Matematikai és Informatikai Intézet

Interfész megoldások imperatív és objektumorientált programozási nyelvek közötti kapcsolattartásra

Készítette
Nagy-Tóth Bence
Szak: Programtervező informatikus
Specializáció: Szoftverfejlesztő informatikus

Témavezető
Dr. Király Roland
egyetemi docens

Eger, 2023

Tartalomjegyzék

Kivonat

Szakdolgozatomban kifejtem az *interlingvális*[1] kommunikáció lehetséges megvalósítási módjait, amelynek legfontosabb mérföldköve a szöveges adatok valamilyen szabvány szerinti közlése, tehát a szerializáció lesz, amelynek segítségével képessé válhatunk bármilyen típusú adatot – legyen az szám, logikai változó, egy teljes, több attribútumból felépülő objektum – ilyen formában a másik program részére közölni. Miután ez a szabványos formátum elkészült, az a feladatunk, hogy a másik oldalt is felkészítsük a közlések fogadására és feldolgozására. Ennek gyakorlati haszna bemutatásra kerül egy, a 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[2], mint később látni fogjuk, nem lesz elegendő, szükség volt egy harmadik, úgynevezett RelayDLL-re, amely áthidalja a két nyelv sajátosságait, és ténylegesen olyan elemeket használ, amelyet mindkét nyelv hasznosítani tud a maga számára. Ezt én készítettem abból a célból, hogy az adatok cseréje tulajdonképpen ezen a „hídon" keresztül, kezelhetőbb módon jöhessen létre a Delphi és a C# programozási nyelvek között.

Kutatásom utolsó fázisában tettem egy kitérőt a C programozási nyelv irányában is, a C-forráskódból elkészült harmadik, `converter32` alkalmasnak bizonyult a Delphiben készített `relayDLL` állománnyal való együttműködésre, így a Delphi és a C programozási nyelvek között is sikerült egy függvényhívás erejéig kapcsolatot teremtenem.

A szoftverek felújításának problémája nem ritkán előfordulhat más kontextusban is, ugyanis vannak vállalkozások, amelyek nem merik leváltani jól bevált, régi (legacy) rendszereiket, mivel egyszerűen túl sok időbe és pénzbe kerülne azokat új köntösbe bújtatni, és felmerülhet, hogy esetleg nem sikerül minden funkciót teljesen, maradéktalanul átültetni egy fejlettebb programozási nyelvre. Nem beszélve arról, hogy az új rendszerre való áttérés során fennállhat az adatvesztés kockázata is. Ezek kiküszöbölésére érdemes lenne – mint a számítógépes hálózatok esetében is – szabványos formátumokat használnunk az adatok tárolására, szállítására és helyreállítására. Ha mégis erre adná a fejét egy szoftverfejlesztő csapat, akkor annak általában az a lefutása, hogy a nulláról elkezdik a választott új nyelvvel leírni a régi rendszert. A megoldás kulcsa jelen álláspontom

[1] Az interlingua kifejezés a latin *inter* (között) és *lingua* (nyelv) szavak összetételéből adódik. Jelentése „nyelvek között".

[2] 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

szerint a szerializáció lehetne, amely lehetővé tenné, hogy szabványos formátumokon dolgozó régi egységeket az új környezetből is meghívhassuk, jelentősen csökkentve az új rendszer implementációjára fordított időt.

1.0. Bevezetés

A kognitív képességek hozzájárulnak elménkben lezajló folyamatok jelentős részéhez, ezért azok fejlesztésére érdemes odafigyelnünk életünk során. Ezen megállapításom különösen beigazolódni egy COVID-világjárvány utáni időszak tendenciáinak tükrében. A járványra reagáló korlátozó intézkedések következtében nőtt a különböző mentális betegségek kialakulásának kockázata. [18] A *demencia* gyűjtőfogalomként alkalmas ezen betegségek együttes megnevezésére.[3]

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 beszédzavarok, az ítélőképesség, a memorizálás és az elvonatkoztatás képességeinek romlása.

Az értelmi hanyatlást különféle egészségkárosító szokások – például az alkoholfogyasztás – szervi károsodás, akár ezek együttese is okozhatja. Sajnos, az ember életkorának előrehaladtával szintén egyre nő a kockázata a demencia kialakulásának.

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 romlását. 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 megfékezésének ezen útját választottuk.

Szakdolgozatom ezen pontján azért érdemes erről a betegségről szót ejtenem, mivel alapvetően számomra már önmagában motivációul szolgál a tudat, hogy egy ilyen nagy volumenű elmélet megvalósításában vehetek részt, amely a demencia megelőzését, a mentális állapot folyamatos rongálódásának megfékezését célozza. [15] [2] [22]

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.

[3] A demencia említésekor nem egyetlen betegségre, hanem több hasonló jellegű problémát csoportosító fogalomra, egy tünetegyüttesre gondolunk. Száznál is több típusát ismerik a demenciának, amelyek közül az Alzheimer-kór a leggyakoribb. [5]

Az imperatív[4] programozási paradigma 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 objektumorientált programozási (röviden: OOP) paradigma igyekszik a világot lemodellezni olyan formában, hogy a modellt objektumok használatával írja le. Az objektumelvű programozás alaptételei:

- **egységbe zárás**: Az OOP egyik alaptézise, hogy az adatszerkezet és a rajtuk végzett műveletek egy egységben, az osztályban összpontosulnak. Ezt másképp úgy is ki lehet fejezni, hogy egy objektumhoz szorosan, logikailag hozzátartozik annak belső állapota – ezt mezőnek hívjuk – és viselkedése (a metódusok). Az osztály tagjait elláthatjuk láthatósági szintekkel, ezek közül a private szint kizárólag osztályon belüli láthatóságot biztosít, ekkor például egy `LEDLight`-típusú objektumhoz tartozó szín mező értékéhez kívülről, közvetlenül nem lehet hozzáférni, csakis az osztályon belül használható. Ha viszont minden csak private szinttel lenne ellátva, nem tudnánk az objektumokat érdemben használni, mivel az osztályok mezőit nem tudnánk sem írni sem olvasni. Ennek kiküszöbölésére publikus szintű metódusokkal engedélyt adhatunk a objektum adattagjaihoz való hozzáférésre, ezzel ellenőrzéseket, validációkat bevezetve eldönthetjük, hogy a hozzáférést valóban megadjuk-e.

- **öröklődés**: Az objektumok gyakran bizonyos tulajdonságaikban hasonlítanak egymásra. Például van egy lámpa eszköz vezérlésére szolgáló `LEDLight`-osztály, amelynek egyetlen példányát leírja annak színe és azonosítója, viszont ezután szeretnénk egy másik osztályt is, amely szintén ezekkel az attribútumokkal rendelkezne, egy ráadással: irányt is szeretnénk hozzárendelni az eszköz beállításaihoz. Látjuk, hogy 3 tulajdonságból 2 megegyezne, ezért az öröklődést választjuk: a megegyező mezőket szervezhetjük úgy, hogy ezek egy úgynevezett szülőosztály részét képezzék (jelen esetben ez a meglévő `LEDLight`-osztály), majd ebből származtatunk ennek egy gyermekosztályát, ami `LEDArrow` névre hallgat. Így a két osztály között kialakult egy öröklődési kapcsolat, a `LEDArrow` kiterjeszti[5] szülője, a `LEDLight` tulajdonságait. Általánosságban elmondható, hogy ennek segítségével a gyermekosztály mindig csak annyit tesz hozzá, amennyi tőle szükséges, miközben megörökli a szülőosztályának logikáját. A gyermekosztály újra felhasználja

[4] Az *imperare* latin szóból eredő szó, jelentése: parancsol

[5] A Java-nyelv szó szerint az „extends” kulcsszóval hangsúlyozza, hogy a gyermekosztály kibővíti szülője funkcionalitását.

a szülőosztály összes mezőjét és metódusát (közös rész), és megvalósíthatja a sajátját (egyedi rész).

