).

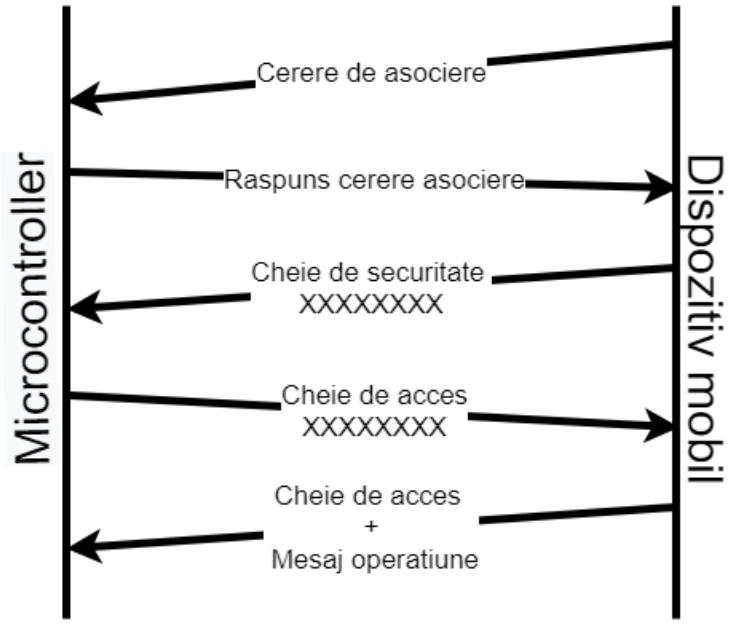

Figura 1.1.

Modulul hardware are în considerare siguranța implementării circuitului electric din punct de vedere electrotehnic și permite crearea întreruperilor programului microcontroller-ului(prin apăsarea unui buton) cu scopul de a afișa starea curentă a modulului Bluetooth(dacă există asocieri sau nu) prin semnale luminoase(se folosește un L.E.D.). Simultan, este folosit un buzzer care alături de dioda electroluminiscentă vor ajuta la localizarea dispozitivului.

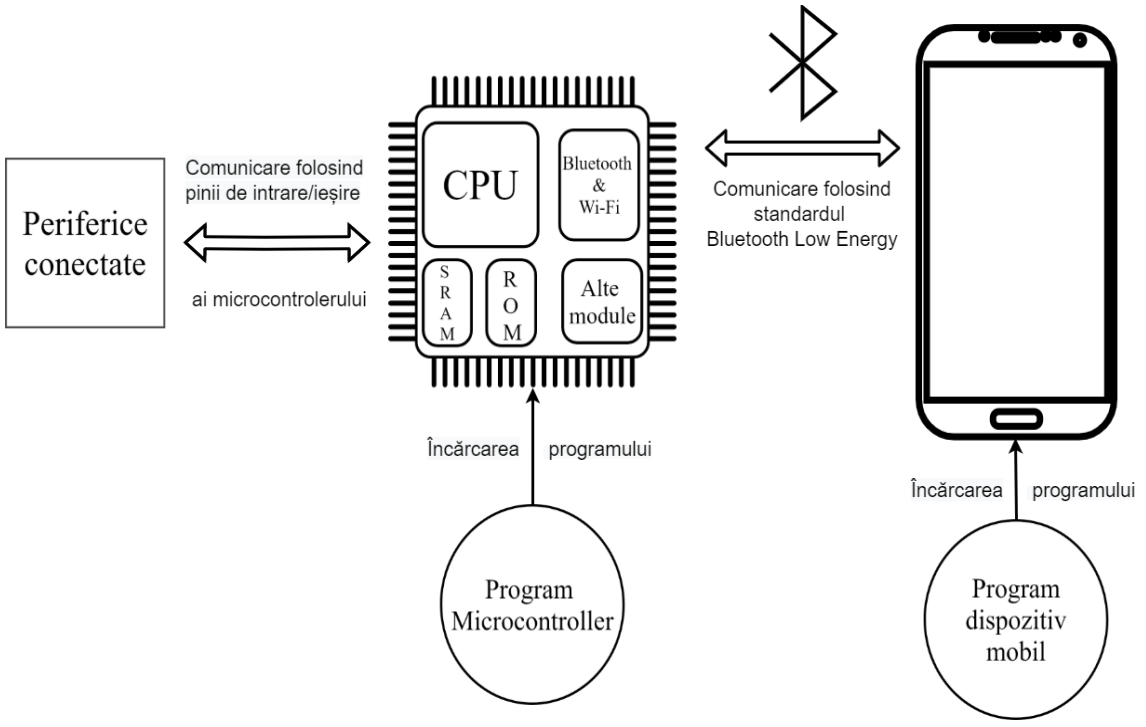

Figura 1.2. Interacțiunea dintre modulele proiectului

1.3. Avantaje și dezavantaje ale metodei alese

1.3.1. Avantaje și dezavantaje ale aplicațiilor pentru Android

Principalul avantaj al dezvoltării unui program destinat sistemului de operare Android este acela că numărul de dispozitive mobile care folosesc această platformă software este majoritară în raport cu restul opțiunilor existente pe piață.

Pe durata studiului tuturor opțiunilor luate în considerare, aplicațiile Android sunt considerate ca având următoarele avantaje:

- Sistemul de operare are codul sursă accesibil(eng.: open source), lucru ce duce la flexibilitatea creării de aplicații mobile;
- Kit-ul nativ de dezvoltare al software-ului și mediul de dezvoltare beneficiază de sprijin deplin din punct de vedere al documentației existente;
- Existența multiplelor medii terțe ce au ca scop dezvoltarea aplicațiilor native scriind un singur program sau a celor care au ca obiect dezvoltarea unui program ce poate rula pe orice platformă(eng.: cross-platform);
- Infrastructura de dezvoltare nu este costisitoare;
- Rapiditate la nivel de implementare;
- Adaptarea rapidă la mediul de dezvoltare folosit;
- Publicarea aplicației este ușor și rapid de gestionat.

Totodată, cel mai conturat dezavantaj este acela că posibilitatea ca aplicația să poată fi fragmentată este mult mai mare datorită numărului mare de dispozitive ce diferă în rezoluție și diagonală. Pentru a asigura compatibilitatea cu toate dispozitivele, perioada de testare a aplicației este lungită considerabil, astfel publicarea aplicației poate fi amânată.

