



MATEMATIKAI ÉS INFORMATIKAI INTÉZET

# Interfész megoldások imperatív és OOP nyelvek közötti kapcsolattartásra

## Készítette

Nagy-Tóth Bence

Szak: Programtervező informatikus

Specializáció: Szoftverfejlesztő informatikus

## Témavezető

Dr. Király Roland  
egyetemi docens

EGER, 2023

# Tartalomjegyzék

<b>Bevezetés</b>	<b>1</b>
0.1. Kommunikáció adatszerkezeteken keresztül . . . . .	3
0.2. Szerializáció . . . . .	3
0.3. Marshalling és szerializáció . . . . .	4
0.4. Adatok konvertálása . . . . .	4
0.4.1. Egész és valós számok, logikai változók . . . . .	4
0.4.2. Szövegek . . . . .	4
0.4.3. Rekordok . . . . .	5
0.5. Managed és unmanaged kód közötti különbség . . . . .	7
0.6. Az ellenőrzött kódok előnyei és hátrányai . . . . .	8
0.7. A Platform Invoke működése . . . . .	8
0.8. Memóriakezelés az InterOp marshaller segítségével . . . . .	10
<b>1. Elosztott rendszerek</b>	<b>11</b>
1.1. Gondolataim a streamalapú kommunikációról . . . . .	11
1.2. RESTful alkalmazások . . . . .	13
1.3. Google RPC és a Proto-nyelv . . . . .	13
1.3.1. Jobb a Proto a JSON-formátumnál? . . . . .	16
1.3.2. Szakdolgozati munkánkról . . . . .	17
1.3.3. Csatlakozók, átviteli eszközök . . . . .	17
1.4. Miről is szól munkánk? . . . . .	18
1.5. Mit jelent az intelligens szoba? . . . . .	19
1.6. Delphi és C# programozási nyelvek összehasonlítása . . . . .	20
1.7. Miket tud a C#, amit a Delphi nem? . . . . .	22
1.8. Miket tud a Delphi, amit a C# nem? . . . . .	23
1.9. Alaphelyzet . . . . .	23
1.10. Az SLDLL-függvények bemutatása . . . . .	23
1.10.1. Open – DLL megnyitása . . . . .	24
1.10.2. Listelem – A tömb beállítása . . . . .	25
1.10.3. Felmeres – Eszközök felmérése . . . . .	25
1.10.4. SetLista - Az eszközbeállítások végrehajtása . . . . .	26

1.10.5. Hangkuldes - A hangszórók vezérlése . . . . .	26
1.10.6. Hibakódok . . . . .	27
1.11. Az általam készített RelayDLL függvényeinek bemutatása . . . . .	27
1.11.1. Open . . . . .	28
1.11.2. Listelem és Felmeres . . . . .	28
1.11.3. ConvertDEV485ToJson . . . . .	28
1.11.4. ConvertDEV485ToJson_C . . . . .	29
1.11.5. ConvertDEV485ToXML . . . . .	30
1.11.6. SetTurnForEachDeviceJSON . . . . .	31
1.11.7. fill_devices_list_with_devices . . . . .	31
1.12. Az általam készített SLFormHelperDLL bemutatása . . . . .	32
1.12.1. A WinForm bemutatása . . . . .	32
1.12.2. DLL-ek üzenetküldése Win32-ben . . . . .	33
1.13. Problémák és megoldások . . . . .	34
1.13.1. Egyénileg definiált típusok és struktúrák . . . . .	34
1.13.2. JSON-formátumra konvertáló függvények hívása .NET keretrendszerből . . . . .	35
1.13.3. Az SLDLL_Open-függvény paramétereinek előállítása . . . . .	35
1.13.4. Stringek átadása két nyelv között . . . . .	36
1.13.5. A Delphiben eldobott kivételek nem kezelhetőek . . . . .	36
1.13.6. uzfeld-metódus megfelelője C#-ban . . . . .	37
1.13.7. Eszközök azonosítóinak, típusainak átadása . . . . .	37
1.13.8. A hangszóró egy egész hanglistát kezel . . . . .	38
1.13.9. Több csatlakoztatott eszköz felismerése, vezérlése . . . . .	40
<b>Összegzés</b>	<b>43</b>

## Kivonat

Szakdolgozatomban kifejtem az *interlingvális*<sup>1</sup> kommunikáció lehetséges megvalósítási módjait, amelynek legfontosabb mérföldköve a szöveges adatok valamilyen szabvány szerinti közlése, tehát a szerializáció lesz, mivel a folyamat segítségével képessé válhatunk bármilyen típusú adatot – legyen az szám, logikai változó, esetleg egy teljes objektum – ilyen formában a másik program részére közölni. Innentől csak annyi a dolgunk, hogy a másik oldalt is felkészítsük ezen szabványos formátumú közlések fogadására és feldolgozására. Ennek gyakorlati haszna bemutatásra kerül szakdolgozatunkhoz készített szoftver keretein belül, amelynek célja, hogy különböző típusú eszközöket vezéreljünk a C# nyelvi adottságaival SLFormHelper, valamint Delphi-Assembly-kódból készült SLDLL állományok munkájának összehangolásával. Ez a két DLL<sup>2</sup>, mint később látni fogjuk, nem lesz elegendő, szükség volt egy harmadik, úgynévezett RelayDLL-re, amely áthidalja a két nyelv sajátosságait, és ténylegesen olyan elemeket használ, amelyet minden két nyelv hasznosítani tud a maga számára. Ezt én készítettem abból a célból, hogy a kommunikáció tulajdonképpen ezen a „hídon” keresztül, szabályozott módon teremtődjön a Delphi és a C# programozási nyelvek között.

---

<sup>1</sup> Az interlingua kifejezés a latin *inter* (között) és *lingua* (nyelv) szavak összetételéből adódik. Jelentése „nyelvek között”.

<sup>2</sup> A DLL-ek – ez a *Dynamically Linked Library* kifejezés rövidítése – csak Windows operációs rendszerben hasznosítható fájlok, amelyek segédkönyvtárként használhatóak, lefordított, gépi kódú állapotban vannak, ezeket a valódi futtatható programok használhatják. Mivel ezek már lefordított ál

# Bevezetés

Kognitív képességeink fejlesztésére különösen oda kell figyelnünk életünk során. Ezen megállapításomat alátámasztja egy COVID-világjárvány utáni időszak, amely időszakban sorra jelentek meg olyan jelentések [22], amelyek azt támasztják alá, hogy a lezárások ideje alatt nőtt a különböző mentális betegségek kialakulásának kockázata. A *demencia* gyűjtőfogalomként alkalmas ezen betegségek együttes megnevezésére.<sup>3</sup>

A különböző digitális eszközök használata számtalan alkalommal hosszas órákon keresztül képes lekötni a figyelmünket, ezért bizonyos külső ingerekre egyre lassabban, kisebb amplitúdóval, azaz nagyobb közömbösséggel tudunk reagálni.

A demencia, azaz a kognitív képességek leépülésének tünetei közé tartoznak a különböző beszédezavarok, az ítélezőképesség, a memorizálás és az elvonatkoztatás képességeinek romlása.



*Forrás: Kutatási jelentés a demencia és a demenciagondozás aktuális helyzetéről  
Kecskeméten, 2020 - Modus Alapítvány*

1. ábra. Valamely demenciafajtával diagnosztizáltak számának emelkedése

<sup>3</sup> A demencia említések nem egyetlen betegségre, hanem több hasonló jellegű problémát csoporthoz kötődő fogalomra, egy tünetegyüttetésre gondolunk. Száznál is több típusát ismerik a demenciának, amelyek közül az Alzheimer-kór a leggyakoribb. [5]

Az értelmi hanyatlást különféle egészségkárosító szokások – például az alkoholfüggysztás – szervi károsodás, akár ezek együttese is okozhatja, a legjelentősebb kockázati tényező ilyen téren azonban az ember életkora.

Szakemberek véleménye szerint a megfelelő étrendek megalkotásával, a társas interakciók intenzitásának és tartalmi színvonalának növelésével, az emberek képzésével mind iskolai, mind munkahelyi szinten megelőzhetjük agyi teljesítőképességünk romlássát. Kifejezetten preventív jelleggel hatnak a különféle társasjátékok, valamint a rejtvényfejtés.

Egy szó, mint száz: bármi, amivel munkára bírjuk agyunkat, alkalmas arra, hogy gátat vessen annak romlásának. Végül és abszolút nem utolsó sorban említhetjük a mozgás fontosságát, munkánkban a demencia megfélezésének ezen útját választottuk.

Szakdolgozatom ezen pontján azért érdemes erről a betegségről szót ejtenem, mivel alapvetően számomra már önmagában motivációjul szolgál a tudat, hogy egy ilyen nagy volumenű elmélet megvalósításában vehetek részt, amely a demencia megelőzését, a mentális állapot folyamatos rongálódásának megfélezését célozza. [17] [2] [26]

Az általunk készített szoftver támogatást nyújthat csoportfoglalkozásokon, tulajdonképpen egy tornaórát le tudunk vezényelni az eszközöknek köszönhetően. Természetesen az eszközök által kibocsájtott különböző fény- és hangjelzések önmagukban nem hordoznak semmiféle jelentést, a jelzések konkrét a tornaórákat vezénylő szakemberek dolga közölni a csoport számára.

Bevezetésem ezen részén szeretném egy kicsit górcső alá venni témám címét. Ehhez először is meg kell ismerkednünk a címben szereplő két programozási paradigmával. Az imperatív<sup>4</sup> programozási paradiigma arról szól, hogy a számítógépet egyértelműen, előre meghatározott utasításkészlettel vezéreljük, le kell írnunk a problémát megoldó lépéssorozatot, algoritmust, így az elkészült forráskódot ilyen lépéssorozatok együttese teszi ki. Az algoritmusok implementálásához változókat és vezérlési szerkezeteket (elágazás, ciklus és szekvencia) használunk. Az efféle megközelítést támogató nyelveket nevezzük imperatívnak. Ilyenek például a Java, a C, C++, C# és a Delphi.

Az objektum-orientált programozási paradiigma igyekszik a világot lemodellezni olyan formában, hogy a modellt objektumok használatával írja le. Az OOP egyik alaptézise, hogy az adatszerkezet és a rajtuk végzett műveletek egy egységen, az osztályban összpontosulnak. Az osztály a belőle készíthető objektumok „tervrajzának” nevezhető. Ennek megfelelően egy osztályba tartoznak a változók és a rajtuk végrehajtható metódusok. Változóinkat és metódusainkat védettségi szintekkel láthatjuk el, hogy a hozzáférést bizonyos erőforrásokhoz korlátozzuk, ellenőrzött módon végezzük. Az osztályok között kapcsolatokat teremthetünk: öröklődést vagy birtoklást. Azon nyelvet, amelyek ezt a paradiigmát támogatják, objektum-orientáltnak minősítjük. A fentebb említett nyelvek ezt a paradigmát is támogatják a C nyelv kivételével.

---

<sup>4</sup> *imperare* latin szóból eredetű szó, jelentése: parancsol

## 0.1. Kommunikáció adatszerkezeteken keresztül

A *Marshalling* egy olyan folyamat, amely összetett típusok átalakítására szolgál, hogy egy nyelv adatszerkezetét a másikkal megértesük. Magyar fordításával nem találkoztam ennek a kifejezésnek, de leginkább talán az 'átalakítás' szóval tudjuk leírni, ami adatszerkezetek esetében annyit tesz, hogy más nyelv által is értelmezhetővé tesszük, kevésbé hagyatkozunk az adott nyelv különlegességeire.[36]

A kód további portolhatósága, újrahasznosíthatósága végett megéri szabványos formátumokon keresztül kommunikálni, mint azt tesszük API-k<sup>5</sup> vagy konfigurációs fájlok esetében, ilyenek például az általam használt JSON- vagy XML-formátumok, hogy más felületekre való költözötés esetén kompatibilitási probléma többé már ne merülhessen fel.

A Marshalling primitív típusok esetén abszolút működőképes, összetett adatszerkezetekre, viszont a programok átültethetősége végett érdemes szabványos formátumokkal kommunikálni. Ha ezt elfogadjuk, akkor fontos, hogy a serializáció és deserializáció folyamataiba is betekintést nyerjünk.

## 0.2. Szerializáció

A sserializáció az a folyamat, amikor valamilyen adatszerkezetet bájtok folyamára<sup>6</sup> alakítunk, hogy. Mivel az adatok valamilyen elgondolás szerint bitekként vannak kódolva a memóriában, ezért szükségünk van a sserializáció inverz műveletére<sup>7</sup>: ez a deserializáció, amely folyamán az objektumot leíró bitek a programban értelmezhető, kezelhető adatokká alakulnak. Az általunk készített szoftverben JSON, valamint XML formájában mutatunk példát az adatok (szöveges) sserializálására. Ennek megfelelően egy string is lehet érvényes formátum, de nem feltétlenül muszáj ilyen költségesen leírni objektumokat. A folyamatnak létezik már egy sokkal hatékonyabb módja is, mint ezt látni fogjuk a GRPC esetében a 1.3.1 fejezetben.

Abszolút jogosnak tartom felvetni azt a kérdést, hogy a programok nem eleve bitek sorozatát kezelik? Ha igen, akkor mi értelme ezt az adatfolyamot még egyszer szintén bitek sorozatává alakítani egy teljesen más eljárás mentén?

Az objektumok tárolásának módját az adott nyelvhez írt, aktuális verziójú fordítóprogram, az operációs rendszer, valamint a processzor kezeli. Ha ezek közül bármelyik

---

<sup>5</sup> Az Application Programming Interface utasítások, szabványok és metódusok halmaza, amely leírja a kommunikációt más, külső szoftverek részére. Az API-k lehetőséget biztosítanak két vagy több szoftver közötti adatok cseréjére, valamint adatokon végzett műveletek végrehajtására úgy, hogy annak mikéntjéről az API írója gondoskodik. Másként fogalmazva: az API a szerver azon része, amely felelős a kérések feldolgozásáért és kiszolgálásáért. Forrás: [25]

<sup>6</sup> A bájtfolyam angolul byte stream néven ismert.

<sup>7</sup> Az eredeti művelet hatását visszafordító, visszavonó.

változik, az adattárolás formátuma ezzel együtt dinamikusan átalakul. Egy objektum szerializált változatban statikus, állandó, kód által meghatározott formátumú, amit több programnyelv is képes értelmezni és feldolgozni. [4]

## 0.3. Marshalling és szerializáció

A Marshalling egy olyan folyamat, amely hidat teremt a felügyelt és felügyelet nélküli kódok között, lásd: 0.5. A Marshalling felelős az üzenetek átviteléért managed környezetből unmanaged környezetbe, és ugyanezért visszafelé is. Ez a CLR egyik alapvető szolgáltatása. Mivel az unmanaged típusok közül soknak nincs is párja a managed környezetben, így létre kell hozni konvertáló függvényeket, amelyek oda és vissza átalakítást végeznek az üzeneteken. Ennek megvalósított folyamatát nevezzünk Marshallingnak. [37]

## 0.4. Adatok konvertálása

### 0.4.1. Egész és valós számok, logikai változók

A primitívek közül numerikus értékek esetében egyszerűbb a Marshalling, mivel ezek minden nyelvben gyakorlatilag ugyanúgy vannak értelmezve. Egyszerű adattípusok azok, amelyek nem tartalmaznak más adattípusokat. Ezek az alapjai minden más típusnak. Példák a menedzselt primitív szám típusokra: `System.Byte`, `System.Int32`, `System.UInt32`, és `System.Double`. [24]

### 0.4.2. Szövegek

A szöveges unmanaged típusok Marshalling-folyamata bonyolultabb, mint a numerikusoké, mert ezek külön bánásmódot igényelnek.