- **absztrakció**: Ami a felhasználó kód számára nem lényeges információ, azt elrejtjük. Az egységbe zárás következtében az osztályban védettségi szinteket tudunk beállítani annak mezőire és metódusaira. a protected az öröklődési láncban gyermekek és szülőosztályok számára engedélyezi a láthatóságot, míg a public láthatósági szinttel biztosítjuk, hogy az adott osztály tagja kívülről is elérhető, hivatkozható. Például szolgálhat a saját autónk: nem feltétlenül kell értenünk, hogy a háttérben pontosan mi zajlik le, az autó mely alkotóelemei működnek összehangoltan, amikor azt működtetjük. Ezek a részletek az autóvezetők elöl elrejthetőek, mivel ezekről nem kell tudnunk ahhoz, hogy vezetni tudjuk járműveinket. A későbbiekben említeni fogom a szerver-kliens architektúrán alapuló alkalmazásokat, amelyben általában metódusok fejlécét, meghívási módját ismerheti a kliens, viszont azt, hogy pontosan milyen módon végzi a szerver a számításokat, az a kliens számára nem lényeges, így el van rejtve, azok eredménye számít igazából.

- **polimorfizmus**: A polimorfizmus görög eredetű szó, többalakúságot jelent. Van egy metódusunk, amelyet a szülő osztályra implementáltunk - de szeretnénk használni a gyermekosztályokra is. Az általunk készített programból kiindulva ilyen a `LEDLight GetDeviceSettings`-függvénye. Azt szeretnénk elérni, hogy a `LEDLight` beállításai mellett jelenjen meg a `LEDArrow` objektumokban eltárolt irány is valamilyen formában, hogy JSON segítségével ezt is át tudjuk adni a vezérlőfüggvénynek. Ezt polimorfizmus használatával oldjuk meg, ez lehetőséget teremt arra, hogy egy gyermekosztályt pontosan úgy használjunk, mint a szülőjét, így nem lesz zavar a típusok keveredésével.[8]

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, relációkat teremthetünk: öröklődést vagy birtoklást. Azon nyelvet, amelyek ezt a paradigmát támogatják, objektumorientáltnak minősítjük. A fentebb említett nyelvek ezt a paradigmát is támogatják a C nyelv kivételével.

1.1. Kommunikáció adatszerkezeteken keresztül

A *Marshalling* egy olyan folyamat, amely összetett típusok átalakítására szolgál, hogy egy nyelv adatszerkezetét a másikkal megértessük. 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.[30]

A kód további portolhatósága, újrahasznosíthatósága végett megéri szabványos formátumokon keresztül kommunikálni, mint azt tesszük API-k[6] vagy konfigurációs fájlok esetében, ilyenek például az általam használt JSON- vagy XML-formátumok, hogy más felületekre való költöztetés 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 szerializáció és deszerializáció folyamataiba is betekintést nyerjünk.

1.2. Szerializáció

A szerializáció az a folyamat, amikor valamilyen adatszerkezetet bájtok folyamára[7] alakítunk, hogy. Mivel az adatok valamilyen elgondolás szerint bitekként vannak kódolva a memóriában, ezért szükségünk van a szerializáció inverz műveletére[8]: ez a deszerializáció, 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) szerializálására. 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 1.10.0 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 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]

[6] 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 miként tjéről az API írója gondoskodik. Másként fogalmazva: az API a szerver azon része, amely felelős a kérések feldolgozásáért és kiszolgálásáért. Forrás: [21]

[7] A bájtfolyam angolul byte stream néven ismert.

[8] Az eredeti művelet hatását visszafordító, visszavonó.

1.3. Marshalling és szerializáció

A Marshalling egy olyan folyamat, amely hidat teremt a felügyelt és felügyelet nélküli kódok között, lásd: 1.5. 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. [31]

1.4. Adatok konvertálása

1.4.0. 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`. [20]

1.4.1. 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 ékezetes 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 értelmezhetetlenné válnak, és bizonyos esetekben programhibához is vezethetnek. Használatuk ebből következően jelenleg már nem javasolt. [11]

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 felügyelet nélküli (unmanaged) kód esetében így működik: egy úgynevezett `MarshalAsAttribute`-attribútumot fűzhetünk a string paraméterek, valamint visszatérési értékek elé, hogy felkészítsük a marshallert, hogy pontosan milyen típusú stringet kell várnia. Több `UnmanagedType`-érték is megadható a karakterláncok fajtájától függően.[14]
Ezek közül hármat mindenképp érdemes megemlíteni:

- `UnmanagedType.BStr` Előre meghatározott hosszúságú, Unicode-karakterkódolású szöveg található a túloldalon. Delphiben ez a WideString típusnak felel meg.
- `UnmanagedType.LPStr` (alapértelmezett) Egy pointer ANSI-karakterekből álló tömbre, amelyet a nullkarakter (`\0`) zár. Delphiben ez a PAnsiChar típusnak felel meg.
- `UnmanagedType.LPWStr` Egy pointer Unicode-karakterekből álló tömbre, amelyet szintén a nullkarakter zár. Delphiben ez a PWideChar típusnak felel meg.

Ennek gyakorlati alkalmazása C#-ban megfigyelhető a(z) 1.20 ábrán.

1.4.2. 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 kódrészleten keresztül szemléltetem, hogy hogyan is néznek ki a gyakorlatban az összetett típusok. Az általam létrehozott `ListaElem` nevezetű struktúra két mezőt tartalmaz: egy primitívet[9], valamint egy összetett típust, ami itt történetesen szintén `ListaElem` típusú, de bármilyen más rekordtípust tartalmazhatna.

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

1.1. ábra. Példa egy rekord definíciójára

Lévén összetett típusról van szó, gondolhatjuk, hogy a struktúrák marshallingolása ebből fakadóan nehezebb folyamat, és ez minden bizonnyal így is van. További fejtörést okozhat például az alábbi ábrán látható Delphiben deklarált rekord típus felépítése

[9] Az `uint8_t` C-ben a char, azaz a byte típussal ekvivalens, 1 bájton tárolt, előjel nélküli egész értéknek felel meg.

C# nyelvben. A probléma ebben az esetben az, hogy míg Delphiben értelmezve van a unionok[10] használata, addig C#-ra már ez nem igaz.

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

1.2. ábra. Példa egy union típus definíciójára. 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[11] keresztül, és a másik nyelv számára is átalakítható a maga nyelvi sajátosságainak megfelelően (más szóval: kódolható és dekódolható). Látni fogjuk, hogy a projektben én ezt úgy oldottam meg,

1.5. 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 Runtimenak nincs közvetlen hatása ezek működésére: a memóriacímeket közvetlenül lehet kezelni, felülírni, így bizonyos esetekben olyan memóriaterületre is hivatkozhat egy változó, amelyet az operációs rendszer nem a program részére foglalt le, ezt `segfault`nak nevezzük.[12] [16]

[10] 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 `handtbp` 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: [23]

[11] Csatorna alatt érthetünk akár egy XML vagy JSON formátumú fájlt, akár egy stringet, akár valami más koncepción alapuló bájtfolyamot is, mint például a Protocol Buffers esetében.

[12] Másnéven *access violation* hibának is nevezik, a Delphi inkább ezzel a kifejezéssel írja le a hibát.

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[13], és így tovább. A fent említett szolgáltatások a fejlesztőket szolgálják, hogy biztonságosabb és optimálisabb kódot készíthessenek. [31]

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 eszközöket 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 ütköznénk, 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 kevesebbszer hagyatkozzunk az unmanaged kódrészletekre. [32]

1.6. 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 szemétgyűjtés automatikusan lezajlik, ezért nem is kell memóriaszivárgástól tartanunk.

– A dinamikus, azaz futásidejű típusellenőrzés biztosítva van.

Átokként és áldásként is felróható, 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 optimalizálást előbb végezze el.

[13] A Garbage Collection, azaz szemétgyűjtés 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.

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

1.7. 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 bejelentkezési adatok vagy éppen a felhasználó által használt kliensprogram verziószáma, elavultabb verziók esetén akár figyelmeztetheti is a felhasználót, hogy telepítse a program egy frissebb változatát. [3]

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

1.8. 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[14]: 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 keresztül: *"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"*. [24]

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

[14] Amennyiben valamelyik fél eltér a meghatározott szabványtól, a kommunikáció nem fog létrejönni, a kliens és a szerver nem fogja érteni egymást.

nyelvnek a kommunikációban résztvevő két fél között.