1.3.2. Avantaje și dezavantaje ale microcontrollerului ESP32

Primul avantaj al folosirii acestui microcontrolelr este acela că cipul integrează un set de module ce permit dezvoltatorilor să creeze aplicații scalabile și adaptabile în timp. Simultan cu această idee, microcontroller-ul dispune de mai multe avantaje, precum:

- Wi-Fi și Bluetooth integrat, module ce reduc necesitatea de periferice;
- Microprocesor cu două nuclee pe 32 de biți ce funcționează la frecvențe între 160 și 240 MHz;
- Pornire securizată ce are ca scop menținerea integrității hardware;
- Pornire prin întreruperi produse la nivelul pinilor de intrare/ieșire;
- 34 de pini de intrare/ieșire programabili;
- Generator de semnale P.W.M.;
- Memoria flash este criptată;
- Consum de energie redus.

Dezavantajul scos în evidență este acela că prețul de achiziție al cipului este aproape dublu comparativ cu predecesorul său, microcontroller-ul ESP8266, fapt motivat de îmbunătățirile și noile funcționalități aduse.

1.3.3. Avantaje și dezavantaje ale tehnologiei Bluetooth Low Energy

Prin studiul făcut a fost conturat evenimentul adoptării standardului tehnologic de cât mai multe companii de profil, lucru datorat avantajelor aduse, dintre care enumerăm:

- Facilitează consumul redus de energie;
- Nu există interferențe cu alte dispozitive fără fir;
- Permite conectarea mai multor dispozitive la un singur dispozitiv;
- Securitatea transferului de date este asigurată de faptul că nu există intermediari care să transmită datele de la un capăt la altul;
- Compatibilitatea între dispozitivele ce folosesc profilurile existente.

Simultan, au fost identificate și dezavantaje ale acestei tehnologii, dintre care enunțăm:

- Conexiunea dintre dispozitive se face doar pe distanțe scurte, ceea ce duce la pierderea acesteia dacă dispozitivele sunt prea depărtate;
- Lățimea de bandă este scurtă;
- Dispozitivele rămase pornite ce nu au un cod de securitate pentru asociere pot fi foarte ușor penetrate de atacuri cibernetice, astfel datele stocate pot fi compromise.

1.4. Limite în care metoda aleasă va funcționa

Având în vedere analiza făcută și avantajele și dezavantajele anterior menționate se înțelege că aplicația mobilă este destinată dispozitivelor Android, fapt care creează limitări din punct de vedere al platformei pe care poate fi pus în execuție programul. Concomitent, existența unei legături între mecanisme este limitată de distanța dintre acestea, definitivând neputința asocierii dispozitivelor dacă ele nu sunt în proximitate.

Nu în ultimul rând, se are în vedere și capabilitatea tehnologiei Bluetooth de a putea oferi suportul ca un dispozitiv-receptor să poată avea mai multe conexiuni. Acest lucru nu va fi luat în calcul deoarece este neutru din punct de vedere al obiectevelor proiectului, fapt ce duce la o limitare de funcționare.

1.5. Componentele proiectului

O componentă reprezintă o parte a unui întreg. Componentele software sunt părți ale unei aplicații sau ale unui sistem, fiecare având un scop unic, astfel complexitatea unei probleme este împărțită în fragmente ușor de gestionat. O componentă software poate fi implementată independent și este supusă compoziției de către terți. Simultan, o componentă hardware este o unitate fizică autonomă ce poate fi încorporată într-un sistem complex(de exemplu un microcontroller și modulele încorporate de acesta).

1.5.1. Componentele software

1.5.1.1.Aplicația mobilă

Pentru schițarea aplicației mobile au fost luate în considerare obiectivele principale ale proiectului, astfel dezvoltarea a început cu ideea că prin interfața cu utilizatorul se vor permite descoperirea dispozitivelor din jur, opțiunea asocierii cu unul sau mai multe dispozitive descoperite, limitarea asocierii prin autorizare folosind o cheie de securitate, ștergerea unei asocieri existente. Astfel s-a creat diagrama de flux(eng.: flow diagram) a ecranelor aplicației, diagramă ilustrată în Figura 1.3.

1.5.1.1.1. Tehnologia aleasă pentru implementare

Platforma de lucru aleasă pentru program este **React Native** deoarece aceasta pune la dispoziție uneltele necesare dezvoltării de aplicații pentru orice platformă prin crearea unui singur proiect. La etapa de compilare trebuie specificată platforma pentru care se creează respectiva aplicație, mediul apoi făcând legăturile dintre librăriile folosite/implementări și librăriile native(Android, iOS, Windows, ș.a.m.d.).

Avantaje ale framework-ului React Native:

- Asemănare cu dezvoltarea aplicațiilor web;
- Flexibilitate în ceea ce presupune managementul codului;
- Rapiditate la migrarea pentru dezvoltarea aplicațiilor native;
- Schimbările aduse codului sunt reflectate în previzualizarea aplicației;
- Sprijin deplin din partea dezvoltatorului;
- Eficient din punct de vedere al costului și timpului;
- Folosește un limbaj multi-paradigmă;
- Extensibilitatea aplicației pe mai multe platforme;

Atuul acestui framework este acela că permite împărțirea elementelor vizuale ale aplicației(eng.: view) în componente independente ce pot fi refolosite. O componentă este o clasă sau o funcție **JavaScript** ce are ca scop definirea comportamentului și a aspectului unui obiect de pe interfață. Pentru ca o componentă să fie definită corect, aceasta trebuie să conțină o modalitate prin care să se returneze un obiect **J.S.X.**(**JavaScript X.M.L.**). JavaScript X.M.L. este o extensie adusă limbajului JavaScript prin care se poate descrie aspectul interfeței utilizator, în același timp oferind și posibilitatea de a lega variabilele de elementele vizuale(eng.: data binding).