Az ANSI és a Unicode két olyan karakterkódolási szabvány, amelyek a legelterjedtebbek. Az ANSI-hoz képest a Unicode számít újabb kódolásnak, amelyet a modern rendszerek használnak. Amikor a számítógépek világszerte elterjedtek, nyilvánvalóvá vált, hogy az ANSI képtelen minden nemzet különböző karaktereit (példának okáért a magyar nyelvben található ékezes betűket) kódolni, mivel a lehetséges variációk száma elfogyott, ezért az Unicode-szabványok (ide tartoznak az UTF-8, UTF-16, illetve UTF-32 néven ismert kódolások) váltották fel az ANSI-szabványt.

Az ANSI legnagyobb hátránya, hogy a használt nyelvtől/régiótól függően több úgynevezett kódlapot foglal magában. Amennyiben az összes számítógép ugyanazt használja, abban az esetben nincs probléma, de ha a kódlapok eltérnek, akkor a leírt szövegek értelmezhetetlené válnak, és bizonyos esetekben programhibához is vezethetnek.

Használatuk ebből következően jelenleg már nem javasolt. [12]

A stringek alapértelmezetten BSTR-típusként (Unicode-szabványú karakterként) kerülnek fogadásra a .NET-keretrendszerben. Én a biztonság kedvéért a DLL-ben ettől függetlenül is jelzem a Marshaller számára [**MarshalAs(UnmanagedType.BStr)**]-attribútum használatával.

A stringek marshalling-folyamata feliügyelet nélküli (unmanaged) kód esetében így működik: egy úgynevezett **MarshalAsAttribute**-attribútumot fűzhetünk a string paraméterek, valamint visszatérési értékek elő, hogy felkészítsük a marshallert, hogy pontosan milyen típusú stringet kell várnia. Több **UnmanagedType**-érték is megadható a karakterláncok fajtájától függően.[15]

Ezek közül hármat mindenépp érdemes megemlíteni:

- **UnmanagedType.BStr** (alapértelmezett) Előre meghatározott hosszúságú, Unicode-karakterkódolású szöveg található a túloldalon.
- **UnmanagedType.LPStr** Egy pointer ANSI-karakterekből álló tömbre, amelyet a nullkarakter (\0) zár.
- **UnmanagedType.LPWStr** Egy pointer Unicode-karakterekből álló tömbre, amelyet szintén a nullkarakter zár.

Ennek gyakorlati alkalmazása C#-ban a következő, általam készített példaprogram kód részletén megfigyelhető:

```
//ha paraméterben van referenciaként átadva WideChar (Delphiben)
[DllImport("MyDLLplus.dll", CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Unicode)]
public static extern byte DBConnect1(string inputString, [MarshalAs(UnmanagedType.BStr)] out string outputStr);

//ha a visszatérési érték PWideChar (Delphiben)
[DllImport("MyDLLplus.dll", CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Unicode)]
[return: MarshalAs(UnmanagedType.LPWStr)]
public static extern string DBConnect2(string inputString);
```

2. ábra. Példa két metódusra, amelyben stringeket használunk.

#### 0.4.3. Rekordok

A rekordok már az összetett adattípusok kategóriájába tartoznak, amelyek primitív és/vagy összetett adattípusokból épülnek fel. Ilyen például egy osztály (*class*) vagy egy struktúra (*struct*) – C++ nyelvben megtalálható még esetleg a union is –, amelyek magukban foglalnak egyszerű vagy egyéb összetett típusokat. Az alábbi C nyelvű kód részleten keresztül szemléltetem, hogy hogyan is néznek ki a gyakorlatban az összetett típusok. Az általam létrehozott **ListaElem** nevezetű struktúra két mezőt tartalmaz:

egy primitívet<sup>8</sup>, valamint egy összetett típust, ami itt történetesen szintén `ListaElem` típusú, de bármilyen más rekordtípust tartalmazhatna.

```
struct ListaElem
{
    uint8_t ertek;
    struct ListaElem* kovetkezo;
};
```

3. ábra. Példa egy rekord definíójára

Lévén összetett típusról van szó, gondolhatjuk, hogy a struktúrák marshallingolása ebből fakadóan nehezebb folyamat, és ez minden bizonnal így is van. További fejtörést okozhat például az alábbi ábrán látható Delphiben deklarált rekord típus felépítése C# nyelvben. A probléma ebben az esetben az, hogy még Delphiben értelmezve van a unionok<sup>9</sup> használata, C# tekintetében már ez nem opcio.

```
LISELE = packed record
    azonos: Word;
    case Integer of
        1:
        (
            vilrgb: HABASZ;
            nilmeg: Byte;
        );
        2:
        (
            handrb: Byte;
            hantbp: PHANGL;
        );
    end;
```

4. ábra. Példa egy union típus definíójára. Forrás: SLDLL

A rekordok szempontjából inkább a szerializáció a nyerő stratégia, tehát ezeket az egységeket átalakítjuk egy bájtalapú adatfolyamra (streamre), amely tárolható, könnyen átküldhető valamilyen csatornán<sup>10</sup> keresztül, és a másik nyelv számára is át-

<sup>8</sup> Az `uint8_t` C-ben a `char`, azaz a `byte` típussal ekvivalens, 1 bájton tárolt, előjel nélküli egész értéknek felel meg.

<sup>9</sup> A union egy olyan adatszerkezet, amelyben bizonyos mezőkből vagy az egyik vagy a másik kerül érvényesítésre attól függően, hogy az adott rekord milyen tulajdonságokkal rendelkezik. A példában ha az eszköz – azonosítójából fakadóan – lámpa (vagy nyíl), akkor a `vilrgb` és `nilmeg` mezők kerülnek érvényesítésre, amennyiben viszont hangszóró, akkor a másik kettő, a `handrb` és `hantbp` mezők értékére vagyunk kíváncsiak, a két-két mező ugyanazon a memóriaterületen tárolódik, tehát egyrészt memóriát tudunk megspórolni azzal, hogy egyszerre csak az egyiket használjuk, másrészt az osztályok öröklődését helyettesíthetjük unionok segítségével. Forrás: [29]

<sup>10</sup> Csatorna alatt érhetünk akár egy XML vagy JSON formátumú fájlt, akár egy stringet, akár valami más koncepción alapuló bájtfolyamot is, mint például a Protocol Buffers esetében.

alakítható a maga nyelvi sajátosságainak megfelelően (más szóval: kódolható és dekódolható). Látni fogjuk, hogy a projektben én ezt úgy oldottam meg,

## 0.5. Managed és unmanaged kód közötti különbség

Először is fontos tisztázni, hogy a C# szemszögéből nézve a *Unmanaged Code* (nem felügyelt kód) elnevezést a .NET-keretrendszer mindenféle külső komponens megbélyegzésére használja, amelyet nem a .NET futtatókörnyezete felügyel és vezérel. Ettől még ezek lehetnek jó kódok, csak szimplán a Common Language Runtime-en nincs közvetlen hatása ezek működésére: a memóriacímeket közvetlenül lehet kezelní, felülírni, így bizonyos esetekben olyan memóriaterületre is hivatkozhat egy változó, amelyet az operációs rendszer nem a program részére foglalt le, ezt `segfaultnak` nevezzük<sup>11</sup> [18] A Common Language Runtime használatával fejlesztett kódot kezelt vagy felügyelt – managed – kódnak nevezzük. Más szóval, ez az a kód, amely a .NET futtatókörnyezete által végrehajtásra kerül. A menedzselt kód futási környezete számos szolgáltatást nyújt a programozó számára: ilyen a kivételkezelés, típusok ellenőrzése, a memória lefoglalása és felszabadítása, így a Garbage Collection<sup>12</sup>, és így tovább. A fent említett szolgáltatások a fejlesztőket szolgálják, hogy biztonságosabb és optimálisabb kódot kézíthessenek. [37]

A felügyelet nélküli kód ennek pontosan az ellentéte: pointereket kezelhetünk a kódban, ezeket tetszésünk szerint lefoglalhatjuk, felszabadíthatjuk, megváltoztathatjuk a pointer értékét (azaz más területre mutathatunk), viszont ilyen típusú kódok növelhetik alkalmazásaink teljesítményét, mivel az extra ellenőrzések ki vannak kapcsolva a kód lefutásának idejére (például egy tömb túlindexelését vizsgáló algoritmus). A nem biztonságos blokkokat tartalmazó kódot az `AllowUnsafeBlocks` fordítói opcióval kell lefordítani.

Amikor eszközökkel vezérlő kódrészleteket szeretnénk C#-on keresztül hívni, legyen ez C++ vagy éppen esetünkben Delphi nyelven készítve, egyszerűen megkerülhetetlen a felügyelet nélküli kódok futtatása, mivel a C#-nak ezekre ráhatásai nincsenek, nyelvi korlátokba ütközünk, ha lámpákat és nyilakat kéne vezérelnünk C#-forráskóddal. A .NET úgy áll ezekhez a kódokhoz, hogy „megengedett, de nem ajánlott” kategóriába sorolja őket, nekünk annyit kell tennünk az ügy érdekében, hogy minél kevesebb szer hagyatkozzunk az unmanaged kódrészletekre. [38]

---

<sup>11</sup> Másnéven *access violation* hibának is nevezik, a Delphi inkább ezzel a kifejezéssel írja le a hibát.

<sup>12</sup> A Garbage Collection, azaz szemetgyűjtés egy automatikusan végbemenő optimalizálási folyamat, amely során futásidő közben felszabadulnak olyan memóriaterületek, amelyek a kód korábbi részén lefoglalásra kerültek, azonban a program későbbi szakaszban már egyáltalán nem történik hivatkozás rájuk.

## 0.6. Az ellenőrzött kódok előnyei és hátrányai

- A futtatott kód biztonságosabb, erről ellenőrző algoritmusok gondoskodnak (például ilyen az *array boundaries check*, ami a tömbök túlindexelését vizsgálja).
- A szemétygyűjtés automatikusan lezajlik, ezért nem is kell memóriaszivárgástól tartanunk.
- A dinamikus, azaz futásidéjű típusellenőrzés biztosítva van.

Átokként és áldásként is felerőható, hogy nem enged direkt ráhatást memóriacímekre: hogy a lefoglalások és felszabadítások mikor, a szekvenciának mely pontjain történnek meg, arról nem tudunk konkrét információkat, mivel a Garbage Collector ezt elrejti előlünk, a program bizonyos pontjain csak sürgetni tudjuk, hogy az optimális eljárást előbb végezze el.

Ami ténylegesen hátrányként említhető, hogy nem enged hozzáférést alacsonyabb szintű részletekhez, így különböző mikrokontrollerek, hardverelemek vezérlésének szemögléből kiindulva a programozás ezen módja nem tekinthető opciónak.

## 0.7. A Platform Invoke működése

A managed és unmanaged kódok közötti adattovábbítást az úgynevezett Platform Invoke (röviden PInvoke) végzi. A PInvoke-keretrendszer a projekt szempontjából egy fekete doboznak (black box) számít, mivel nem ismerjük ennek megvalósítási módját, ezért mindenkor érdemes körüljárunk, hogy mi történik a háttérben, amikor a `[DllImport]`-attribútumot ráteszünk egy-egy metódus szignatúrájára. Induljunk ki egy konzolos alkalmazásból, amely egy felügyelet nélküli (unmanaged) Win32-eljárást hív meg: egy Sleep-metódust, amelyet egy úgynevezett `'kernel32.dll'`-állományban lett definiálva, meghirdetve és implementálva `??`. Ha ezt a kódot megfuttatnánk, programot két metódus írná le: az egyik a `Program.Main`<sup>13</sup>, a másik a `Program.Main` törzsében meghívott DLL-ből érkező `Sleep`-eljárás, mindkettő IL-kódra fordul.

A .NET-keretrendszer 4-es verziója óta (projektünkben 4.7.2-es verziót használtam a DLL elkészítésére) minden PInvoke-kal hívott metódust köztes nyelvre fordítanak, és Just in Time kerül lefordításra a processzor-specifikus futtatható kódra.

A következő fogalmak kifejtésre szorulnak:

1. **Call Stack:** A Call Stack (hívási verem) információkat tárol a számítógépes

---

<sup>13</sup> A `Program.Main` a program belépési pontja C# esetében, innentől kezdődik elkészített kódjaink futtatása Lényegében ha a futtatást mappákra vetítenénk le, ő számítana a gyökérkönyvtár (root directory), amiből nyílhatnak további könyvtárak.

```

class Program
{
    [DllImport("kernel32.dll")]
    static extern void Sleep(uint dwMilliseconds);

    static void Main(string[] args)
    {
        Console.WriteLine("Press any key ...");

        while (!Console.KeyAvailable)
        {
            Sleep(1000);
        }
    }
}

```

5. ábra. Példa egy külsős, azaz PInvoke-eljárás definíciójára és használatára

Forrás: [39]

program aktív (meghívott) függvényeiről.<sup>14</sup> Bár a hívási verem karbantartása fontos a legtöbb szoftver megfelelő működéséhez, a részletek nem láthatók, magas szintű programozási nyelveken ráadásul automatikusan működnek. Számos processzor utasításkészlete eltérő utasításokat tartalmaz a programverem kezelésére. Amennyiben a Call Stack üressé válik, azt mondhatjuk, hogy kiléptünk a programból, mivel a Main-függvény is visszatért hívási helyére, azaz a vezérlés az operációs rendszerhez kerül. [28]

2. **Call Frame:** Olyan adatszerkezetek, amelyek az alprogramok állapotára vonatkozó információkat tartalmazzák: változók, visszatérési típus, visszatérés címe. minden egyes keret egy olyan függvényhívásnak felel meg, amely még nem tért vissza értékkel. Ha például egy DrawLine nevű metódus éppen fut, amelyet a DrawSquare hívta meg, akkor a Call Stack tetejére a DrawLine-metódus tulajdonságait összefoglaló hívási keret kerül. Amikor a metódus lefutott, azaz visszatért a hívás helyére, a Call Stack tetejéről törlődik a metódusnak létrehozott keret. A Call Stack tehát Call Frame elemekből épül fel. [28]
3. **CLR:** A Common Language Runtime .NET-keretrendszer futtatókörnyezete. A fejlesztett kódot, amely a CLR fordítóján megy keresztül, felügyelt (managed) kódnak nevezük. [16]
4. **Just-In-Time-fordítás:** A Just-In-Time-fordító .NET-ben a Common Language Runtime (CLR) része, amely a .NET programok végrehajtásának kezeléséért felelős, függetlenül a .NET programozási nyelvtől. A nyelvspecifikus fordító a for-

---

<sup>14</sup> Végrehajtási veremként, programveremként, vezérlő veremként, gépi veremként is találkozhatunk a fogalommal, ezek mind ugyanazon jelentéssel bírnak a fejlesztők körében.

rászkódot a köztes nyelvre alakítja át. Ezt a köztes nyelvet aztán a Just-In-Time (JIT) fordítóprogram alakítja át gépi kóddá. Ez a gépi kód specifikus arra a számítógépes környezetre, amelyen a JIT-fordító fut. Előnyei, hogy egyrészt gyorsítja a kód futtatását, másrészt segítségével platformfüggetlen C#-kódok készíthetőek. [7]

Amikor a CLR találkozik egy PInvoke metódussal, egy hívási keretet (Call Frame) tárol a Call Stackre (InlinedCallFrame), amely - többek között - a nem menedzselt függvény címét tartalmazza, mielőtt meghívna a tényleges.

