
**Budapesti Műszaki és Gazdaságtudományi Egyetem**

Villamosmérnöki és Informatikai Kar

Nagy Richárd

# Külső eszközök vezérlése agy-mobil interfész felhasználásával

KONZULENS

Dr. Forstner Bertalan

Szegletes Luca

BUDAPEST, 2013

# Tartalom

# Összefoglaló

A megváltozott mozgásképességű emberek számára egy olyan alapvető dolog, mint például a villany felkapcsolása is problémát jelenthet. Számukra különösen hasznos lehet egy olyan megoldás, aminek segítségével a különböző eszközöket távolról is irányíthatják akár egy mozdulat nélkül. A dolgozatban egy ilyen megoldást megvalósító rendszer kerül bemutatásra. A rendszer 3 részből áll: egy EEG eszközből, egy tabletből, valamint a vezérelni kívánt eszközből. Az EEG segítségével a felhasználó agyi aktivitását mérjük és a kapott adatokat a tableten dolgozzuk fel, aminek a segítségével vezéreljük az irányítani kívánt eszközt. A dolgozatban kitérek arra, hogy hogyan lehet csatlakoztatni az EEG-t a tablethez, feldolgozni a beérkező jeleket illetve, hogy a tablet hogyan kommunikál az irányítandó eszközzel. A rendszer implementálásához egy Android operációs rendszert futtató tabletet használok, amihez egy Emotiv EEG kerül csatlakoztatásra. Az irányítandó eszköz pedig egy vezérelhető elosztó lesz, ami Bluetooth-on keresztül kommunikál a tablettel.

Mivel az EEG nem rendelkezett Androidos illesztőprogrammal ezért első lépésként ezt kellett elkészítenem. Az illesztőprogram elkészítése során megismertem, hogy hogyan lehet USB porton keresztül hardvereket csatlakoztatni az Androidos tabletekhez, valamint eközben az Android NDK (Native Development Kit) használatát is sikerült elsajátítanom.

Következő lépésként egy olyan alkalmazást kellett készítenem, aminek segítségével olyan agy tevékenység váltható ki, ami detektálható. Az alkalmazás 4 piktogramot jelenít meg mátrixos formában. Ezek különböző fázisban, ugyanazon a frekvencián villognak. Amikor az alany rátekint az egyik villogó képre, az inger hatására P300 jel keletkezik a laterális lebenyen, amit az EEG eszközzel mérünk. Az alkalmazás megtanulja, hogy az egyes piktogramok felvillanásakor milyen jel keletkezik, és ez alapján detektálni tudja, hogy a felhasználó éppen melyik piktogramot nézi. A felhasználó által nézett kép alapján pedig parancsokat küld a vezérelni kívánt eszköznek. A vezérelt eszköz ez esetben egy elosztó, aminek az egyes aljzatait Bluetooth-on keresztül küldött parancsokkal ki illetve bekapcsolhatunk.

# Bevezetés

Napjainkban egyre elterjedtebbek az okostelefonok, tabletek és az egyéb hordozható infokommunikációs eszközök. Ezek egyre több funkcióval és számítási kapacitással rendelkeznek, így olyan komplex alkalmazások is implementálhatóak rájuk, amikhez eddig PC-re volt szükség. Az utóbbi időben megjelentek a kis méretű EEG (elektorenkefalográf) eszközök is. Ezeket többnyire az idegtudományi kutatások során használják egy PC-hez csatlakoztatva, azonban mára a különböző mobil eszközök is elérték azt a fejlettségi szintet, hogy egy ilyen EEG csatlakoztatható legyen hozzájuk. PC-re már számos olyan alkalmazás készült, amelyek funkciói az EEG által mért adatok alapján érhetőek el. A legtöbb ilyen alkalmazás az úgynevezett P300 jelet használja ki, ami bizonyos ingerek hatására keletkezik az agyban. Ezt a reakciót már az 1960-as évek közepén felfedezték az idegtudománnyal foglalkozó kutatók [1] [2]. A P300 jel egy pozitív feszültség változás az agyban, ami az inger után 250-350 milliszekundummal keletkezik az agyban. Az első igazán nagy áttörést jelentő számítógép-agy interfész 1988-ban került bemutatásra [3]. Itt egy 6 sorból és oszlopból álló mátrixban jelenítették meg az ABC betűit. A mátrixban a különböző sorokat, illetve oszlopokat villogtatták, ha az alany által figyelt karakter az éppen felvillanó sorban vagy oszlopban helyezkedett el, akkor az agyban P300 jel keletkezett. Ahhoz, hogy az EEG-ből érkező zajokat kiszűrjék nem csak egy jel keletkezését várta meg a rendszer, hanem több mérési eredményt rögzített és azok átlaga alapján döntött. Ezzel a módszerrel az alanyoknak 3,4-4,3 karakter/perc gépelési sebességet sikerült elérniük. A P300 jel detektálásának tökéletesítésén a mai napig dolgoznak a kutatók [4]. A dolgozatban bemutatásra kerülő rendszer nagyban támaszkodik az imént említett kutatási eredményekre, algoritmusokra.

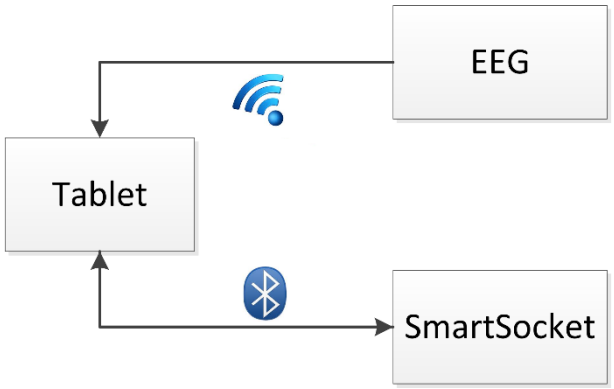
A megváltozott mozgásképességű emberek számára egy olyan alapvető dolog, mint például a villany felkapcsolása is problémát jelenthet. Számukra különösen hasznos lehet egy olyan megoldás, aminek segítségével a különböző eszközöket távolról is irányíthatják akár egy mozdulat nélkül. Erre is nyújthat megoldást az EEG használata. Az EEG segítségével, az agyi aktivitást mérve utasítást küldhetünk a különböző eszközöknek. Ezzel a megoldással az olyan mértékben mozgássérült személyek is képessé válhatnak a környezetükben található tárgyak kezelésére, akik még a kezüket sem tudják megmozdítani. A rendszer segítségével a mozgáskorlátozottak életminősége nagymértékben javítható, mivel használatával nagyobb önállóságra tehetnek szert. Különösen fontos a mozgássérült gyerekek megfelelő oktatása, felkészítése a felnőtt életre, ezért a rendszer tervezése során olyan intézmények is bevonásra

kerültek, amelyek már régóta foglalkoznak ezzel a problémával. Ilyen például a Mozgásjavító Általános Iskola, Szakközépiskola, Egységes Gyógypedagógiai Módszertani Intézmény és Kollégium, amely már közel 110 éve foglalkozik mozgáskorlátozott gyerekek oktatásával és évről-évre egyre több tanulója van. A másik nagyobb intézmény, amelyikkel együttműködés van folyamatban a Mozgássérült Emberek Rehabilitációs Központja, melynek főcélja a mozgássérült emberek rehabilitációja és társadalmi integrációjának elősegítése. A rendszer iránt nem csak nagy érdeklődés van, hanem igény is.

A rendszer tervezése során azért döntöttem a hordozható eszközök alkalmazása mellett, mert így nem lesz a felhasználó a számítógéphez kötve, hanem szabadon változtathatja helyzetét. Ezért fontos az, amit már korábban is említettem, hogy a mobil eszközök napjainkra már elérték azt a fejlettségi szintet, hogy egy olyan komplex rendszer is megvalósítható legyen rajtuk, ami a dolgozatban bemutatásra kerül. A hardverelemek kiválasztásánál figyelembe vettem, hogy a piacon jelenleg melyek a legelterjedtebb mobileszközök [5]. Ezért választottam egy Android operációs rendszert futtató eszközt. Az Android operációs rendszert a Google fejleszti, és a forráskódja is szabadon elérhető. Az Android alapú eszközök a legtöbb esetben rendelkeznek USB porttal, valamint WiFi és Bluetooth modullal, amelyekkel könnyedén megvalósítható a vezeték nélküli kommunikáció.

# Rendszerterv

A rendszer 3 hardver elemből áll. A központi egység egy Android operációs rendszert futtató tablet (Motorola Xoom). Ehhez csatlakozik további perifériaként egy vezeték nélküli Emotiv EPOC típusú 14 csatornás EEG (elektorenkefalográf), aminek segítségével a felhasználó agyában található neuronok elektromos aktivitása mérhető. Az EEG-hez tartozik egy USB (standard A típusú) portra csatlakoztatható 2,4 GHz-es rádióvevő is, ezt kell csatlakoztatni a tablethez. A harmadik elem pedig egy úgynevezett SmartSocket, ami egy vezérelhető elosztó. A SmartSocket egyes aljzatai ki- illetve bekapcsolhatóak Bluetooth-on keresztül küldött parancsok segítségével. Az 1. ábra mutatja a rendszer hardveres felépítését.