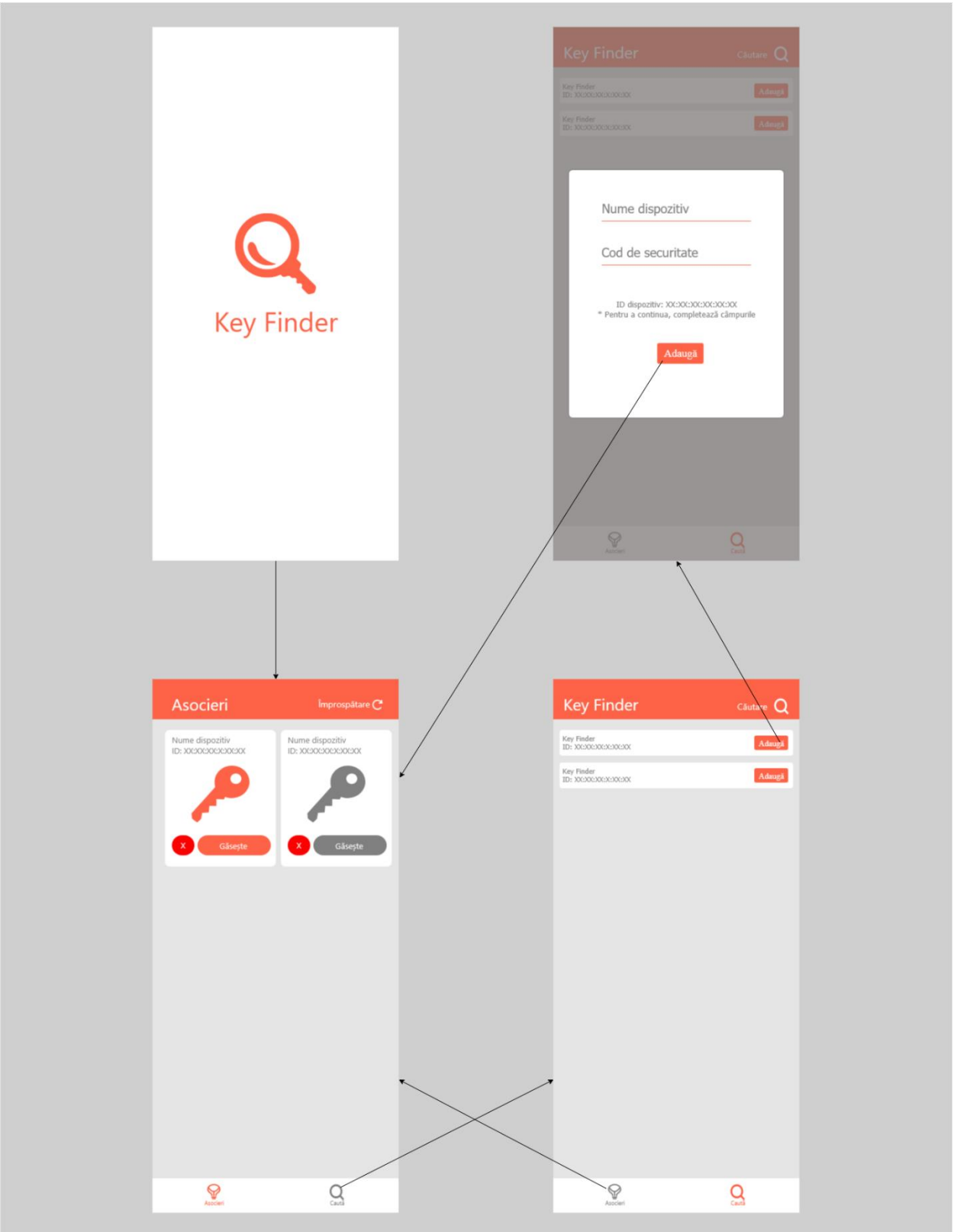

Figura 1.3. Diagrama de flux a ecranelor aplicației

1.5.1.1.2. Componenta pentru funcționalități

Clasa **BtManager**(Figura 1.4) are o singură instanță pe tot parcursul programului, deoarece se dorește ca o unică entitate din program să poată accesa componenta Bluetooth a dispozitivului. La momentul creării instanței clasei se încarcă datele din fișierul „devices-data.json”, fișier ce conține id-urile serviciilor Bluetooth de interes și mesajul ce descrie tipul operațiunilor executate de dispozitivele țintă.

Din câte se observă, clasa **BtManager** nu conține un câmp cu o instanță proprie care să fie returnată atunci când este nevoie de aceasta, fapt neconform cu șablonul de proiectare **Singleton**. Funcționalitatea asemănătoarea este îndeplinită prin crearea unei instanțe și exportarea acesteia(funcționalitate specifică limbajului JavaScript).

Metoda **searchForDevices**(Figura 1.5) returnează un **Promise** și are ca rol descoperirea dispozitivelor Bluetooth Low Energy din apropiere al căror identificator de serviciu de advertising se găsește în lista de id-uri de advertisement încărcată la momentul creării instanței **BtManager**. Scanarea se va face timp de trei secunde urmând ca apoi să se returneze lista dispozitivelor din proximitate.

```
class BtManager {
   constructor() {
      this.data = require("../../assets/data/devices-data.json");
      this.manager = new BleManager();
      this.blEnabled = true;
      this.manager.onStateChange(state => {
         this.blEnabled = state == 'PoweredOn' ? true : false;
      });
   }
   searchForDevices { [...] }
   addDevice(deviceID, securityCode) { [...] }
   findDevice(deviceID, accessCode) { [...] }
   throwErrorByType(error) { [...] }
}
const instance = new BtManager();
export default instance;
```

Figura 1.4. Clasa BtManager

```
searchForDevices() {
   return new Promise((resolve, reject) => {
      let devices = [];
      this.manager.startDeviceScan(this.data.adv_service_uuids, null, (error, device) => {
         if (error) {
            if (error.message == 'BluetoothLE is powered off')
               reject(new BluetoothError('Bluetooth-ul nu este                                    pornit'));
            else if (error.message == 'Device is not authorized to use BluetoothLE')
               reject(new LocationError('Nu s-a permis accesul la locatie'));
            else if (error.message == 'Location services are disabled')
               reject(new LocationServicesError('Serviciile de localizare nu sunt pornite'));
            reject(error);
         }
         if (!devices.find(d => d.id == device.id))
            devices.push(device);
      });
      setTimeout(() => {
         this.manager.stopDeviceScan();
          resolve(devices);
      }, 3000)
});}
```

Figura 1.5. Metoda searchForDevices din clasa BtManager

Metoda **addDevice**(Figura 1.6) va primi ca parametri de intrare id-ul dispozitivului cu care se dorește asocierea și cheia de securitate, apoi va realiza conexiunea între capete și va descoperi serviciile și caracteristicile Bluetooth ale dispozitivului receptor. Urmează trimiterea cheii de securitate către dispozitiv succedată de citirea și returnarea rezultatului obținut în urma validării cheii. În cazul în care cheia de securitate este validă funcția va returna un cod de acces prin intermediul căruia se pot face cereri către serviciul dispozitivului respectiv fără a mai fi nevoie de o viitoare autorizare.