A csonk viszont lekéri ezt a a `StubHelpers.GetStubContext()` segítségével, és meghívja a nem felügyelt függvényt. [39] [14]

## 0.8. Memóriakezelés az InterOp marshaller segítségevel

Az Interop marshaller a .NET-keretrendszer egyik komponense, amely a managed adattípusok unmanaged típusokká történő konverziójáért felelős. A memóriakezelést illetően az Interop marshaller biztosítja, hogy a memória helyesen kerül lefoglalásra, és felszabadításra a konverzió folyamatában, különösen, ha nem felügyelt kóddal dolgozik. Amikor felügyelt objektumokat ad át nem felügyelt kódnak, az Interop az objektumot mély másolással<sup>15</sup> hozza létre, hogy a nem felügyelt kód a managed objektum befolyásolása nélkül módosíthassa az objektumot.

Amikor viszont nem felügyelt objektumokat adunk át a felügyelt kód részére, az Interop marshal egy sekély másolatot hoz létre, így a felügyelt kód hozzáférhet a memóriához. Ez azt jelenti, hogy írhatja és olvashatja egyaránt ezen változók értékeit.

Fontos megjegyezni, hogy a memóriakezelést az Interop Marshaller automatikusan elvégzi, így a fejlesztőknek nem kell aggódniuk memóriaszivárgás miatt. Bizonyos esetekben azonban szükség lehet a használt memória explicit módon történő felszabádítása. [21]

---

<sup>15</sup> A mély másolás (angoul deep copy) azt jelenti, hogy objektumonként a referencia szerint tárolt mezők is klónozódnak, úgyhogy két objektum között egyáltalán nem lehet közös referencia. Ennek ellenére a shallow copy, azaz a sekély klónozás, amelyben a tárolt objektumok memória-címe egy az egyben átmásolásra kerül, de ekkor egy objektum ilyen típusú mezőjének változása a klónban ugyanazt a változást eredményezi.

## 1. fejezet

# Elosztott rendszerek

Az elosztott rendszerek lényege, hogy van egy alkalmazás, amivel különböző kliensprogramok (ügyfelek) kommunikálhatnak, ez a szerveralkalmazás (másnéven host vagy kiszolgáló), amely legtöbb esetben egy másik számítógépen fut, amely számítógép ráadásul egy másik hálózat tagja, tehát a két gép közötti adatátvitel internetkapcsolat segítségével valósítható meg.

Az kliens és a szerver kommunikációja kétirányú: az ügyfelek többnyire kéréseket küldenek a kiszolgáló valamely erőforrásának (adatbázis) eléréséhez, a szerver is küld a kliensprogram részére üzeneteket, tájékoztatást hibaüzenetről, a végrehajtott művelet sikerességéről, annak eredményéről, és így tovább.

A szerveralkalmazás is fogadhat bizonyos adatokat a kliensektől, ezek lehetnek bejelentkezási adatok vagy éppen a felhasználó által használt kliensprogram verziószáma, elavultabb verziók esetén akár figyelmezheti is a felhasználót, hogy telepítse a program egy frissebb változatát. [3]

Kliens-szerver-kommunikáción alapuló alkalmazásokat különböző módokon építhetünk fel, amelyek a téma szempontjából csak annyiból fontosak, hogy például szolgályanak programozási nyelvek közötti kapcsolatteremtés alkalmazási lehetőségeire.

### 1.1. Gondolataim a streamalapú kommunikációról

Ebben a megközelítésben a kliens és szerver kódjának nyelve megegyezik, a téma szempontjából már ez is egy tanulsággal szolgál: kell egy közös nyelv, egy szerződés a két fél között, amelytől egyik fél sem térhet el<sup>1</sup>: a streamalapú kommunikáció ezt az alábbi módon valósítja meg: szövegesen eljuttatja a szerver számára a végrehajtani kívánt metódus nevét és paramétereit.

Például egy kliensalkalmazásban elküldhetjük az alábbi üzenetet a streamen keresz-

---

<sup>1</sup> Amennyiben valamelyik fél eltér a meghatározott szabványtól, a kommunikáció nem fog létrejönni, a kliens és a szerver nem fogja érteni egymást.

tül: "*ADD/2/3*", ezt a szerver a várakozás állapotában megkapja, rá is illeszti az üzenet első tagját – esetünkben ez az "*ADD*" – az általa ismert parancsokra, és amennyiben ismeri az "*ADD*" parancsot, meghívja rá saját *Add(int a, int b)*-függvényét, majd egy hasonló stream üzenetet küld vissza a kliens számára: "*OK/5*". [30]

Láthatjuk, hogy van hasonlóság a küldött üzenetek között, egy szabvány, ami leírja a kommunikáció formáját. Elméletem szerint ez már önmagában nevezhető egy közös nyelvnek a kommunikációban résztvevő két fél között.

A szerver – így a kliens is – ismeri a következő halmazeit:

$$\mathbb{A} = \{"ADD", "SUB", "MUL", "DIV", "EXIT"\}$$

Ezeket véleményem szerint bátran nevezhetjük a közös nyelv **kulcsszavainak**, (amelyek használatával a kliens a távoli számítógép implementált metódusait hívhatjuk meg), és a résztvevő feleket ezekre feltétlenül fel kell készítenünk, hogy ezen parancsok meghívási módja, tehát az üzenetküldésre használt nyelv **szintaxisa**, valamint a várható eredmény ismert legyen.

Ha továbbgondoljuk, a nyelv szemantikája is megadható: tegyük fel, hogy az előbb felsorolt parancsok használatához szükség van bejelentkezésre. Bejelentkezik

*"LOGIN|username|password"*

üzenettel a szerverre, amely eldönti, hogy a bejelentkezási adatok passzolnak-e az általa eltárolt felhasználói rekordok bármelyikével, amennyiben igen, bejelentkezteti a felhasználót a szerverre. Ekkor már a többi parancs is megnyílik a számára, a "*LOGIN*"-parancs letiltásra kerül. Ha a bejelentkezést követően a felhasználó a

*"LOGIN|username|password"*

parancsot ismételten kiadná, kapna a szervertől egy "*ERROR|ALREADY\_LOGGED\_IN*" üzenetet.

Szintaktikailag<sup>2</sup> ugyan helyes parancsot adott ki a felhasználó – azaz az általa futtatott kliensprogram –, de már a parancs meghívása ebben a kontextusban (ebben a párhuzamban kontextusnak számíthatjuk a sessiont)<sup>3</sup> már egyszer megtörtént, így a

<sup>2</sup> **Szintaxis:** A szavak és mondatok helyességét vizsgáló tudományág. Programozásban az elgépeált kulcsszavak, azonosítók, az idézőjelek és zárójelek helytelen használata, és így tovább. A Chomsky 2-es típusú (kontextustól független) nyelvtanok leírják a legtöbb programozási nyelv szintaxisát.

<sup>3</sup> Egy többfelhasználós rendszerben munkamenetnek (session) nevezzük azt az időszakot, ami eltelik a felhasználó be- és kijelentkeztetése között, ekkor az szabadon végezheti a dolgát, nem kell minden egyes funkció használatához bejelentkeznie.

megkapott hibaüzenetben tulajdonképpen egy *szemantikai hibáról*<sup>4</sup> kap tájékoztatást a felhasználó. [27]

## 1.2. RESTful alkalmazások

A szerver-kliens-architektúrák megvalósításának egyik legnépszerűbb módja egy RESTful alkalmazás készítése. A szerver különféle erőforrásai (adatbázisban lévő táblák, az ezeket módosító függvények) különböző Unified Resource Identifiers (URI) címeken keresztül érhetők el. A kliensnek elegendő ezen URI-címek valamelyikét meghívni, a hívással egy kérést intéz a szerver felé, majd az valamilyen szabványos szöveges formátummal – ilyen például a JSON is – képes visszaküldeni a kérés eredményét a kliens-program számára.

## 1.3. Google RPC és a Proto-nyelv

A Google RPC, röviden gRPC a Google jóvoltából készült nyílt forráskódú RPC-keretrendszer. Remote Procedure Callnak<sup>5</sup> nevezzük, amikor *A* processz meghívja *B*-nek valamely meghirdetett szolgáltatását, így *B* a kért műveletet implementációja szerint elvégzi, majd *A* részére visszajuttatja az általa kért művelet eredményét. Így *A*-nak nem kell törődnie a végrehajtott algoritmus részleteivel, ennek implementációja és futtatása *B* részére delegálódik. Ez lehetőséget teremt arra, hogy bizonyos kód részleteket, implementációkat egy másik számítógépre, egy szerverre helyezzünk át, ettől lesz a hívás ”távoli”.

Hogy az RPC-t jobban megértsük, gondoljunk egy távirányítóna: a távirányítóval képesek vagyunk távolról vezérelni a televíziót, be- és kikapcsolhatjuk, csatornát váltathunk a segítségével, változtassunk annak hangerején, és így tovább. A távirányító a televízió azon tulajdonságait tudja elérni, amelyek számára ”meg vannak hirdetve”, azaz amely gombokra reagál a tévékészülék. A tévé által meghirdetett szolgáltatásait meg tudjuk hívni távirányítón keresztül. A ”távoli hívás” segítségével elérünk, hogy ne kelljen minden alkalommal fizikailag közel lennünk a készülékhez, amikor annak beállításain módosítani kívánunk. Ugyanakkor a távirányító használata alternatív megoldásként is szolgál olyan esetekre, amikor fizikai interakció nem vagy nehezebben kivitelezhető

---

<sup>4</sup> **Szemantika (jelentéstan):** A mondatok jelentésének vizsgálata egy kontextuson belül. A programozási nyelvek terén ez azt jelenti, hogy kifejezéseket vizsgálhatunk, hogy annak az van-e értelme az adott kontextuson belül. Például amikor egy ’i’ nevű változónak növeljük az értékét ( $i = i + 1$  vagy  $i++$ ), viszont ’i’ korábban nem volt definiálva, akkor *szemantikai hibáról* beszélünk. A Chomsky 1-es típusú (kontextusfüggő) nyelvtanok képesek leírni a legtöbb programozási nyelv szemantikáját.

<sup>5</sup> magyarul: Távoli eljáráshívás

(például meghibásodott a nyomógomb vagy túl magasan helyezkedik el a televízió).

A gRPC-t számos programozási nyelv támogatja, a teljesség igénye nélkül ezek a Java, C#, C, C++, Kotlin, Python és PHP-nyelvek. Még Delphiben is létezik egy külső komponens, amit a projekthez hozzáadva használhatjuk a gRPC nyújtotta szolgáltatásokat. [33]

A keretrendszer működtetője az úgynevezett *Protocol Buffers*, amely nyelv- platformfüggetlen adatcserét biztosít. A gRPC-ben ennek a segítségével teremthetünk kapcsolatot különböző vagy azonos programozási nyelven készített alkalmazásaink között, egy szerveralkalmazáshoz különféle platformokra írt kliensprogram is kapcsolódhat, mivel a kérések és válaszok nyelvezete közös, szabványos.

Ami weboldalaknál a HTML és a JSON-formátum, az a gRPC esetén a Proto-nyelv, ami átájárást biztosít akár más nyelven írt programok részére is. Amely nyelv képes implementálni a gRPC-t, a Proto-nyelvből le tudja gyártani maga forráskódját, valamint a forráskódjából is generálható a Proto-formátum, az alkalmas ezen a nyelven keresztül összekapcsolódni más, ugyanezt a keretrendszert alkalmazó szerverprogrammal is. A Proto-nyelv választott témám szempontjából krucialis, mivel kifejezetten erre a célra lett kitalálva, hogy programozási nyelvek felett álljon, ezért úgy gondolom, érdemes tovább vizsgálnunk a nyelvben rejlő lehetőségeket.

Miért kell ehhez egy külön nyelv? Nem lehetne maradni a jól bevált JSON-nél? A Proto-nyelv további előnyökkel jár, amelyeket sem a JSON, sem az XML, sem bármiilyen más szerializált formátum nem tud biztosítani: az adatszerkezeteket szigorúan típusosan írja le.

Ezenkívül nemcsak adatok, hanem szolgáltatások leírására is egyaránt alkalmas a Proto-nyelv, ezért is érdemes az RPC ezen változatát használni, mivel nagyobb rugalmasságot biztosít, mint egy átlagos szerializált formátum.

```
service Adopt {
    rpc Login(User) returns (SessionID);
    rpc Logout(SessionID) returns (Result);
    // ...
}

//bejelentkezéskor user ezt kapja, ezt használjuk a felhasználó körültekercsére
message SessionID{
    string id = 1;
    string username = 2;
}

//CRUD-műveletek visszajelzése szöveges formában
message Result{
    string response = 1;
}

//bejelentkezéskor ezt küldi a kliens a szervernek lekérdezésre
message User{
    string name = 1;
    string passwd = 2;
    bool isAdmin = 3;
}
```

1.1. ábra. Beandandó projektben készített .proto állomány (részlet)

Lényegében elég a fentebb látható .proto kiterjesztésű fájlt elkészítenünk, ennek környezetét, a kliens- és szerveroldali stubok forráskódjának vázát az adott nyelv sa-

bályrendszerre szerint képes lesz legyártani ebből. Ennek lehetősége a legtöbb RPC-keretrendszerrel nem áll fenn.

Mint azt említettem, Proto-nyelvben a változókat csak típusossal együtt vehetünk fel<sup>6</sup>, osztályokat `message`, a meghívható metódusokat a kifejezőbb `rpc` kulcsszóval látja el. Mint láthatjuk, a mezők nevéhez 1-től kezdődően számokat rendelünk, ez a serializációhoz szükséges. Binárisan ez úgy fog kinézni, hogy a mező azonosítóját annak típusával kombinálják, ezzel lényegében egy mező leírásához pusztán 1 bájtnyi terület elegendő. Ha üzeneteinket úgy szervezzük, hogy azonosítókat 1-től maximum 15-ig terjedően oszthatunk ki (tehát legfeljebb 15 mezőt engedünk meg osztályonként), akkor ez a tulajdonság garantált.

Természetesen ez is a gRPC előnyeihez sorolható, hogy kevesebb adatot kerül átadásra szervertől kliensig, és fordítva. Az adatokat a lehető legtömörebb bináris formátumban képes leírni és szállítani a hálózaton, a JSON-formátum ezzel szemben egy az egyben szövegesen kerül átadásra, ami jóval több adatforgalmat tesz ki.

Emellett nyugodtan kijelenthetjük, hogy ez a megoldás típusbiztonsági szempontból is előnyösebb, mivel a Proto-nyelvet használva a mezőket eleve típusokkal kötjük, így nem az érkezés helyein kell validálni a megérkezett adatokat. Szöveges szerializáció esetében minden alkalommal, amikor például JSON érkezik, elsősorban ellenőriznünk kell, hogy a fogadott JSON-stringben az értékek típusai rendre megegyeznek-e az osztályban definiált mezők típusaival. Ez a Proto-nyelv esetében automatikusan teljesül, mivel a típusossal kapcsolatos hibák már a küldéskor kiütköznek.

