Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Gaják Tibor István

TÁVOLI ADATGYŰJTŐ RENDSZER KÉSZÍTÉSE PICOSCOPE 2207A TÍPUSÚ OSZCILLOSZKÓPPAL

KONZULENS

Dr. Orosz György

BUDAPEST, 2017

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Gaják Tibor István** szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző, cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2017. 12. 06

...…….

Gaják Tibor István

Kivonat

A dolgozat témája távolról is vezérelhető adatgyűjtő rendszer készítése PicoScope 2207A típusú PC-s oszcilloszkóp felhasználásával. A megvalósított rendszer alkalmas jelgenerálásra, adatok gyűjtésére, megjelenítésére, feldolgozására. Mindezek mellett lehetőséget biztosít távoli vezérelhetőséghez, fizikailag a mérés helyszínén tartózkodás nélkül végezhető méréshez. A választott PicoScope adottságainak köszönhetően egy rendkívül mobilis rendszer létrehozására került sor, ugyanis a műszer önmagában tartalmazza a jelgenerátor és mérőműszer képességeit, ráadásul fizikailag elfér akár egy zsebben is.

A mérőrendszer tartalmaz egy olyan szoftvercsomagot, ami felhasználói kényelmet fokozva egyetlen grafikus kezelői felület segítségével az összes funkció vezérelhető. A szoftvercsomag Python nyelven készült, amely egy szerver oldali és egy kliens oldali részből tevődik össze. A két egységet TCP/IP kapcsolat köti össze, amelyen keresztül elvégezhető az összes szükséges hálózati kommunikáció.

A feladat célja egy olyan kísérleti mérőrendszer megalkotása volt, ami laboratóriumi demonstrációkhoz segítséget nyújthat, továbbá későbbi továbbfejlesztéssel akár ipari felhasználásra alkalmas intelligens mérőrendszer lehessen.

Abstract

This thesis is about development of remote data acquisition system using PicoScope 2207A oscilloscope. The system is capable of generating signals, acquiring data, display data and process the acquisition signal. Moreover the system suitable for remote controlling and measurement without staying at the measured system. Due to the capability of the selected oscilloscope the created system is mobile because the PicoScope contains the ability of the measuring instruments and the signal generators.

The measuring system contains a software package, which is able to control all features by the help of the graphical user interface. It was developed in Python programming language, which has two parts. One of them is the client side and the other is the server side. These two parts are connected by the network. The sides communicate through TCP/IP protocol.

The purpose of my task has created an experimental measurement system, which can help demonstrate the laboratory tasks and it could be an intelligent measurement system by future developments even for industrial purposes.

1 Bevezetés

A mérés az emberiség fejlődését végigkísérve mindig jelen volt. Egészen az ókori csillagászati módszereken alapuló időméréstől kezdve a modernkori mérőrendszerek kifejlődéséig. A technológia fejlődésével párhuzamosan a mérőrendszerek fejlődésére is folyamatos igény van. Az orvoslástól kezdve az építőiparon keresztül egészen a minőségbiztosításig egyre nagyobb teret hódít.

1.1 Mérőeszközök fejlődése

Ha rápillantunk a hőmérőre, vagy amikor megnézzük mennyi az idő, mérést végzünk. Ezekben az esetekben a hőmérő és az óra mérőeszközök, más néven műszerek. A történelembe beleásva magunkat felfedezhetjük, hogy már a régi civilizációkban kialakultak térfogat, hosszúság és tömeg mérésére szolgáló eszközök. Nagyon sokféle mértékegység és mérőeszköz volt használatban, ezért egyesítésük a XIX. században kezdődött és mind a mai napig tart. A pontosabb időmérők fejlődését 1280 körül Angliában az ingaóra feltalálása indította el. Az 1300-as évektől kezdve tengeri hajózáshoz helyzet-meghatározására fejlesztett eszközöket már műszereknek nevezhetjük. Az áttörést az elektronikus műszerek fejlődéséhez az elektroncső és a katódsugárcső feltalálása indította el a XX. században. Az utóbbi 30 évben történt robbanásszerű fejlődést a mikroelektronikának és a számítástechnikának köszönhetjük. A ma kapható elektronikus műszerekben általában a legtöbb feladatot mikroprocesszorok látják el. Mindezek mellett elterjedtek az úgynevezett virtuális műszerek is. Ezeket mérési feladatok megoldására alkalmas áramköri elemek és szotverek tesznek intelligens műszerré. [1]

1.2 Műszervezérlés

Napjainkban már számos lehetőség nyitva áll teszt és mérőberendezéseink kommunikációjára. Talán az egyik legelterjedtebb interfész a GPIB, azaz Bájt-soros bitpárhuzamos adatátvitel. Sok esetben kell mérési eredményeket feldolgozni, mérőműszereket vezérelni. Amennyiben ezek elvégzése nem igényel nagy távolságot, ki lehet használni a párhuzamos adatátvitel előnyeit. [2]

Az ANSI/IEEE 488 szabvány bejelentésének köszönhetően a műszereket számítógépekről is lehet vezérelni, gyártótól függetlenül. A fejlődésért tett erőfeszítéseknek köszönhetően a GPIB jelenleg egy bevált vezérlőinterfész. Előnyei a kompatibilitás, kis késleltetés, jó átviteli teljesítmény. [3]

A GPIB mutatja a legjobb kombinációt költség, egyszerűség és teljesítmény szempontjából, azonban nem minden kategóriában a legjobb. Manapság három új műszervezérlő busztechnológia hódít. Az USB, az Ethernet és a vezetékes PCI Express. Az USB vezérelt eszközök alkalmasak lehetnek hordozható mérőrendszerek megvalósítására, illetve gépjárművekben végezhető adatgyűjtésekre. Azonban nagy hátránya, hogy a vezeték hossza korlátos és a csatlakozók minősége sem a legmegbízhatóbbak.

Az Ethernet előnye, hogy széles körben elosztott műszerhálózatok hozhatóak létre vele, akár földrajzilag nagy távolságokat átívelve. Azonban ennek is akadnak hátrányai. Ide sorolható a nagyon nagy késleltetés, ugyanis ez jelentős torlódásokat okozhat. [3]

1.3 Célok

Látható, hogy a mérőrendszerek világa mennyire sokrétű, komplex, lehetőségekkel teli terület, amely folyamatos fejlődés alatt áll. Minden jelfeldolgozás alapja, hogy rendelkezésünkre álljanak adatok, amiket ki lehet értékelni, fel lehet dolgozni. Így a mérőrendszerek egyik fontos és elengedhetetlen feladata az adatgyűjtés.

Célom egy olyan távolról is vezérelhető adatgyűjtő rendszer elkészítése, amely kompakt, komfortos és felhasználóbarát. Ezek a szempontok alapján elkészült mérőrendszer szívét egy PicoScope 2207A típusú oszcilloszkóp alkotja, amely lehetővé teszi, hogy a mérés összeállításánál ne kelljen fizikailag külön jelgenerátort, adatgyűjtő eszközt alkalmazni. További célkitűzés volt egy olyan kezelői felület megalkotása, amely alkalmas egy programablakból vezérelni szinte minden funkciót. Tovább fokozva a kényelmet a felhasználó számára, képes legyen akár távolról is konfigurálni az eszközt. Nem utolsó sorban pedig lehetőséget nyújtson későbbi továbbfejlesztéshez.

A felhasznált oszcilloszkóp típusa nem volt tervezési szempont, ugyanis a szakdolgozat készítésének helyet adó laboratórium egyik műszeréről van szó, és a

laboratóriumban megoldandó feladatok során felmerült az igény a konkrét eszköz távoli vezérlésére.

A szakdolgozat felépítését tekintve az általános és rendszerszintű szemlélettől kezdve az egyes részkomponensek részletes ismertetése felé haladva mutatja be az elkészült mérőrendszert. A második fejezetben ismertetem a megtervezendő rendszerrel szemben támasztott követelményeket, és a specifikációknak megfelelően elkészített rendszertervet. A következő fejezetben bemutatom munkám során felhasznált hardver és szoftvereszközöket. A negyedik fejezetben található a szoftverrendszer és hálózatkezelés általános bemutatása, azt követő két fejezetben pedig a szerver és kliensoldal részletes ismertetése következik. Legvégül az elkészült mérőrendszer tesztelésének és továbbfejlesztésének bemutatására kerül sor.