A szerver – így a kliens is – ismeri a következő halmaz elemeit:

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

Ezen halmaz elemeit 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 bejelentkezési 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 "$LOGIN$" letiltásra kerül. Ha a bejelentkezést követően a felhasználó ugyanazon parancsot ismételten kiadná, kapna a szervertől egy "$ERROR|ALREADY\_LOGGED\_IN$" üzenetet.

Szintaktikailag[15] 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)[16] már egyszer megtörtént, így a megkapott hibaüzenetben tulajdonképpen egy *szemantikai hibáról*[17]

1.9. 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 érhetőek el. A kliensnek elegendő ezen URI-címek valamelyikét meghívni, a hívással egy kérést intéz a szerver felé, majd az valamilyen szabványos szöveges formá-

[15] **Szintaxis:** Programozásban az elgépelt kulcsszavak, azonosítók, az idézőjelek és zárójelek helytelen használata, és így tovább.

[16] 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.

[17] **Szemantika (jelentéstan)**: 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.

tummal – ilyen például a JSON is – képes visszaküldeni a kérés eredményét a kliensprogram számára.

1.10. Google RPC és a Proto-nyelv

A Google RPC, röviden gRPC a Google jóvoltából készült nyílt forráskódú RPC-keretrendszer. Remote Procedure Callnak[18] 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 kódrészleteket, implementációkat egy másik számítógépre, egy szerverre helyezzünk át, ettől lesz a hívás "távoli".

Hogy az RPC-t jobban megértsük, gondoljunk egy távirányítóra: a távirányítóval képesek vagyunk távolról vezérelni a televíziót, be- és kikapcsolhatjuk, csatornát válthatunk 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 elértük, 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, például meghibásodott a nyomógomb, vagy éppen 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. [27]

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, az a gRPC esetén a Proto-nyelv, ami átjárást biztosít 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,

[18] magyarul: Távoli eljáráshívás

ugyanezt a keretrendszert alkalmazó szerverprogrammal is. A Proto-nyelv választott témám szempontjából kifejezetten lényeges, mivel pontosan arra a célra lett kitalálva, hogy programozási nyelvek felett álljon, ezért úgy gondolom, érdemes tovább vizsgálnunk a nyelvben rejlő lehetőségeket.

De miért kell ehhez egy külön nyelv? Nem maradhatunk a jól bevált formátumoknál, amilyen a JSON is? A Proto-nyelv olyan előnnyel jár, amelyeket sem a JSON, sem az XML, sem bármilyen más szerializált formátum nem tud biztosítani: az adatszerkezeteket szigorúan típusosan írja le. Ezenkívül nemcsak adatszerkezetek, hanem szolgáltatások – azaz metódusok szignatúrájának – leírására is egyaránt alkalmas a Proto-nyelv, ezért is lehet érdemes az RPC ezen megközelítését használni, mivel nagyobb lehetőséget biztosít a kommunikációra, mint az előbb felsorolt opciók.

```
service Adopt {

  rpc Login(User) returns (SessionID);
  rpc Logout(SessionID) returns (Result);
  // ...
}

//bejelentkezéskor user ezt kapja, ezt használjuk a felhasználó kiléptetésére
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;
}
```

1.3. ábra. Beadandó 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 szabályrendszere szerint képes lesz legyártani ebből. Ennek lehetősége a legtöbb RPC-keretrendszernél nem áll fenn.

Mint azt említettem, Proto-nyelvben a változókat csak típussal együtt vehetünk fel[19], osztályokat `message`, a meghívható metódusokat a kifejezőbb `rpc` kulcsszóval látja el. Mint láthatjuk, a mezők nevéhez 1-től kezdődően számokat rendelünk, ez a szerializációhoz 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.

Az adatokat a Protocol Buffers használatával a lehető legtömörebb bináris formátumban tudjuk leírni és továbbítani a hálózaton, a JSON-formátum ezzel szemben egy

[19] Ezen tulajdonsága miatt a Proto szigorúan típusos nyelvnek számít.

az egyben szövegesen kerül átadásra, ami jóval több adatforgalmat tesz ki. Természetesen ez is előnyös a gRPC részére, hogy ugyanaz az adat, amit szövegesen közölnénk, tömörebb formában kerül átadásra szervertől kliensig, és fordítva. Ennek használatával véleményem szerint csökkenthetjük az adatforgalmat, és növelhetjük a szerver kapacitásait, mivel annak egyetlen kérés kiszolgálására kevesebb energiát kell fordítania.

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 típussal 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. [10]

1.10.0. Jobb a Proto a JSON-formátumnál?

Ezt a kérdést árnyalja, hogy gRPC használatával négyfajta metódushívást tudunk leírni, ezeket érdemes megvizsgálni:

- *Unáris hívások*: Itt egyszerű kérés-válasz-üzeneteket cserélünk. Ez Protoban úgy néz ki, hogy a szolgáltatások paramétere és visszatérési értéke is message típusú. Például az alábbi ábra szerint a legelső, `Login`-függvény egyetlen `User`-objektumot[20] vár, majd ez a függvény egyetlen `SessionID`-objektummal tér vissza.

- *Kliens streamelés*: A kliens egy adatfolyamot – gyakorlatban ez egy lista – küld paraméterben a szerver részére, a kiszolgáló pedig egyetlen objektummal (message) tér vissza. Erre példa az ábrán látható harmadik függvény, az `Adopt`, amely a gyakorlatban annyit jelent, hogy a kliens egy kérésben több állatot, állatok listáját képes örökbe fogadni. A szerver erre a kérésre egyetlen `Result`-objektummal válaszol.

- *Szerver streamelés*: A kliens elküld egy kérést, és a szervertől válaszként streamet fogad. Erre például szolgál az ábra második függvénye, amelyben a kliens lekéri az összes állatot tartalmazó listát a szervertől, hogy azokat a felhasználó számára megjeleníthesse. A szerver minden esetben egy listával, egy streammel fog válaszolni, legyen az üres, egyelemű vagy többelemű.

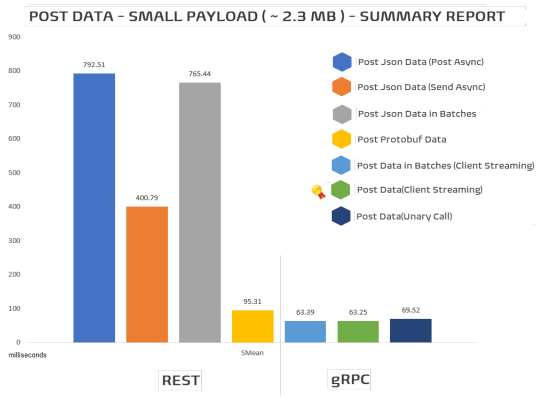
[20] a gRPC kontextusában ez message néven ismert

– *Kétirányú streamelés*: A kliens és a szerver is streamben kommunikál egymással. Például azt mondjuk, hogy az állatok listájából egyszerre több szűrési feltétel megengedett, az állat telephelye és faja szerint is végezhetünk szűrést. Ekkor a kliens szűrések listáját küldi el, és a szerver olyan állatok listájával tér vissza, amelyekre a szűrési feltételek egyszerre teljesülnek. Ezt mutatja az ábra 4. metódusa.

```
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);
}
```

1.4. ábra. gRPC metódushívásainak típusai

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[21] kliens streamelt hívás 63.39 ms és 69.52 ms eredményekkel) is hasonlóan teljesítettek, és pár ezredmásodperccel maradtak le az első helyezetthez képest. [1]



1.5. ábra. Szerializáció teljesítményének összehasonlítása JSON és Proto esetében

Forrás: [1]

[21] 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:[26]

1.10.1. DLL-ek üzenetküldése Win32-ben

A DLL (Dynamic-Link Library) egy forráskódokból előállított, gépi kódra lefordított könyvtár, amely olyan kódokat és adatokat tartalmaz, amelyeket egyszerre akár több program is használhat. Ebben a tekintetben az én meglátásom szerint a DLL egyfajta "helyi szerverként" is felfogható, amelynek meghirdetett, hívható szolgáltatásai (metódusai) vannak, valamint a benne tárolt adatokon műveletet végez. A DLL azonban csak Windows operációs rendszereken hívható segítségül, például a különböző Linux-disztribúciók számára ez ismeretlen, ezek a .so (shared object) kiterjesztésű fájlokat használják ugyanerre a célra.