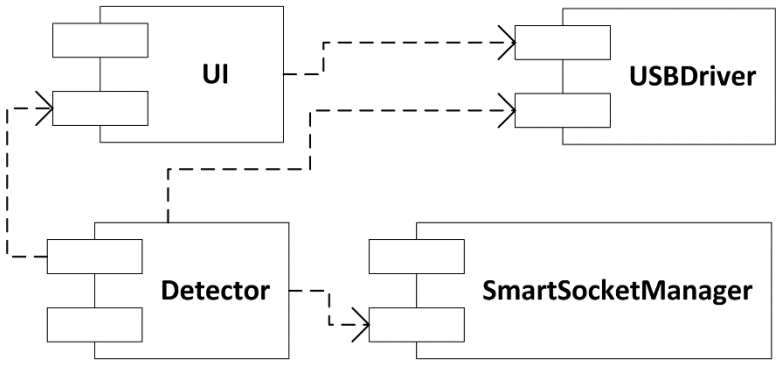



*1. ábra A rendszer hardveres felépítése*

Ahogy az ábrán is látható a tablet és az EEG között a kommunikáció egy irányú. Az EEG eszköz nem igényel semmilyen vezérlést, a csatlakoztatást követően beolvashatóak róla a mért adatok. Ezzel szemben a SmartSocket vezérléséhez kétirányú kommunikáció szükséges, mivel az eszköz a fogadott parancsok végrehajtásának eredményéről minden esetben üzenetet küld az őt vezérlő egységnek.

A rendszer szoftveresen egyetlen alkalmazásból áll. Az alkalmazás feladata az EEG-vel mért adatok feldolgozása és ez alapján a SmartSocket vezérlése. Az alkalmazás 4 piktogramot jelenít meg mátrixos formában. Ezek különböző fázisban, ugyanazon a frekvencián villognak. Amikor az alany rátekint az egyik villogó képre, az inger hatására p300 jel keletkezik a laterális lebenyen, amit az EEG eszközzel mérünk. Az alkalmazás megtanulja, hogy a felhasználó által nézett piktogram felvillanásakor milyen jel keletkezik, és később ez alapján detektálni tudja, hogy a felhasználó éppen melyik piktogramot nézi. A detektált kép alapján parancsot küld a SmartSocketnek.

Az alkalmazás 4 nagyobb komponensre bontható. Az USBDriver komponens az EEG illesztőprogramja, ennek segítségével csatlakozhatunk az EEG-hez. A SmartSocketManager komponens a SmartSockettel történő kommunikációért felelős. A Detector komponens dolgozza fel az EEG-vel mért adatokat, és egy rövid tanulási folyamat után ez dönti el, hogy a felhasználó melyik képet nézi. Az utolsó és egyik legfontosabb komponens a UI, feladata csupán annyi, hogy a piktogramokat megfelelően villogtassa [3] a felhasználónak.




*2. ábra Komponensek*

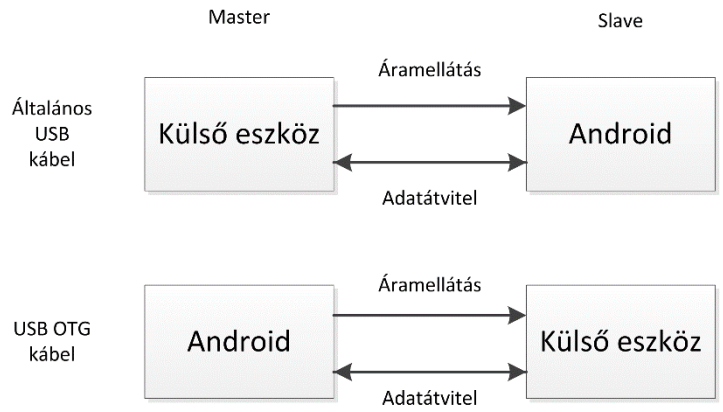
A 2. ábra szemlélteti az alkalmazás felépítését. Ahogyan az ábrán is látható a Detector komponens vezérli a SmartSocketManagert, valamint megkapja az USBDriver által mért adatokat, továbbá lekérdezheti a UI komponenstől, hogy a képek milyen sorrendben villognak. A UI komponens kihasználja, hogy az USBDriver nem csak a mért adatokat adja meg, hanem egy csomagsorszámot is, ami a fix mintavételi frekvencia miatt órajelként használható a képek villogtatásához.

# EEG illesztőprogram Android operációs rendszerre

## Az EEG csatlakoztatása

Ahogy azt már korábban említettem az EEG eszközhöz tartozik egy 2,4GHz-en üzemelő USB portra csatlakoztatható rádióvevő, aminek segítségével az EEG vezeték nélkül csatlakoztatható számítógépekhez. Mielőtt további részletekbe bocsátkoznék, fontosnak tartom az USB porton csatlakozó eszközök általános működésének bemutatását, mivel ez volt az első probléma, amibe az illesztőprogram megírása során beleütköztem. Az USB porton történő kommunikáció során a két eszköz között master-slave kapcsolat alakul ki. Ez annyit jelent, hogy a master eszköz vezérli a teljes kommunikációt, míg a slave eszköz csak végrehajtja a kapott parancsokat, illetve adatokat küld a masternek, ha az arra kéri. Általában az eszközöket úgy tervezik meg, hogy csak az egyik szerepet tudják betölteni például a számítógépek tipikusan a master szerepet töltik be, a perifériák (pl.: nyomtató, billentyűzet, egér) pedig a slave-ként üzemelnek. A slave eszközök áramellátásáról is gyakran a master eszköz gondoskodik.

Mivel az Android operációs rendszert futtató tabletek alapértelmezés szerint USB kábel csatlakoztatásakor slave üzemmódban működnek, ezért a rádióvevőt egy speciális kábellel kell csatlakoztatni a tablethez, ez az úgynevezett USB OTG (On-The-Go) kábel. Ez a kábel nem csak azt a problémát oldja meg, hogy a rádióvevő standard A típusú csatlakozóval rendelkezik és a tabletnek micro B típusú bemenete van, hanem azt is, hogy a tablet ne slave hanem master módban üzemeljen. A csatlakozás módját a 3. ábra szemlélteti.




*3. ábra Eszköz csatlakoztatása USB portra*

Miután sikerült csatlakoztatni az eszközt a tablethez egy újabb probléma merült fel. Az Android rendszere történő fejlesztés során az alkalmazások USB porton keresztül

telepíthetőek a céleszközökre és a debuggolás is ezen keresztül megy az Android SDK részét képező adb (Android Debug Bridge) program segítségével. Mivel a rádióvevő elfoglalta az egyetlen redelkezésre álló USB portot így más alternatívát kellett találni a fejlesztő környezet és a tablet összekapcsolására. Szerencsére az újabb verziójú adb már támogatja a hálózaton keresztüli csatlakozást a céleszközhöz. Ehhez csak annyit kellett tennem, hogy a csatlakoztattam a tabletet a számítógéphez és a parancssorban kiadtam a következő utasításokat.

```
adb tcpip 5555
```

Az előbbi parancs hatására a tablet és az adb átvált hálózaton keresztüli csatlakozási módra és a tablet leválasztható a számítógépről. A tcpip paraméter után adható meg, hogy melyik porton keresztül akarok csatlakozni a tablethez, a kommunikáció során TCP protokollt fog használni a fejlesztői környezet. Nem elég a csatlakozás módját megadni, ha hálózaton keresztül akarunk csatlakozni a céleszközhöz, akkor még a következő paranccsal csatlakoztatni is kell az eszközt, ez USB módban nem szükséges.

```
adb connect 192.168.1.2:5555
```

A connect paraméter után a céleszköz IP címét kell megadni, majd egy „:” után az első parancsban megadott portot. Ezután ugyanolyan egyszerűen telepíthetőek és debugolhatóak az alkalmazások a fejlesztői környezetből mintha USB porton lenne csatlakoztatva a céleszköz.

Az USB porton keresztüli csatlakozás az alábbi paranccsal állítható vissza.

```
adb usb
```

## Az EEG illesztőprogramja

Az EEG-hez elérhető egy C nyelven írt illesztőprogram. Mivel az Android rendszere Java nyelven készíthetőek alkalmazások így a rendelkezésre álló illesztőprogramot csak részben tudtam felhasználni. Az Android rendszerre natív nyelven is (C, C++) írhatóak programok az Android NDK (Native Development Kit) segítségével. Javaból is meghívhatjuk a natív nyelven írt függvényeket ehhez a JNI (Java Native Interface) használatára van szükség.