2 Rendszerterv és specifikáció

2.1 Specifikáció

A PicoScope 2207A típusú USB vezérlésű oszcilloszkóppal felépített rendszer főbb specifikációi és követelményei a következők:

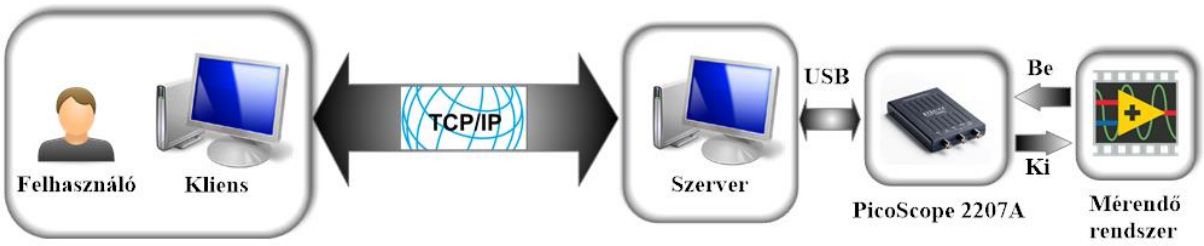
- Python nyelven készített grafikus kezelői felülettel bíró szoftvercsomag elkészítése
- Kezelői felületen keresztül a mérőrendszer vezérlése, paraméterek beállítása
- Két üzemmód választási lehetősége:
  - távoli elérésű üzemmód
  - helyi elérésű üzemmód
- Adatgyűjtő funkciók:
  - blokkos beolvasás
  - valós idejű beolvasás
- Jelgenerálási lehetőség
  - beépített jelalakok generálása
  - tetszőleges jelalak generálása
- Adatok és parancsok átvitele interneten keresztül távoli elérésű üzemmódban
- Mérési eredmények megjelenítése
- Mért adatok utólagos feldolgozási lehetőségeinek megteremtése

2.2 Mérőrendszer felépítése

A célok kitűzése után a mérőrendszer elkészítésének első fázisa magának a rendszer felépítésének a megtervezése. Ez egy jól átlátható egységet alkot a specifikációban meghatározott részfunkciókat megvalósító elemek között. Továbbá a mérőrendszer működésének megértésében segítséget nyújt. Kétféle üzemmódot különböztethetünk meg. Az egyik a helyi elérésű, amikor helyben történik a mérés. Ebben az esetben a PicoScope-pal kapcsolatban lévő számítógép megegyezik a felhasználó által használt kezelői felületet futtató számítógéppel. Másik a távoli elérésű,

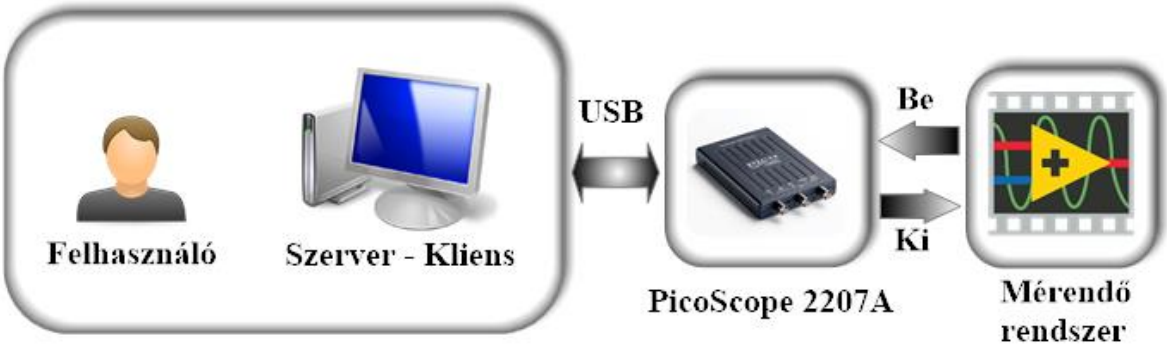
aminél a kezelői felületet futtató számítógép fizikailag máshol helyezkedik el, mint a mérőműszer. Szemléletes bemutatását az elkészült konstrukcióknak az alábbi ábrákon láthatjuk:

1) Távoli elérésű üzemmód:



2-1. ábra Távoli elérésű rendszerterv

2) Helyi elérésű üzemmód:



2-2. ábra Helyi elérésű rendszerterv

Látható, a mérőrendszer felépítése gyakorlatilag két fő részre bontható szét a szerkezeti felépítés alapján. Az egyik a kliens oldal. Itt található a kezelői felület a felhasználó számára, ahol beállítási, konfigurálási lehetőségekkel találkozhatunk.

Másik nagyobb alkotóelem a szerver oldal. Ezen a részen történik lényegében a közvetlen kommunikáció a PicoScope 2207A típusú oszcilloszkóppal USB interfészen keresztül. A PC-s oszcilloszkópot, mint mérőműszer, illetve jelgenerátor formájában egyaránt használhatjuk. A 2–1. és 2–2. ábrákon is látható módon egy BNC kábel segítségével a jelgenerátor kimenetét[1] rákötve a mérendő rendszer bemenetére, majd a

[1] 2-1. és 2-2. ábrákon „Ki”, PicoScope 2207A „AWG” csatlakozója

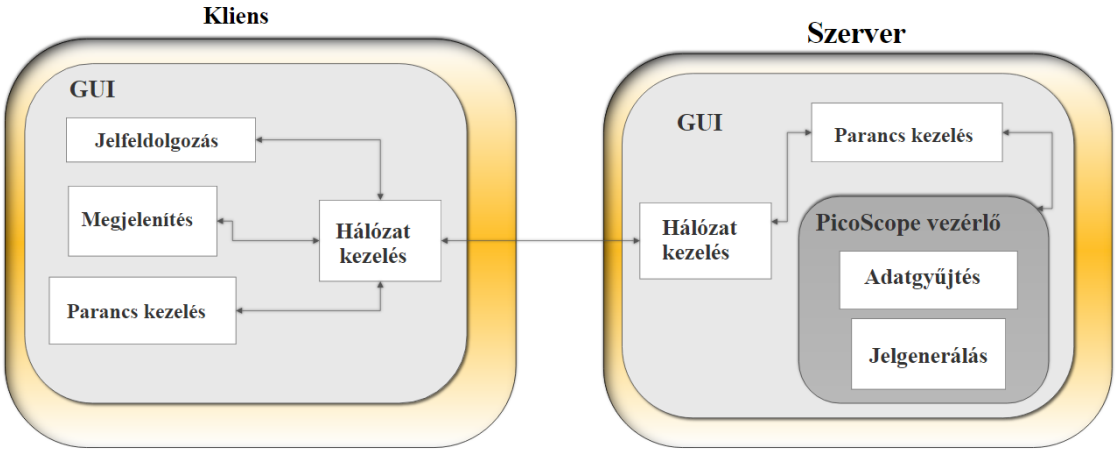
mérendő rendszer kimenetét rákötve szintén egy BNC kábel segítségével az oszcilloszkóp egyik bemenetére[2] adatokat nyerhetünk ki a mérendő rendszerből. Az így kapott adatokat USB-n keresztül a szerver oldali számítógépen megkapjuk későbbi feldolgozáshoz.

A két oldal között TCP/IP protokoll teremt kapcsolatot. Ez lehetővé teszi egyrészt a szerver oldalon mért adatok továbbítását a kliens oldalra, másrészt a kliens oldalon beállítani kívánt paraméterek szállítását a szerver oldalra a megfelelő működés érdekében.