Egy üzenetlistán[22] 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 tulajdonságaiból (ablak pozíciója, szélessége és magassága) tudjuk detektálni például, hogy az egér az ablak területére érkezett.[23]

[22] A Message Queue (üzenetsor) egy sor (queue) adatszerkezetben, érkezési sorrendben tárolja az üzeneteket, majd ezek ugyanezt a sorrendet megtartva kerülnek le róla.

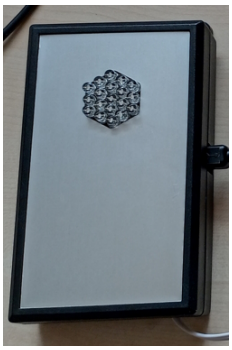
[23] Erre vonatkozó esemény a `MouseEnter`-event Windows Forms esetében.

Ha erre a felhasználói bemenetre fel van készítve a programunk által üzenetküldésre használt metódus[24], 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.

1.10.2. Szakdolgozati munkánkról

A vezérelni kívánt eszközök



1.6. ábra. SLDLL segítségével vezérelhető lámpa típusú eszköz

1.7. ábra. SLDLL segítségével vezérelhető nyíl típusú eszköz

[24] Egy-egy Win32-üzenet feldolgozását Windows Form asztali alkalmazás esetében a `WndProc`-metódus szolgálja.

1.8. ábra. SLDLL segítségével vezérelhető hangszóró típusú eszköz

Csatlakozók, átviteli eszközök

Speciális, két eres (2 position, 2 contact, azaz 2p2c) telefonkábel két végére RJ9-es csatlakozókat krimpeltünk, ezzel kapcsolhatjuk össze az eszközöket, amelyek a programban egyetlen tömbben lesznek kezelve.

1.9. ábra. Telefonkábel RJ9 csatlakozókkal

Nyomatókábel USB-A és USB-B végekkel, előbbi a számítógéphez, míg utóbbi az elsőnek szánt eszközhöz csatlakoztatandó:

1.10. ábra. Nyomtatókábel USB-A és USB-B csatlakozókkal

Tápkábelt használunk a további, számítógéphez csak közvetve nem csatlakozó eszközök feszültségre kapcsolásához:

1.11. ábra. Tápkábel a csatlakozókkal

Ezen dokumentum mellé készített szakdolgozati szoftverprojektemet Sipos Levente hallgatótársam részére, az ő feladatának megkönnyítésének céljából készítem. 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[25] DLL-projekt lesz, amelyeket Levente az ő Windows Forms keretrendszerrel készített grafikus alkalmazásában tud meghívni. Továbbá szükség volt egy úgynevezett relayDLL-re, amelyet én állítottam össze.

1.11. 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 későbbiekben.

A készített alkalmazásunk gyakorlatilag különböző fény- és hangjelzések kibocsátására alkalmas eszközök (lámpák, nyilak és hangszórók) vezérléséből áll, egy úgynevezett *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.

[25] 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.

1.12. Mit jelent az intelligens szoba?

Az intelligens szoba elnevezés olyan helységet takar, amelynek mind a négy falán jeladókat helyeztünk el. Ezek különböző típusú eszközök: fények, nyilak és hangok. Ezek különállóan vagy együttesen jeleket küldenek, amely jelekre speciális mozdulatokat kell a foglalkoztatottnak végrehajtani. Az eltérő színek más feladatokat írnak le, mást kell tenni piros, és szintén 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 eszközt 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 egyszerűen (0,0,0) RGB-színkód[26] közlését, míg hangeszköz esetén egy 0 dB hangerősségű tetszőleges frekvenciájú hangjelzés kiküldését fogja jelenteni. Somodi László edzővel való együttműködésünk Dr. Király Roland tanár úr jóvoltából jöhetett létre, akinek az alkalmazással kapcsolatos múltbéli tapasztalatait és ötleteit folyamatos egyeztetések, konzultációk útján tudtuk segítségül hívni.

1.13. Delphi és C# 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.[27] [33] 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 egységbe zárunk, ezt az egységet a továbbiakban osztálynak nevezzük. Az osztály mezőit és metódusait különböző láthatósági szintekkel vértezhetjük fel, ezzel tudjuk

[26] Az RGB-színkódolás egy szín leírását három komponens, vörös (**R**ed), zöld (**G**reen) és kék (**B**lue) alapszínek arányától teszi függővé.

[27] Az umbrella term (magyarul gyűjtőfogalom) olyan kifejezés, amely több fogalmat rendel önmaga alá, így már fogalmak egy csoportját, kategóriáját jelenti összefoglalóan.

védeni az osztályunk kritikus részeit más osztályokkal szemben. Az osztályok között akár öröklődéssel, akár objektum-összetétellel viszonyokat alakíthatunk ki.

A két nyelv **erősen típusosnak** számít, mivel egy változó definiálásakor meg kell mondanunk 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 változó hatókörén belül nem is változhat meg. Hatókör, illetve *scope* alatt a program azon részét, kontextusát értjük, amely magában foglalja az adott változót. Ezen kontextuson belül a változó "életben van", tehát a memóriában hely van lefoglalva számára, nevére és/vagy memóriacímére hivatkozva értékét felülírhatjuk, kiolvashatjuk. A változó típusa meghatározza, hogy az értéket hogyan kell értelmezni a memóriában, a megadott típusnak mely műveletei vannak értelmezve, így például egy `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, lévén ezek fordított (nem interpretált) nyelvek, ezért ezek nem szimultán módon, nem egyszerre történnek. A Delphi fordítóprogramja azonnal gépi kódot[28] készít, míg a C# esetében első lépésben egy köztes nyelvű[29] 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 kizárólag a fordítás folyamatában észlelhetünk.

Ha ennél is tovább megyünk, akkor a C#-nak nagy előnye származik abból a Delphivel való összevetésben, hogy aszinkron[30] kódolási lehetőséget is biztosít, amely felgyorsítja a végrehajtást, jobban ki tudja használni a rendelkezésre álló CPU teljesítményét.[12] A LINQ[31] használata `yield` kulcsszóval az iterációkban biztosítja, hogy csak akkor van végrehajtva a kód, amikor valóban szükség van rá (lusta kiértékelés). A delegate-ek

[28] A gépi kódban már az utasításokat is számok jelzik, ezen nyelv utasításkészlete már a számítógépben működő processzor típusától is erősen függ. Gépi kód általában véve fordítóprogram eredménye, a hardverközeli vezérlőprogramok elkészítéséhez is inkább az eggyel magasabb szinten lévő Assembly nyelvet használják.

[29] 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.

[30] 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.

[31] 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.

használata tovább növelheti a C#-kódok teljesítményét, bár a Delphi is rendelkezik ehhez hasonló funkciókkal. [25]

A fejlesztés folyamán elkészített **modulokat** a különböző programozási nyelvek eltérő megnevezéseket használnak.[32] 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, 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 funkcionalitás végrehajtási részleteit kapszulázzák és elrejtsék. Ez azt jelenti, hogy a modulok tiszta felületet biztosítanak a funkcionalitással való interakcióhoz, miközben a mögöttes kódot elrejtik és védik 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 segítségükkel 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,

[32] Az említett példákon kívül a Java-nyelvben *package*-ként, míg a Pythonban *module*-ként hivatkoznak az osztályokat összegyűjtő egységre.

elég csak azzal foglalkozniuk, hogy mi az a problémakör, amit a program ezen szelete lefed. [7]

Ami közös még a két nyelvben, hogy a fejlesztés moduljaiból Win32-szabványnak megfelelő DLL-ek készülhetnek a segítségükkel. Ezek lefordított, gépi kódú állományok, amelyek más szoftver forráskódjában felhasználhatóak, lényegében a felhasználáskor, futásidőben válnak futtatható állománnyá Windows operációs rendszerek esetében.

1.14. 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 szintaxissal 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. [19]

1.15. 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.

– *Case insensitive*, azaz a kód nem érzékeny kis- és nagybetűre, ezért a változók, a különböző kulcsszavak, metódushívások leírhatóak tetszőleges formátumban.