Metoda **findDevice**(Figura 1.7) permite trimiterea mesajului de activare al funcției de găsire a dispozitivului cu id-ul primit ca parametru prin concatenarea codului de acces. Parametrul este primit la momentul autorizării și mesajul operației de găsire încărcat la momentul creării instanței clasei din care face parte.

Se observă că metodele **findDevice** și **addDevice** vor arunca erori în cazul în care operația cerută nu poate fi executată(din motive precum: Bluetooth-ul/ serviciile de localizare nu sunt pornite, dispozitivul nu este în apropiere sau accesul la serviciile de localizare nu este permis).

Totodată, a fost creată și metoda **throwErrorByType**(Figura 1.8), prin care se analizează tipul de eroare aruncat de instanța clasei **BleManager**. Necesitatea acesteia este datorată faptului că la momentul apariției unei erori librăria creează un tip generic de eroare(**BtError**), astfel identificarea erorii trebuie făcută prin evaluarea mesajului acesteia, ci nu prin tipul instanței.

```
addDevice(deviceID, securityCode) {
  if (this.blEnabled) {
    return this.manager.connectToDevice(deviceID)
      .then(() => {
        return this.manager.discoverAllServicesAndCharacteristicsForDevice(deviceID);
      })
      .then(() => {
        return this.manager.writeCharacteristicWithoutResponseForDevice(
          deviceID,
          this.data.gatt_service[0].uuid,
          this.data.gatt_service[0].characteristic_uuid,
          btoa(securityCode));
      })
      .then(() => {
        return this.manager.readCharacteristicForDevice(
          deviceID,
          this.data.gatt_service[0].uuid,
          this.data.gatt_service[0].characteristic_uuid
        );
      })
      .then(char => {
        this.manager.cancelDeviceConnection(deviceID);
        return atob(char.value);
      })
      .catch((e) => {
        this.manager.isDeviceConnected(deviceID)
          .then(isConnected => {
            if (isConnected)
              this.manager.cancelDeviceConnection(deviceID);
          });
        this.throwErrorByType(e);
        throw new DeviceNotInRangeError('Dispozitivul nu poate fi contactat.');
      });
  }
  else throw new BluetoothError('Bluetooth-ul nu este pornit');}
```

Figura 1.6. Metoda addDevice din clasa BtManager

```
findDevice(deviceID, accessCode) {
  if (this.blEnabled) {
    return this.manager.connectToDevice(deviceID)
      .then(() => {
        return this.manager.discoverAllServicesAndCharacteristicsForDevice(deviceID);
      })
      .then(() => {
        return this.manager.writeCharacteristicWithoutResponseForDevice(
          deviceID,
          this.data.gatt_service[0].uuid,
          this.data.gatt_service[0].characteristic_uuid,
          btoa(accessCode.concat(this.data.write_message.FIND)))
          .catch(err => {
            console.log(err);
          });
      })
      .then(() => {
        this.manager.cancelDeviceConnection(deviceID)
          .catch(e => {
            print('Eroare la deconectare');
            console.error(e);
          });
        return { success: true };
      })
      .catch((e) => {
        this.manager.isDeviceConnected(deviceID)
          .then(isConnected => {
            if (isConnected)
              this.manager.cancelDeviceConnection(deviceID);
          });
        this.throwErrorByType(e);
        throw new DeviceNotInRangeError('Dispozitivul nu poate fi contactat.');
      });
  }
  else throw new BluetoothError('Bluetooth-ul nu este pornit');
}
```

Figura 1.7. Metoda findDevice din clasa BtManager

```
throwErrorByType(error) {
  if (error.message == 'BluetoothLE is powered off')
    throw new BluetoothError('Bluetooth-ul nu este pornit');
  else if (error.message == 'Device is not authorized to use BluetoothLE')
    throw new LocationError('Nu s-a permis accesul la locatie');
  else if (error.message == 'Location services are disabled')
    throw new LocationServicesError('Serviciile de localizare nu sunt pornite');
}
```

Figura 1.8. Metoda throwErrorByType din clasa BtManager

1.5.1.1.3. Componenta elementelor vizuale

Datorită asemănării în implementare a componentelor vizuale, se va prezenta o singură componentă ce va cuprinde elemente similare cu celelalte implementări, dar și elemente unice încorporate, astfel se alege pentru descriere clasa **AppOverlay**(Figura 1.9).

Pentru ca o clasă să poată fi o componentă React validă, aceasta trebuie să extindă clasa **Component** și totodată să suprascrie metoda **render**, metodă ce va returna obligatoriu un obiect **J.S.X.**. Unei astfel de componente îi pot fi trimise proprietăți(asemănător cu declararea obiectului **Overlay**), caracteristică asemănătoare cu trimiterea de parametri unei funcții. Valorile/referințele proprietăților primite putând fi accesate prin intermediul câmpului props al clasei (de exemplu: **this.props.<nume_proprietate>**).

Rolurile proprietăților clasei AppOverlay:

- isVisible – proprietate ce determină dacă componenta trebuie afișată la momentul actualizării ecranului;
- onBackdropPress – proprietate ce are rol de callback pentru evenimentul de apăsare pe marginea Overlay-ului;
- deviceNameValue/securityCodeValue – proprietăți primite pentru a afișa valoarea introdusă în casetele text la momentul actualizării ecranului;
- onDeviceNameChange/onSecurityCodeChange – callback pentru a gestiona evenimentele de schimbare a valorilor casetelor text;
- deviceNameErrorMessage/securityCodeErrorMessage – proprietăți trimise pentru a afișa mesaje de eroare în ceea ce prevede valorile introduse în caseta text
- deviceId – proprietate ce are ca scop afișarea id-ului dispozitivului selectat
- onAddPress – callback ce gestionează evenimentul de apăsare pe buton
- isLoading – proprietate ce determină vizualizarea butonului

```
export default class AppOverlay extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <Overlay
        backdropStyle={{ backgroundColor: 'rgba(125,125,125,0.6)' }}
        animationType='fade'
        transparent={true}
        statusBarTranslucent={true}
        isVisible={this.props.isVisible}
        onBackdropPress={this.props.onBackdropPress}>
        <View style={styles.container}>
          <Input
            value={this.props.deviceNameValue}
            onChangeText={this.props.onDeviceNameChange}
            errorMessage={this.props.deviceNameErrorMessage}

            inputStyle={styles.inputStyle}
            inputContainerStyle={styles.inputContainerStyle}
            placeholder='Nume discpozitiv'
          />
          <Input
            value={this.props.securityCodeValue}
            onChangeText={this.props.onSecurityCodeChange}
            errorMessage={this.props.securityCodeErrorMessage}
```

```
                inputStyle={styles.inputStyle}
                inputContainerStyle={styles.inputContainerStyle}
                placeholder='Cod de securitate'
              />
              <View style={styles.formSubmit}>
                <Text>ID dispozitiv: {this.props.deviceId}</Text>
                <Text style={{ color: assets.color.inactive }}> *Pentru a continua, completează câmpurile.</Text>
                <Button
                  onPress={this.props.onAddPress}
                  title='Adaugă'
                  loading={this.props.isLoading}
                  buttonStyle={styles.buttonStyle} />
              </View>
            </View>
          </Overlay>
      );
    }
}
```

Figura 1.9. Clasa AppOverlay

1.5.1.1.4. Componenta de încapsulare a vizualizării și funcționalității programului

În funcție de tipul de platformă pe care este rulată o aplicație mobilă, Android sau iOS, ferestrele principale ale aplicației se numesc activități(eng.: activities), respectiv controller-e vizuale(eng.: view controller). Pentru a avea o referință comună asupra ambelor denumiri, elementele vizuale ale interfeței vor fi denumite ecrane(eng.: screens).

Așadar, aplicația va fi compusă din două ecrane principale, unul prin care se permite descoperirea dispozitivelor din proximitate și asocierea cu acestea(clasa **SearchScreen**), iar altul prin care se oferă posibilitatea de a vedea care sunt dispozitivele asociate din apropiere și totodată contactarea acestora(clasa **DevicesScreen**). Datorită similitudinii între implementările acestor componente, se va descrie amănunțit un singur obiect-ecran, clasa **SearchScreen**(Figura 1.10).

Trecerea de la un ecran la altul se face prin intermediul unui container de navigare(eng.: navigation container) care încorporează un navigator de tab-uri (eng.: tab navigator) ce are în componență ecranele anterior menționate.

Spre deosebire de clasele pur vizuale, clasele ecran sunt proiectate să conțină variabile de stare(eng.: state) prin intermediul cărora este posibilă redesenarea interfeței în momentul în care una din stări își modifică valoarea/referința.

Stările clasei **SearchScreen** au următorul scop:

- overlayVisible – valoare booleană prin care se determină dacă se va afișa overlay-ul ce conține formularul de adăugare a unui dispozitiv;
- isLoading – valoare booleană prin care se permite afișarea unui spinner în locul butonului de căutare cât timp se efectuează scanarea de dispozitive;
- isCheckingSecurityCode – valoare booleană prin care se se permite afișarea unui spinner în locul butonului de adăugare din cadrul formularului cât timp se așteaptă terminarea execuției acțiunii butonului;
- devices – lista dispozitivelor găsite la momentul ultimei scanări;
- deviceId – id-ul dispozitivului pentru care se dorește asocierea;

- deviceName/securityCode – variabile ce memorează valorile introduse în câmpurile formularului de adăugare a unui dispozitiv;
- securityCodeErrorMessage/deviceNameErrorMessage – variabile ce permit afișarea unor mesaje de eroare în cazul în care câmpurile formularului sunt completate necorespunzător.

În corpul metodei **onSearchButtonPress** se apelează metoda **searchForDevices** a clasei **BtManager** și se așteaptă scanarea dispozitivelor din proximitate. La finalul scanării, variabila de stare „devices” va fi actualizată cu noile dispozitive găsite în apropiere(lista poate fi goală) și va cauza o redesenare a interfeței.

Metoda **onAddDeviceButtonPress** are rolul de a gestiona evenimentele ce pot să apară în momentul în care butonul din formularul de asociere este apăsat. Sunt verificate câmpurile din formular așa încât să fie corect completate și se actualizează variabilele de stare ce au rol de a afișa mesajele de eroare în privința completării incorecte. Apoi se încearcă adăugarea dispozitivului și memorarea cheii de acces a acestuia în cazul în care operația a fost realizată cu succes. Totodată, dacă va apărea o eroare, variabilele de stare „overlayVisible”, „deviceName” și „securityCode” sunt resetate.

```
Class SearchScreen extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      overlayVisible: false,
      isLoading: false,
      isCheckingSecurityCode: false,
      devices: [],
      deviceId: null,
      deviceName: '',
      securityCode: '',
      securityCodeErrorMessage: '',
      deviceNameErrorMessage: '',
    }
  }
  componentDidMount() { [...] }
  onDeviceNameChange = (value) => { [...] }
  onSecurityCodeChange = (value) => { [...] }
  onToggleOverlay = () => { [...] }
  onSearchButtonPress = () => { [...] }
  onAddDeviceButtonPress = (deviceId) => { [...] }
  onSubmitAddDevice = () => { [...] }
  updateDevices = () => { [...] }
  render() { [...] }
}
```

Figura 1.10. Clasa SearchScreen

1.5.1.2. Programul pentru microcontroller

1.5.1.2.1. Tehnologia aleasă pentru implementare

Platforma de lucru aleasă pentru program este **Espressif IoT Development Framework**(ESP-IDF) deoarece aceasta este interfața de programare oferită de dezvoltatorii microcontroller-ului. Prin aceasta se pun la dispoziție un sistem de monitorizare al aplicației și librăriile „low-level" necesare dezvoltării(Figura 1.11). Totodată această alegere este motivată și de faptul că restul opțiunilor luate în calcul nu au o manevrabilitate completă asupra microcontroller-ului, dezvoltatorii acestora oferind interfețe peste librăriile native ce încapsulează doar o parte de funcționalități.