A C nyelven írt illesztőprogram több olyan osztálykönyvtárra hivatkozik, ami Androidra nem érhető el. Az egyik ilyen a hidapi, ami az USB-s HID eszközök kezelését teszi lehetővé. Az Android NDK-ból az USB nem érhető el közvetlenül, ezért a hidapi nem használható. A

másik az mcrypt osztálykönyvtár, ami különböző titkosítási algoritmusok implementációját tartalmazza. A későbbiekben részletesen is bemutatom, hogy hogyan sikerült kiküszöbölni az előbb említett osztálykönyvtárak hiányát.

Androidon az USB-s eszköz csatlakoztatásakor a felhasználónak ki kell választania, hogy melyik alkalmazás használhatja azt. Ahhoz, hogy az általam készített program is megjelenjen a választható alkalmazások listájában fel kellett vennem két intent filtert az alkalmazás manifest állományában, ami a következő pár sorral tehető meg.

```
<intent-filter>
<action android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED" />
</intent-filter>
<meta-data android:name="android.hardware.usb.action.USB_DEVICE_ATTACHED"
android:resource="@xml/device_filter" />
```

Valamint meg kellett adni azt is, hogy az alkalmazás használja az USB portot. Ez a következő sorral tehető meg, amit szintén a manifest állományban kell elhelyezni.

```
<uses-feature android:name=" android.hardware.usb.host" android:required="true" />
```

A csatlakozó eszközök listája az UsbManager osztállyal kérdezhető le. Egy eszközhöz egy UsbDevice típusú objektum tartozik. Az eszköznek több tulajdonsága is lekérdezhető, amire nekem szükségem van az a VID (Vendor ID), PID (Product ID), valamint az eszköz sorozatszáma. A VID és a PID csupán arra szolgál, hogy azonosítsam, hogy az EEG-hez tartozó rádióvevő lett-e csatlakoztatva a tablethez, vagy egy másik eszköz. A sorozatszám azért fontos, mert a rádióvevőről érkező adatok AES titkosítással vannak ellátva, aminek a kulcsa a sorozatszámból nyerhető ki. A kapcsolat a tablet és az USB eszköz között az UsbConnection osztály segítségével hozható létre. Az előbb említett osztályok segítségével teljes mértékben sikerült áthidalnom a hidapi hiánya okozta problémát. Az USB portról történő olvasást külön szálon valósítottam meg, hogy a felhasználói felület ne akadjon meg.

Az EEG-ről az adatok 32 bájtos csomagokban érkeznek, amiket dekódolni kell. Mint azt már korábban említettem a kódolás AES titkosítást használ, aminek dekódolása az elérhető illesztőprogramban az mcrypt osztálykönyvtár segítségével van megvalósítva. Mivel ez Java alatt nem elérhető és olyan további osztálykönyvtárakat használ, amik az NDK korlátozásai miatt nem használhatóak, ezért meg kellett keresnem a forráskódjában az AES titkosításhoz tartozó részeket. Miután sikerült ezeket azonosítani le tudtam cserélni az illesztőprogram dekódolásához tartozó részét úgy, hogy ne használja az mcrypt által nyújtott funkciókat, hanem csak az abból kinyert AES dekódolót. Ezzel az mcrypt hiányát is sikerült kiküszöbölni.

Ezen lépések után az illesztőprogram két külön álló részből állt, amik még összekötésre vártak. Az egyik fele a Java oldalon beolvasta az adatokat az USB portról, a másik pedig készen állt az adatok dekódolására. A két rész összekötésére a JNI segítségével került sor.

Az előzőekből is jól látható, hogy az illesztőprogram elkészítése nem volt triviális feladat, sok utánajárást, mérést és türelmet igényelt. Talán ennek köszönhető, hogy ismereteim szerint az eszköz ily módon történő illesztését máshol még nem végezték el.

A továbbiakban bemutatom az illesztőprogram implementációs részleteit a Java oldallal kezdve. Javában két osztályra van szükség. Az egyik az UsbDriver osztály, amit a Thread osztályból származtattam le. A másik pedig az Event osztály, ami mérési adatokat tartalmazhat. Egy interfészt is definiáltam EventListener néven. Az EventListeneren keresztül tudja az UsbDriver jelezni egy Event objektummal, hogy új mérési eredmény áll rendelkezésre. Az interfész egyetlen függvényt tartalmaz ez az onEvent(Event e).

Az UsbDriver konstruktora két paramétert vár az egyik egy UsbManager típusú objektum, a másik pedig egy EventListenert megvalósító objektum.

```
public UsbDriver(UsbManager man, EventListener aListener) {
	manager = man;
	connected = false;
	listener = aListener;
}
```

Az USB eszközhöz a connect() függvény segítségével csatlakozhatunk, ha sikerül csatlakozni, akkor true-val különben false-zal tér vissza.

```
public boolean connect() {
	// Csatlakoztatott eszközök listája
	HashMap<String, UsbDevice> deviceList = manager.getDeviceList();
	if(deviceList.isEmpty()) // Nincs csatlakoztatva eszköz
	{
		return false;
	}
	Iterator<UsbDevice> deviceIterator = deviceList.values().iterator();
	while(deviceIterator.hasNext()){
		device = deviceIterator.next();
		// Nem a megfelelő eszköz van csatlakoztatva
		if(device.getProductId()!=0xed02 && device.getVendorId()!=0x1234)
			continue;
		connection = manager.openDevice(device); // Csatlakozás az eszközhöz
		serial = connection.getSerial();
		if(!serial.startsWith("SN")) // A sorozatszám nem megfelelő formátumú
		{
			connection.close();
			continue;
		}
		// Bemeneti interfész keresése (host felé)
		for(int i=0;i<device.getInterfaceCount();i++)
```

```
            {
                  if(device.getInterface(i).getEndpoint(0).getDirection()==
                  UsbConstants.USB_DIR_IN)
                  {
                        inInterface = device.getInterface(i);
                        inEndPoint =inInterface.getEndpoint(0);
                  }
            }
            if(inInterface !=null & inEndPoint!=null) // Interfész lefoglalása
            {
                  connection.claimInterface(inInterface, true);
            }
            initEmotivDriver(serial.getBytes()); // Natív oldali inicializálás
            initializeSensor(1); // Natív oldali inicializálás
            connected = true;
            return true; // Sikeres csatlakozás
      }
      connected = false;
      return false; // Van csatlakoztatott eszköz, de nem az EEG vagy nem sikerült
                     //csatlakozni
}
```

Csatlakozáskor először ellenőrzésre kerül, hogy vannak-e csatlakoztatott eszközök, ha igen megkeresi az EEG rádióvevőjét a VID és PID tulajdonság alapján. Amennyiben meg van a keresett eszköz lekérdezi a sorozatszámát. Ha a sorozatszám formátuma nem megfelelő, akkor tovább folytatja a keresést, különben tovább folyik a kapcsolat felépítése a bemeneti interfészek lefoglalásával. Végezetül inicializálja az illesztő program natív részét.

Az adatok beolvasása a readData() függvénnyel történik.

```
private byte[] readData(){
      if(connected) // Kapcsolat ellenőrzése
      {
            byte[] data = new byte[32];
            int readBytes = connection.bulkTransfer(inEndPoint, data,
data.length, 0); // Olvasás az eszközről
            if(readBytes<32) // Sikertelen olvasás vagy nincs új adat
                  return null;
            return data;
      }
}
```

Az eszközről mindig 32 bájtos csomagokban érkeznek az adatok, ha ennél kevesebb érkezik vagy nem volt mit olvasni, akkor az olvasás sikertelenségét úgy jelzi, hogy null-lal tér vissza.

A tényleges olvasás a szál indítása után folyamatosan megy, amihez a Thread osztályból származó run() függvényt írtam felül.

```
public void run() {
      while(active)
      {
            packet = readData();
            if(packet!=null)setNewData(packet);
```

```
	}
}
```

Az olvasáskor csupán annyi történik, hogy a beolvasott bájt tömböt továbbítja a rendszer a setNewData() függvényhívással a natív résznek. Még egyetlen függvényt kell Java oldalon ismertetnem, ami a rawEegO1O2Data(). Mivel az alkalmazás csak az O1 és O2 csatornáról érkező adatokat használja ezért csak azokat, valamint a csomagsorszámot adja át a natív rész a Java oldalnak ezen a függvényen keresztül, ami továbbítja az EventListenert megvalósító objektumnak.

```
public void rawEegO1O2Data(double o1, double o2, int buffer)
{
	Event e = new Event(o1,o2,buffer);
	listener.onEvent(e);
}
```

Meg kell még jegyeznem, hogy a natív függvények eléréséhez a Java osztálynak be kell töltenie a natív osztálykönyvtárakat, amit úgy lehet megtenni, hogy az osztály deklarációt kiegészítjük az alábbi kódrészlettel.

```
static {
		System.loadLibrary("c"); // C könyvtár
		System.loadLibrary("gnustl_shared"); // STL támogatás
		System.loadLibrary("InnolearnSensorsLib"); // Saját osztálykönyvtár
	}
```

A hívások paramétereiben a betölteni kívánt osztálykönyvtár nevét kell megadni. Továbbá azt is jelezni kell, hogy melyik függvényeket használja az osztály a betöltött osztálykönyvtárakból. Erre a következő kódrészletben látható példa.

```
public native String initEmotivDriver(byte[] serial);
public native void setNewData(byte[] raw);
public native void initializeSensor(int type);
```

Meg kell adni, hogy a függvény publikus, natív, a visszatérési értékét, nevét és paramétereit. Ezután a függvény úgy használható, mint bármelyik Javában írt metódus. Ezzel a Java oldali része az illesztőprogramnak véget is ért, most bemutatom a natív oldalt.