– Az utasításblokkok határait a `begin` és `end` kulcsszavak jelzik, azokat átláthatóbbá téve.[34]

1.16. Alaphelyzet

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

1.17. Az SLDLL-függvények bemutatása

A következőkben a Keresztes Péter tanár úr által Delphiben implementált metódusokat, ezek kezelésének lehetőségeit fogom részletezni. Általánosságban elmondható, hogy minden függvény egész típusú értékkel tér vissza, amely érték tájékoztat a lefutás eredményességéről: 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[33] keretein belül. Ezen hibakódok projektünkre vonatkozó részét az alábbi táblázatban 1.12a összegyűjtöttem. [17]

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 sikerességéről szóló információt. Ha az üzenethez tartozik más adat akkor arra a Win32-üzenet LParam értékével hivatkozhatunk. Ilyen esettel `SLDLL_Listelem` függvény hívásakor találkoztam, amikor az eszközök darabszámát tároló változónak adtunk értéket (drb485).

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

1.17.0. Open – DLL megnyitása

A program azzal nyit, hogy felméri az USB-porton csatlakoztatott, egymással telefonkábellel[34] összekapcsolt eszközöket, ezeket egy-egy azonosítóval látja el, hogy külön-külön vezérelni tudjuk őket.

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 Delphiben írt konzolos alkalmazás elkészítése is, amely a .NET-ben készült grafikus felüle-

[33] A Win32-es hibakódok szabványa szerint minden hibakódnak a $0x0000$ (decimálisan: 0) és $0xFFFFFF$ (decimálisan: 16 777 215) közötti tartományban kell lennie.

[34] A telefonkábel egy sajátos változatát használtuk, mivel a benne található 4 érből pusztán a középső kettőt vettük igénybe, a két szélső eret ki kellett iktatnunk annak érdekében, hogy azok a soros kommunikációt ne zavarják. A kábelek végeit RJ9 szabványú csatlakozókkal zártuk le, az eszközök a csatlakozóknak megfelelő portokon keresztül lettek egymáshoz kapcsolva.

tünkkel kommunikált volna. Ebben az esetben kizárólag az asztali alkalmazás Handle-paraméterét tudtuk volna átadni, tehát akkor is a C# üzeneteket feldolgozó függvényével (`WndProc` – bővebben a metódus működéséről: 1.21.5) lenne dolgunk.

1.17.1. 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 azonosítóinak ismeretével az eszközöket vezérelni tudjuk. Az `SLDLL_Listelem` függvény ezen adatok feltöltését végzi el számunkra. Ennek a függvénynek mindenképpen később, az `SLDLL_Open` után kell lefutnia, különben működése hibakódot eredményez.

1.17.2. Felmeres – Eszközök felmérése

A hívási sorrend (szekvencia) következő lépése az `SLDLL_Felmeres`-függvény meghívása, amely az USB-porton észlelt eszköz(ök) felmérését indítja el. A nyomtatókábel USB-B vége a számítógép bármely – szintén – USB-B portjára csatlakoztatható. Kifejezetten előnyös tulajdonsága a metódusnak, hogy képes eldönteni, 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 sikerességéről 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.

1.17.3. SetLista - Az eszközbeállítások végrehajtása

Amennyiben a dev485 tömb elemeit hozzárendeljük egy másik, `LISELE`-típusú elemeket 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. 6

1.17.4. 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[35] számít.

[35] 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.

1.17.5. Hibakódok

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

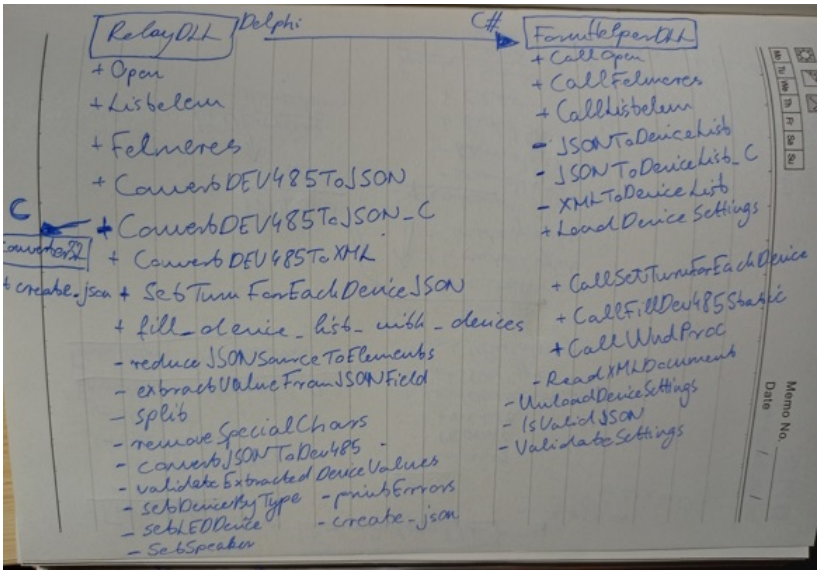
- `Dev485Exception`: Eszközök tömbjére vonatkozó hibaüzenetek.
- `SLDLLException`: Az SLDLL működésével kapcsolatos hibaüzenetek.
- `USBDisconnectedException`: Hibaüzenet arról, hogy az egyik USB-porton sem észlelhető eszköz.

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 megnyitni.	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		

1.12. ábra. Win32-hibakódok magyarázata

Saját szerkesztés

Itt látható a teljes kép az összes DLL – a `FormHelperDLL` példányszintű metódusait leszámítva – minden metódussal.



1.13. ábra. Az elkészült állományok metódusai

1.18. A RelayDLL publikus függvényei

A RelayDLL állomány elkészülését Dr. Király Roland tanár úr javasolta abból a célból, hogy a paraméterezést, így a függvényhívások módját egyszerűsítsük, valamint a meghívható függvények listáját bővítsük.

1.18.0. 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;
```

1.18.1. Listelem és Felmeres

A `SLDLL_Listelem` és `SLDLL_Felmeres`-függvények meghívása a feladatuk. Fontos, hogy a Listelem-függvényt C#-ból is hívni tudjuk, mivel a Win32-üzenetek a Form részére kerülnek elküldésre, és a hívást egy beépített üzenetfeldolgozó metódus, a WndProc folyamán kell elvégeznünk. A Listelem egy egyparaméteres függvény, amely az eszközök darabszámát várja. Ezt az értéket az asztali alkalmazás fogja ismerni, mivel neki fog visszaszólni a Win32-üzenet, miután az `SLDLL_Listelem` meghívódott, így neki kell beállítania a `relayDLL` számára is az eszközök darabszámára vonatkozó változót. A Felmeres nem vár paramétert, pusztán meghívja az `SLDLL_Felmeres`-függvényt.

```
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;
```

1.18.2. ConvertDEV485ToJSON

A metódus az SLDLL `dev485`-tömbjét C# által is értelmezhető JSON formátumú szöveggé alakítja[36]. A kódban is látszik, hogy a JSON-ben kizárólag az eszközök azonosítóit küldjük át, mivel ez a tulajdonság önmagában elegendő ahhoz, hogy azokat vezérelni tudjuk. Az eszköz azonosítójából kikövetkeztethető az is, hogy milyen eszközzel van dolgunk.

Az eredmény egy referenciaként átadott string paraméterből olvasható ki a függvény hívása után. Ezzel a függvénnyel közli a Delphi a C# számára, hogy milyen eszközöket tároltunk a `dev485`-tömbben a `SLDLL_Listelem` folyamán. Ha ez a tömb valamilyen oknál fogva nem került beállításra – ez a `SLDLL_Listelem` dolga –, úgy üres JSON-listával fog visszatérni a függvény, és tájékoztatja a felhasználót egy üzenetablakkal.