Az adatokat a lehető legtömörebb bináris formátumban képes leírni és szállítani a hálózaton, a JSON-formátum ezzel szemben egy az egyben szövegesen kerül átadásra, ami jóval több adatforgalmat tesz ki. [11]

GRPC-vel négyféle metódushívást tudunk leírni

- *Unáris hívások*: Itt egyszerű kérés-válasz-üzeneteket cserélünk. Ez Protoban úgy néz ki, hogy a szolgáltatások paramétere és visszatérési értéke is `message` típusú.
- *Kliens streamelés*: A kliens egy streamet, egy adatfolyamot (kvázi egy listát) küld paraméterben a szerver részére, a kiszolgáló pedig egy normál `message`-objektummal tér vissza.
- *Szerver streamelés*: A kliens elküld egy kérést, és a szervertől válaszként streamet fogad.
- *Kétirányú streamelés*: A kliens és a szerver is streamekben kommunikál egymással.

---

<sup>6</sup>Ezen tulajdonsága miatt a Proto szigorúan típusos nyelvnek számít.

```

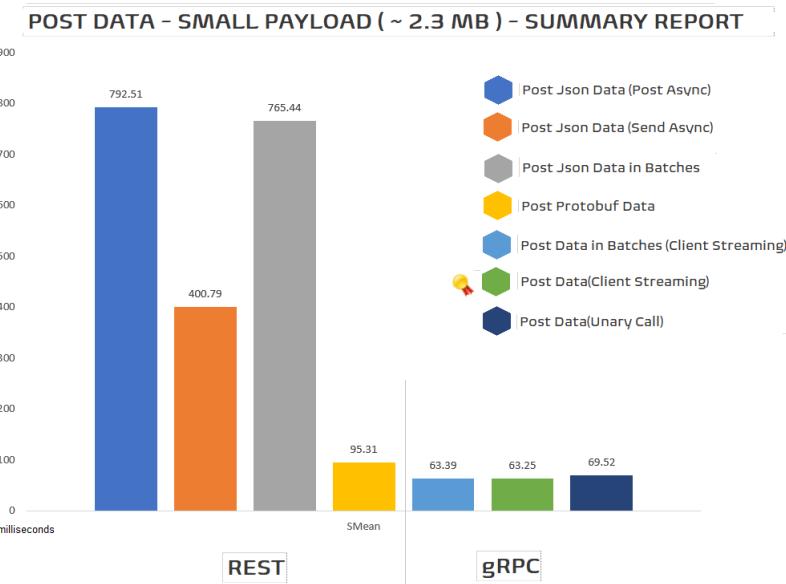
service Adopt{
    // unáris hívás
    rpc Login(User) returns (SessionID);
    // szerver streamelés
    rpc ListAnimals(Empty) returns (stream Animal);
    // kliens streamelés
    rpc Adopt(stream Animal) returns (Result);
    // bidirectional / kétirányú streamelés
    rpc FilterAnimals(stream FilterAttribute) returns (stream Animal);
}

```

1.2. ábra. gRPC metódushívásainak típusai

### 1.3.1. Jobb a Proto a JSON-formátumnál?

Az alábbi ábrán látható, hogy a bináris szerializációt alkalmazó Proto-kérések futásidőben sokkal gyorsabbnak bizonyultak, mint a JSON-t használó REST-hívások, a kísérlet győzteseként a *kliens streamelés* került ki (63.25 ezredmásodperccel), ami körülbelül 13-szor gyorsabb, mint a leglassabb REST-módszer (792.51 ms), és legalább 5-6-szor gyorsabb, mint egy átlagos REST JSON-hívásokkal. A másik két gRPC-hívás (Unáris és kötegelt<sup>7</sup> kliens streamelt hívás 63.39 ms és 69.52 ms eredményekkel) is hasonlóan teljesítettek, és pár ezredmásodperccel maradtak le az első helyezetthez képest. [1]



1.3. ábra. Szerelmezési teljesítményének összehasonlítása JSON és Proto esetében

Forrás: [1]

<sup>7</sup> A *batch*, azaz kötegelt programozásban nagy mennyiségű adatot kötegelt formában dolgozunk fel. Az adatok egy bizonyos időszakban összegyűlnek, majd ezek feldolgozása egy menetben történik. Ilyen például a bérszámfejtés, ami megvárja, hogy egy munkás munkaórái összegyűljenek a hónap során, majd a hónap lezárultát követően a megadható munkabért meghatározza. Forrás:[32]

### 1.3.2. Szakdolgozati munkánkról



1.4. ábra. Lámpa eszköz



1.5. ábra. Nyíl eszköz

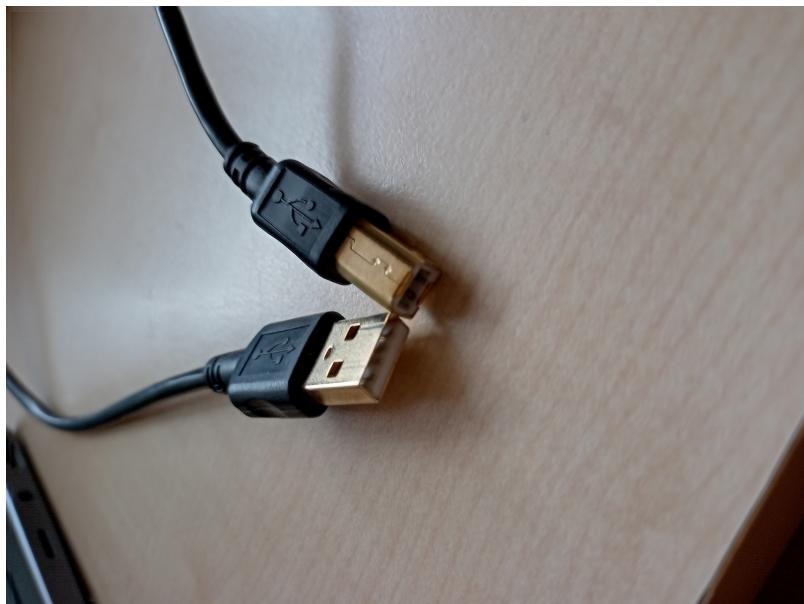
### 1.3.3. Csatlakozók, átviteli eszközök

Eme dokumentum mellé készített szakdolgozati szoftverprojektemet Sipos Levente hallgatótársam részére, az ő feladatának megkönnyítésének céljából készíttem. Munkám arról szól, hogy Delphiben implementált metódusokat lehessen meghívni a .NET-keretrendszerből, ennek lehetséges módját kutassam. A C# és Delphi programozási nyelvek összehangolása az én feladatom, az elkészült termék úgynevezett segítő, azaz helper metódusok implementációit tartalmazó, lefordított<sup>8</sup> DLL-projekt lesz, amelyeket Levente az ő Windows Forms keretrendszerrel készített grafikus alkalmazásában tud meghívni. Jelen állás szerint szükség volt továbbá egy úgynevezett relayDLL-re, amelyet én állítottam össze.

<sup>8</sup> Lévén a C# fordított nyelvnek számít, ezért az alkalmazások futtatásához/használatához az elkészült forráskódokat első lépésben futtatható állományra (gépi kódra) kell fordítani egy fordítóprogram segítségével. Ezt a folyamatot szakszóval *compilingnak* vagy *buildelésnek* is nevezzük.



1.6. ábra. RJ9-es csatlakozó vezetékes telefonkábel végén. Csak a középső 2 ér van csatlakoztatva.



1.7. ábra. nyomatokábel USB-A és USB-B végekkel, előbbi a számítógéphez, míg utóbbi az elsőnek szánt eszközhöz csatlakoztatandó.

## 1.4. Miről is szól munkánk?

Egy mentális egészségfejlesztésre használatos alkalmazást fejlesztésére vállalkoztunk 2022 szeptemberében, amely elméleti alapjait Somodi László futballedző munkásságának köszönhetjük, ezek gondolati vonulatairól titoktartási szerződésünk lévén csak nagyon érintőlegesen fogok beszámolni a későbbiekbén.

A készített alkalmazásunk gyakorlatilag különböző fény- és hangjelzések kibocsátására alkalmas eszközök (lámpák, nyílak és hangszórók) vezérléséből áll, egy úgyneve-



1.8. ábra. Tápkábelt használunk a további, számítógéphez közvetlenül nem csatlakozó eszközök feszültségre kapcsolásához.

zett *intelligens szobában* az eredeti tervek szerint 8 eszköz (a készített program azonban tetszőleges,  $n$  darabszámú eszköz vezérlésére lett felkészítve) együttes vezérlését kell kezelnünk megadott időközönként, amely időközöket egyszerűen ütemnek fogjuk nevezni. A módszer alkalmazása az intelligens szobával együtt működik teljességében.

## 1.5. Mit jelent az intelligens szoba ?

Az intelligens szoba elnevezés olyan helységet takar, amelynek mind a négy falán jelzőkat helyeztünk el. Ezek különböző típusú eszközök: fények, nyilak és hangok. Ezek különállóan vagy együttesen jeleket küldenek, amely jelekre speciális mozdulatokat kell a foglalkoztattnak végre hajtani. Az eltérő színek más feladatokat írnak le, mást kell tenni piros, és színtén mást zöld szín felvillanása esetén, ez emlékeztetheti az embert a forgalmi jelzőlámpák működésére is: minden színhez más jelentést rendelhetünk. A nyilak felvillanásának és a hangszóróból érkező különféle hangok észlelése esetén pedig irányváltásokat kell végezni.

Egy feladatsor több egymást követő ütemből áll, mely a programban azt fogja jelenti, hogy  $x$  másodperc késleltetéssel a felmért eszköz tömb elemeinek tulajdonságait (mezőit) a feladatsor aktuális ütemének megfelelően módosítjuk. Mivel minden eszközöt együttesen vezérlünk, a program ütemenként az összes eszköz állapotát felül fogja írni, ezért szükségünk van egy olyan állapotra is, ami azt közli az adott eszközzel, hogy éppen semmit ne csináljon (azaz várakozzon). Ez a fényeszközök esetében egy-egyszerűen (0,0,0) RGB-színkód<sup>9</sup> közlését, míg hangeszköz esetén egy 0 dB hangerősségű

---

<sup>9</sup> Az RGB-színkódolás egy szín leírását három komponens, vörös (**Red**), zöld (**Green**) és kék

tetszőleges frekvenciájú hangjelzés kiküldését fogja jelenteni. Somodi László edzővel való együttműködésünk Dr. Király Roland tanár úr jóvoltából jöhetett létre, akinek az alkalmazással kapcsolatos múltbeli tapasztalatait és ötleteit folyamatos egyeztetések, konzultációk útján tudtuk segítségül hívni.

## 1.6. Delphi és C# programozási nyelvek összehasonlítása

Először is érdemes leszögeznünk, hogy a továbbiakban Delphi alatt nem a fejlesztői környezetet, amely az Object Pascal nyelvvel dolgozott együtt, hanem inkább a programozási nyelvet értjük, jelenleg az Object Pascal megnevezés úgynevezett „umbrella term” formájában él tovább.<sup>10</sup> [40] A Delphi nyelv (megjelenési éve: 1986, a Borland nevű cég jóvoltából) idősebb nyelvnek számít a C#-hoz (megjelenési éve: 2001, a Microsoft jóvoltából) viszonyítva, ebből fakadóan egy stabilabb, kiforrottabb, időtállóbb eszköznek számít a programozók kezében.

Mindkét nyelv **objektumorientált**, ami azt jelenti, hogy bizonyos, logikailag összetartozó adatokat (mezőket), valamint a rajtuk végezhető műveleteket (metódusokat) egy egységre zárunk, ezt az egységet a továbbiakban osztálynak nevezzük. Az osztály mezőit és metódusait különböző láthatósági szintekkel vörtezhetjük fel, ezzel tudjuk védeni az osztályunk kritikus részeit más osztályokkal szemben. Az osztályok között akár öröklődéssel, akár objektum-összetételelviszonyokat alakíthatunk ki.

A két nyelv **erősen típusosnak** számít, mivel egy változó definiálásakor meg kell mondunk azt is, hogy milyen típusú értékeket szeretnénk abban tárolni, a típusok már a forráskódban explicit módon megjelennek, így az adott nyelv fordítóprogramja fel van készítve a változó típusaira, amely változtató hatókörén<sup>11</sup> belül nem is változhat meg. Ahogy azt már korábban is említettem, a változó típusa meghatározza, hogy az értéket hogyan kell értelmezni a memóriában, a megadott típusnak mely műveletei vannak értelmezve, így például **string** típuson nem értelmezhetünk logikai ÉS (konjunkció)-műveletet.

Az elkészült kódok teljesítménye alapján is érdemes összehasonlítani a két nyelvet. Ehhez először is külön kell választanunk a fordításhoz és a futtatáshoz szükséges időt,

---

(Blue) alapszínek arányától teszi függővé.

<sup>10</sup> Az umbrella term (magyarul gyűjtőfogalom) olyan kifejezés, amely több fogalmat rendel önmaga alá, így már fogalmak egy csoportját, kategóriáját jelenti összefoglalóan.

<sup>11</sup> Hatókör, illetve scope alatt a program azon részét, kontextusát értjük, amely magába foglalja az adott változót. Ezen kontextusban vizsgálva a változó "életben van", tehát a memóriában hely van lefoglalva számára, nevére vagy memóriacímére hivatkozva értékét felülírhatjuk, kiolvashatjuk.

lévén ezek fordított (nem interpretált) nyelvek, ezért ezek nem szimultán módon<sup>12</sup> törtennek. A Delphi fordítóprogramja azonnal gépi kódot<sup>13</sup> készít, míg a C# esetében első lépésekben egy köztes nyelvű<sup>14</sup> kód készül, amelyet a .NET virtuális gép képes futtatni. Bár fordítási időben a Delphi-kód abszolút győztesnek tekinthető – mivel fordítóprogramja közvetlenül futtatható állományt készít –, a két nyelv segítségével készített programok futásideje közel azonos. Így megállapítható, hogy mérvadó teljesítménybeli különbséget kizárolag a fordítás folyamatában észlelhetünk.

Ha ennél is tovább megyünk, akkor a C#-nak nagy előnye származik abból a Delphi-val való összevetésben, hogy aszinkron<sup>15</sup> kódolási lehetőséget is biztosít, amely felgyorsítja a végrehajtást, jobban ki tudja használni a rendelkezésre álló CPU teljesítményét. A LINQ<sup>16</sup> használata `yield` kulcsszóval az iterációkban biztosítja, hogy csak akkor van végrehajtva a kód, amikor ténylegesen szükség van rá (lusta kiértékelés). A delegate-ek használata szintén növelheti a C#-kódok teljesítményét, bár a Delphi is rendelkezik ehhez hasonló funkciókkal. [31]

A fejlesztés folyamán elkészített **modulokat** a különböző programozási nyelvek eltérő megnevezéseket használnak.<sup>17</sup> A vizsgált két nyelv tekintetében is ez áll fenn: a C# `namespace`, míg a Delphi `unit` kifejezéssel illeti, a lényegük ugyanaz lesz:

1. **Elnevezések:** Mivel előfordulhat, hogy két, funkciójában eltérő osztálynak ugyanazt a nevet kellene adnunk<sup>18</sup>, nem tehetnénk meg, mivel a fordítóprogram nem tudná eldönteni, hogy a két változat közül éppen melyiket kívánjuk érvényesíteni. Ezt a problémát orvosoljuk például a kódok modulokra való felosztásával.
2. **Egységbe zárás:** A modulok lehetővé teszik a programozók számára, hogy egy funkcionálitás végrehajtási részleteit kapszulázzák és elrejtsék. Ez azt jelenti,

<sup>12</sup> nem egyidőben, nem egyszerre

<sup>13</sup> A gépi kódban már az utasításokat is számok jelzik, ezen nyelv utasításkészlete már a számítógében működő processzor típusától is erősen függ. Gépi kód az esetek nagy részében fordítóprogram eredménye, a hardverközeli vezérlőprogramok elkészítéséhez is inkább a magasabb szinten álló Assembly nyelvet használják.