A natív oldal úgy lett elkészítve, hogy akár további EEG eszközök illesztőprogramja is könnyedén integrálható legyen bele. Mivel ezek az eszközök többnyire több csatornával rendelkeznek ezért létrehoztam egy általános osztályt, ami a mérési eredmények tárolására szolgál, ez a ChannelSet.

```
class ChannelSet
{
private:
```

```
	double* channelValue;
	int* channelQuality;
	int current;
	int size;
public:
	ChannelSet(int size=1);
	virtual ~ChannelSet();
	void addChannel(double value, int quality);
	double getChannelValue(int id);
};
```

Az osztály konstruktorában meg kell adni, hogy hány csatorna értéke kerül tárolásra, majd az addChannel() függvénnyel adható hozzá egy csatorna mérési eredménye, illetve a getChannelValue(int id)-vel kérdezhető le az adott sorszámú csatornán mért érték. Az EEG-hez is tartozik egy általános osztály, az EEGSensor.

```
enum DeviceType
{
	Emotiv = 1
};

class EEGSensor {
private:
	int sampleRate;
	int channelCount;
	int bufferSize;
	ChannelSet* buffer;
	string* channelNames;
public:
	int currentBufferPosition;
	SignalProcessorPipeline** pipelines;
	int pipelineCount;

public:
	EEGSensor();
	~EEGSensor();
	int getChannelCount() const;
	void setChannelCount(int channelCount);
	int getSampleRate() const;
	void setSampleRate(int sampleRate);
	void addToBuffer(ChannelSet channelSet, JNIEnv* env, jobject caller);
	ChannelSet* getBuffer() const;
	void setBuffer(ChannelSet* buffer);
	int getBufferSize() const;
	void setBufferSize(int bufferSize);
	void initializeSensor(int type);
	DeviceType deviceType;
}
```

Az osztály rendelkezik egy előre beállítható méretű bufferrel, amiben a mért adatok tárolásra kerülnek az imént ismertetett ChannelSet formájában. Megadhatóak továbbá a csatornákhoz tartozó nevek és mintavételi frekvencia. Az initializeSensor() segítségével megadható, hogy milyen típusú EEG eszköz van csatlakoztatva és ez alapján beállíthatóak automatikusan a paraméterek. Az osztály továbbá rendelkezik egy SignalProcessorPipeline típusú

tagváltozóval. Ez az osztály különböző jelfeldolgozással kapcsolatos műveltek elvégzésére képes, mivel az általam készített rendszer nem használja, ezért ezt most nem mutatom be részletesen.

A következő kódrészlet az inicializálást mutatja be.

```
void EEGSensor::initializeSensor(int type)
{
	deviceType = type;
	switch (deviceType)
	{
	case Emotiv:
		channelCount = 14; // csatornaszám beállítása
		sampleRate = 128; // mintavételi frekvencia
		bufferSize =128; // buffer méret
		buffer = new ChannelSet[bufferSize];
		channelNames = new string[channelCount]; // csatorna nevek
		channelNames[0]="F3";
		// További csatorna nevek beállítása
		for(int i=0;i<pipelineCount;i++) //Jelfeldolgozók inicializálása
		{
			pipelines[i]->initialize(128,14,channelNames,sampleRate);
		}
		break;
	default:
		break;
	}
}
```

A buffer feltöltése az addToBuffer függvénnyel történik.

```
void EEGSensor::addToBuffer(ChannelSet channelSet, JNIEnv* env, jobject caller)
{
	buffer[currentBufferPosition]=channelSet;
	// A hívó objektum osztályának lekérdezése
	jclass objectClass = env->GetObjectClass(caller);
	// Java metódus meghívása
	jmethodID objectMethod = env->GetMethodID
(objectClass,"rawEegO1O2Data","(DDI)V");
	env->CallVoidMethod
(caller,objectMethod,channelSet.getChannelValue(12),channelSet.getChannelValue(11)
,currentBufferPosition);
	currentBufferPosition++;
	if(currentBufferPosition==bufferSize) // Buffer megtelt
	{
		for(int i=0;i<pipelineCount;i++) // Jelfeldolgozók futtatása
		{
			pipelines[i]->process(buffer,env, caller);
		}
		currentBufferPosition=0;
	}
}
```

A bejövő adatok bekerülnek a bufferbe, illetve a megfelelő csatornák értékei átadásra kerülnek a Java oldalnak a Java oldalon található rawEegO1O2Data() függvény

meghívásával. Majd ezután, ha megtelt a buffer, akkor a jelfeldolgozók elvégzik a feladataikat a tárolt adatokon. Ahogy a kódban látható a metódusnak van egy JNIEnv és egy jobject típusú paramétere is. Ezek a Java és a natív oldal összekötéséhez szükségesek, aminek a folyamatát később részletesen is bemutatom, de előtte szeretném magának az Emotiv EPOC eszköznek az illesztését bemutatni.

Az eszköz illesztőprogramja C nyelven van írva, ezért az eszköz aktuális állapota nem egy osztályban, hanem a következő struktúrában van tárolva.

```
struct emokit_device {
	unsigned char serial[16]; // Eszköz sorozatszáma
	RI td; // AES dekódoláshoz struktúra
	unsigned char key[EMOKIT_KEYSIZE]; // titkosító kulcs
	unsigned char *block_buffer; // átmeneti tár dekóddoláshoz
	int blocksize; // aktuális blokk méret
	struct emokit_frame current_frame; // utolsó dekódolt adatok
	unsigned char raw_frame[32]; // nyers kódolt adatok
	unsigned char raw_unenc_frame[32]; // nyers dekódolt adatok
	unsigned char last_battery; // az eszköz töltöttsége
	struct emokit_contact_quality last_quality; // csatlakozások minősége
};
```

A dekódolt adatokat pedig a következő struktúra tárolja.

```
struct emokit_frame {
	unsigned char counter; // 0-128 folyamatosan nő
	// A csatornákon mért értékek
	double F3, FC6, P7, T8, F7, F8, T7, P8, AF4, F4, AF3, O2, O1, FC5;
	struct emokit_contact_quality cq; // csatlakozások minősége
	char gyroX, gyroY; // giroszkóp adatai
	unsigned char battery; // töltöttség
};
```

Az eszközhöz tartozó struktúra lefoglalása az emokit_create() metódussal történik.

```
struct emokit_device* emokit_create()
{
	struct emokit_device* s =
	(struct emokit_device*)malloc(sizeof(struct emokit_device));
	memset(s,0,sizeof(struct emokit_device));
	return s;
}
```

A struktúra lefoglalása után inicializálni be kell állítani a titkosítás dekódoló kulcsát. Erre szolgálnak a következő függvény.

```
int emokit_init_crypto(struct emokit_device* s, int dev_type) {
	emokit_get_crypto_key(s, dev_type);
	s->blocksize = _mcrypt_get_block_size(); // 16
	s->block_buffer = (unsigned char *)malloc(s->blocksize);
	_mcrypt_set_key(&(s->td),s->key,EMOKIT_KEYSIZE); // AES inicializáslás
	return 0;
}
```

A fenti kódrészletben található emokit_get_crypto_key(s, dev_type); függvény a titkosító kulcs generálását végzi el.