```
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;
```

[36] Ez maga a szerializáció: egy szabványos, eltárolható, szállítható, majd helyreállítható formátum elkészítése.

1.18.3. ConvertDEV485ToJSON_C

A `ConvertDEV485ToJSON\_C` pontosan azt a műveletet végzi el, amit az imént említett függvény: JSON-szerializációt. A különbség csupán annyi, hogy C-ből hívja meg a JSON-szerializációt végző függvényt. Ezt a szemléltetés kedvéért készítettem, hogy lássuk, a Delphi és a C nyelv között is tudunk kapcsolatot teremteni egy megfelelően előkészített, exportált metódusokat tartalmazó 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-keretrendszert Michael Grieco szoftverfejlesztő építette fel, én kizárólag ehhez egyetlen, külső hívást megengedő, exportált függvényt készítettem úgy, hogy az Delphiből hibamentesen hívható lehessen.[9]

A függvény szignatúrája így néz ki Delphiben:

```
function create_json(device_ids:pinteger;length:integer):pchar; stdcall;
external 'converter32.dll'
//C-ből vettük ezt a függvényt, ez a converter32.dll-ben található
```

Az elkészült C-függvény (`create_json`) meghívása Delphiben:

```
function ConvertDEV485ToJSON_C(out outputStr: WideString): byte; stdcall;
var
i: integer;
device_ids: array of integer;
begin
SetLength(device_ids, drb485);
for i := 0 to drb485 - 1 do
begin
device_ids[i] := dev485[i].azonos;
end;
outputStr := create_json(@device_ids[0], drb485);
result := 0;
end;
```

Természetesen a JSON-formátum kialakításához használhattam volna külső Delphi-könyvtárakat is, mint például a `SuperObject` [6] vagy az `LKJSON` [13] de fontosabbnak tartottam, 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.

1.18.4. ConvertDEV485ToXML

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

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

1.18.5. 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;
```

1.18.6. 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;
```

1.19. A RelayDLL privát függvényei

1. `reduceJSONSourceToElements`: Delphi stringlistát hoz létre a megadott JSON-tömbből

2. `extractValueFromJSONField`: Megkeresi a kulcsot egy JSON-elemben, és a hozzá párosított értékkel tér vissza, például: a "azonos":10 inputból kiszedi 10 értéket.

3. `split`: A stringet megadott elválasztó karakter (delimiter) mentén darabolja fel

4. `removeSpecialChars`: Eltávolítja a felesleges karaktereket a JSON-bemeneti karakterláncból, előkészíti azt. Ezt tekinthetjük úgy is, mint egyfajta Lexert egy fordítóprogram esetében

5. `validateExtractedDeviceValues`: Ellenőrzi, hogy az eszközök JSON-ben érkező beállításai megfelelőek-e.

6. `setDeviceByType`: Eldönti, hogy az adott eszköz melyik típusba tartozik (nyíl, lámpa vagy hangszóró) és meghívja a hozzá megfelelő set-módszert.

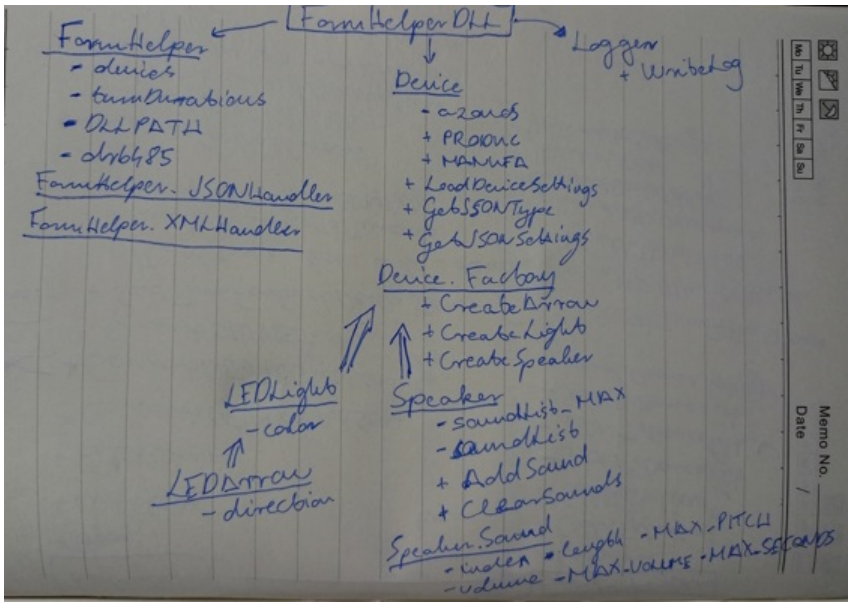
7. `setLEDDevice`: Beállít egy LED-nyíl vagy egy LED-lámpa típusú eszközt

8. `setSpeaker`: Beállít egy hangszórót, az indexet, a hangerőt és a lejátszandó hang hosszát várja paraméterben. Ennek van egy másik változata, ahol hangok listáját

– az előbb említett értékhármasok konkatenáltját – is hozzá lehet rendelni a hangszóró beállításaihoz.

1.20. Az 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.



1.14. ábra. A FormHelperDLL felépítése

1.20.0. Létrehozott osztályok

1. `Device`: absztrakt osztály, minden eszköz őse, ez tárolja az eszközök azonosítóját

2. `Device.Factory`: belső absztrakt osztály, amelynek feladata a különböző eszközök példányosítása.

3. `LEDLight`: lámpa típusú eszközök beállításainak tárolására használt osztály, ebben pusztán – az ősében lévő azonosító mezőn kívül – a LEDek színét tároljuk.

4. `LEDArrow`: nyíl típusú eszközöket tárolunk benne, amely csak közvetetten öröklődik a `Device` osztályból, közvetlen őse a `LEDLight`, mivel nyilak és lámpák esetében ugyanúgy egy szín attribútum jellemzi. A nyilak sajátossága, hogy a felvillanó nyíl irányát is megadhatjuk. Mivel ez csak 3 különböző értéket vehet fel (balra, jobbra és mindkét irányban), ezért én ezt egy `Directions` nevezetű

enumhoz kötöttem, hogy elrejtsek egy olyan alacsonyabb szintű információt, hogy melyik szám pontosan melyik irányt jelenti az SLDLL-ben.

5. `Speaker`: Hangszóró eszközöket leíró osztály, egyetlen, `Sound`-elemeket tartalmazó hanglistát tartalmaz, amelyet – amennyiben annak van eleme – le tud játszani.

6. `Speaker.Sound`: Egy hangszóró által lejátszható hang leírható a következő 3 értékkel:

   a) index/sorszám: Ez egy egész szám, amely egy elemre hivatkozik a frekvenciákat tartalmazó tömbből, a $[0; n-1]$ intervallumból vehet fel értéket, ahol n : a tömb hossza, esetünkben $n = 50$. Ennek segítségével megadhatjuk a lejátszandó hang magasságát. Hogy véletlenül se lehessen intervallumon kívül eső értéket megadni, valamint hogy a hangmagasságok beállítása egyszerűbb, leíróbb lehessen, bevezettem egy enumot `Pitch` (hangmagasság), a Delphi oldalon ezek egész szám formájában jelennek meg.

   b) volume/hangerő: Ez egy egész szám, amely 0 és 63 között vehet fel értéket.

   c) 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 ezredmásodpercekben értendő.

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			

1.15. ábra. Hangszóróval lejátszható hangok listája frekvenciákra bontva

Táblázat: saját szerkesztés
Adatok: SLDLL-állomány

1.20.1. Eszközlista és beállításainak szerializálása

A `dev485`-tömb átalakításra kerül a `relayDLL`-ben, amint be lett állítva az eszközöket tároló tömb, a `dev485`, ez a `Listelem` hívásakor zajlik le. Ennek végül 3 lehetséges változata lett: a `relayDLL`-ben vagy egy XML-fájlt írunk ki a háttértárra (`devices.xml`),

vagy egy JSON-szöveggel tér vissza, ezt teheti úgy, hogy a saját függvényét hívja, de választhatjuk azt az irányt is, hogy egy külső, C-ben implementált függvényt hívunk meg. Az alábbi függvény hívásakor átadott paraméterrel dönthetünk, hogy melyik implementációt választjuk, alapértelmezésként a Delphiben elkészített JSON