Meg kell jegyezni, hogy a szerver és kliens szoftver egyazon PC-n is futtatható, és a TCP/IP protokoll segítségével a lokális IP címen keresztül ugyanúgy lehet akár PC-n belül is kommunikálni a két program között, mintha a két programkomponens külön PC-n futna. Ezzel a módszerrel tehát a bemutatásra kerülő programrendszer futtatható akár egyetlen gépen lokálisan, akár távoli gépeken elosztottan is.

2.3 Mérőrendszer működése

A következőkben a rendszer két fő szoftverblokkjának bemutatására kerül sor. Szemléletes értelmezést segíti az alábbi ábra:



2-3. ábra Funkciók felépítése

[2] 2-1. és 2-2. ábrákon „Be” , PicoScope 2207A „A” vagy „B” csatlakozója

Kliens oldalon találhatunk egy grafikus kezelői felületet[3], ami egyrészt beállítási lehetőségeket tartalmazó, másrészt megjelenítő részből áll. Beállítási lehetőségek hátterében a program parancsok formájában hálózaton keresztül kommunikál a szerver oldallal. Ide tartozik a jelgenerátor konfigurálása. A program egyaránt lehetőséget kínál beépített jelek, például szinusz, négyszög, háromszög jel, illetve tetszőleges jelek paramétereinek kiválasztására, például amplitúdó, frekvencia, ofszet feszültség. A kívánt paraméterek megadása után a program elküldi azokat a szerver oldalnak. Továbbá lehetőségünk van adatgyűjtés paramétereinek beállítására is. Miután elindítottunk egy adatgyűjtést, majd a szerver oldaltól megkaptuk a begyűjtött adatokat, azoknak kijelzésére is lehetőségünk van vagy fel is dolgozhatjuk.

Másik oldalon a szerver található. Itt is találhatunk egy grafikus kezelői felületet, ahol csak a szerver oldal létrehozásával kapcsolatos kezelőszervek láthatóak, minden más funkció a háttérben történik automatikusan a kliens oldalról vezérelve. Szerkezeti felépítés szempontjából, miután hálózaton keresztül beérkezik egy adatcsomag, azt egy parancskezelő blokk veszi át. Ez a rész kapcsolatban áll a PicoScope által közzétett könyvtári függvényekkel. A kliens által küldött parancsoknak megfelelően itt hajtódnak végre az adatgyűjtéshez vagy jelgeneráláshoz szükséges mechanizmusok. Miután vége lett az adatok gyűjtésének, azokat a szerver hálózaton keresztül visszaküldi a kliensnek.

[3] 2-3. ábrán: GUI

3 Eszközök bemutatása

3.1 Hardver

3.1.1 PicoScope 2207A bemutatása

Ahogyan már megismerhettük, a mérőrendszer elkészítéséhez PicoScope 2207A típusú oszcilloszkópot használtam, mint mérőműszer, illetve jelgenerátor. Az oszcilloszkóp az alábbi ábrán látható:

3-1. ábra PicoScope 2207A [4]

Ez egy USB interfészen keresztül kommunikáló PC-s oszcilloszkóp, amihez a gyártó fejlesztői könyvtárat mellékel DLL[4] formátumú fájl formájában. A fejlesztés előrelendítése érdekében a gyártó mellékel még egy programozói útmutatót [4] is, ami néhány általános információ mellett a felhasználható könyvtári függvények leírását is tartalmazza.

[4] Dinamikus csatolású könyvtár

Két bemeneti csatornával[5] rendelkezik, egy jelgenerátor kimenettel[6], és egy USB csatlakozóval, amin keresztül egyrészt vezérelni lehet az oszcilloszkópot, másrészt tápellátást is ezen keresztül kapja meg. Így ennek köszönhetően, nem kell külön vezetéken keresztül feszültség alá helyezni a műszert.

3.1.2 PicoScope 2207A tulajdonságai

Néhány említésre méltó tulajdonságai közül az egyik, hogy mind a két bemeneti csatornája 100 MHz analóg sávszélességű, feszültségtartományaik ±50 mV-tól ±20 V-ig tart 1, 2, 5 osztásokkal. 40 kS[7] méretű buffer memóriával rendelkezik, ahova blokkos mintavételi üzemmód esetén tárolni tudja az oszcilloszkóp a mintákat. Továbbá 1 GS/s[8] maximális mintavételi sebességre képes, ami folyamatos adatgyűjtő üzemmód esetén 9,6 MS/s[9] maximális mintavételi sebességre csökken.

Jelgenerátor funkciója maximum 1 MHz frekvenciájú jel generálására alkalmas, aminek kimeneti feszültségtartománya ofszet feszültséggel együtt maximum ±2 V tartományba eshet. Lehetőség van beépített jelek generálására is, és tetszőlegesen programozható jel generálására egyaránt.

A PicoScope négy mintavételi üzemmódban képes adatok gyűjtésére, amik a következők:

1) Block mode
2) ETS[10] mode
3) Rapid block mode
4) Streaming mode

1) Blokkos mintavételi üzemmód esetén az oszcilloszkóp először a saját belső buffer memóriájába gyűjti az adatokat, majd miután befejezte a gyűjtést

[5] Lásd: 3-1. ábrán Ch A, Ch B

[6] Lásd: 3-1. ábrán AWG (Arbitrary waveform generator)

[7] kS = kiloSamples

[8] GS/s = GigaSamples/sec

[9] MS/s = MegaSamples/sec

[10] Equivalent Time Sampling

továbbmásolja az adatokat egy számítógén lefoglalt memóriaterületre. Végül pedig, ha teljesen befejeződött a művelet, az adatok lekérdezhetőek és megvizsgálhatóak. Abban az esetben, ha újra szeretnénk indítani egy mérést vagy megváltoztatunk valamilyen beállítást, a lefoglalt memóriaterületről törlődni fognak az adatok. Mivel a PicoScope-nak szükséges körülbelül 50 msec beállítási idő egy adatgyűjtési blokk előtt, ennek következtében nem lehet ezzel az üzemmóddal folyamatos mérést végezni, ellenben egy adott mintaszámmal rendelkező rész begyűjtése esetén akár 1 GS/s mintavételi sebességre is képes.

2) Ekvivalens mintavételezési üzemmód lehetővé teszi, hogy ismétlődő jelek esetén az effektív mintavételi sebességet akár tízszer gyorsabb legyen, mint block mode esetén. Ez az üzemmód különbözik a normál blokkos beolvasás folyamatától.
3) Gyors blokkos mintavételi üzemmód egy olyan variációja a normál blokkos mintavételi üzemmódnak, ahol több blokk beolvasására van lehetőség egyszerre. Ennek következtében a mintavett blokkok közötti időrést drasztikusan le lehet csökkenteni. Amíg egy normál blokkos beolvasás esetén egy blokk begyűjtéséhez szükséges beállási idő akár 50msec is lehet, addig rapid block mode esetén 2microsec időrésnyire lehet csökkenteni két blokk közötti időintervallumot.
4) Folyamatos mintavételi üzemmód a blokkos beolvasásoktól eltérően először nem a PicoScope belső memóriájába gyűjti az adatokat, hanem rögtön a számítógépen lefoglalt memória területre. Ennek eredményeképpen valósidejű méréseket lehet végezni, aminek már csak a felhasznált PC memóriája szabhat korlátot.

Látható, hogy viszonylag széles lehetőséggel bíró eszközről van szó számos pozitív és előnyös tulajdonságokkal, azonban mivel a jelenlegi technológiai állás szerint nem a legfejlettebb, legnagyobb teljesítménnyel rendelkező eszköz, ezért egy ipari szintű speciális mérőműszerrel és jelgenerátorral szemben vannak hátrányai is.