<sup>14</sup> A(z) (Common) Intermediate Language a .NET-keretrendszerben a magasabb absztrakciós szintű C# és a legalacsonyabb gépi kód között „félúton” helyezkedik el, ami még processzortól és operációs rendszertől független. A .NET Runtime futtatókörnyezete képes futtatni.

<sup>15</sup> Az aszinkron programozás lehetővé teszi, hogy az alkalmazás egy időigényes folyamat futtatását háttérbe helyezze, így a programot futtató szál, lévén nem várakozik a válaszra, addig ugyanúgy képes a felhasználói interakciókat kiszolgálni. [13]

<sup>16</sup> A **Language Integrated Query** (magyarul: nyelvbe ágyazott lekérdezés) egy gyűjtőfogalom a C# nyelvbe épített szintaktikai elemekre, amely elemek lehetővé teszik, hogy akár lambda kifejezésként, akár SQL-szintaxishoz hasonló módon meg tudjunk fogalmazni lekérdezéseket bizonyos - iterálható, vagyis bejárható, az `IEnumerable`-interfészt megvalósító - szerkezetekre. Ilyen szerkezetek minősül például a `List` is.

<sup>17</sup> Az említett példákon kívül a Java-nyelvben *package*-ként, míg a Pythonban *module*-ként hivatkoznak az osztályokat összegyűjtő egységre.

<sup>18</sup> Ilyenre példa, hogy

hogy a modulok tiszta felületet biztosítanak a funkcionálitással való interakcióhoz, miközben a mögöttes kódot elrejtik és védi a véletlen változásoktól.

3. **Újrafelhasználhatóság:** A kód modulokba történő kapszulázásával a programozók olyan kódot hozhatnak létre, amely több programban is felhasználható. Ez csökkenti a fejlesztési időt és erőfeszítéseket azáltal, hogy a programozók újra felhasználhatják a már megírt és tesztelt kódot.
4. **Karbantarthatóság:** Ha a kódot modulokba szervezik, könnyebb karbantartani és frissíteni, mivel az egyik modulban anélkül lehet változtatásokat végrehajtani, hogy az befolyásolná az azt használó többi modult. Ez javítja a kód karbantarthatóságát és csökkenti a hibák bevezetésének valószínűségét.
5. **Moduláris felépítés:** A modulok használatával kódjainkat több kisebb, könnyebben kezelhető darabra szedhetjük szét. Ez megkönnyíti a kód megértését, és mivel a komponensek így külön fájlokban találhatóak, így a kódot szimultán módon a fejlesztőcsapat több tagja is fejlesztheti.
6. **Importálhatóság:** A modul legyen beemelhető más komponensbe, ehhez szükség van egy olyan összefoglaló névre, amire hivatkozni tudunk, amikor a modult használni szeretnénk máshol is.

Összességében a modulok használata a modern programozás alapja, mivel segítséggükkel kódjaink a későbbiekben felhasználhatóak több helyen is, karbantartásuk gyorsabb és egyszerűbb. A programozók a programokat önálló darabokra tervezik, amelyeket meg tudnak írni, majd akár el is felejthetik egy-egy komponens belső működését, tehát hogy a problémára adott megoldás hogyan lett megoldva, elég csak azzal foglalkozniuk, hogy mi az a problémakör, amit a program ezen szelete lefed.

[6]

Ami közös még a két nyelvben, hogy a fejlesztés moduljaiból Win32-szabványnak megfelelő DLL-ek készülhetnek a segítségükkel, ezek szintén lefordított, gépi kódú állományok, amelyek más szoftver forráskódjában felhasználhatóak, lényegében így válnak futatható állománnyá.

## 1.7. Miket tud a C#, amit a Delphi nem?

- automatikus memóriakezelés - Garbage Collector gondoskodik a memória felszabadításáról futásidőben.
- lambda kifejezések és LINQ: A LINQ, vagyis a Language-Integrated Queries – nyelvbe ágyazott lekérdezések – a C# 3.0 nyelv és keretrendszer újdonsága, amely

használatával egyszerűbb szyntaxissal végezhetünk tömbökön, listákon, és adatbázisokon egyaránt lekérdezéseket. A típusellenőrzést a .NET fordítási időben végzi. A lekérdezések egymáshoz láncolhatóak, tehát egyetlen utasítással egészen összetett szűréseket tudunk végezni letárolt adatainkon. [23]

## 1.8. Miket tud a Delphi, amit a C# nem?

- Alacsonyabb szintű, Assembly nyelvű kódokat is futtathatunk Delphi-kódból, ez történik a programunkban használt SLDLL-állományban is.
- A kód nem érzékeny kis- és nagybetűre, ezért a változók, a különböző kulcsszavak, metódushívások leírhatóak tetszőleges formátumban.<sup>19</sup>
- Az utasításblokkok határait a `begin` és `end` kulcsszavak jelzik.

[41]

## 1.9. Alaphelyzet

Amikor a munkát megkezdtem, rendelkezésemre állt Dr. Király Roland tanár úr által fejlesztett Delphis asztali alkalmazás, amely – mint kiderült – a mai napig teljes mértékben használható. Ezen alkalmazás forráskódjában követtem végig az hardvereket működtető függvények hívási sorrendjét (szekvenciáját), ezek módját és eredményeit, az esetlegesen előforduló hibaüzeneteket. Ezen alkalmazás C#-nyelvű megfelelőjének elkészítését és továbbgondolását kaptuk feladatként Leventével.

## 1.10. Az SLDLL-függvények bemutatása

A következőkben a Keresztes Péter tanár úr által Delphiben implementált metódusokat, ezek kezelésének lehetőségeit fogom részletezni. Általanosságban elmondható, hogy minden függvény egész típusú értékkel tér vissza, amely érték tájékoztat a lefutás eredményességről: amennyiben a hívott metódus sikeresen (hiba, kivétel nélkül) lefutott, 0-val tér vissza, ahogy ezt egyébként az operációs rendszerek processzeinél is megszokhattuk. Ettől eltérő értékek az egyes hibatípusokat hivatottak meghatározni a Win32-szabvány<sup>20</sup> keretein belül. Ezen hibakódok projektünkre vonatkozó részét az alábbi táblázatban 1.9a összegyűjtöttem. [19]

---

<sup>19</sup> Angolul: case insensitivity

<sup>20</sup> A Win32-es hibakódok szabványa szerint minden hibakódnak a 0x0000 (decimálisan: 0) és 0xFFFFFFF (decimálisan: 16 777 215) közötti tartományban kell lennie.

A DLL, miután használatba vettük, az `SLDLL_Open` hívásakor paraméterben megadott címre (A Form Handle-címe) küldött Win32-üzenetekkel tartja a kapcsolatot a felhasználóval (hívó féllel). Az üzenet (Message) WParam értéke tartalmazza minden esetben a lefutás sikereségéről szóló információt. Míg a negatív értékek hibaüzenetnek, addig a pozitívak tájékoztatásnak számítanak az elvégzett feladat sikerességét illetően. Ha az üzenethez tartozik paraméter, adat, akkor arra az üzenet LParam értéke hivatkozik. Erre vonatkozó példával `SLDLL_Listelem` függvény hívásakor találkoztam, amikor is az eszközök darabszámát jelentő egész számot kell kigyűjtenünk a Win32-üzenetben beállított LParam értéke szerint.

- FELMOK (1) -> a felmérés lezajlott, a Win32-üzenet WParam mezője a talált eszközök számát – drb485 leendő értékét – is közli.
- FIRMUZ (3) -> a frissítés adatait tartalmazó struktúra (UPGPCK) címe
- LEDRGB (5) -> az általános állapotinformáció struktúra (ELVSTA) címe
- NYIRGB (6) -> az általános állapotinformáció struktúra (ELVSTA) címe
- HANGEL (7) -> az általános állapotinformáció struktúra (ELVSTA) címe
- STATKV (8) -> az általános állaptinformáció struktúra (ELVSTA) címe
- LISVAL (9) -> a táblázat végrehajtás végének hibakódja
- FELMHK (-3) -> A felmérés hibával zárult.

Az egyes funkciók visszatérési kódja tájékoztat a hívás sikeres vagy sikertelen voltáról. Ha a visszatérési kód `NO_ERROR` (0), akkor a hívás sikeres volt. Eltérő érték esetében a kód tájékoztat a sikertelenség okáról. A lehetséges értékek az adott hívás leírásában ismertetésre kerülnek.

A program azzal nyit, hogy felméri az USB-porton csatlakoztatott, egymással telefonkábelrel sorba kötött eszközöket, őket a típusának megfelelő azonosítóval látja el. A telefonkábel egy sajátos változatát használtuk, mivel a 4 érből pusztán a középső kettőt vettük igénybe, a két szélső eret ki kellett iktatnunk annak érdekében, hogy azok a soros kommunikációt ne zavarják. Az eszközök az RJ11 szabvány csatlakozók 4 pines változatával az erre szolgáló portokon keresztül lettek egymáshoz kapcsolva. Az azonosító meghatározza, hogy egy eszköz milyen típusú.

### 1.10.1. Open – DLL megnyitása

A program indulásakor legelőször lefutó `SLDLL_Open`-függvényt meghívva elkezdhetjük az `SLDLL` további metódusainak használatát. A hívást érdemes kódból elvégezni,

ennek kezelését nem kell a felhasználóra bíznunk, mivel egyrészt elfelejtheti, másrészt felesleges, hogy a felhasználó tudjon a kötelező metódushívási sorrend betartásáról.

A függvény továbbá elvégzi a Form felcsatlakoztatását a Win32 üzenetküldési láncra a paraméterben kapott Handle memóriacíme alapján, innentől a DLL üzenetküldésre is alkalmas az asztali alkalmazásunk részére. Tervben volt részünkről egy Delphi-ben írt konzolos alkalmazás elkészítése is, amely a .NET-ben készült grafikus felületünkkel kommunikált volna. Ebben az esetben kizárolag az asztali alkalmazás Handle-paraméterét tudtuk volna átadni, tehát akkor is a C# üzeneteket feldolgozó függvényével (WndProc – bővebben a metódus működéséről: 1.13.6) lenne dolgunk.

### **1.10.2. Listelem – A tömb beállítása**

Ez a függvény felelős a dev485 tömb beállításáért. Mivel itt találkozunk először a dev485-tömbbel, ezért itt fejtem ki, hogy mi a tömb funkciója. Ez a tömb úgynevezett DEVSEL típusú elemeket vár, ez a felmért eszközeink adatait tartalmazó struktúra. A DEVSEL leírja egy eszköz:

- azonosítóját: ez az eszköz legfontosabb attribútuma a vezérlés szempontjából, az azonosítón keresztül állíthatjuk be, és küldhetünk ki jelet egy bizonyos eszköz számára, amelyet vezérelni szeretnénk.
- verzióleíróját: ez egy VERTAR típusú objektum, amely leírja az eszközön futó vezérlőszoftver verzióját, a fejlesztés dátumát 1.00 17/11/28 formátumban.
- forgalmazóját: Ez számunkra egy konstans szövegérték ("Somodi László") lesz.
- gyártóját: Szintén egy konstans szövegérték ("Pluszs Kft.") lesz.

Optimalizációs szempontból az utóbbi két – ha a verziószám minden eszköznél egyezik, akkor három – attribútumot véleményem szerint nem is érdemes DEVSEL-példányonként letárolni, elegendő lenne pusztán azokat, amelyek eszközönként változó értékek. Az eszközök azonosítónak ismeretével az eszközöket vezérelni tudjuk. Az SLDLL\_Listelem függvény ezen adatok feltöltését végzi el számunkra. Ennek a függvénynek mindenéppen később, az SLDLL\_Open után kell lefutnia, különben működése hibakódot eredményez.

### **1.10.3. Felmeres – Eszközök felmérése**

A hívási sorrend (szekvencia) következő lépése az SLDLL\_Felmeres-függvény meghívása, amely az USB-porton észlelt eszközök felmérését indítja el. A nyomtatókábel USB-B vége a számítógép bármely – szintén – USB-B portjára csatlakoztatható. Kifejezetten előnyös tulajdonsága a metódusnak, hogy képes eldönten, pontosan melyik

porton történt meg a csatlakoztatás, így nem kell elkülönítenünk egy portot ezen eszközök detektálására.

Elméletem szerint ezen függvény meghívása következett volna az SLDLL\_Open-t követően, viszont az SLDLL\_Listelem a gyakorlatban ezt megelőzi, mivel a vezérlés már a WndProc-függvényben erre kerül. A függvény nem vár paramétert, visszatérési értéke egész szám, amely a lefutás sikereségéről tájékoztat, ezek a következők:

- NO\_ERROR: Nem történt hiba, a végrehajtás sikeres volt.
- ERROR\_DLL\_INIT\_FAILED: Az SLDLL\_Open még nem lett meghívva.
- egyéb - Windows műveleti hibakódok.

#### **1.10.4. SetLista - Az eszközbeállítások végrehajtása**

Amennyiben a dev485 tömb elemeit hozzárendeljük egy másik, LISELE-típusú elemet tartalmazó tömbhöz. A LISELE-struktúrában a következő attribútumok kerülnek tárolásra:

1. eset: Az eszköz lámpa vagy nyíl típusú
  - vilrgb: Az eszköz milyen színnel világítson ?
  - nilmeg: Az eszköz milyen irányban világítson ? - Háromféle értéket vehet fel:  
0 – Balra, 1 – Jobbra, 2 – Mindkét irányban. Amennyiben lámpáról van szó, ez az érték konstans 2 lesz.
2. eset: Az eszköz hangszóró típusú
  - handrb: Hány hangot szeretnénk lejátszani a hangszóró segítségével.
  - hantbp: A hangbeállítások tömbjének mutatója. Egy lejátszható hangot leír a hangerő, a hang hossza, valamint a hanglistából kiválasztott elem indexe.  
??

#### **1.10.5. Hangkuldes - A hangszórók vezérlése**

A fejlesztés során a SLDLL\_SetLista függvény meghívása nem volt elegendő a hangszórók megszólaltatásához, ezért a SLDLL\_Hangkuldes függvényre is szükségem volt. Ennek 3 paramétere van, az első a lejátszani kívánt hangok száma, a második a lejátszani kívánt hangok tömbje (ennek mérete az előző paraméterből következik), valamint az eszköz azonosítója, amelyet meg szeretnénk szólaltatni.

Az előzőekkel ellentétben az utóbbi 2 függvény nem került exportálásra, a Delphiben az általam készített SetTurnForEachDevices-függvény gondoskodik ezek helyes meghívásáról.

Az eszköz azonosítójának ismeretével kiszámítható annak típusa, így ez származtatható attribútumnak<sup>21</sup> számít.

### 1.10.6. Hibakódok

Az alábbi táblázatban látható hibakódokat C#-ban a következő megfelelő, egyénileg definiált kivételekkel, és sokkal kifejezőbb üzenetekkel váltom fel:

- **Dev485Exception**: Eszközök tömbjére vonatkozó hibaüzenetek.
- **SLDLLEception**: A DLL működésével kapcsolatos hibaüzenetek.
- **USBDisconnectedException**: Hibaüzenet arról, hogy az egyik USB-porton sem észlelhető eszköz.