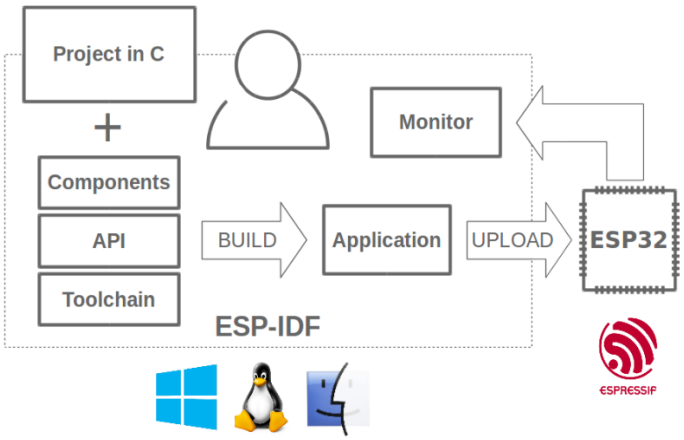

Figura 1.11. Uneltele ESP-IDF

Trebuie luat în vedere și faptul că, deși, platforma este foarte potentă din punct de vedere al creării unui program, aceasta este și dificil de folosit datorită necesității de a avea cunoștințe asupra limbajului de programare pe care îl are la bază(limbajul **C**).

Mediul de dezvoltare ales pentru a crea proiectul este **PlaformIO**, o extensie adusă editorului **Microsoft Visual Studio Code** ce accelerează și simplifică crearea și livrarea produselor încorporate.

Luând la cunoștință paragrafele anterior enunțate, primul pas în proiectarea programului a fost acela de a crea o diagramă prin care să fie redat fluxul programului(Figura 1.12).

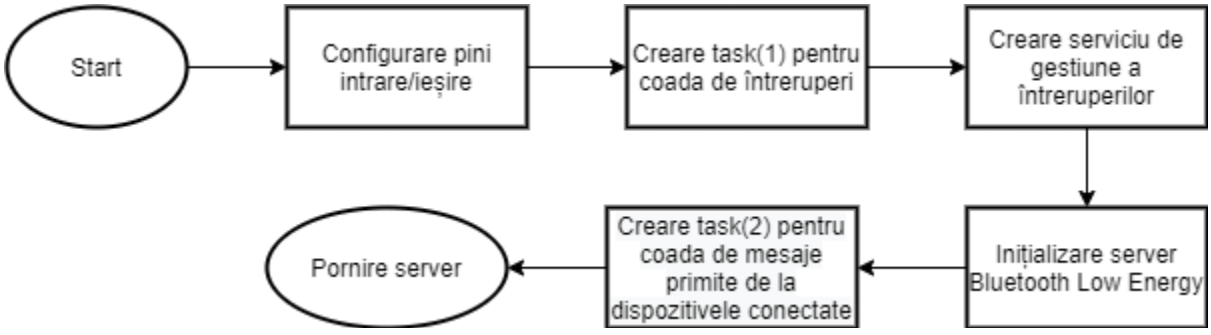

Figura 1.12. Diagrama de flux a programului microcontroller-ului

1.5.1.2.2. Configurarea pinilor de intrare/ieșire

Utilizarea pinilor de intrare/ieșire ai microcontroller-ului necesită cunoașterea dispozitivelor legate de aceștia, fapt enunțat în capitolul 1.5.2, astfel pentru a folosi pinii la care sunt conectate dioda și butonul a fost creată funcția **kf_config_gpio**(Figura 1.13). Configurarea acestora se face prin utilizarea unei structuri de configurație(**gpio_config_t)** care are în componență o mască de 64 de biți, fiecare poziție din aceasta semnificând numărul pinului folosit ca intrare/ieșire. Valoarea(„1 sau 0") găsită la fiecare poziție arată dacă pinul este folosit sau nu.

Simultan, structura conține detalii despre activarea întreruperilor, modul de utilizare(intrare/ieșire) și tipul de rezistor folosit pentru fiecare pin menționat în mască. Totodată, în această funcție sunt create coada de întreruperi, serviciul ce adaugă elemente în coadă la momentul producerii uneia și task-ul ce gestionează elementele aflate în coadă. Configurarea pinului la care este conectat buzzer-ul pasiv se face prin folosirea librăriei „ledc" deoarece prin aceasta se pot genera semnale **P.W.M.**, semnale fără de care buzzer-ul pasiv nu poate funcționa.

```
static void kf_config_gpio()
{
    gpio_config_t io_conf;
    io_conf.intr_type = GPIO_INTR_DISABLE;
    io_conf.mode = GPIO_MODE_OUTPUT;
    io_conf.pin_bit_mask = OUTPUT_MASK;
    io_conf.pull_down_en = 0;
    io_conf.pull_up_en = 0;
    gpio_config(&io_conf);

    io_conf.intr_type = GPIO_INTR_POSEDGE;
    io_conf.mode = GPIO_MODE_INPUT;
    io_conf.pin_bit_mask = INPUT_MASK;
    gpio_config(&io_conf);

    gpio_evt_queue = xQueueCreate(10, sizeof(uint32_t));
    xTaskCreate(kf_button_action_task, „kf_button_action_task", 2048, NULL, 10, NULL);
    gpio_install_isr_service(ESP_INTR_FLAG_LOWMED);
    gpio_isr_handler_add(BUTTON_PIN, gpio_isr_handler, (void *)BUTTON_PIN);
}
static ledc_channel_config_t ledc_channel = {
    .channel = LEDC_CHANNEL_0,
    .duty = 0,
    .gpio_num = BUZZER_PIN,
    .speed_mode = LEDC_HS_MODE,
    .hpoint = 0,
    .timer_sel = LEDC_HS_TIMER
};
static ledc_timer_config_t ledc_timer = {
    .duty_resolution = LEDC_TIMER_13_BIT,
    .freq_hz = FREQUECY,
    .speed_mode = LEDC_HS_MODE,
    .timer_num = LEDC_HS_TIMER,
    .clk_cfg = LEDC_AUTO_CLK,
};
static void kf_config_ledc()
{
    ledc_timer_config(&ledc_timer);
    ledc_channel_config(&ledc_channel);
    ledc_fade_func_install(0);
}
```

Figura 1.13. Configurarea pinilor de intrare/ieșire

O întrerupere este un semnal trimis către procesor, emis de hardware sau software, indicând un eveniment care necesită atenție imediată. Ori de câte ori are loc o întrerupere, controller-ul finalizează executarea instrucțiunii curente și începe executarea unei rutine de întrerupere a serviciului(eng.: Interrupt Service Routine – I.S.R.). Prin I.S.R. se comunică procesorului ce să facă atunci când are loc întreruperea. Schema logică a rutinei de întrerupere este prezentată în Figura 1.14.