Előnyök:

- Kicsi, kis helyet foglal
- kompakt, fizikailag egyben van a mérőműszer és a jelgenerátor

- hordozható
- nem igényel külön tápellátást
- viszonylag olcsó

Hátrányok:

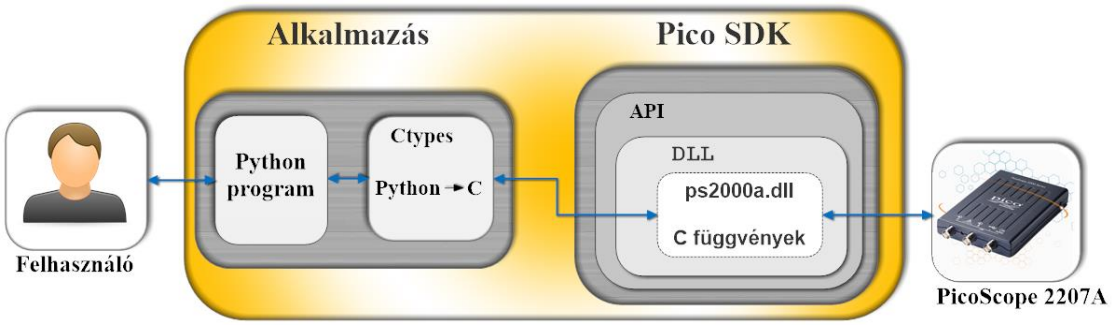
- Korlátos képességek
- Teljesítményében alulmúlja egy nagy, ipari, precíziós mérőműszerét
- Jelgenerátor kimeneti jele zajosabb, mint a különálló, ipari jelgenerátoroké

3.2 Szoftver

3.2.1 Könyvtári függvények

A gyártó könyvtári függvényeket biztosított mindenki számára, aki saját programot szeretne fejleszteni. Ezeknek a függvényeknek a leírását, illetve néhány példakódot, fejlesztési tanácsokat egy programozói útmutatóban találhatjuk meg. Ráadásul ingyenesen elérhetőek a gyártói honlapról[11] letöltve.

A szoftverfejlesztői csomag[12] tartalmaz egy 'ps2000a' nevű, DLL típusú fájlt, aminek segítségével alkalmazásprogramozási[13] felületen keresztül tudunk a programunkkal a PicoScope-hoz csatlakozni, majd vezérelni. Ennek a szerkezeti felépítését, kapcsolati elemeit az alábbi ábrán láthatjuk:



3-2. ábra Alkalmazás - PicoScope könyvtárak kapcsolata

[11] www.picotech.com/downloads

[12] SDK = Software Developement Kit

[13] API = Application Programming Interface

Az ábrán is látható módon a PicoScope-pal egy 'ps2000a.dll' nevű fájlon keresztül tudunk kommunikálni, ami C nyelvű függvényeket exportál. Ennek következtében ezeknek a függvényeknek a meghívása kizárólag csak olyan programnyelvekkel lehetséges, amelyek támogatják a C-konverziót. Mivel a Python támogatja a C nyelvű függvények meghívását, ezért ez volt az egyik szempont, ami szerint a Pythont választottam programozási nyelvnek. A C-konverziót végrehajtó modulnak a neve Ctypes. Ennek segítségével már tudunk C függvényeket hívni, amiket a dll fájl tartalmaz.

A következőekben könyvtári függvények általános alkalmazásának bemutatása következik, egy konkrét példán keresztül. Az alábbi példában szereplő kódrészlet a PicoScope azonosítását, majd összekapcsolását az alkalmazással eredményezi, aminek következtében megkapunk egy azonosító számot, amivel a későbbiekben a többi függvényt is használhatjuk.

`ps2000aOpenUnit()` függvény hívása:

1) Ctypes modul importálása:

```
from ctypes import *
```

2) DLL fájl betöltése:

```
mydll = windll.LoadLibrary("c:\\Program Files (x86)\\Pico Technology\\PicoScope6\\PS2000a.dll")
```

3) paraméterek definiálása:

```
serialNullTermStr = None
c_handle = c_int16()
PICO_STATUS = c_byte()
```

4) C függvény hívása:

```
PICO_STATUS = mydll.ps2000aOpenUnit(byref(c_handle), serialNullTermStr)
```

A függvények egy státusz értékkel térnek vissza, amit hexadecimális számok formájában kapunk meg. Ezeknek az értelmezéséhez a programozói útmutató végén található táblázat nyújt segítséget.

3.2.2 Python

A Python egy nagyon magas szintű programozási nyelv, mely egyaránt alkalmas általános, rövid, kiterjedt programok elkészítésére. Kifejezetten könnyen tanulható, hatékony nyelv, mely az olvashatóságot és a programozói munka megkönnyítését, gyorsítását helyezi előtérbe. A Python szintaxisa nagyon hasonlít a Matlab programozási nyelvéhez, ami elősegíti a nyelvnek a gyorsabb elsajátíthatóságát, amennyiben már korábban programoztunk Matlab nyelven. Ezen felül, egyszerű és letisztult felépítés jellemzi, ami mind elősegíti, hogy kiváló lehetőség legyen akár kezdő nyelvnek is. A Python egyaránt támogatja a funkcionális, procedurális, imperatív, és az objektumorientált programozási paradigmákat. Mivel interpreteres nyelv, továbbá elegáns szintaxissal, dinamikus típusossággal rendelkezik, ezeknek köszönhetően egy kiváló script nyelvről beszélhetünk. Tehát nincs különválasztva forrás és tárgykód, hanem byte kódot fordít, és azt futtatja. Ebből fakadóan azonnal futtatható. Maga a programozási környezet platformfüggetlen, azaz minden gyakran[14] használt operációs rendszeren fut. Azonban manapság már egyre inkább kezdenek megjelenni platformspecifikus tulajdonságok is. [5]

Kétféle Python programozási nyelv is elterjedt. Az egyik a 2.7-es verzió, a másik pedig Python 3, ahol ez nem az elsőként megemlített verziónak a továbbfejlesztett változata, hanem egy külön nyelv. Az általam használt verzió a 2.7.5 volt.[15]

A Python egyik legnagyobb előnye, hogy ingyenes és nyílt forráskódú, azaz mindenki számára egyformán elérhető és bárki továbbfejlesztheti. Ez mind a mai napig igaz, és rengetegen fejlesztik folyamatosan. Ezért egyre újabb és újabb készen elérhető programnyelvet kapunk, ami egyre több modullal rendelkezik. Ezeknek a segítségével, ha matematikai algoritmusokat, esetleg jelfeldolgozási algoritmusokat szeretnénk alkalmazni programunkba, akkor ezeknek a kódoknak a kifejlesztésre nem kell időt és energiát pazarolni, mivel már készen megtalálhatjuk, és fel is tudjuk használni. Jelfeldolgozási téren számos szoftvert alkalmaznak, különböző okoknál fogva mind az előnyeiket és hátrányaikat figyelembe véve. Az egyik legelterjedtebb a Matlab, mivel nagyon sok támogatással rendelkezik és széleskörű alkalmazási lehetőségeket nyújt.

[14] Például Windos, Linux stb.

[15] Letölthető: https://www.python.org/download/releases/2.7.5/

Ellenben viszonylag drága. Mivel sokat számítanak a költségek, ennek következtében remek megoldás lehet egy általános célú programozási nyelv, a Python.

A Python modulok gyakorlatilag olyan fájlok, amikben előre megírt kész függvények állnak rendelkezésünkre, amiket meghívva[16] az adott területen való munka megkönnyítésében nyújtanak segítséget. A félév során több ilyen modullal is részletesebben is megismerkedhettem. A következőekben néhány általam felhasznált modul bemutatására kerül sor:

1) A begyűjtött adatok kijelzéséhez használt modul a **Matplotlib**. Ez a modul függvények Matlabszerű ábrázolásában és grafikonok megjelenítésében, szerkesztésében nyújt széleskörű támogatást. Ennek a modulnak a felhasználása nagy előny volt, amikor a begyűjtött adatokat grafikonon szerettem volna megjeleníteni.

2) Alkalmazásom fejlesztését nagyban gyorsította a **Scipy** modul megléte. A programozható jelalak generálásánál, a pontosabb hullámforma kijelzése érdekében a modul által tartalmazott lineáris interpoláció nyújtott segítséget.

3) Feladatom során előkerült fontos modul még a **socket** volt. Ez a modul egyaránt támogatja az UDP és a TCP kapcsolat felépítését. Ez a modul tette lehetővé, hogy távolról is vezérelhető legyen a PicoScope.