A nyers kódolt adatok dekódolását a következő függvény mutatja be.

```
int emokit_get_next_raw(struct emokit_device* s) {
	// Két darab 16 bájtos blokkra kell bontani a nyers adatot
	// Elsó blokk
	if (memcpy (s->block_buffer, s->raw_frame, s->blocksize)) {
		_mcrypt_decrypt(&s->td,s->block_buffer);
		memcpy(s->raw_unenc_frame, s->block_buffer, s->blocksize);
	}
	else {
		return -1;
	}
	// Második blokk
	if (memcpy(s->block_buffer, s->raw_frame + s->blocksize, s->blocksize)) {
		_mcrypt_decrypt(&s->td,s->block_buffer);
		memcpy(s->raw_unenc_frame + s->blocksize, s->block_buffer,
		s->blocksize);
	}
	else {
		return -1;
	}
	return 0;
}
```

A nyers adatot két darab 16 bájtos blokkban kell dekódolni. A dekódolás után kapott nyers adatokat ezután fel kell dolgozni. Ezt a folyamatot a következő függvény végzi el.

```
struct emokit_frame emokit_get_next_frame(struct emokit_device* s) {
	struct emokit_frame k;
	memset(s->raw_unenc_frame, 0, 32);

	if (emokit_get_next_raw(s)<0) { // Sikertelen dekódolás
		k.counter=0;
		return k; // Üres struktúrával tér vissza
	}
	memset(&k.cq,0,sizeof(struct emokit_contact_quality));
	// Csak a 128-as számű csomagben jön töltöttség jelzés
	if (s->raw_unenc_frame[0] & 128) {
		k.counter = 128;
		k.battery = battery_value( s->raw_unenc_frame[0] );
		s->last_battery=k.battery;
	} else {
		k.counter = s->raw_unenc_frame[0];
		k.battery = s->last_battery;
	}
	//Csatornák értékeinek lekérése
	k.F3 = get_level(s->raw_unenc_frame, F3_MASK);
	k.FC6 = get_level(s->raw_unenc_frame, FC6_MASK);
	//...
	k.FC5 = get_level(s->raw_unenc_frame, FC5_MASK);
	// Giroszkóp adatok
	k.gyroX = s->raw_unenc_frame[29] - 102;
	k.gyroY = s->raw_unenc_frame[30] - 104;
	// Csatlakozás minőségének frissítése
```

```
	k.cq=handle_quality(s);
	return k;
}
```

Ha nem sikerül dekódolni az adatokat, akkor egy üres struktúrával tér vissza a függvény. Az EEG akkumulátorának töltöttségi szintjéről az eszköz 1 másodpercenként, vagyis minden 128. csomagban küld értesítést. A get_level() függvény segítségével kapható meg az egyes csatornákhoz tartozó mért érték a dekódolt adatok és a csatornához tartozó maszk megadásával, amik a forrásállományban konstansként vannak deklarálva. Megjegyezném, hogy a fenti kódrészletben nincs feltűntetve az összes csatorna lekérdezése.

Végezetül még be kell mutatnom, hogy hogyan is kerül a Java és a natív oldal összekötésre. Azokat a függvényeket, amik elérése szükséges a Java oldalról speciális makrókkal kell ellátni és az elnevezésük is kötött. Ez nem az Android NDK sajátossága, hanem a JNI követeli meg. A függvény deklarációnak a következő formátumnak kell megfelelnie.

```
JNIEXPORT visszatérési érték típusa JNICALL Java_package_neve_függvénynév(
		JNIEnv* env, jobject object, további paraméterek)
```

A package nevében található pontokat „_” karakterre kell cserélni. Ahogy az a Java oldal ismertetésénél látszott 3 natív oldali függvényt hívható meg Javából.

```
JNIEXPORT void JNICALL
Java_hu_android_bme_innolearn_smartsocketeeg_UsbDriver_initializeSensor(
		JNIEnv* env, jobject object, jint type) {
	eegSensor = new EEGSensor();
	switch (type) { // EEGSensor inicializálása a megadott típussal
	case 1:
		eegSensor->initializeSensor(Emotiv);
		break;
	default:
		break;
	}
}
```

Ahogy az látható a Java oldalról egyetlen int típusú paraméter érkezik, ami alapján az általános EEGSensor objektum inicializálásra kerül. Értelmetlennek tűnhet, hogy csak egy case ág van, de ez azért van, mert mint azt már korábban említettem az illesztőprogram ezen része általános, és ha egy másik típusú EEG-t is támogatni szeretnénk, akkor elegendő felvenni az enumerációk közé egy új értéket, és elkészíteni a hozzátartozó specifikus részeket. Ebből kifolyólag az illesztőprogram további funkcióit (pl.: jelfeldolgozók) nem kell külön elkészíteni az új hardverhez. Jelenleg csak az Emotiv EEG támogatott, ezért van csak egy case ág.

```
JNIEXPORT void JNICALL
Java_hu_android_bme_innolearn_smartsocketeeg_UsbDriver_initEmotivDriver(
```

```
            JNIEnv* env, jobject object, jbyteArray serial)

            {
      device = emokit_create(); // Eszközhöz tartozó struktúra lefoglalása
      signed char* tmp = (signed char*) malloc(16);
      // Java oldali byte tömb másolása
      env->GetByteArrayRegion(serial, 0, 16, tmp);
      jsize l = env->GetArrayLength(serial); // Tömb méretének lekérése
      int i;
      for (i = 0; i < l; i++) { // Eszköz sorozatszámának beállítása
            device->serial[i] = tmp[i];
      }
      emokit_init_crypto(device, 1); // Dekódoló inicializálása
      free(tmp); // tmp felszabadítása
}
```

Az előző függvény a korábban ismertetett Emotiv EEG-hez tartozó adatstruktúrákat inicializálja. Az utolsó Javából hívható függvény az USB portról érkező adatok átadására szolgál.

```
JNIEXPORT void JNICALL
Java_hu_android_bme_innolearn_smartsocketeeg_UsbDriver_setNewData(
            JNIEnv* env, jobject object, jbyteArray rawData) {
      switch (eegSensor->deviceType) {
      case Emotiv:
            setEmotivData(env, object, rawData);
            break;
      default:
            break;
      }
}
```

Az adatok itt továbbításra kerülnek az eszköz specifikus feldolgozáshoz, valamint a korábban említett JNIEnv és jobject típusú változók is, hogy az illesztőprogram visszajelzést küldhessen a Java oldalnak, ahogyan az EEGSensor addToBuffer() függvényében látható volt. A JNIEnv típusú objektum valójában a Java környezetet reprezentálja, a jobject típusú pedig a hívást kezdeményező objektumra ad egy mutatót.

```
void setEmotivData(JNIEnv* env, jobject object, jbyteArray rawData) {
      signed char* tmp = (signed char*) malloc(32); // Átmeneti buffer foglalása
      env->GetByteArrayRegion(rawData, 0, 32, tmp); // Java byte tömb másolása
      jsize l = env->GetArrayLength(rawData); // Tömb méret lekérdezése
      for (int i = 0; i < l; i++) { // Kódolt adatok átadása a struktúrának
            device->raw_frame[i] = tmp[i];
      }
      free(tmp); // Átmeneti buffer felszabadítása
      emokit_get_next_raw(device); // Dekódolás
      // Nyers adar feldolgozás
      device->current_frame = emokit_get_next_frame(device);
      // Az adatok átalákítása az általános ChannelSet formátumba
      ChannelSet* channels = new ChannelSet(eegSensor->getChannelCount());
      channels->addChannel(device->current_frame.F3, device->last_quality.F3);
      // ... További csatornák átadása
      channels->addChannel(device->current_frame.FC5, device->last_quality.FC5);
```

```
	//Csatlakozások minőségének jelzése a Java oldalnak
	if (device->current_frame.counter == 0)
	{
		jclass objectClass = env->GetObjectClass(object);
		jmethodID objectMethod = env->GetMethodID(objectClass,
				"emotivChannelsQuality", "(IIIIIIIIIIIII)V");
		env->CallVoidMethod(object, objectMethod, device->last_quality.F3,
				device->last_quality.FC6, device->last_quality.P7,
				device->last_quality.T8, device->last_quality.F7,
				device->last_quality.F8, device->last_quality.T7,
				device->last_quality.P8, device->last_quality.AF4,
				device->last_quality.F4, device->last_quality.AF3,
				device->last_quality.O2, device->last_quality.O1,
				device->last_quality.FC5);
	}
	// Az adatok mentése az eegSensor bufferébe és a Java változók továbbítása
	eegSensor->addToBuffer(*channels, env, object);
}
```

A setEmotivData() függvény, amit a Javából hívott setNewData() hív meg annyit tesz, hogy az Emotiv EEG-hez tartozó speciális adatstruktúrát átalakítja úgy, hogy az illeszkedjen az általánosan használható EEGSensor által használt struktúrához.

Az EEG periféria illesztése mobil eszközhöz nemcsak az én projektem szempontjából bír jelentőséggel. Segítségével számtalan újszerű alkalmazás előtt nyitottam meg a lehetőséget, hogy neurofeedback alkalmazásával tegye könnyebbé a megváltozott képességűek eszköz- illetve szoftverhasználatát, vagy megújítsa a tablettel történő önálló tanulást. Erre kíván megoldást adni a tanszéken indított InnoLearn projekt, amelynek fejlesztésében szintén aktívan tevékenykedem. Az illesztőprogram bemutatása során említett jelfeldolgozással kapcsolatos, szintén általam írt részek abban a projektben kerültek sikeres felhasználásra. Az InnoLearn projektben elért eredmények alapján további két TDK dolgozat is készült. Az egyik dolgozat szerzői Angeli Róbert és Fekete András, a másiké pedig Szita Ádám.