Hibakód	Hiba címe	Hiba jelentése (SLDLL esetén)	Gyakorlati példa
0 NO_ERROR / ERROR_SUCCESS		A függvény sikeresen lefutott.	Program indításakor nem érkezik hibaüzenet.
13 ERROR_INVALID_DATA		Az azonosító típusjelölő bitpárosa hibás, vagy nincs ilyen eszköz.	SLDLL_SetLampa függvényt hangszóró típusú eszközre akartuk meghívni.
24 ERROR_BAD_LENGTH		Hanghossz nem [1;16] intervallumból kap egész értéket.	SLDLL_SetHang függvény rosszul lett felparaméterezve.
71 ERROR_REQ_NOT_ACCEP		Jelenleg fut már egy függvény végrehajtása.	SLDLL_Felmeres-t hívja meg, miközben az SLDLL_SetLista fut.
85 ERROR_ALREADY_ASSIGNED		Az új azonosító más panelt azonosít, ezeknek egyedieknek kell lenniük.	Nem találkoztam még ezzel a hibával.
110 ERROR_OPEN_FAILED		A megadott fájlt nem sikerült megnynítni.	Nem találkoztam még ezzel a hibával.
126 ERROR_MOD_NOT_FOUND		A megadott fájl nem a firmware frissítési adatait tartalmazza.	Nem találkoztam még ezzel a hibával.
1114 ERROR_DLL_INIT_FAILED		Az SLDLL_Open még nem lett meghívva.	Bármely SLDLL-függvényt úgy hívjuk meg, hogy előtte az SLDLL_Open nem futott le.
1247 ERROR_ALREADY_INITIALIZED		A függvény meghívása már megtörtént	SLDLL_Open függvény egymás után 2x való meghívása.
1626 ERROR_FUNCTION_NOT_CALLED		Nincs csatlakoztatott USB-eszköz.	SLDLL_Open függvény hívásakor nincsenek csatlakoztatva az eszközök.
egyéb Windows műveleti hibakódok			

1.9. ábra. Win32-hibakódok magyarázata

Saját szerkesztés

## 1.11. Az általam készített RelayDLL függvényeinek bemutatása

A RelayDLL állomány elkészülését Dr. Király Roland tanár úr javasolta, majd én készítettem el abból a célból, hogy az SLDLL hívható függvényeinek listáját bővítsük, a hívások módját egyszerűsítsük.

<sup>21</sup> Származtatható attribútumnak számít adatbázisok esetében például egy személy életkora, amennyiben a születésének dátuma (év, hónap és nap) eltárolásra kerül. Ha egy adat egy vagy több másikból egyenesen következik, tehát kiszámítható, abban az esetben azt az adatot felesleges letárolni.

### 1.11.1. Open

Az SLDLL\_Open függvény meghívásáért felelős, a híváshoz szükséges paraméterek számát egyre redukáltam, elegendő csupán az elkészült asztali alkalmazás Handle-címét átadni, hogy a Win32-üzenetküldés elkezdődhessen.

```
function Open(wndhnd:DWord): word; stdcall;
var
nevlei, devusb: pchar;
begin
result := SLDLL_Open(wndhnd, UZESAJ, @nevlei, @devusb);
end;
```

### 1.11.2. Listelem és Felmeres

A SLDLL\_Listelem és SLDLL\_Felmeres-függvények meghívása a feladatuk. Fontos, hogy a Listelem-függvényt is C#-ból hívassuk, mivel a Win32-üzenetek a Form részére kerülnek elküldésre, és a hívást egy beépített üzenetfeldolgozó metódus, a WndProc folyamán kell elvégeznünk.

```
function Listelem(var numberOfDevices: byte): word; stdcall;
begin
result := SLDLL_Listelem(@dev485);
drb485 := numberOfDevices;
devListSet := false;
end;

function Felmeres(): word; stdcall;
begin
result := SLDLL_Felmeres();
devListSet := false;
end;
```

### 1.11.3. ConvertDEV485ToJson

A metódus az SLDLL dev485-tömbjét C# által is értelmezhető JSON formátumú szöveggé alakítja (szerializálja).

Az eredmény egy referenciaiként átadott string paraméterből olvasható ki a függvény hívása után. Ezzel a függvénytel közli a Delphi a C# számára, hogy milyen eszközök kerültek eltárolásra a SLDLL\_Listelem folyamán.

```

function ConvertDEV485ToJSON(out outputStr: WideString): byte; stdcall;
var
buffer: WideString;
i: byte;
begin
if (drb485 = 0) or (not Assigned(dev485)) then
begin
showmessage(format('Dev485 is empty drb485 = %d dev485 = %p', [drb485, @dev485]));
outputStr := '[]';
result := DEV485_EMPTY;
exit;
end;
buffer := '[';
i := 0;
while i < drb485 do
begin
buffer := buffer + Format('{"azonos":%d}, ', [dev485[i].azonos]);
inc(i);
end;
buffer[length(buffer)] := ']';
showmessage(format('JSON-buffer = %s', [buffer]));
outputStr := buffer;
result := EXIT_SUCCESS;
end;

```

#### **1.11.4. ConvertDEV485ToJSON\_C**

A metódus nevéből is kitalálható, hogy ez pontosan azt a műveletet végzi el, amit az imént említett függvény: JSON-szerializációt. A szemléltetés kedvéért megcsináltam egy JSONParser hívásának másik lehetséges módját, hogy elismerjük, a Delphi és a C nyelv között is tudunk kapcsolatot teremteni DLL segítségével, mint azt alapvetően a Delphi és a C# között tettük.

Ehhez a relayDLL egy C nyelvű komponenst, az általam készített és lefordított converter32.dll állományt hívja segítségül. Az ebben használt JSON-keretrendszer Michael Grieco szoftverfejlesztőtől építette fel, én kizárolag ehhez egyetlen, külső hívást megengedő (exportált) függvényt készítettem úgy, hogy az hibamentesen hívható lehessen Delphiből.[8]

Természetesen a JSON-formátum kialakításához használhattam volna külső könyvtárakat is, mint például a SuperObject [9] vagy az LKJSON [10] de fontosabbnak tar-

tottam, hogy egyrészt kitaláljam ennek működését, másrészt megteremtsek esetlegesen egy újabb nyelvi kapcsolatot a C programozási nyelvvel.

### Hívott függvény: create\_json

Mivel a Delphi 7-es verziójába alapból nem lett beépítve a JSON-formátum kezelése, ezért segítségül hívunk C-ből egy olyan függvényt, ami azt a JSON-formátumot készíti el, amire nekünk szükségünk van:

```
[{"azonos":16385}, {"azonos":32770}, {"azonos":49153}, ...]
```

A C-nyelvű implementáció azonban hatékonyabban elvégzi ugyanezt a JSON-szerializációt.

### 1.11.5. ConvertDEV485ToXML

A dev485 tömbből egy XML formátumú szöveget készít, amelyet egy úgynévezett `devices.xml` állományban ment ki a háttértárra.

```
function ConvertDEV485ToXML(var outPath:WideString): byte; stdcall;
var
  xmlDoc : IXMLDOCUMENT;
  rootNode, node : IXMLNODE;
  i : byte;
begin
  if(drb485 = 0) or (not Assigned(dev485)) then
  begin
    result := DEV485_EMPTY;
    exit;
  end;
  xmlDoc := NewXMLDocument();
  xmlDoc.Encoding := 'utf-8';
  xmlDoc.Options := [doNodeAutoIndent];
  rootNode := xmlDoc.AddChild('devices');
  for i := 0 to drb485 - 1 do
  begin
    node := rootNode.AddChild('device');
    node.Attributes['azonos'] := dev485[i].azonos;
  end;
  xmlDoc.SaveToFile(outPath);
  showmessage('saved XML to location: ' + outPath);
  result := EXIT_SUCCESS;
end;
```

### **1.11.6. SetTurnForEachDeviceJSON**

Az USB-portról felmért eszközöket összerendeli a `devList` elemeivel azonosítójuk szerint, majd ennek a `devList` tömböt beállítja az eszközök beállításait leíró JSON-szerint. Ezt a függvényt meghívva, ha a JSON-formátum helyes, lényegében „távoli hívással” vezérelhetjük C# nyelvből az eszközökre kiküldött információkat: ilyen például lámpák esetében a színkód.

```
function SetTurnForEachDeviceJSON(var json_source: WideString):word; stdcall;
var
  i, j: byte;
  jsonArrayElements: TStringList;
  json_element1, json_element2: string;
  actDeviceType: string;
  actDeviceSettings: string;
begin
  jsonArrayElements := reduceJSONSourceToElements(json_source);
  i := 0; j := 0;
  while(j < drb485) do
  begin
    json_element1 := jsonArrayElements[i];
    json_element2 := jsonArrayElements[i+1];
    actDeviceType := extractValueFromJSONField(json_element1, 'type');
    actDeviceSettings := extractValueFromJSONField(json_element2, 'settings');
    if devListSet = false then
    begin
      devList[j].azonos := dev485[j].azonos;
    end;
    result := setDeviceByType(j, actDeviceType, actDeviceSettings);
    printErrors(result);
    inc(j);
    inc(i, 2);
  end; //case
  devListSet := true;
  result := SLDLL_SetLista(drb485, devList);
end;
```

### **1.11.7. fill\_devices\_list\_with\_devices**

Statikusan feltölti 3 eszközzel a `dev485` tömböt attól függetlenül, hogy van-e csatlakoztatva egyáltalán bármi eszköz a számítógéphez. Ezt csupán a többi függvény működé-

sének tesztelésére használtuk.

```
function fill_devices_list_with_devices(): byte; stdcall;
begin
if drb485 > 0 then
begin
result := DEV485_ALREADY_FILLED; //array is already filled
exit;
end;
drb485 := 3;
SetLength(dev485, drb485);
dev485[0].azonos := $c004; //hangszóró típusú eszköz
dev485[1].azonos := $8004; //nyíl típusú eszköz
dev485[2].azonos := $4004; //lámpa típusú eszköz
result := EXIT_SUCCESS;
end;
```

## 1.12. Az általam készített SLFormHelperDLL bemutatása

Az állomány a RelayDLL-en keresztül meghívható SLDLL-függvényekre épült, nagyon eltúlozva egy keretrendszert biztosít Levente grafikus felületéhez támogatólag. Az eszközök JSON- (vagy XML-) formátumban érkeznek, ezek beállításai szintén JSON-ben kerülnek átadásra a RelayDLL számára.

### 1.12.1. A WinForm bemutatása

A *WinForm*, teljes nevén a *Windows Forms* a .NET GUI-fejlesztést<sup>22</sup> támogató keretrendszer, amelynek segítségével egy asztali alkalmazást egyszerűbb módon el tudunk készíteni.

A weblapfejlesztésből már ismert, valamint ezek tárházát bővítő vezérlőelemekkel (gombok, legördülő listák, adattáblázatok, és így tovább) gondoskodik azok újrahasznosíthatóságáról. A Visual Studio fejlesztői környezet egy Designer-felülettel is rendelkezik, amellyel gyorsan és különösebb képzelőerő nélkül elkészíthetjük grafikus alkalmazásaink vázát.

---

<sup>22</sup> A GUI a **Graphical User Interface** (grafikus felhasználói interfész) kifejezés rövidítése, asztali alkalmazásokat értünk alatta, amely átlagfelhasználók számára kényelmesebb megközelítés a konzolos alkalmazásokkal szemben

Egy Windows Forms alkalmazásban a *Form* (továbbiakban ūrlapnak is fogom nevezni) egy vizuális felület, amely információkat jelenít meg a felhasználó számára. Egy Windows Forms alkalmazás általában úgy készíthető el, hogy egy Formhoz vezérlőelemeket (Control) adunk, a felhasználó által végrehajtott műveletekre, például egérkattintásokra és billentyűleütésekre adott válaszokat implementálunk. A *vezérlőelem* különálló GUI-elemek gyűjtőfogalma, amely adatokat jelenít meg vagy adatokat fogad bevitelre.

Amikor a felhasználó műveletet hajt végre egy ūrlapon vagy annak vezérlőelemein, ez a művelet eseményeket generál. Az alkalmazás kódban reagálhat ezekre az eseményekre, amennyiben azok bekövetkeznek, de figyelmen kívül is hagyhatja azokat.[20]

### 1.12.2. DLL-ek üzenetküldése Win32-ben

Egy üzenetlistán<sup>23</sup> keresztül kommunikál az operációs rendszer a futó programmal, ezt projektünk szempontjából Levente ablakos alkalmazása fogja jelenteni. Az operációs rendszer ráteszi a lenyomott gomb által kiváltott üzenetet erre a listára, tehát például amikor az egér bal gombjával kattintunk, akkor azt ténylegesen nem a futó program fogja észlelni, mivel az operációs rendszer a központ, ahova az I/O-kérések befutnak.

Az operációs rendszer eleve azért felelős, hogy elossza az erőforrásokat és kezelje a kimeneti-bemeneti perifériákat, így az egerünk által kiadott jel is az operációs rendszerhez érkezik be. Az alábbi lépéssorozat fog lejátszódni:

1. Az operációs rendszer ráteszi a WM\_LBUTTONDOWN-üzenetet az üzenetsorra.
2. A programunk meghívja a GetMessage-függvényt.
3. A GetMessage leveszi a WM\_LBUTTONDOWN-üzenetet az üzenetsorról, és az érkező információkból feltölti a Message-adatszerkezetet.
4. A programunk meghívja a TranslateMessage- és DispatchMessage-függvényeket, utóbbiban az operációs rendszer meghívja az asztali alkalmazás WndProc függvényét. Ez minden esetben lezajlik, attól függetlenül, hogy a függvény ki van fejtve vagy sem.
5. Az ablakos alkalmazásban válaszolunk az I/O-kérésre (például egy gombra kattintva újabb ablakot nyitunk meg) vagy éppen figyelmen kívül is hagyhatjuk, ekkor a felhasználó belátja, hogy lényegében nem is történt semmi.

Már az is Win32-üzenetet vált ki, ha szimplán mozgatjuk az egerünket, ekkor az egér új pozíciója is az üzenetben tárolásra kerül, innen és a Form előre meghatározott

---

<sup>23</sup> A Message Queue (üzenetsor) egy sor (queue) adatszerkezetben, érkezési sorrendben tárolja az üzeneteket, majd ezek ulyanezt a sorrendet megtartva kerülnek le róla.

tulajdonságaiból (ablak pozíciója, szélessége és magassága) tudjuk detektálni például, hogy az egér az ablak területére érkezett<sup>24</sup>.

Ha erre a felhasználói bemenetre fel van készítve a programunk által üzenetküldésre használt metódus<sup>25</sup>, akkor az érzékeli, hogy erre az eseményre reagálnia kell, így egy másik állapotba lép. Természetesen a készített programban lehetőségünk van arra is, hogy egyszerűen ignoráljuk az operációs rendszer felől érkező üzeneteket.

A felhasználó ebből az egész folyamatból csak annyit érzékelhet, hogy a lenyomott gomb hatására valami esemény történt a programban, így ő azt gondolhatja, hogy közvetlenül a program érzékelte az interakciót. Lényegében ez is történik, csak az operációs rendszer végzi az I/O-eszközökről érkező jelek feldolgozását, és erről egy Win32-üzenet formájában tájékoztatja az éppen futó asztali alkalmazást is.

## 1.13. Problémák és megoldások

Ebben az alfejezetben a következő, munkánkat időnként meg-megakasztó, kutatómunkát igénylő tényezőkről kívánok szót ejteni.

### 1.13.1. Egyénileg definiált típusok és struktúrák

**A probléma leírása:**

A Delphis projekt erősen függ az SLDLL-ben megkívánt típusoktól, ezért a teljes kódot nem tudjuk C#-ra átültetni.