4) C konverziót tartalmazó modul a Ctypes. Ez a modulcsomag DLL fájlok kezelését, megnyitását és onnan adatok beolvasást, továbbá C típuskonverziókat is lehetővé tesz. Ennek a modulnak a használata azért volt elengedhetetlen munkám során, ugyanis a C függvényeket tartalmaznak a könyvtári függvények.

3.2.3 GUI[17]

A python számos pozitív tulajdonságai mellett, amik az előző pontban bemutatásra kerültek, további előnyt jelent, hogy több lehetőséget is kínál grafikus kezelői felület elkészítéséhez.

---

[16] Importálva import parancs által

[17] ***graphical user interface = grafikus felhasználói felület***

Az egyik lehetőség a **PyGTK**. Nagyon kifinomult és dokumentált, ennek következtében remek választás lehet. A legtöbb Linux disztribúcióban már megtalálhatóak. Azonban semmi sem tökéletes, így ennek is van egy nagy hátránya, hogy Windows operációs rendszerre telepítése nem a legegyszerűbb.

Másik lehetőség a **PyTk** lehet. Ennek az előnye az előző esettel szemben, hogy Windows alatt a Pythonnal együtt megtalálható, így az elkészült programunk Windows alatt biztosan futtatható. Azonban Linux alatt nem alapértelmezetten telepített, így arról nekünk kell gondoskodni. Legnagyobb hátránya viszont, hogy nem túl jól dokumentált, és nem a legfejlettebb megoldás.

Harmadik lehetőségünk a **PyQt**. Egyik előnye, hogy Windows, Linux és Mac operációs rendszerek alatt egyaránt elérhetőek. Kifejezetten kifinomult, elegáns kinézettel rendelkezik. Számos lehetőséget rejt magában.

Ezeken kívül még több lehetőség is adott, például wxWidget, azonban a választásom az előnyöket összevetve végül a PyQt-ra esett, egészen konkrétan a PyQt4 verziójára. [6] Feltelepítve a számítógépre találhatunk egy Designer nevű programot, ami gyakorlatilag egy grafikus szerkesztő program, aminek segítségével össze tudjuk állítani a kívánt programunk kinézetét. Adottak alap eszközkészletek külön témánként felbontva, például gombok, jelölőnégyzetek, szövegbeviteli mezők, kijelzők és még rengeteg egyéb. Ezeknek a segítségével el tudjuk készíteni a kívánt programunk kinézetét. Ezek után, ha szükséges, akkor lehetőségünk van úgynevezett jelek elhelyezésére két objektum között, amiknek segítségével ki tudjuk választani, hogy milyen kapcsolatban álljanak egymással. Például ha be szeretnénk állítani, hogy az egyik gomb megnyomására a kijelző törlődjön, akkor azt megtehetjük úgy, hogy a gombtól a kijelző fele behúzunk egy jelet és a felugró listából kiválasztjuk a megfelelő végrehajtandó műveleteket. Először a kiinduló oldalon, azaz a gombra vonatkozóan, ha rákattintunk, az esemény bekövetkezik, akkor csináljon valamit. Majd kiválasztjuk a jelnek a túloldalán a bekövetkezendő eseményt, azaz a példa szerint, hogy törlődjön a kijelző. Amennyiben viszont bonyolultabb feladatok végrehajtását szeretnénk elvégeztetni, komplexebb összefüggésekben, akkor ebben az esetben már Designer program nem kínál fel lehetőséget, így azokat magunknak kell megírni az elkészül kódunkban. Miután kitaláltuk a kinézeti koncepciót és elrendeztük a gombok és egyéb

Widgetek[18] helyét, lehetőségünk van a kinézetet is szépíteni. Ugyanis egészen a feliratok szövegszerkesztő szintű formázásától a háttérkép, színek és formák szerkesztésére is lehetőségünk van. Miután mindennel elkészültünk a program értelmezi és generál számunkra egy továbbfejleszthető és felhasználható Python kódot.

[18] Gombok, kijelzők stb. összefoglaló néven

4 Szoftverintegráció

Előző fejezetek alapján megismerhettük hogyan épül fel a mérőrendszer, milyen elemekből tevődik össze, azok milyen tulajdonságokkal rendelkeznek, milyen feladatot látnak el. Ezzel szerezhettünk egy rendszerszintű átlátást a feladatra vonatkozóan. A következő néhány fejezetben a mérőrendszer vezérlését végző kezelői szoftvercsomag fejlesztésének és felépítésének részletes bemutatását következik. Ebben a fejezetben megismerhetjük, milyen részegységekből tevődik össze a program, és ezek hogyan integrálódtak egységes rendszerré.

4.1 Applikáció felépítése

Megismerhettük, hogy a program Python nyelven készült grafikus kezelői felület szerkesztőprogram segítségével. Miután a program kinézetét elkészítettem, majd megformáztam, az elkészült felületet Python kódját legeneráltam. Az így megkapott kódot egy design.py nevű fájlba mentettem, amit később, mint modul beimportálva a főprogramba tovább tudtam fejleszteni. Magának a főprogramnak a felépítése a következőképpen néz ki:

1) Widgetek vezérlését és kezelését szolgáló osztály felépítése először az osztály létrehozásával az örökölt ősosztályok megnevezésével, majd a konstruktor létrehozásával történik:

```
class ControllApp(QtGui.QMainWindow, design.Ui_MainWindow):
    def __init__(self):
        super(self.__class__, self).__init__()
        self.setupUi(self)
```

Egyéb függvények is ebben az osztályban lettek definiálva, amiket a kezelői felületen található Widgetek által létrehozott események aktiválnak. Például egy blokkos beolvasáshoz szükséges mintavételi időalap beállítására szolgáló mező bemutatásával lehet szemléltetni, hogyan is épülnek fel azok a függvények. Először is fel kell iratkozni egy eseményre a megfelelő signal[19] alkalmazásával.

[19] Jel

A példában alapján létrehozunk egy olyan jelet, ami akkor fog eseményt generálni, ha a mezőre kattintva az enter billentyűt lenyomtuk:

```
self.lineEdit_timebase_A.returnPressed.connect(self.lineEdit_tim
ebase_A_event)
```

Az esemény függvénye a következőképpen néz ki:

```
def lineEdit_timebase_A_event(self):
    global timebaseA
    timebaseA = unicode(self.lineEdit_timebase_A.text())
    self.textBrowser.append("<span>timebase:</span>")
    self.textBrowser.append(timebaseA)
    s.mysend("Sent_timebaseA")
    s.mysend(timebaseA)
```

2) Ezután következett a szálkezelt programrészek definiálása, ahol a run() függvényhez írt kódrészlet fog végrehajtódni egy külön szálon:

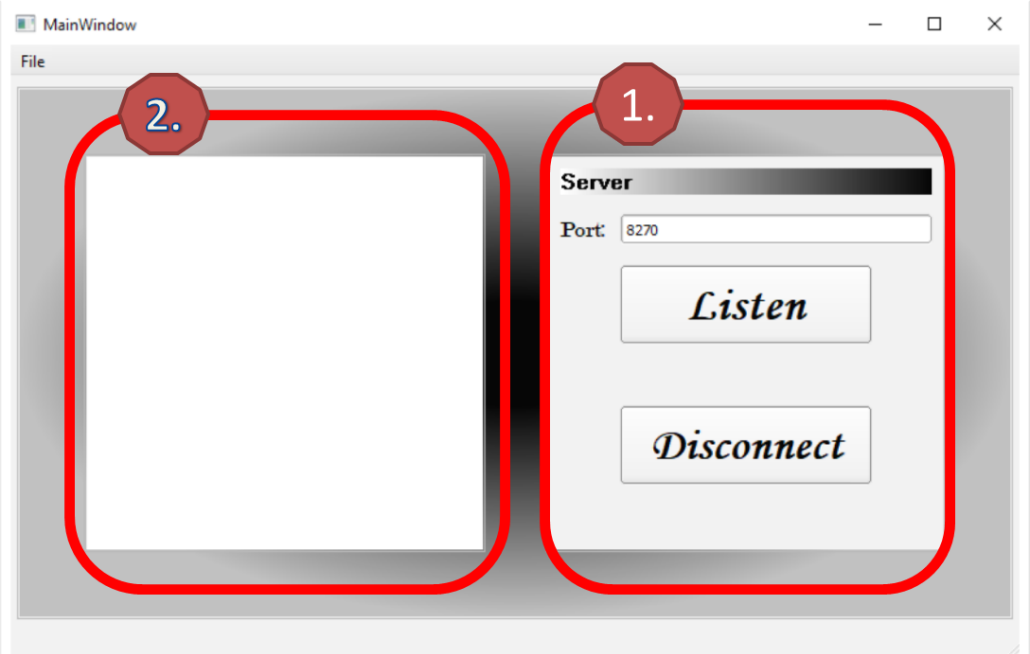
```
class ClientThread_3(QThread):
    def __init__(self):
        QThread.__init__(self)
    def __del__(self):
        self.wait()
    def run(self):
          (...)
```