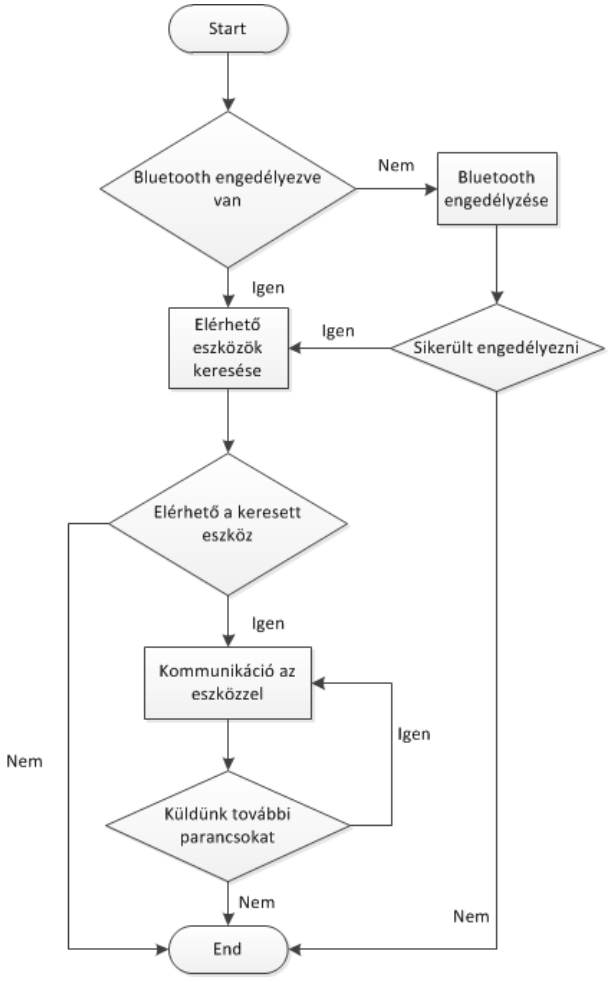
# SmartSocket vezérlése

A SmartSocket egy a BME Automatizálási és Alkalmazott Informatikai Tanszék hallgatói által készített vezérelhető elosztó. A SmartSocket Bluetooth-on keresztül képes a parancsok fogadására. Szerencsére az Android operációs rendszer rendelkezik Bluetooth API-val, amivel a kommunikáció könnyedén megvalósítható.

Először is a manifest állományban kell jelezni, hogy az alkalmazás használni akarja a Bluetooth-ot. Ezt a következő két sorral tehetjük meg.

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
```

A Bluetooth használatának folyamatát a 4. ábra szemlélteti.




*4. ábra Bluetooth használata*

A Bluetooth egy Intent segítségével engedélyezhető, amire a következő kódrészlet mutat példát.

```
Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
```

Az engedélyezést követően meghívódik annak az Activity-nek az onActivityResult függvénye, amelyikből az engedélyezési folyamat indult. A művelet sikerességét az imént említett függvény felülírásával ellenőrizhető. A következő kódrészlet erre sikeres engedélyezés esetén elkezdi a Bluetooth eszközök keresését.

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
	super.onActivityResult(requestCode, resultCode, data);
	if(requestCode == REQUEST_ENABLE_BT)
	{
		if(resultCode == RESULT_OK)
		{
			adapter.startDiscovery(); // Eszközök keresésének indítása
		}
	}
}
```

Amennyiben található a közelben Bluetooth eszköz akkor arra egy BroadcastReceiverrel reagálhatunk.

```
IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
registerReceiver(mReceiver, filter);
```

Az előbbi kódrészlet mutatja, hogy hogyan regisztrálható be egy BroadcastReceiver, ami az eszköz megtalálására reagál.

```
private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
	public void onReceive(Context context, Intent intent) {
	String action = intent.getAction();
	if (BluetoothDevice.ACTION_FOUND.equals(action)) {
		device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
		if(device!=null){
			if(device.getName().contains("SmartSockets"))
			{
			adapter.cancelDiscovery(); // Eszköz keresés leállítása
			boolean socketConnected = false;
			BluetoothSocket socket = null;
			while(!socketConnected)
				{
				try {
					socket =
device.createRfcommSocketToServiceRecord(UUID.fromString("00001101-0000-1000-8000-
00805F9B34FB")); // Csatlakozás a SPP szolgáltatáshoz
					socket.connect();
					socketConnected = socket.isConnected();
				} catch (IOException e) {
					socketConnected = false;
				}
				if(socket != null && socket.isConnected())
				{
				connectionThread = new SmartSocketManager(socket);
				connectionThread.start(); // Kommunikációs szál indítása
				}
```

```
                    }
                }
            }
};
```

Ha keresés közben az alkalmazás olyan eszközt talál, aminek a nevében szerepel a SmartSocket szó, akkor megtalálta a keresett eszközt és leállítom a keresését, majd nyitok egy BluetoothSocketet az eszköz felé SPP (Serial Port Profile) szolgáltatáson keresztül. Amennyiben a socket megnyitása sikeres volt elindítok egy új szálat. Ezen a szálon fog végbemenni a kommunikáció.

A SmartSocket számos hasznos funkcióval rendelkezik, de ezek részletes bemutatása nem releváns jelen dolgozat szempontjából, ezért csak az alkalmazás által használt funkciókat és a hozzájuk tartozó parancsokat mutatom be. Az egyik ilyen funkció a SmartSocket üzemmódjának lekérdezése és megváltoztatása. Kétféle üzemmódja van az elosztónak a konfigurációs és a felhasználói üzemmód. A másik használt funkció az egyes aljzatok ki- és bekapcsolása. Mivel utóbbi funkció elérésére csak felhasználói üzemmódban van lehetőség, ezért szükség van a SmartSocket állapotának lekérdezésére, szükség esetén megváltoztatására. Az elosztó a parancsokat 2 bájtban várja, az 1. táblázatnak megfelelően.

*1. táblázat Bluetooth paracsformátum*

| Bájt 0 | Bájt 1 |
|---|---|
| Parancskód | Argumentum |

Nem minden parancskódhoz tartozik argumentum, ebben az esetben a második bájt értéke bármi lehet, mivel az eszköz figyelmen kívül hagyja.

*2. táblázat Parancsok*

| Parancs | Parancskód | Argumentum |
|---|---|---|
| Üzemmód lekérdezés | 0x54 | - |
| Üzemmód váltás | 0x49 | - |
| Aljzat bekapcsolása | 0x4B | aljzat sorszáma (1-5) |
| Aljzat kikapcsolása | 0x4C | aljzat sorszáma (1-5) |

A 2. táblázat mutatja a használt parancsokat, a hozzájuk tartozó parancskódot, valamit a parancshoz tartozó argumentumot. Az elküldött parancsokra az eszköz minden esetben egy 10 bájtos választ ad, melynek felépítése a 3. táblázatban látható.

*3. táblázat Bluetooth válaszformátum*

| Bájt 0 | Bájt 1 | Bájt 2 | Bájt 3 | Bájt 4 | Bájt 5 | Bájt 6 | Bájt 7 | Bájt 8 | Bájt 9 |
|---|---|---|---|---|---|---|---|---|---|
| Válaszkód | Végpont | Argumentumok | | | | | | 0x0D | 0x0A |

Az üzemmód lekérdezésre és váltásra 0x36 válaszkódú üzenetet küld az elosztó, aminek a bájt 3-as argumentuma jelzi az eszköz aktuális üzemmódját. Amennyiben az értéke 0 konfigurációs módban van, ha pedig 1, akkor felhasználóiban. Az aljzat ki- és bekapcsolása parancsra 0x37 válaszkódú üzenettel reagál a SmartSocket. A válaszban a 2 és 6 közötti sorszámú bájtok adják meg az egyes aljzatok állapotait (a bájt 2 az első a bájt 6 pedig az ötödik aljzat). Amennyiben az adott bájt értéke 1 az aljzat áram alatt van, ha pedig 0, akkor nem.