```
public static void CallListelem(
    ref byte drb485,
    ToDeviceList_UsedMethod method = ToDeviceList_UsedMethod.JSON){
    ushort result = Listelem(ref drb485);
    if (result == 254)
        throw new Dev485Exception("Az eszközöket tartalmazó dev485 tömb üres!");
    if (result == (ushort)Win32Error.ERROR_DLL_INIT_FAILED)
        throw new SLDLLException(
            "Hiba az eszközök beállítása közben: SLDLL_Open még nem került meghívásra.");
    if (result == (ushort)Win32Error.ERROR_SUCCESS) //sikeres lefutás esetén
    {
        devices.Clear();
        turnDurations.Clear();
        switch (method)
        {
            case ToDeviceList_UsedMethod.JSON:
                JSONToDeviceList();
                break;
            case ToDeviceList_UsedMethod.JSON_C:
                JSONToDeviceList_C();
                break;
            case ToDeviceList_UsedMethod.XML:
                XMLToDeviceList();
                break;
            default:
                break;
        }
        turnDurations.Add(2000); //alapból beállítjuk egy ütem hosszát 2 mp-re
    }
}
```

1.16. ábra. A különböző szerializálási módok meghívása az SLFormHelperben

A C# oldalon ez egy úgynevezett `SerializedDevice`-osztálynak kerül átadásra, amelyben az átadott azonosító alapján létrehozza a típusának megfelelő példányokat, majd ezt a Levente által iterálható `Devices`-listához fűzi, ezzel elértem azt, hogy a Devices-listában különböző eszközök lettek létrehozva, ezek beállításait minden további nélkül felül lehet írni azok vezérléséhez.

Ilyenkor felvetődik egy másik kérdés: hogyan válnak a C#-ban módosított eszközbeállítások a Delphi által is értelmezhetővé? Ezt szintén JSON-formátummal oldottuk meg, erre az `SLFormHelper SerializedTurnSettings`-osztálya biztosít lehetőséget, hogy a JSON-ben átadásra kerüljön egyrészt az eszközlistában szereplő eszközök típusa, másrészt azok minden beállítása típusuktól függően. Ugyanebben a formátumban feltüntettem ütemenként annak időtartamát is, ennek a JSON beolvasásakor és kimentésekor van igazán jelentősége, hogy az ütemek időzítése is rögzíthető és helyreállítható lehessen. A kiolvasott időtartamok meghatározzák, hogy az eszközbeállításokat ütemenként kiküldő `CallSetTurnForEachDevice`-függvény milyen késleltetéssel legyen meghívva a C#-ban, tehát mikor változzanak az eszközre kiküldött beállítások, például egy pirosan világító lámpa 2 másodperc elteltével, a következő ütemben kapcsoljon ki.

Ennek az ütemezésnek köszönhetően számunkra is látható, tapasztalható eredménye is van az eszközökre kiküldött jeleknek. Az eszközök ütemenként változó beállításait a `relayDLL` az erre a formátumra felkészített `CallSetTurnForEachDevice`-függvénnyel fogja a Delphi nyelvezetével az eszközök részére közölni.


```
[
  {
    "devices": [
      {
        "type": "H",
        "settings": "0|63|500"
      },
      {
        "type": "N",
        "settings": "255|0|0|0"
      },
      {
        "type": "L",
        "settings": "0|128|0"
      }
    ],
    "time": 2000
  }
]
```


1.17. ábra. Egy lámpa és egy hangszóró 1 ütemre vonatkozó beállításai JSON-formátumban

1.21. Problémák és megoldások

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

1.21.0. Egyénileg definiált típusok és struktúrák

A probléma leírása

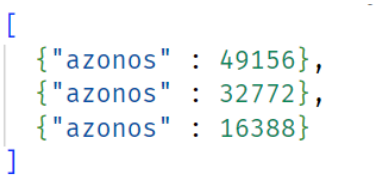
A Delphis projekt erősen függ az SLDLL-ben megkívánt típusoktól, ezért a teljes kódot nem tudjuk C#-ra átültetni.

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 mindkét nyelv számára. Ilyenek a primitívek, az egész, lebegőpontos, logikai és szöveges típusok).



1.18. ábra. Eszközök azonosítóinak átadása JSON-formátumon keresztül

1.21.1. 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 (`dev485`) ugyanúgy elérhetőnek kell lennie.

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 megoldásom 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 máshol definiálni, 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 mindent 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.

1.21.2. Az SLDLL_Open-függvény paraméterezése

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 Windows Form – mint függvényeket meghívó fél – egy Handle nevezetű pointerrel[37] rendelkezik, amit átadhatunk az SLDLL-nek.

1.21.3. 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 bármilyen más típus formátuma is – így a rekordoké is – leírható és feldolgozható.

Megoldás

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.[28]

```
//converting dev485 to JSON-format - JSON-serializing
function ConvertDEV485ToJSON(
        out outputStr: WideString): byte; stdcall;
        external RELAY_PATH;
```

1.19. ábra. A függvény helyes Delphi-deklarációja

Magyarázat a `DllImport`nak átadott paraméterekről:

- `Charset.Unicode`: utasítja a marshallert arra, hogy UTF-16 formátumban értelmezze a szöveget, hacsak nincs más utasítás.
- `CallingConvention.StdCall`: Az adott függvény által elfogadott ”hívási konvencióra” utal. Azt jelenti, hogy a hívott fél, a relayDLL végzi a hívási verem (call stack) felszabadítását.
- `DLLPATH`: konstans érték, a relayDLL relatív elérési útja az SLFormHelperDLL mappájához viszonyítva, arra vonatkozik, hogy hol keresse a marshaller az adott függvény implementációját.
- `EntryPoint='ConvertDEV485ToJSON'`: A marshaller ezen a néven keresi az adott függvény implementációját a relayDLL-ben.

[37] A pointer egy olyan változó, amely memóriacímet képes tárolni. Mérete a memóriacímek méretezésétől függ (például 64bites rendszereken 8 bájt ír le egy memóriacímet, ezért egy pointer tárigénye is ennyi lesz.).

– `[MarshalAs(UnmanagedType.BStr)]`: A kimenő paraméterként átadott változót ilyen formában küldi át a Delphinek, és ilyen formában várja is azt vissza. A Delphiben készült függvény WideString paraméterére a C#-ban stringként átadott változó így fog illeszkedni. Ez a string átadásának egyik lehetséges párosítása.

```
[DllImport(DLLPATH, CallingConvention = CallingConvention.StdCall,
        CharSet = CharSet.Unicode, EntryPoint = "ConvertDEV485ToJSON")]
extern private static byte ConvertDeviceListToJSON(
        [MarshalAs(UnmanagedType.BStr)][Out] out string outputStr);
```

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

1.21.4. 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 relayDLL-függvény, ebből a hibakódból a C# tudni fogja, hogy milyen típusú kivételt dobjon az asztali alkalmazás számára. Ezekből egyedi kivételfajták készíthetők.

1.21.5. 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[38] (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:

[38] 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.

1. A DLL működését elindító `SLDLL_Open` kivált egy Win32-üzenetet. A paraméterben átadjuk a Form Handle-mezőjét, ezzel az operációs rendszer tudja, hogy melyik alkalmazással kell kommunikálnia.

2. Az üzenet felkerül a Win32 üzenetsorra (Message Queue), ezt kiolvasva operációs rendszer meghívja a Form `WndProc` metódusát.

3. Amennyiben a felmérés sikeresen lezajlott, a `WndProc` meghívja a relayDLL `dev485` és `drb485` változóit beállító `SLDLL_ListElem`-függvényt, mielőtt az `SLDLL_Felmeres`-re kerülne a sor. A megfelelő hívási szekvencia így automatikusan biztosított.

1.21.6. 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 azonosítója, tehát az azonosítót XML-fájlba kimentjük [29], vagy JSON-formátumú szöveggé alakítjuk. Az azonosító önmagában meghatározza az eszköz típusát, úgyhogy elég azt átadni. Egy eszköz azonosítója típusának megfelelően a következő intervallumokból kaphat értéket:

- `SLLELO` (lámpa): 0x4000 -> $[0x4000,0x4fff]$ ez decimálisan ezt a tartományt fedi le: [16 384,20 479]

- `SLNELO` (nyíl): 0x8000 -> $[0x8000,0x8fff]$ ez decimálisan ezt a tartományt fedi le: [32 768,38 683]

- `SLHELO` (hangszóró): 0xc000 -> $[0xc000,0xcfff]$ ez decimálisan ezt a tartományt fedi le: [49 152,53 257]

Az eszköz típusának meghatározásában nem játszik szerepet az utolsó 3 bitnégyes, tulajdonképpen a legfelső hex[39] azonosítja a típust. Ha szigorúan hexadecimálisan nézzük, 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.

[39] 4 bit azonosít egy hexadecimális számjegyet – 0-tól F-ig terjedően –, így a bitnégyest hexnek is szokás nevezni.

1. Algorithm Egy eszköz típusának meghatározása

function CreateDevice(azonos: int)
  $deviceType : int \leftarrow azonos \ \& \ 0xc000$; ▷ &: bitenkénti AND-művelet
  if deviceType = $0x4000$ **then**
    $CreateDevice \leftarrow Device.Factory.CreateLight(azonos)$;
  else if deviceType = $0x8000$ **then**
    $CreateDevice \leftarrow Device.Factory.CreateArrow(azonos)$;
  else if deviceType = $0xc000$ **then**
    $CreateDevice \leftarrow Device.Factory.CreateSpeaker(azonos)$;
  else
    $CreateDevice \leftarrow null$;
  end if
end function

A megoldás a RESTful alkalmazásokból inspirálódik, amelyről a(z) 1.9 alfejezetben ejtettem szót, a C# és a Delphi között akár az XML, akár a JSON közös nyelvként (hídként) szolgál. Ebben az analógiában a RelayDLL számít a szervernek, ennek függvényeit az SLFormHelper kliensként hívhatja.

1.21.7. A hangszóró egy egész hanglistát kezel

A probléma leírása

A hangszórók nem feltétlenül csak egy bizonyos hang, hanem egy egész hangsorozat lejátszására is képesek.

Megoldás - JSON-formátumban

Egy lejátszandó hangot az előbb említett értékhármas (sorszám, hangerő és időtartam) ír le. 6 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;
```