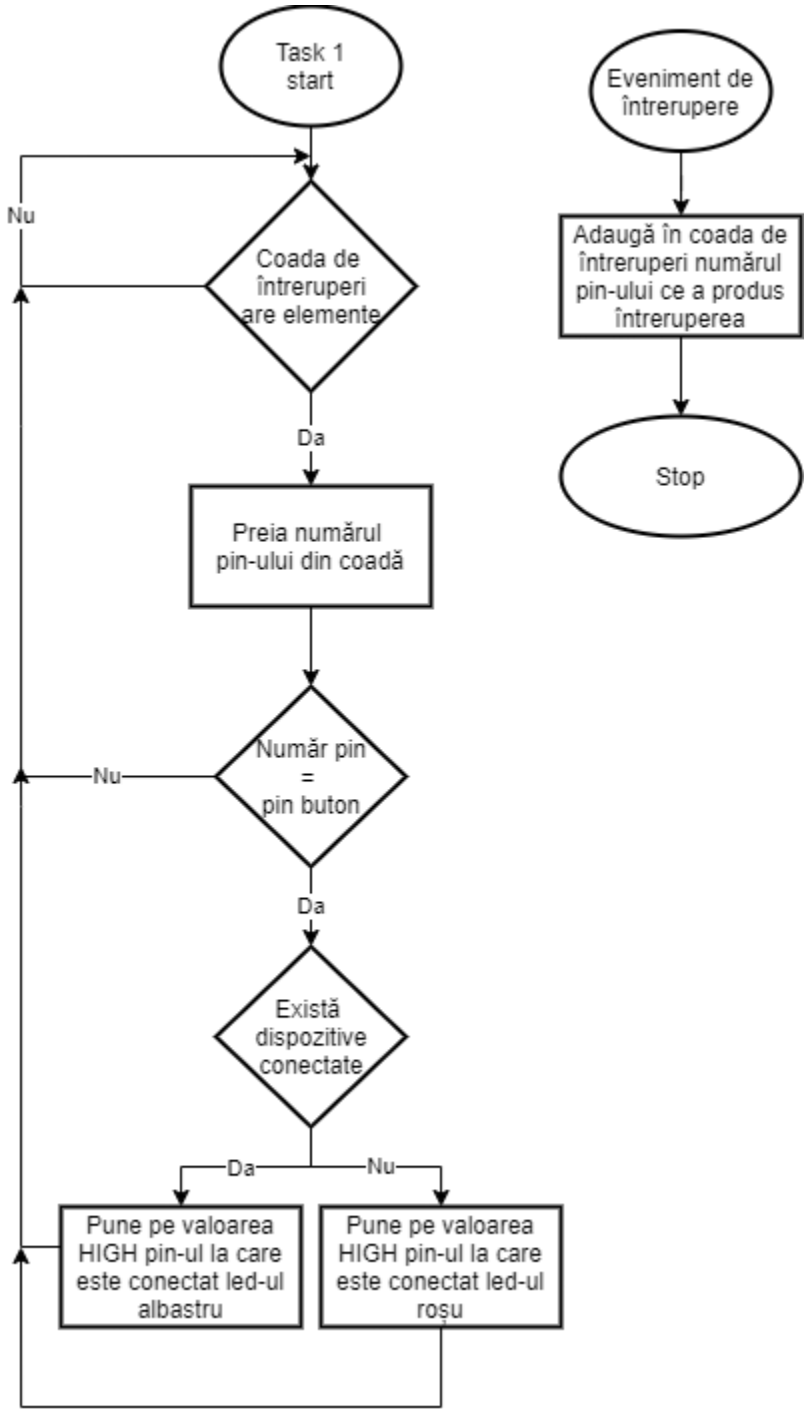

Figura 1.14. Schema logică a rutinei de întrerupere

1.5.1.2.3. Configurarea server-ului Bluetooth Low Energy

Pentru a crea server-ul, a fost preluat și adaptat exemplul din documentația oferită de dezvoltatorii ESP-IDF, astfel au fost adăugate următoarele funcționalități:

- Citirea caracteristicii serviciului de advertising va returna codul de asociere al serviciului dacă cererea a fost precedată de o autorizare realizată cu succes;
- Trimiterea unui pachet de date către caracteristica serviciului va avea ca efect trimiterea mesajului primit către un task. Task-ul va interpreta mesajul și va lua măsuri în funcție de natura acestuia(conform schemei logice ilustrată în figura 1.15);
- Evenimentul de deconectare va reseta variabila ce are ca scop memorarea răspusului validării cheii de securitate.