A kommunikációt a tablet és az elosztó között a SmartSocketManager osztály valósítja meg, ami a Thread osztály leszármazottja.

```
public SmartSocketManager(BluetoothSocket socket) {
	mmSocket = socket;
	InputStream tmpIn = null;
	OutputStream tmpOut = null;
	// Be- és kimeneti streamek lekérdezése
	try {
		tmpIn = socket.getInputStream();
		tmpOut = socket.getOutputStream();
	} catch (IOException e) {
		// Hibakezelés
	}
	mmInStream = tmpIn;
	mmOutStream = tmpOut;
	try { // Üzemmód lekérdezés parancs küldése
		mmOutStream.write(0x54);
		mmOutStream.write(0x00);
		mmOutStream.flush();
	} catch (IOException e) {
		// Hibakezelés
	}
	socketState = new boolean[5]; // Aljzat állapotok tárolásának inicializálása
}
```

Az előbbi kódrészlet a SmartSocketManager osztály konstruktorából származik. A legfontosabb dolog, amit elvégez, hogy rögtön lekérdezi az eszköz állapotát, mivel csak a megfelelő üzemmódban küldhető neki az aljzatok ki- illetve bekapcsolását előidéző utasítás. A kommunikáció a sockethez tartozó InputStream és OutputStream segítségével történik. Az olvasás a szál leállításáig, illetve az első olvasási hibáig tart. A következő kódrészlet az eszköz felől érkező válaszok feldolgozást szemlélteti.

```
public void run() {
		byte[] buffer = new byte[1024];  // buffer lefoglalása
		int bytes;
		while (true) {
			try {
				// Read from the InputStream
				bytes = mmInStream.read(buffer);
				switch (buffer[0]) { // Válaszkód
				case 0x36: // Üzemmód válasz
					if(buffer[3] == 0) //Átváltás felhasználói módba
					{
						mmOutStream.write(0x49);
						mmOutStream.write(0x00);
						mmOutStream.flush();
					}
					break;
				case 0x37: // Aljzat ki- és bekapcsolás válasz
					socketState[0] = buffer[2] == 1;
					socketState[1] = buffer[3] == 1;
					socketState[2] = buffer[4] == 1;
					socketState[3] = buffer[5] == 1;
					socketState[4] = buffer[6] == 1;
					break;
				}
			} catch (IOException e) {
				// Hiba jelzés
				break;
			}
		}
	}
```

Az alkalmazás által küldött parancsokra kétféle válasz érkezhet. Ha a válasz üzemmód lekérdezés vagy üzemmód váltás parancsra érkezet, akkor ellenőrzi, hogy felhasználói módban van-e a SmartSocket és ha nem akkor utasítja az eszközt az üzemmód váltásra. Amennyiben az elosztó aljzat ki- vagy bekapcsolására reagált, akkor az aljzatok tárolt állapotát. Végezetül a parancsküldés kerül bemutatásra a következő kódrészletben.

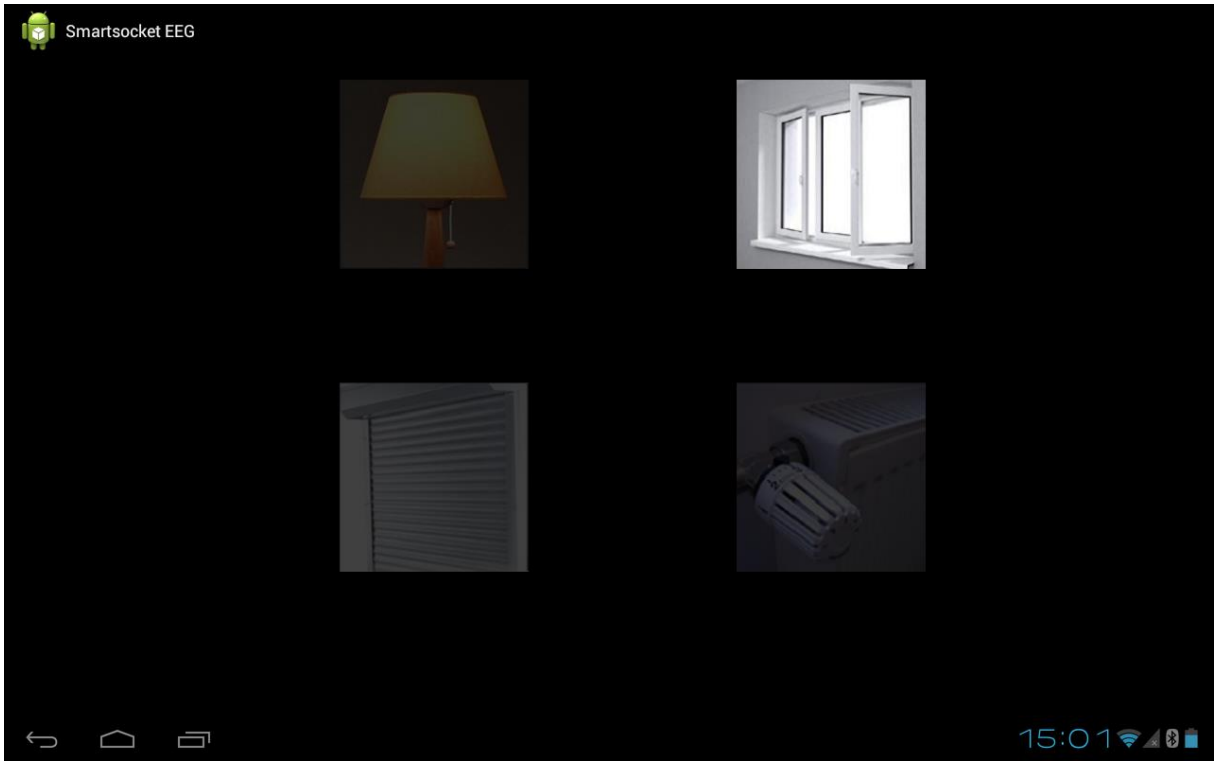
```
public void toggleSocket(int i){
		try {
			if(!socketState[i]) // Aljzat állapotának ellenőrzése
				mmOutStream.write(0x4b); // Bekapcsoló parancs
			else
				mmOutStream.write(0x4c); // Kikapcsoló parancs
			mmOutStream.write(i+1); // Az aljzatok 1-től vannak számozva
			mmOutStream.flush();
		} catch (IOException e) {
			// Hibakezelés
		}
	}
```

Az előbbi függvény a megadott sorszámú aljzatot ki- vagy bekapcsolja attól függően, hogy az éppen milyen állapotban van.

Bár egy konkrét Bluetooth-on keresztül vezérelhető eszköz csatlakoztatását és irányítását mutattam be, a folyamat során felhasznált elvek és eljárások segítségével könnyedén elkészíthető bármilyen hasonló eszköz illesztése amennyiben ismert a kommunikációs protokoll.

# Az alkalmazás

Ahogy azt már korábban említettem az alkalmazás négy képet jelenít meg mátrixos elrendezésben (5. ábra) és ezeket véletlenszerű sorrendben villogtatja, aminek hatására a felhasználó agyában P300 jel keletkezik a figyelt kép felvillanásakor. Ez az eljárás hasonló a [3] cikkben leírt rendszer működéséhez, ahol egy 6 sorból és oszlopból álló mátrixban az ABC betűi kerültek megjelenítésre, és a sorok illetve oszlopok villogtatásával váltották ki a P300 jelet.




*5. ábra Az alkalmazás futás közben*

Miközben az alkalmazás villogtatja a képeket a csatlakoztatott EEG segítségével rögzíti a felhasználó agyi tevékenységét. A villogtatások sorozatokba vannak rendezve, egy sorozaton belül minden egyes kép egyszer villan fel. A felvillanások sorrendje véletlenszerű. Egy kép 250 milliszekundumig látható majd ugyanennyi szünet után egy másik villan fel. Ha minden kép felvillant egyszer, akkor a sorozat befejeződik és az EEG által mért adatok feldolgozásra kerülnek. A figyelt kép villanásait a felhasználónak számolnia kell, erre azért van szükség, hogy a többi kép villanása ne terelje el a figyelmét, és ez ne váltson ki felesleges P300 jelt, ami tönkre tenné a mérést. Az első 90 sorozatnál a felhasználónak egy előre kijelölt képet kell néznie, és számolnia a villanásait, ez a kalibrációs szakasz. A kalibrációs szakaszban az alkalmazás megtanulja, hogy az adott felhasználónál, hogyan néz ki a P300 jel, mivel ez

egyénenként változó. A mérési adatok feldolgozása 2 másodperces blokkokban történik, ami megegyezik egy sorozat hosszával. A blokkban mért értékeknek kiszámítja az átlagát, majd az összes mérési eredményből levonja a kapott értéket. Erre azért van szükség, mert az EEG a fejen könnyedén elmozdulhat, ami miatt a nyugalmi állapotban mért eredmény megváltozhat, és ez tönkre teheti a P300 jel detektálását. Ha a levonás után marad olyan érték, ami az átlagtól 100-nál jobban eltér, akkor az adott sorozat mérési eredményét hibásnak tekinti, és figyelmen kívül hagyja. A kalibrációs szakaszban két mintát tanul meg a rendszer. Az egyik a figyelt kép felvillanása során keletkezett minta, ez a cél minta. A másik pedig az a minta, amelyik a többi kép felvillanásakor keletkezik, ez a nem-cél minta. A minták úgy alakulnak ki, hogy a kalibrációs szakasz 90 sorozatának mintáit átlagolja a rendszer. A kalibrációs szakasz befejeztével a rendszer átvált detektálási szakaszba. Egy detektálási szakasz 40 sorozatból áll és minden egyes kép felvillanásához rögzít egy mintát a kalibrációs szakaszhoz hasonló módon. A szakasz végén a rendszer azt a képet detektálja a felhasználó által figyelt képnek, amelyiknek a mintája a legközelebb van a cél mintához, valamint közelebb van a cél mintához, mint a nem-cél mintához. A közelséget a mintákból alkotott vektorok Euklideszi távolságának alapján méri a rendszer. Matematikailag a következő képletek írják le a rendszer működését.