**Megoldás:**

Erre a problémára megoldást nyújthat akár a marshalling, akár a szerializáció, mi a projektben utóbbi módszert választottuk az előző problémából levont következtetés miatt: ha stringet át tudunk adni, akkor bármilyen típust képesek vagyunk leírni szöveges formában, ezt átadva újból fel tudjuk építeni a másik nyelvben. A C#-os objektumok példányai egy erre dedikált forrásból (JSON-formátumú szövegből) fognak értékeket kapni.

A lényege a megoldásnak, hogy szerializálással küszöböljük ki a két nyelv közötti kompatibilitási hiányosságokat, ezért csak olyan adatokkal kommunikálunk, amelyek ismertek, feldolgozhatóak minden két nyelv számára. Ilyenek a primitívek, az egész, lebőrpontos, logikai és szöveges típusok).

---

<sup>24</sup> Erre vonatkozó esemény a MouseEnter-event Windows Forms esetében.

<sup>25</sup> Egy-egy Win32-üzenet feldolgozását Windows Form asztali alkalmazás esetében a WndProc-metódus szolgálja.

```
[  
  {"azonos" : 49156},  
  {"azonos" : 32772},  
  {"azonos" : 16388}  
]
```

1.10. ábra. Eszközök azonosítóinak átadása JSON-formátumon keresztül

### 1.13.2. JSON-formátumra konvertáló függvények hívása .NET keretrendszerből

#### A probléma leírása:

Az SLDLL funkcióit szeretnénk kibővíteni az XML- és JSON-szerializációt végző függvényekkel egy külön állományban. Ebben a felmért eszközök tömbjének ugyanúgy elérhetőnek kell lennie (dev485).

#### Megoldás:

A korábbi SLDLL-hez fűzzük őket, vagy tegyük egy új DLL-állományba őket, és ezeket is meg kell hirdetnünk a C#-os futtatókörnyezet számára is. A választott megoldáson az új DLL-re esett, egy úgynevezett `relayDLL` nevezetű állomány forráskódját megírtam Delphiben, amelyen keresztül az SLDLL-függvényeket meg tudjuk hívni.

Ennek legfőbb indoka az volt, ami eldöntötte, hogy érdemes a hívó (.NETben készült `FormHelperDLL`) és hívott (`SLDLL`) felek között egy köztes réteget elkészíteni, hogy Delphiben különböző globális változók kapnak értéket, amelyeket nem tudunk másolni definálni, csakis Delphiben, tehát a hívás környezetét így tudjuk megteremteni, hogy az értékadások ténylegesen működjenek, ne fussunk `NullReferenceException`-hibákra.

A `relayDLL` használatával ellenőrzöttebb módon tudom meghívni az SLDLL-ben meghirdetett függvényeket, ebből nem is kell minden meghirdetni, csak annyit, amennyit ténylegesen a vezérléshez használnunk szükséges. Ezenkívül több, általam definiált hibakódot is vissza tudok adni, amelyek kiváltó okáról C#-ban megfelelő kivételek segítségével tájékoztatni tudjuk a felhasználókat.

### 1.13.3. Az `SLDLL_Open`-függvény paramétereinek beállítása

#### A probléma leírása:

Az `SLDLL_Open`-függvény várja az ablakos alkalmazástól egy úgynevezett Handle-t, amely az üzenetküldést engedélyezi az SLDLL-függvények részéről.

**Megoldás:**

C#-ban a Formnak (a függvényeket meghívó félnek) létezik egy Handle nevezetű pointerre, amit átadhatunk az SLDLL-nek.

#### 1.13.4. Stringek átadása két nyelv között

**A probléma leírása:**

Különösen fontos, hogy Delphiből valamilyen formában át tudjunk adni a C# részére szöveg típusú változókat, mivel ha ez működik, akkor lényegében segítségükkel bármi-lyen más típus formátuma is – így a rekordoké is – leírható és feldolgozható.

**Megoldás:**

```
//converting dev485 to JSON-format - JSON-serializing
function ConvertDEV485ToJson(out outputStr: WideString): byte; stdcall; external RELAY_PATH;
//converting dev485 to XML-format - XML-serializing
function ConvertDEV485ToXML(var outputPath:WideString): byte; stdcall; external RELAY_PATH;
```

1.11. ábra. A függvény helyes Delphi-deklarációja

Stringek formájában alapértelmezetten a C# marshalling komponense Delphiből PWideChar típusú változókat vár. Amennyiben Delphiben a metódus WideStringet kér paraméterben, akkor azt a C#-oldali marshallerrel a [MarshalAs(UnmanagedType.BStr)] attribútum segítségével tudjuk közölni.[34]

```
[DllImport(DLLPATH, CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Unicode, EntryPoint = "ConvertDEV485ToXML")]
extern private static byte ConvertDeviceListToXML([MarshalAs(UnmanagedType.BStr)][In] ref string outputStr);

[DllImport(DLLPATH, CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Unicode, EntryPoint = "ConvertDEV485ToJson")]
extern private static byte ConvertDeviceListToJson([MarshalAs(UnmanagedType.BStr)][Out] out string outputStr);
```

1.12. ábra. A függvény helyes szignatúrája C#-ban

#### 1.13.5. A Delphiben eldobott kivételek nem kezelhetőek

**A probléma leírása:**

Delphiben dobott kivételeket a C# úgy érzékeli, mint hiba a külső komponensben, a tényleges kiváltó okát nem tudjuk meg.

**Megoldás:**

Hibakódokkal jelezzük a C# felé, ha esetleg hibára futott valamelyik függvény, ebből a hibakóból a C# tudni fogja, hogy milyen típusú kivételt dobjon az asztali alkalmazás számára, amelyek már lehetnek egyénileg elkészített Exception-objektumok.

### **1.13.6. uzfeld-metódus megfelelője C#-ban**

#### **A probléma leírása:**

Delphiben az ablakos alkalmazásnak volt egy úgynevezett `uzfeld` nevezetű metódusa, amely a DLL üzeneteinek kezelésére szolgált, ez hívódik meg az `SLDLL_Open` meghívása után<sup>26</sup> (ezért került átadásra a formot azonosító Handle). Ez azért fontos, mert az `SLDLL_Listelem`-függvény ezen az uzfeld-metóduson keresztül kerül meghívásra, a `drb485` és a `dev485` a program ezen pontján kapnak ténylegesen értelmezhető értékeket.

#### **Megoldás:**

Elméletem szerint létezik ennek egy szabványos megfelelője, C#-ban ez `WndProc`-metódus, ami szintén Message-típusú változót vár referencia szerint átadva, ezzel tud az `SLDLL` futásidőben kommunikálni. Innentől kezdve a következő lépések játszódnak le:

1. `SLDLL_Open` kivált egy üzenetet (Win32 Message).
2. Ezen az üzeneten keresztül az operációs rendszer meghívja a Form `WndProc` metódusát<sup>27</sup>.
3. Amennyiben a felmérés sikeresen lezajlott, a `WndProc` meghívja a `dev485` és `drb485` változókat beállító `SLDLL_ListElem`-függvényt, mielőtt az `SLDLL_Felmeres` re kerülne a sor. A megfelelő hívási szekvencia így automatikusan biztosított.

### **1.13.7. Eszközök azonosítóinak, típusainak átadása**

#### **A probléma leírása:**

Hogyan közöljük a felmért (USB-re csatlakoztatott) eszközök azonosítóját és típusát a C# részére?

#### **Megoldás:**

A Delphiben tárolt tömb (`dev485`) eszközeinek legfontosabb mezőjét (ami az eszközt azonosítja, tehát az azonosítót XML-fájlba kimentjük [35], vagy JSON-formátumú szöveggé alakítjuk. Az azonosító önmagában meghatározza az eszköz típusát, úgyhogy elég azt átadni. Az eszközök típusát a következő módon tudjuk meghatározni kizárálag beállított azonosítóikon keresztül:

<sup>26</sup> Ez a tény, hogy a `Listelem`-függvény tulajdonképpen a Felmeres előtt kell, hogy meghívódjon, a Delphi7-es fejlesztői környezetének debug és breakpoint funkcióinak segítségével derült ki. Ennek használatával sorról sorra tudtam követni a program végrehajtását.

<sup>27</sup> Ezt megteheti, mivel az Open hívásakor átadott Handle révén tudja, hogy merre továbbítsa az üzeneteket.

- SLLELO: \$4000 -> lámpa azonosítók \$4000 .. \$4fff (decimálisan: 16 384 .. 20 479) tartományon kaphatnak értéket.
- SLNELO: \$8000 -> nyíl azonosítók \$8000 .. \$8fff (decimálisan: 32 768 .. 38 683) tartományon kaphatnak értéket.
- SLHELO: \$c000 -> hangszóró azonosítók \$c000 .. \$cff (decimálisan: 49 152 .. 53 257) tartományon kaphatnak értéket.

Az eszköz típusának meghatározásában nem játszik szerepet az utolsó 3 bitnégyses, tulajdonképpen a legfelső hex<sup>28</sup> azonosítja a típust. Ha szigorúan hexadecimálisan nézük, akkor például ha egy szám négyjegyű és balról tekintve a legelső számjegye C, akkor az azonosító hangszórót fog jelölni.

Mivel az azonosítóból egy számítással kikövetkeztethető eszköz típusa, ezért elegendő a C# DLL-nek ezt az értéket átadni, amely értékből le tudja gyártani a maga példányait.

---

### 1. Algorithm Eszköz típusának meghatározása

---

```
function CREATEDEVICE(azonos: int)
    deviceType : int ← azonos & 0xc000;           ▷ &: bitenkénti AND-művelet
    if deviceType == 0x4000 then
        CreateDevice ← new LEDLight(azonos);
    else if deviceType == 0x8000 then
        CreateDevice ← new LEDArrow(azonos);
    else if deviceType == 0xc000 then
        CreateDevice ← new Speaker(azonos);
    else
        CreateDevice ← null;
    end if
end function
```

---

A megoldás a RESTful alkalmazások működéséből inspirálódik, amelyről a(z) 1.2 alfejezetben ejtettem szót, a C# és a Delphi között akár az XML, akár a JSON közös nyelvként (hídként) funkcionál. Ebben az analógiában a RelayDLL számít a szervernek, ennek függvényeit az SLFormHelper kliensként hívogatja.

### 1.13.8. A hangszóró egy egész hanglistát kezel

#### A probléma leírása:

A hangszórók nem feltétlenül csak egy bizonyos hang, hanem egy egész hangsorozat lejátszására is képesek. Egy hang lejátszásához 3 értéket kell tudnunk:

---

<sup>28</sup> 4 bit azonosít egy hexadecimális számjegyet – 0-tól F-ig terjedően –, így a bitnégystet hexnek is szokás nevezni.

1. index – sorszám: Ez egy egész szám, amely egy elemre hivatkozik a frekvenciákat tartalmazó tömbből, tehát az értéke  $[0; n - 1]$  intervallumból kerülhet ki, ahol  $n$ : a tömb hossza, esetünkben  $n = 50$ . 1.14a Ennek segítségével megadhatjuk, hogy milyen magas hangot szeretnénk a hangszóró részére lejátszásra kiküldeni.
2. volume – hangerő: Ez egy egész szám, amely 0 és 63 között vehet fel értéket.
3. length – időtartam: Szintén egész szám, amely 0 és 10000 között enged értéket beállítani az általam készített SLFormHelper DLL-ben. Ezzel megadhatjuk, hogy a hangszóró a lejátszáshoz mennyi időt vegyen igénybe, ez milliszekundumban, azaz ezredmásodperc egységen értendő.

### Megoldás - JSON-formátumban

Egy hanglistát több *index/volume/length*-hármas ír le, a pipe, azaz '|'-karakter mentén széttördelt szövegen 3 lépésközzel iterálhatunk végig.

```

k := 0;
j := 0;
while j < elements.Count - 2 do
begin
  H[k].hangso := strToIntDef(elements[j], 0); //5 index
  H[k].hanger := strToIntDef(elements[j+1], 0); //63 volume
  H[k].hangho := strToIntDef(elements[j+2], 0); //1000 length
  inc(k);
  inc(j, 3);
end;

```

1.13. ábra. Megoldás a problémára - RelayDLL

### Megoldás - kódban

A Speaker-osztályban az eszköz viselkedését leíró 3 mezőt (index, volume és length) egy külön objektumba szervezzük ki (Sound), és a Speaker – azonosítóján kívül – kizárolag Sound-típusú objektumok listáját tartalmazza, a listát JSON-ben "*index/volume/length*" rendezett hármasok összefűzésével szerializálja, a listához kizárolag listakezelő függvényekkel lehet hozzáérni (törles, módosítás, hozzáadás), majd az SLDLL\_Hangkuldes számára kiküldött H-tömböt (hanglistát) Delphiben egy ciklussal a JSON-nel átadott adatok szerint a megfelelő sorrendben feltölthjük, az előző probléma megoldásában ennek mikéntjét kifejtettem.

Hangmagasság	Frekvencia (Hz)	Hangmagasság	Frekvencia (Hz)	Hangmagasság	Frekvencia (Hz)
C'''	4186.0090	G''	1567.9817	D'	587.3295
H'''	3951.0664	FISZ''	1479.9777	CISZ'	554.3653
B'''	3729.3101	F''	1396.9129	C'	523.2511
A'''	3520.0000	E''	1318.5102	H	493.8833
GISZ'''	3322.4376	DISZ''	1244.5079	B	466.1638
G'''	3135.9635	D''	1174.6591	A	440.0000
FISZ'''	2959.9554	CISZ''	1108.7305	GISZ	415.3047
F'''	2793.8259	C''	1046.5023	G	391.9954
E'''	2637.0205	H'	987.7666	FISZ	369.9944
DISZ'''	2489.0159	B'	932.3275	F	349.2282
D'''	2349.3181	A'	880.0000	E	329.6276
CISZ'''	2217.4610	GISZ'	830.6094	DISZ	311.1270
C'''	2093.0045	G'	783.9909	D	293.6648
H''	1975.5332	FISZ'	739.9888	CISZ	277.1826
B''	1864.6550	F'	698.4565	C	261.6256
A''	1760.0000	E'	659.2551	33	szünet
GISZ''	1661.2188	DISZ'	622.2540		

1.14. ábra. Hangszóróval lejátszható hangok listája frekvenciákra bontva

Forrás:

táblázat: saját szerkesztés,

adatok: SLDLL-állomány

```
public partial class Speaker : Device
{
    private readonly List<Sound> soundList;
    public Speaker(uint azonos) : base(azonos) {
        this.soundList = new List<Sound>();
    }
    public void AddSound(Pitch pitch, byte volume, ushort length)
    {
        if (soundList.Count > 30)
            throw new Exception("A lejátszható hangok listája megtelt.");
        Sound soundToAdd;
        try
        {
            soundToAdd = new Sound((byte)pitch, volume, length);
            this.soundList.Add(soundToAdd);
        }
        catch(ArgumentOutOfRangeException ex)
        {
            Console.WriteLine($"Nem adom hozzá a hanglistához a következő miatt: {ex.Message}.");
        }
    }
    public void ClearSounds()
    {
        soundList.Clear();
    }
}
```

1.15. ábra. A hangszóró hanglistát kezel - megoldás C# nyelven

### 1.13.9. Több csatlakoztatott eszköz felismerése, vezérlése

A probléma leírása:

A rajzon látható módon kötöttük be az eszközöket, és háromból 2 eszközöt érzékelt a dev485 tömbben, valamint csak egy, az USB portra közvetlenül csatlakozó (legelső) elemnek küldte ki a jeleket, a többi egyszerűen „elnyelődött”.