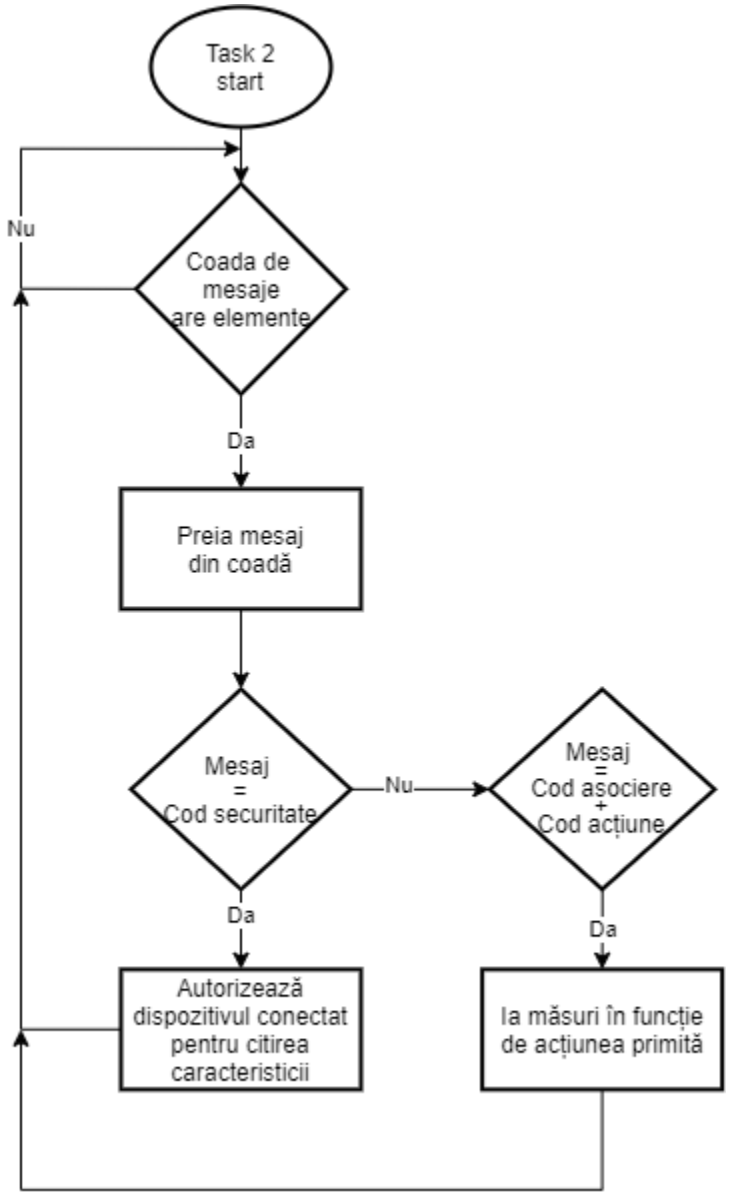

Figura 1.15. Schema logică a task-ului ce interpretează mesajele aflate în coadă

1.5.2. Componenta hardware

ESP32 este o serie de sisteme „low-cost” cu consum redus de energie, având cip integrat Wi-fi și Bluetooth cu mod dual. Seria ESP32 utilizează un microprocesor Tensilica Xtensa LX6 atât în variante dual-core, cât și în variante single-core și include comutatoare de antenă încorporate, „balun R.F.”, amplificator de putere, amplificator de recepție cu zgomot redus, filtre, și module de gestionare a alimentării.

Pentru a facilita procesul de dezvoltare al aplicației a fost utilizată o placă de dezvoltare NodeMCU ESP-32S ce are la bază un microcontroller ESP-WROOM-32. Aceasta permite încărcarea și monitorizarea programului microcontroller-ului printr-un cablu USB eliminând nevoia de a folosi alte periferice. Totodată, la pinii de intrare/ieșire ai plăcii de dezvoltare au fost conectate o diodă R.G.B., un buzzer pasiv și un buton(Figura 1.16).

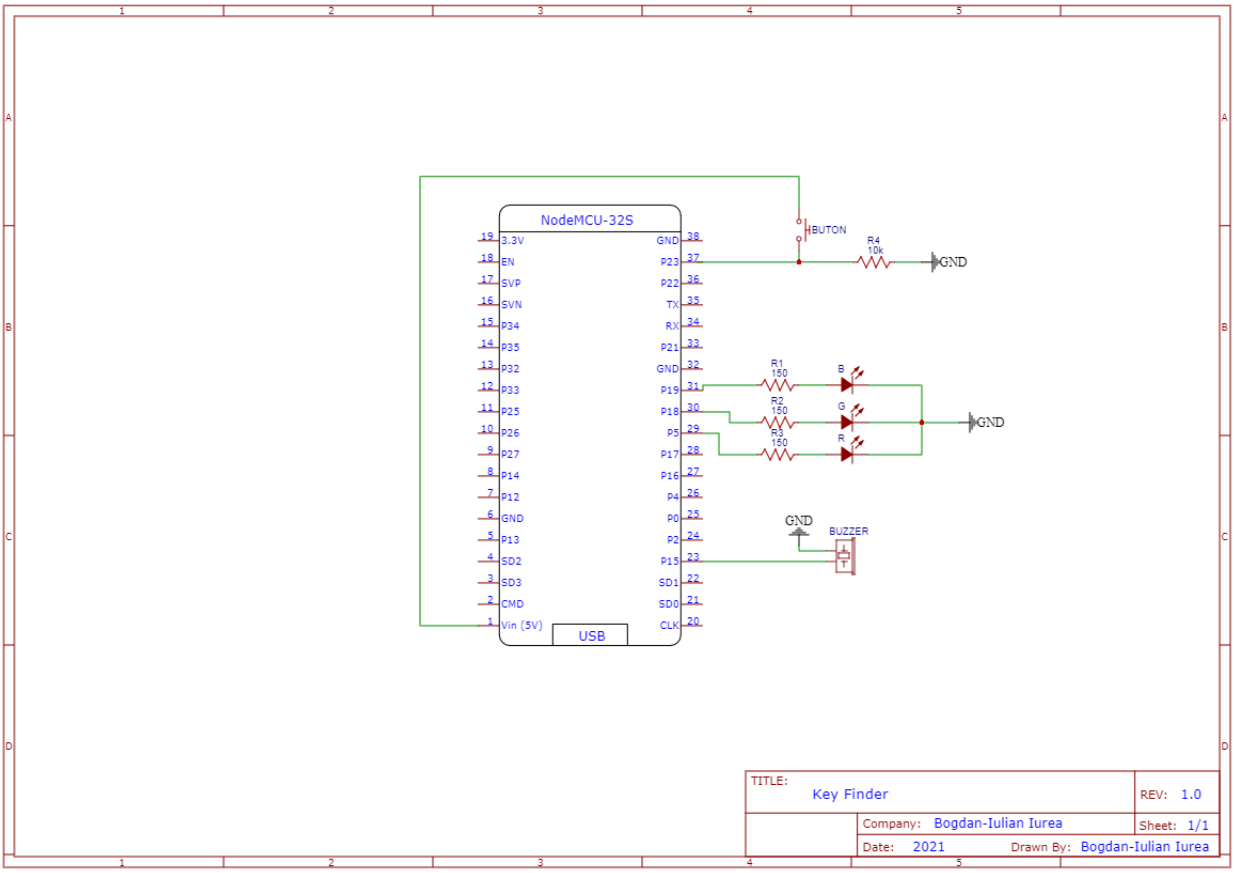

Figura 1.16. Schematica componentei hardware