3) Legvégül pedig a főprogram látható, ami a következőképpen néz ki:

```
def main():
    app = QtGui.QApplication(sys.argv)
    form = ControllApp()
    form.show()
    app.exec_()
```

4.1.1 Szerver oldal

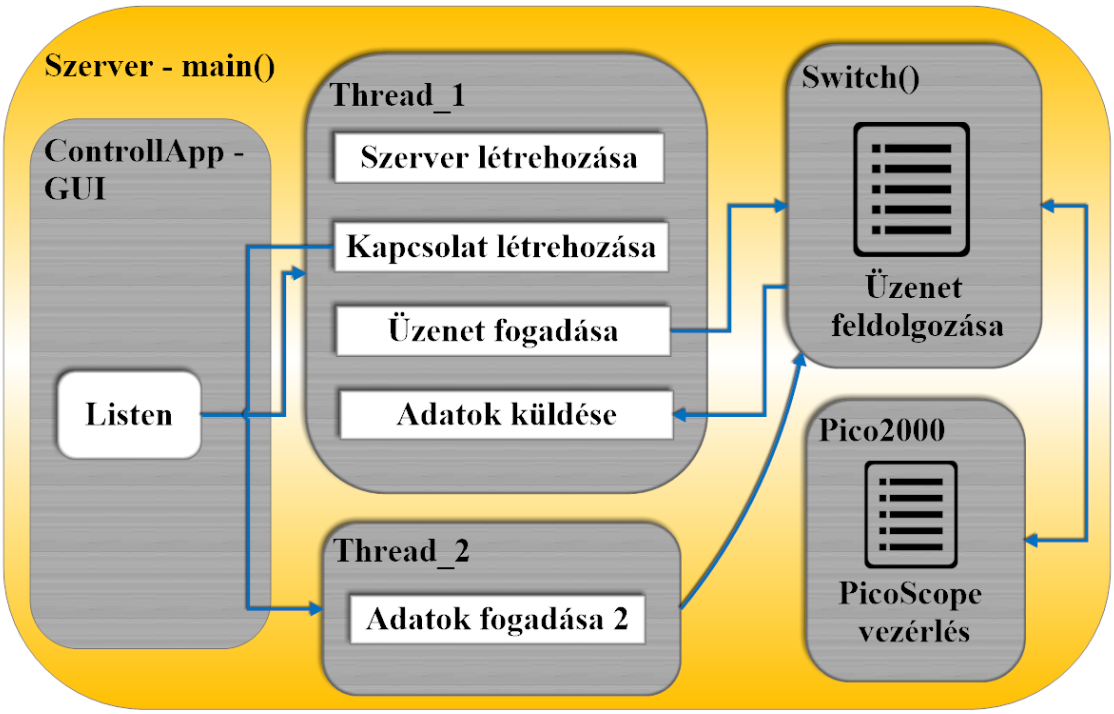
A szerver oldal kommunikál a PicoScope-pal, így lényegében itt történik a mérés lényegi része. A következő ábrán láthatjuk a szerver oldali kezelői felület kinézetét és elrendezési struktúráját:



4-1. ábra Szerver oldali kezelőfelület

Az 1.) jelöli az kezelői részt, ahol port megadása, socket létrehozása és kapcsolat bontása lehetséges. A 2.) rész a visszajelző ablak, ahol a program a felhasználó számára érdekes információkat képes megjeleníteni.

A programstruktúra szemléletesebb megértését az alábbi ábra szolgálja:



4-2. ábra Szerver oldali programstruktúra

Az ábrából láthatjuk, hogy miután a Listen gombot aktiváltuk – ami program struktúrailag a main függvény ControllApp osztályában helyezkedik el – a kattintás által létrejött esemény elindítja az **első szál**at[20], (ami programozástechnikailag a második szál, mivel az elsőn a főprogram és a GUI fut, azonban tekintsük azt nulladik szálnak) ami először végrehajtja a hálózat szerver oldali részét socketek segítségével. Miután a kliens is csatlakozott és létrejött a kapcsolat, elindul a **második szál**[21]. Ez a szál a TCP kapcsolat alatt végig várja a beérkező üzeneteket és feldolgozza azokat. Amint Streaming mintavételi üzemmódot indítottunk, és annak leállító parancsa beérkezik, leállítja a folyamatos adatgyűjtést. Erre azért volt szükség, mivel elsődlegesen az első szálon várjuk és dolgozzuk fel az adatokat, azonban, ha folyamatos adatgyűjtés indul, akkor az a szálat teljesen lefoglalja, így nem lehetne leállítani. Visszatérve az **első szál**on futó folyamatokra miután felépült a kapcsolat a klienstől beérkező üzeneteket várja, majd ha érkezett egy, akkor azt továbbküldi egy feldolgozó egységnek, ami végrehajtja az üzenetben szereplő parancsokat. Ez a rész[22] egy külön modulban lett megírva, ahol csak az üzenetek értelmezése történik. Amint kiderült, hogy milyen parancs érkezett be, meghívja az üzenetnek megfelelő PicoScope vezérlő függvényt, ami végrehajtja a műveletet. Az lehet például egy paraméterbeállítás vagy akár mérésindítás is. Ezek a függvények szintén egy külön modulban lettek megírva[23]. Végül, ha lefutott egy adatgyűjtési ciklus, a begyűjtött adatokat a hálózaton keresztül elküldi a kliens oldalnak, ahol további kiértékelést vagy utófeldolgozást lehet végrehajtani.

Így épül fel a szerver oldal egyes részelemeinek rendszerbe való integrálásával.

4.1.2 Kliens oldal

Kliens oldal feladata a felhasználóval való közvetlen kapcsolat megteremtése és lekezelése. Tehát magának a funkciók beállításainak a lehetőségei itt találhatóak meg.