1.21. ábra. Megoldás a hangszóró hanglistájának kezelésére - 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 – kizárólag Sound-típusú objektumok listáját tartalmazza, a listát JSON-ben *"index/volume/length"* rendezett hármasok összefűzésével szerializálja, a listához kizárólag listakezelő függvényekkel lehet hozzáférni (törlés, módosítás, hozzáadás), majd az `SLDLL_Hangkuldes` számára kiküldött H-tömböt (hanglistát) Delphiben egy ciklussal a JSON-nel átadott adatok szerint a megfelelő sorrendben feltöltjük, az előző probléma megoldásában ennek miként jét kifejtettem.

```
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();
    }
```

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

1.21.8. Több csatlakoztatott eszköz felismerése, vezérlése

A probléma leírása

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

Az első esetben szimplán nem érzékelte az USB-porton lévő eszközöket – ez az 1626-os (`ERROR_FUNCTION_NOT_CALLED`) 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 ellentétes volt, továbbá a két szélső, a sárga és fekete ereken 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 RJ9-es csatlakozóhoz erősíteni. Az ily módon elkészült kábel alkalmassá teszi az eszközöket a soros kommunikációra.

Két eszköz megfelelő összekapcsolásának módja az alábbi ábrán látható. A laptopot nyomtatókábellel kapcsoljuk a legelsőnek szánt eszközhöz, majd ezt követően

1.23. ábra. Két eszköz csatlakoztatásának módja.

1.21.9. Az elkészített forráskódok

a) `FormHelperDLL` és `relayDLL` állományok forráskódjai itt érhetőek el: Github

b) `converter32DLL` állomány forráskódja itt érhető el: Github

1.22. 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.

Örömömre szolgált, hogy Somodi László minket bízott meg elméleteit kivitelező szoftveres háttérrel. Szakdolgozatunk nem kizárólag arról szól, hogy szakmai tudásunkat gyarapítsuk és mutassuk bizonyítványként 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 jellemezhető, azaz több szakterület, jelesül a mozgáskoordináció, 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.

Inspirációnkat, motivációnkat 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 1.10 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 mindenképp érdemes lenne munkánkat esetleg egy másik szakdolgozat keretein belül a gRPC irányába elmozdítani.

1.23. Köszönetnyilvánítás

Szeretném ezúton is megköszönni konzulensemnek, Dr. Király Roland Tanár Úrnak a tanulmányaimban 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 nagy türelemmel támogatott és motivált engem hozzájárulva ahhoz, hogy szakdolgozatom időről időre minél jobbá válhasson.

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 lelki épségben 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] ekapujiw2002. Delphi 7 json superobject. `https://github.com/ekapujiw2002/delphi7-json-parser-superobject`, utolsó elérés: 2023-04-04.

[7] ELTE. Delphi programozási nyelv. `http://nyelvek.inf.elte.hu/leirasok/Delphi/index.php`, utolsó módosítás: 2014-06-04.

[8] FreeCodeCamp. How to explain object-oriented programming concepts to a 6-year-old. `https://tinyurl.com/34pj477d`, utolsó módosítás: 2018-06-27.

[9] Michael Grieco. Programming challenges - 28.1 - json data (c). `https://www.youtube.com/watch?v=mnIvV-IBI7Y`, utolsó elérés: 2021-04-01.

[10] Jeremy Hillpot. grpc vs. rest: How does grpc compare with traditional rest apis? *DreamFactory*, 2022-11-11. `https://tinyurl.com/2p8w28s3`.

[11] Ben Joan. Difference between ansi and unicode.
http://www.differencebetween.net/technology/software-technology/difference-between-ansi-and-unicode/, utolsó elérés: 2023-02-23.

[12] Jonathan Johnson. Asynchronous programming: A beginner's guide.
https://www.bmc.com/blogs/asynchronous-programming/, 2020.

[13] Leonid Koninin. Json delphi library.
https://sourceforge.net/projects/lkjson/, utolsó elérés: 2023-04-04.

[14] Microsoft Learn. Default marshalling for strings.
https://learn.microsoft.com/en-us/dotnet/framework/interop/default-marshalling-for-strings, 2022-05-20.

[15] Yen Nee Lee. Covid could trigger a spike in dementia cases, say alzheimer's experts. *CNBC*, 2022-09-17. https://tinyurl.com/ynm3cecx.

[16] 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#.

[17] Microsoft. Win32 error codes.
https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-erref/18d8fbe8-a967-4f1c-ae50-99ca8e491d2d, 2021.

[18] David Nield. The brains of teenagers look disturbingly different after lockdown. *Science Alert*, 2022.
https://tinyurl.com/29m7r88a.

[19] ELTE Prognyelvek portál. Linq lekérdezések.
http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=25, utolsó módosítás: 2010.

[20] ELTE Prognyelvek portál. Marshalling használata csharp alatt.
http://nyelvek.inf.elte.hu/leirasok/Csharp/index.php?chapter=6#section_3, utolsó módosítás: 2010.

[21] RedHat. What is an api?
https://tinyurl.com/m5n7wfyp, utolsó módosítás: 2022-06-02.

[22] 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/.

[23] Jacob Sorber. When do i use a union in c or c++, instead of a struct? https://www.youtube.com/watch?v=b9_0bqrm2G8, utolsó elérés: 2021-06-09.

[24] 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.

[25] 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.

[26] TIBCO. What is batch processing? https://www.tibco.com/reference-center/what-is-batch-processing, utolsó elérés: 2023-02-11.

[27] ultraware (username). Delphigrpc github repository, 2018-10-17. https://github.com/ultraware/DelphiGrpc.

[28] 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, 2019.

[29] Abdul Salam (username). how to create xml file in delphi. https://stackoverflow.com/questions/8354658/how-to-create-xml-file-in-delphi, 2012.

[30] Brian G (username). What is object marshalling? https://stackoverflow.com/questions/154185/what-is-object-marshalling, 2008-09-30.

[31] Peter (username). What is the difference between serialization and marshaling? https://stackoverflow.com/questions/770474/what-is-the-difference-between-serialization-and-marshaling/770509#770509, 2009.

[32] Pokus (username). What is managed or unmanaged code in programming? https://stackoverflow.com/questions/334326/what-is-managed-or-unmanaged-code-in-programming, 2008.

[33] Tracing (username). Object pascal vs delphi? https://stackoverflow.com/questions/15699788/object-pascal-vs-delphi, 2012.

[34] FreePascal Wiki. Pascal for csharp users.
https://wiki.freepascal.org/Pascal_for_CSharp_users#Types_Comparison, 2023-02-08.

Ábrák jegyzéke