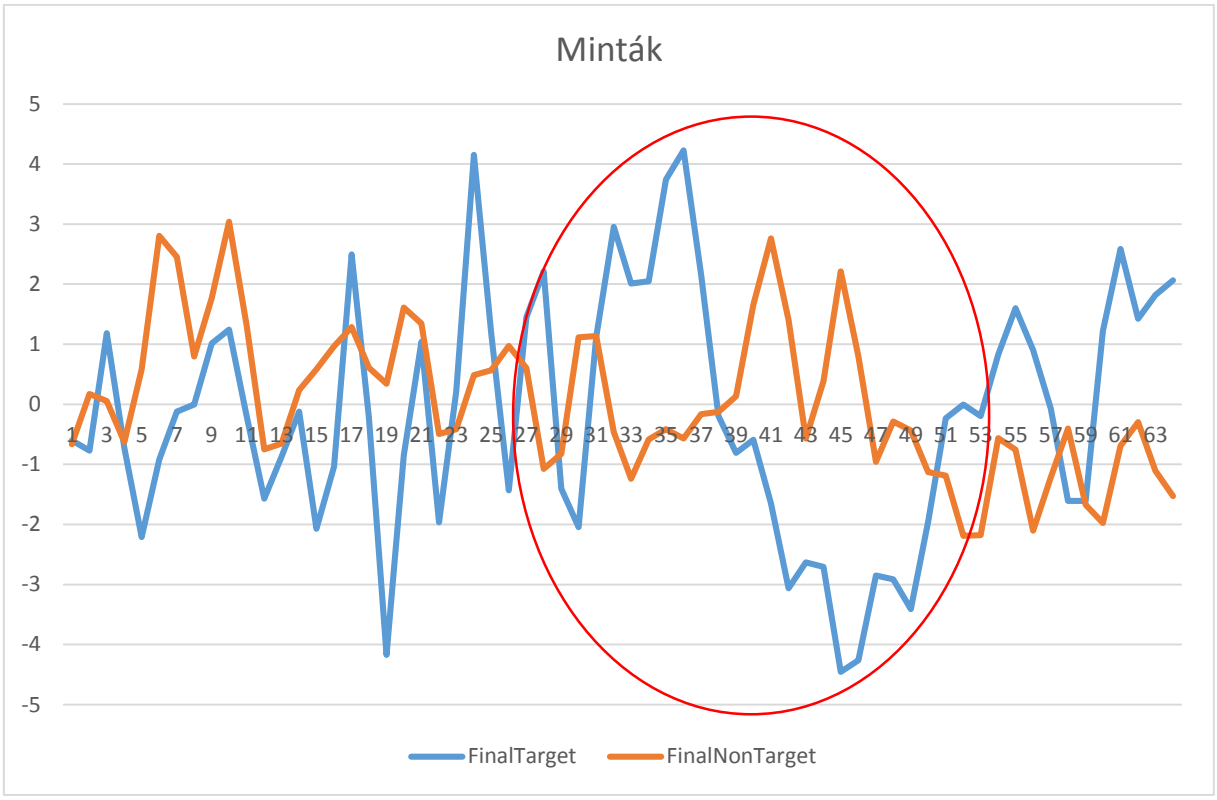
$$|c(t) - m_i(t)| - |nc(t) - m_i(t)| < 0$$

$$\min(|c(t) - m_i(t)|)$$

A képletekben c(t) jelöli a cél minta vektorát, az nc(t) a nem-cél minta vektorát. $m_i(t)$ pedig az i-edik képhez tartozó mintát. A minták 64 elemet tartalmaznak, ami 500 milliszekundumos időtartamnak felelnek meg és a kép felvillanásakor kezdődik el a rögzítése.

A detektált kép alapján pedig parancsot küld a SmartSocketnek.

A képek villanásának hosszát, a kalibrációs és detektálási szakaszok hosszát rengeteg mérés alapján határoztam meg. Arra törekedtem, hogy minél gyorsabb legyen az alkalmazás, ezért gyors villogtatással kezdtem, de ilyenkor a minták túlzottan átlapolódtak, és a detektálás nem adott egyértelmű eredményt. A méréseket saját magamon, illetve családtagjaimon végeztem el. A 6. ábra diagramja egy kalibrációs szakasz végén kapott cél és nem-cél mintákat ábrázolja. A kék görbe a cél a narancssárga pedig a nem-cél mintát ábrázolja. A piros ellipszissel jelölt részben jól látható az inger által kiváltott P300 jel.




*6. ábra Cél és nem-cél minták*

# Összefoglalás és továbbfejlesztési lehetőségek

Az EEG mobileszközhöz történő csatlakoztatására és ilyen módon történő felhasználására tudomásom szerint még nem volt példa. Már az EEG használata is segítséget jelent a mozgáskorlátozottak számára. Azzal, hogy ez egy mobileszközhöz van csatlakoztatva nem csak a környezetükben található eszközök vezérlése válik lehetővé, hanem az is, hogy mindezt az aktuális helyzetüktől függetlenül tegyék, hiszen egy tablet könnyedén felszerelhető például egy kerekesszékre. A rendszer ötletét és a megvalósításának lehetőségét több mozgáskorlátozottakkal foglalkozó intézmény (pl.: Mozgásjavító Általános Iskola, Szakközépiskola, Egységes Gyógypedagógiai Módszertani Intézmény és Kollégium, Mozgássérült Emberek Rehabilitációs Központja) nagy örömmel fogadta és részt vesz a leginkább segítséget nyújtó alkalmazás kialakításában.

A rendszer elkészítése során rengeteg új technológiával sikerült megismerkednem, amelyek használata a legtöbb esetben nem volt triviális, de a felmerülő akadályokat sikerült leküzdenem. Különösen érdekes volt az EEG eszköz illesztésének elkészítése, mivel ilyen jellegű feladattal korábban még nem találkoztam. A detektálási algoritmus paramétereinek megfelelő beállításához rengeteg mérést kellett végeznem, de végül sikerült megtalálnom az ideális értékeket. Véleményem szerint egy olyan innovatív és úttörő rendszer prototípusát sikerült elkészítenem, ami könnyedén továbbfejleszthető és bővíthető. Úgy érzem, hogy a projekt megvalósítása nagyban hozzájárult a szakmai fejlődésemhez és rengeteg hasznos tapasztalatot sikerült szereznem.

Az alkalmazás működőképes, de közel sem nevezhető gyorsnak. A kalibrációs szakasz sokáig tart és a detektálás is hosszadalmas, ezért azt tervezem, hogy ezeket a részeket különböző jelfeldolgozási módszerekkel gyorsabbá teszem.

Mivel jelenleg csak négy kép található a mátrixban, ezért csak négyféle utasítás adható ki. Ahhoz, hogy a kiadható utasítások számát megnöveljem, további képeket kell felvennem a mátrixba, de a kis képernyő méret miatt ez rendkívül körülményes, mivel a képek mérete és távolsága is befolyásolhatja a P300 jel kialakulását. Erre megoldás lehet egy olyan menü kialakítása, ami ilyen mátrixokból épül fel.

Az EEG felhasználása mellett hamarosan befejezem hangfelismerés integrálását is a rendszerbe, ami sokkal gyorsabb lehet, mint az EEG használata, de nem nyújt megoldást azok

számára, akik betegségükből adódóan nem csak mozgáskorlátozottak, hanem beszélni sem tudnak.

A projekt során használt SmartSocket az Automatizálási és Alkalmazott Informatikai Tanszék Intelligens otthon projektjének részeként került kifejlesztésre, amely során további vezérelhető eszközöket is készítettek. Ezek csatlakoztatását is tervezem a rendszerhez.

# Irodalomjegyzék


[1] Chapman, R.M. & Bragdon, H.R. (1964). Evoked responses to numerical and non-numerical visual stimuli while problem solving. Nature, 203, 1155-1157..

[2] Sutton, S., Braren, M., Zubin, J., & John, E.R. (1965). Evoked-Potential Correlates of Stimulus Uncertainty. Science, 150, 1187-1188..

[3] L. A. Farwell and E. Donchin, "Talking off the top of your head: A mental prosthesis utilizing event-related brain potentials," Electroencephalogr. Clin. Neurophysiol., vol. 70, pp. 510–523, 1988..

[4] Polich, J. (2007). Updating P300: An integrative theory of P3a and P3b. Clinical Neurophysiology, 118(10), 2128-2148..

[5] „http://www.nielsen.com/us/en/newswire/2013/whos-winning-the-u-s-smartphone-market-.html," [Online].