---

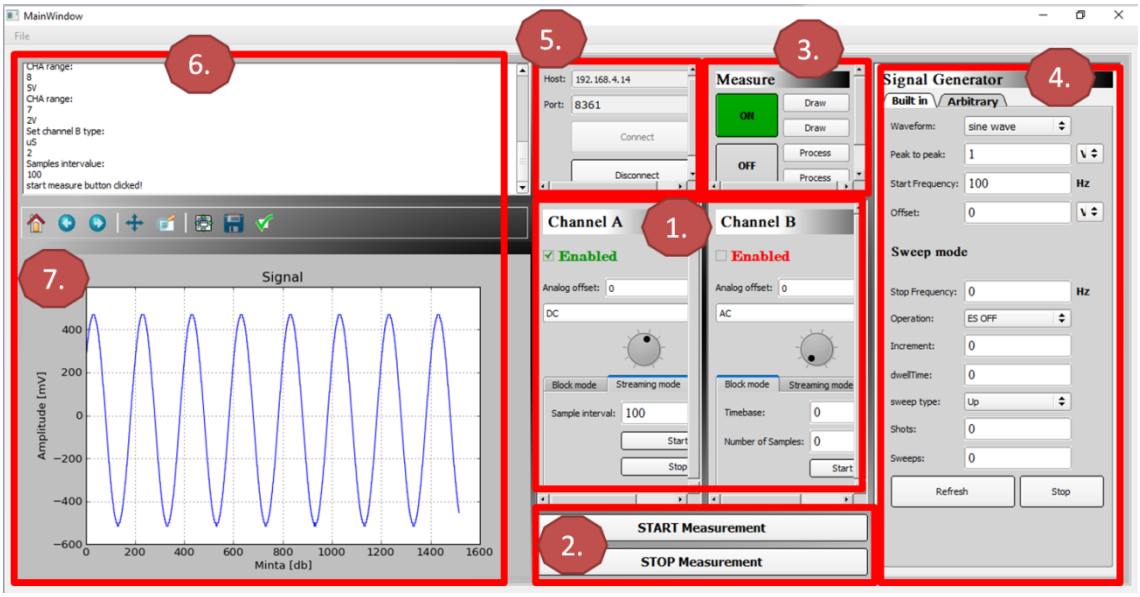
[20] Thread_1

[21] Thread_2

[22] 4-2. ábrán switch()-el jelölt

[23] 4-2. ábrán Pico2000-rel jelölt

Kliens oldal grafikus kezelői felület elrendezése és kinézete a következő ábrán látható:



4-3. ábra Kliensoldali kezelői felület

A kezelői felület két nagyobb logikai részre bontható szét. Az egyik a beállításokkal, foglalkozó rész:

1.) Az első beállítási lehetőség az adatgyűjtéshez kapcsolódik, ahol a mintavételezés üzemmódját és hozzá tartozó paramétereket tudjuk kiválasztani. Az itt indított mérésekkel úgynevezett vizsgáló üzemmódra van lehetőségünk. A paraméterek beállítása, mérés indítása után, az eredmény kiértékelésével, ha nem sikerült a beállítás, akkor tudunk változtatni.

2.) Miután a megfelelő beállítások sikerültek, lehetőségünk van hosszú távú folyamatos mérés indítására, ahol már csak elindítjuk a mérést, majd leállítás után egyben megkapjuk a begyűjtött adatokat.

3.) A begyűjtött adatokat megjeleníthetjük vagy utófeldolgozást végezhetünk.

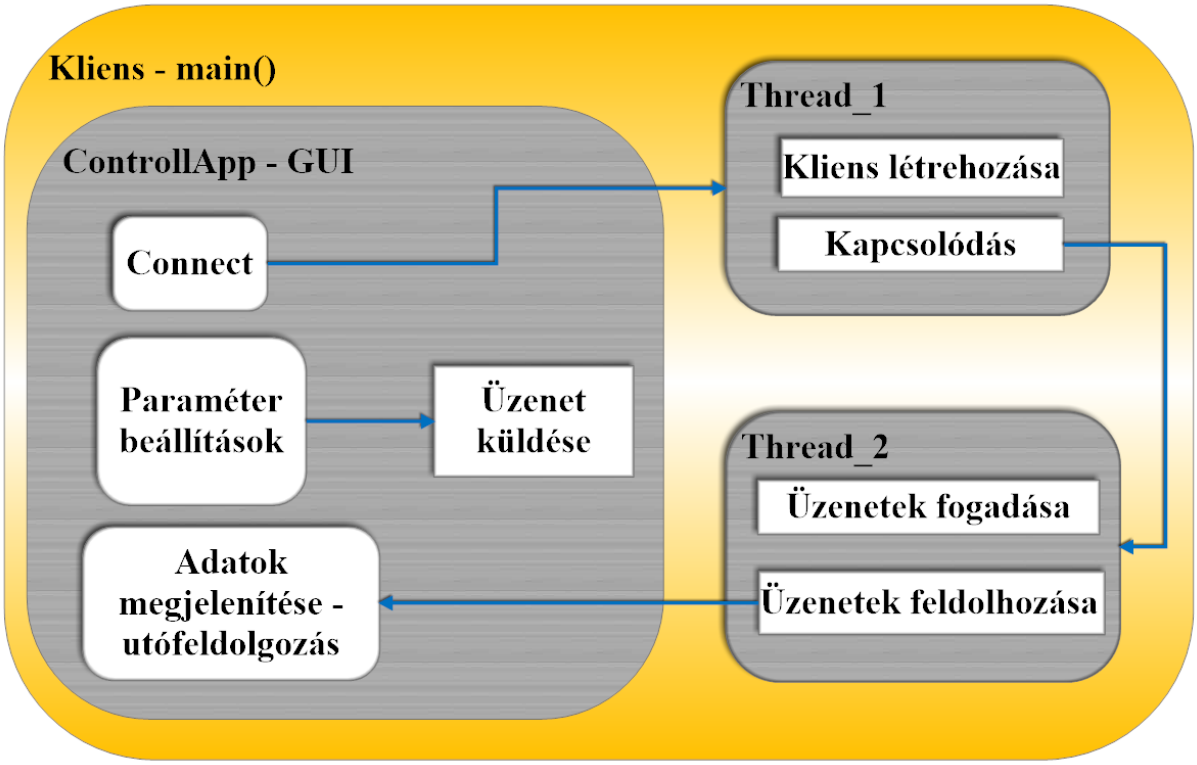
4.) Amennyiben szükséges, jelek generálására is lehetőség van, beépített jelek vagy akár tetszőlegesen programozató jelalak generálására. Ezeket a beállításokat a felület jobb oldalán megtalálhatjuk.

5.) A beállítások alá sorolható a hálózati kapcsolat létrehozásához szükséges beállítási felület.

Másik nagyobb logikai egység a visszajelzéssel, kijelzéssel kapcsolatos:

6.) Ide tartozik a felhasználó felé visszajelzést közvetítő felület, ahol meggyőződhet róla a felhasználó, hogy valóban rákattintott az adott gombra, és a műveletet a program végre is hajtotta.
7.) Végül pedig a kijelző rész, ahol a kívánt adatok megjelenítésére grafikusan is lehetőség van.

A kliens oldali programstruktúra részletesebb megértését az alábbi ábra szolgálja:



4-4. ábra Kliens oldali programstruktúra

Az ábra alapján megfigyelhetjük, hogy lényegében három nagyobb rész integrálásával készült el a kliens oldali szoftver. Ez a három egység párhuzamosan, egymástól független szálakon dolgoznak. A főprogramban találhatunk egy ControllApp nevű osztályt, ami tartalmazza a beviteli Widgetek által generált eseményeket. Miután rákattintottunk a Connect gombra elindul az első szál[24] (szerver oldalhoz hasonlóan a GUI-n futó szálat tekinthetjük nulladiknak), ahol felépül a hálózati kapcsolat.

[24] Thread_1

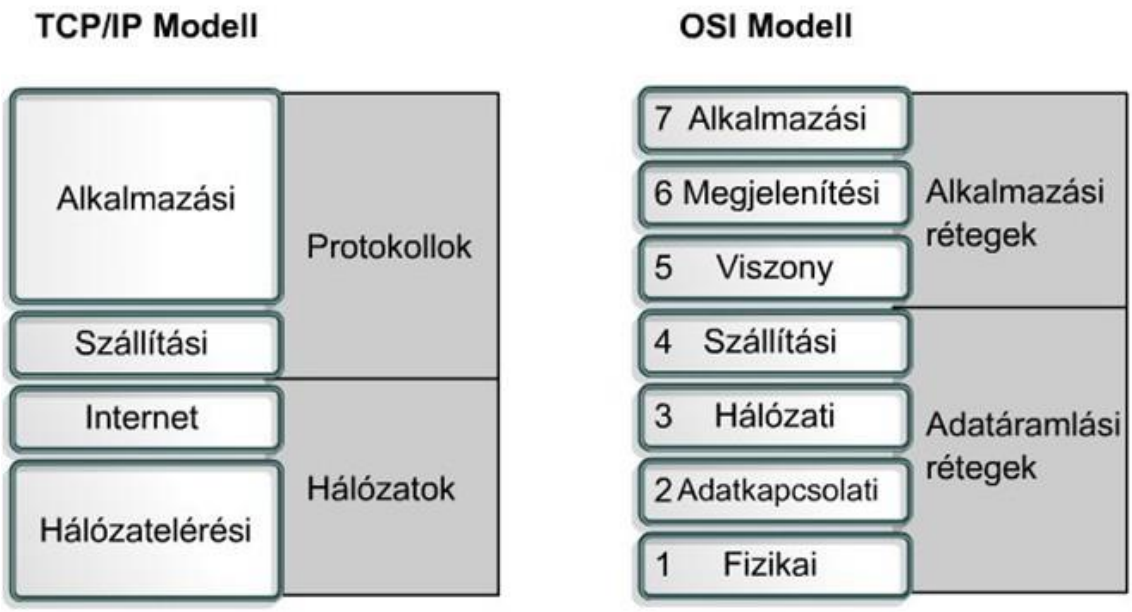
Miután felépült a hálózati kapcsolat, elindul a második szál, ahol folyamatosan várjuk a bejövő üzeneteket a szerver oldaltól, amik lehetnek begyűjtött adatok vagy visszajelző üzenetek. Ha érkezett egy üzenet, akkor azt feldolgozzuk és az üzenet tartalmának megfelelően vagy megjeleníthetjük a begyűjtött adatokat vagy további műveleteket hajthatunk végre.