Amikor 2 eszközöt, egy lámpát és egy nyilat csatlakoztattam, akkor egyrészt a telefonkábel folyamatosan, érezhetően melegedett, másrészt

- a) vagy állandóan piros színnel világított,
- b) vagy piros - zöld - kék jelzésekkel követően kikapcsolt



1.16. ábra. Két eszköz csatlakoztatásának módja.

Forrás: saját kézi rajzom.

Az első esetben szimplán nem érzékelte az USB-porton lévő eszközöket – ez az 1626-os hiba az `SLDLL_Open` függvény lefutásakor –, az alábbi hibakódot dobja az eszközök felmérésekor.

### Megoldás:

Az általunk használt telefonkábelek két végén a középső erek bekötése egymással el-lentétes volt, továbbá a két szélső, a sárga és fekete erekben átfolyó stabil egyenáram zajként szolgált, egyúttal ez okozta a kábel folyamatos melegedését is. Elegendő volt csupán a két középső, a piros és zöld ereket rendre – balról jobbra – bekötni, majd egy krimpelő fogó segítségével az RJ10-es csatlakozóhoz erősíteni. Az ily módon elkészült kábel alkalmassá teszi az eszközöket a soros kommunikációra.

# Összegzés és kitekintés

Két ismert programozási nyelv között megteremthető kommunikáció kiaknázásával elértem, hogy egy olyan szoftver készülhessen el, amelynek segítségével szakemberek képessé válhatnak emberek mentális egészségének fejlesztésére.

Örömömben szolgált, hogy Somodi László minket bízott meg elméleteit kivitelező szoftveres háttérrel. Szakdolgozatunk nem kizárolag arról szól, hogy szakmai tudásunkat gyarapítuk és mutassuk bizonyítvánnyként egyetemi oktatóink felé, hanem ezzel a projekttel sok ember számára tudunk további segítséget nyújtani magunk legjobb tudása szerint.

Kifejezetten tetszik, hogy munkánkat több tudományág szakértője segítette, így kutatásunk interdiszciplinárisként is jellemzhető, azaz több szakterület, jelesül a mozgás-koordináció, beágyazott rendszerek vezérlése, együttes munkájának az eredményének részesei lehettünk.

A szoftverfejlesztés elméleti és gyakorlati alapjait egyrészt az Egyetem oktatóitól, szakmai és technikai vonulatait Dr. Király Roland Tanár Úrral folytatott konzultációinkon szerzett információk biztosították.

Inspirációkat, motivációkat Somodi Lászlóval való beszélgetéseinkből, valamint „Mozgáskoordináció- és gyorsaságfejlesztő gyakorlatok óvodától a felnőtt korig” címet viselő könyvében leírt gondolatokból merítettük.

Mint minden szoftverre és emberi termékre jellemző, a megoldásaink természetesen nem mondhatóak tökéletesnek, programunk épp annyira időnként karbantartásra és további fejlesztésekre szorulhat. Érdekes volt felfedezni a gRPC és a Proto-nyelv kapcsán a 1.3 fejezetben, hogy ötször-hatszor jobb teljesítmény érhető el az adatok bináris módon történő szerializációjával a szöveges formátumokhoz képest – jelen állás szerint mégis az utóbbi megoldás számít elterjedtebbnek –, én azt gondolom, hogy minden-képp érdemes lenne munkánkat esetleg egy másik szakdolgozat keretein belül a gRPC irányába elmozdítani.

# Köszönetnyilvánítás

Szeretném ezúton is megköszönni konzulensemnek, Dr. Király Roland Tanár Úrnak a tanulmányaim során nyújtott támogatását és szakmai útmutatását. Szakértelme és tapasztaltsága segített engem abban, hogy munkámat megérthessem, elkezdhessem, folytathassam, végül sikeresen lezárhassam.

Külön köszönet illeti a barátnőmet, aki ezt a dolgozatot gondosan elolvasta, valamint a dokumentumban fellelhető hibákról tájékoztatott engem hozzájárulva ahhoz, hogy szakdolgozatom időről időre minél jobb lehessen.

Szeretnék köszönetet mondani barátaimnak és családomnak is a szeretetükért, türelmükért és támogatásukért.

Köszönöm a tanulmányomban részt vállaló valamennyi oktatónak az idejüket és igyekezetüket, hogy megosszák tudásukat velünk.

Végezetül szeretnék hálát adni a jó Istennek, hogy vigyázott rám, így testi, szellemi és lelke épségen el tudtam végezni ezt a munkát is. Belé vetett hitem segített abban, hogy szakdolgozatomat minden akadályozó tényező ellenére is folytatni tudjam.

# Irodalomjegyzék

- [1] Srikanth Chakilam. Comprehensive performance bench-marking experiment to compare REST + JSON with gRPC +Protobuf. *LinkedIn*, 2020-03-10.  
<https://tinyurl.com/3htxsp5n>.
- [2] Cleveland Clinic. Study suggests mechanistic overlap between alzheimer's and covid-19. *Cleveland Clinic*, 2021-06-16.  
<https://tinyurl.com/yckjs8tx>.
- [3] V2 Cloud. Client-server application.  
<https://v2cloud.com/glossary/client-server-application>, ismeretlen.
- [4] edward (username). What is serialization?  
<https://stackoverflow.com/questions/633402/what-is-serialization>, 2009.
- [5] Egészségvonal. Demencia típusai.  
<https://egeszsegvonal.gov.hu/d/1541-a-demencia-tipusai.html>, utolsó módosítás: 2022-02-18.
- [6] ELTE. Delphi programozási nyelv. <http://nyelvek.inf.elte.hu/leirasok/Delphi/index.php>, utolsó módosítás: 2014-06-04.
- [7] Geeks For Geeks. What is just-in-time (jit) compiler in .net.  
<https://tinyurl.com/yexwvpra>, utolsó módosítás: 2021-02-03.
- [8] Michael Grieco. Programming challenges - 28.1 - json data (c).  
<https://www.youtube.com/watch?v=mnIvV-IBI7Y>, utolsó elérés: 2021-04-01.
- [9] Michael Grieco. Programming challenges - 28.1 - json data (c).  
<https://github.com/ekapujiw2002/delphi7-json-parser-superobject>, utolsó elérés: 2023-04-04.
- [10] Michael Grieco. Programming challenges - 28.1 - json data (c).  
<https://sourceforge.net/projects/lkjson/>, utolsó elérés: 2023-04-04.

- [11] Jeremy Hillpot. grpc vs. rest: How does grpc compare with traditional rest apis? *DreamFactory*, 2022-11-11.  
<https://tinyurl.com/2p8w28s3>.
- [12] Ben Joan. Difference between ansi and unicode.  
<http://www.differencebetween.net/technology/software-technology/difference-between-ansi-and-unicode/>, utolsó elérés: 2023-02-23.
- [13] Jonathan Johnson. Asynchronous programming: A beginner's guide.  
<https://www.bmc.com/blogs/asynchronous-programming/>, 2020.
- [14] Microsoft Learn. Interoperating between native code and managed code.  
[https://learn.microsoft.com/en-us/previous-versions/visualstudio/windows-sdk/ms717435\(v=vs.100\)?redirectedfrom=MSDN](https://learn.microsoft.com/en-us/previous-versions/visualstudio/windows-sdk/ms717435(v=vs.100)?redirectedfrom=MSDN), 2013-02-04.
- [15] Microsoft Learn. Default marshalling for strings.  
<https://learn.microsoft.com/en-us/dotnet/framework/interop/default-marshalling-for-strings>, 2022-05-20.
- [16] Microsoft Learn. Common language runtime (clr) overview.  
<https://learn.microsoft.com/en-us/dotnet/standard/clr>, utolsó elérés: 2023-02-17.
- [17] Yen Nee Lee. Covid could trigger a spike in dementia cases, say alzheimer's experts. *CNBC*, 2022-09-17. <https://tinyurl.com/ynm3cecx>.
- [18] ParTech Media. Managed and unmanaged code: Key differences. *ParTech*, 2022-09-17.  
<https://www.partech.nl/en/publications/2021/03/managed-and-unmanaged-code---key-differences#>.
- [19] Microsoft. Win32 error codes.  
[https://learn.microsoft.com/en-us/openspecs/windows\\_protocols/ms-erref/18d8fbe8-a967-4f1c-ae50-99ca8e491d2d](https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/18d8fbe8-a967-4f1c-ae50-99ca8e491d2d), 2021.
- [20] Microsoft. Desktop guide (windows forms .net). *Microsoft Learn*, 2022.  
<https://learn.microsoft.com/en-us/dotnet/desktop/winforms/overview/?view=netdesktop-6.0>.
- [21] Microsoft. Default marshalling behavior. *Microsoft Learn*, 2022-09-17.  
<https://learn.microsoft.com/en-us/dotnet/framework/interop/default-marshalling-behavior#memory-management-with-the-interop-marshaler>.

- [22] David Nield. The brains of teenagers look disturbingly different after lockdown. *Science Alert*, 2022.  
<https://tinyurl.com/29m7r88a>.
- [23] ELTE Prognyelvek portál. Linq lekérdezések.  
<http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=25>, utolsó módosítás: 2010.
- [24] ELTE Prognyelvek portál. Marshalling használata csharp alatt.  
[http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=6#section\\_3](http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=6#section_3), utolsó módosítás: 2010.
- [25] RedHat. What is an api?  
<https://tinyurl.com/m5n7wfyp>, utolsó módosítás: 2022-06-02.
- [26] Tori Rodriguez. Impact of the covid-19 pandemic on people living with dementia and caregivers. *Neurology Advisor*, 2022-07-08.  
<https://www.neurologyadvisor.com/topics/alzheimers-disease-and-dementia/impact-covid-19-pandemic-people-living-with-dementia-caregivers/>.
- [27] Dr. Király Roland. Formális nyelvek és automaták jegyzet.  
[https://aries.ektf.hu/~serial/kiralyroland/download/formalis\\_nyelvek\\_es\\_automatak\\_KR\\_TAMOP\\_20121116.pdf](https://aries.ektf.hu/~serial/kiralyroland/download/formalis_nyelvek_es_automatak_KR_TAMOP_20121116.pdf), 2012.
- [28] Isaac Computer Science. The call stack and stack frames.  
[https://isaaccomputerscience.org/concepts/prog\\_sub\\_stack?examBoard=all&stage=all](https://isaaccomputerscience.org/concepts/prog_sub_stack?examBoard=all&stage=all), utolsó elérés: 2023-02-23.
- [29] Jacob Sorber. When do i use a union in c or c++, instead of a struct?  
[https://www.youtube.com/watch?v=b9\\_0bqrm2G8](https://www.youtube.com/watch?v=b9_0bqrm2G8), utolsó elérés: 2021-06-09.
- [30] Dr. Király Sándor. Szolgáltatásorientált programozás (régi név: Az internet eszközei) elektronikus tananyag. 2. Streamalapú kommunikáció,  
<https://elearning.uni-eszterhazy.hu/course/view.php?id=1128>.
- [31] Wim ten Brink. Which (in general) has better performance csharp or delphi?  
<https://www.quora.com/Which-in-general-has-better-performance-C-or-delphi>, 2019.
- [32] TIBCO. What is batch processing?  
<https://www.tibco.com/reference-center/what-is-batch-processing>, utolsó elérés: 2023-02-11.
- [33] ultraware (username). Delphigrpc github repository, 2018-10-17.  
<https://github.com/ultraware/DelphiGrpc>.

- [34] user1635508 (username). Passing string from delphi to csharp returns null.  
[https://stackoverflow.com/questions/53529623/  
passing-string-from-delphi-to-c-sharp-returns-null-however-\  
it-works-fine-when-i](https://stackoverflow.com/questions/53529623/passing-string-from-delphi-to-c-sharp-returns-null-however-it-works-fine-when-i), 2019.
- [35] Abdul Salam (username). how to create xml file in delphi.  
[https://stackoverflow.com/questions/8354658/  
how-to-create-xml-file-in-delphi](https://stackoverflow.com/questions/8354658/how-to-create-xml-file-in-delphi), 2012.
- [36] Brian G (username). What is object marshalling?  
[https://stackoverflow.com/questions/154185/  
what-is-object-marshalling](https://stackoverflow.com/questions/154185/what-is-object-marshalling), 2008-09-30.
- [37] Peter (username). What is the difference between serialization and marshaling?  
[https://stackoverflow.com/questions/770474/  
what-is-the-difference-between-serialization-and-marshaling/](https://stackoverflow.com/questions/770474/what-is-the-difference-between-serialization-and-marshaling)  
770509#770509, 2009.
- [38] Pokus (username). What is managed or unmanaged code in programming?  
[https://stackoverflow.com/questions/334326/  
what-is-managed-or-unmanaged-code-in-programming](https://stackoverflow.com/questions/334326/what-is-managed-or-unmanaged-code-in-programming), 2008.
- [39] Ruurd (username). PInvoke: beyond the magic. *devops.lol*, 2014-03-16.  
<https://www.devops.lol/pinvoke-beyond-the-magic/>.
- [40] Tracing (username). Object pascal vs delphi?  
[https://stackoverflow.com/questions/15699788/  
object-pascal-vs-delphi](https://stackoverflow.com/questions/15699788/object-pascal-vs-delphi), 2012.
- [41] FreePascal Wiki. Pascal for csharp users.  
[https://wiki.freepascal.org/Pascal\\_for\\_CSharp\\_users#Types\\_  
Comparison](https://wiki.freepascal.org/Pascal_for_CSharp_users#Types_Comparison), 2023-02-08.

# Ábrák jegyzéke

1.	Valamely demenciafajtával diagnosztizáltak számának emelkedése . . . . .	1
2.	Példa két metódusra, amelyben stringeket használunk. . . . .	5
3.	Példa egy rekord definíójára . . . . .	6
4.	Példa egy union típus definíójára. Forrás: SLDLL . . . . .	6
5.	Példa egy külsős, azaz PInvoke-eljárás definíójára és használatára . . . . .	9
1.1.	Beadandó projektben készített .proto állomány (részlet) . . . . .	14
1.2.	gRPC metódushívásainak típusai . . . . .	16
1.3.	Szerializáció teljesítményének összehasonlítása JSON és Proto esetében	16
1.4.	Lámpa eszköz . . . . .	17
1.5.	Nyíl eszköz . . . . .	17
1.6.	RJ9-es csatlakozó vezetékes telefonkábel végén. Csak a középső 2 ér van csatlakoztatva. . . . .	18
1.7.	nyomatókábel USB-A és USB-B végekkel, előbbi a számítógéphez, míg utóbbi az elsőnek szánt eszközhöz csatlakoztatandó. . . . .	18
1.8.	Tápkábelt használunk a további, számítógéphez közvetlenül nem csatlakozó eszközök feszültségre kapcsolásához. . . . .	19
1.9.	Hibakódok és magyarázataik . . . . .	27
1.10.	Eszközök azonosítóinak átadása JSON-formátumon keresztül . . . . .	35
1.11.	A függvény helyes Delphi-deklarációja . . . . .	36
1.12.	A függvény helyes szignatúrája C#-ban . . . . .	36
1.13.	Megoldás a problémára - RelayDLL . . . . .	39
1.14.	Hangszóróval lejátszható hangok listája frekvenciákra bontva . . . . .	40
1.15.	A hangszóró hanglistát kezel - megoldás C# nyelven . . . . .	40
1.16.	Két eszköz csatlakoztatásának módja. . . . .	41