4.2 Hálózat felépítése

Korábban bemutatott szerver és kliens oldalalkat a hálózat kapcsolja egységes rendszerré. A számítógép-hálózatok lehetővé teszik elosztott alkalmazások fejlesztését. A felhasználók által alkalmazott hálózati eszközöket végrendszereknek, a hálózatot pedig szolgáltatásinfrastruktúrának nevezzük, ugyanis szolgáltatás, hogy a számítógépek egymással tudnak kommunikálni. A számítógépes hálózat szemléltetése az alábbi ábrán megfigyelhető:

4-5. ábra Számítógépes hálózat

A számítógépes hálózatok rétegzett architektúra alapján írhatóak le. Az egyes rétegeket protokolloknak nevezzük. Az egyetemi tanulmányok során két hivatkozási modellel is megismerkedhetünk, az egyik az úgynevezett OSI modell, másik pedig a TCP/IP. Munkámhoz az utóbbit választottam. TCP/IP hálózati protokoll felépítését az alábbi ábrán vehetjük szemügyre:

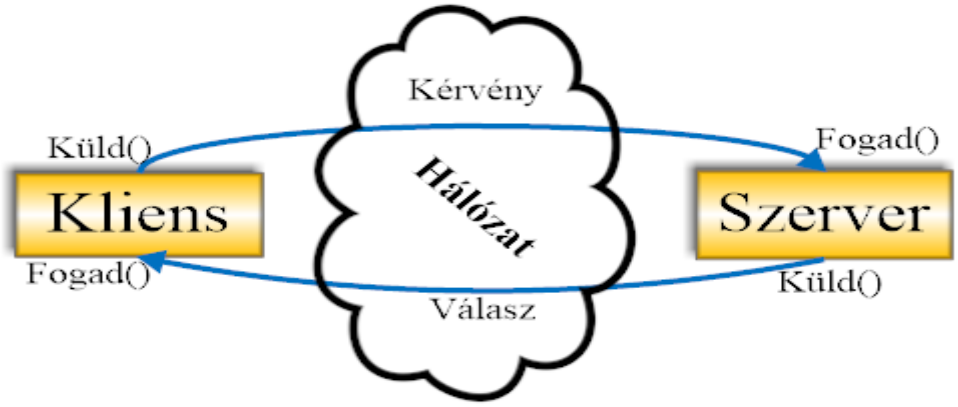


4-6. ábra OSI és TCP/IP modell összehasonlítása [8]

Az egyes rétegek részletes kifejtése, bemutatása szakdolgozatomnak nem része, azonban, ha érdeklődés mutatkozik a téma felé, akkor mélyebb rálátást és szemléletet kaphatunk számos fellelhető szakirodalom tanulmányozása által. [7] [8]

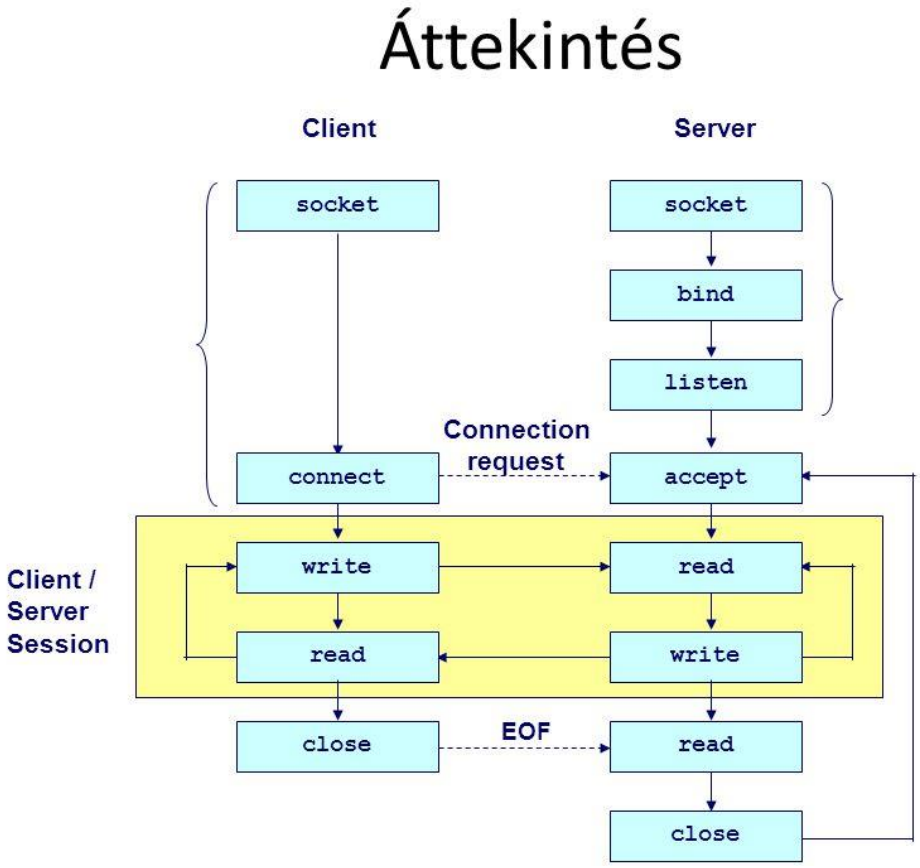
A szállítási réteghez sorolhatjuk a socketek segítségével hálózati kapcsolat-felépítést, aminek megértését az alábbi ábrák segíthetik:

1) Már felépült kapcsolati kommunikáció:



4-7. ábra Hálózati kommunikáció

2) Socketek segítségével a hálózati kapcsolat felépítése:



4-8. ábra Hálózati kapcsolatépítés [10]

A 4-8. ábrán látható módon lehet szerver kliens kapcsolatot létrehozni. Egészen a kapcsolat felépítésétől a már aktív kapcsolat alatt lévő kommunikációnak a lezajlási folyamatán át – szemléletesebben lásd: 4-7. ábrán – a kapcsolat bontásáig megfigyelhetünk az ábrák alapján minden lényegi információt.

4.2.1 Szerver oldal létrehozása Pythonban

Ahogyan már a 3.2.2. fejezetben megismertük hálózati kapcsolat felépítéséhez rendelkezésünkre áll a socket modul. Ennek segítségével már csak a megfelelő sorrendben kell meghívni a függvényeket a szerver oldal felépítéséhez. A szerver oldal felépítéséhez szükséges kódrészletek a következők:

```
import socket
host = ''
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
print host , port
s.listen(1)
conn, addr = s.accept()
print('Connected by', addr)
while True:
    try:
        data = conn.recv(1024)
        if not data: break
        print "Client Says: "+data
        conn.sendall("Server Says:hi")
    except socket.error:
        print "Error Occured."
        break
conn.close()
```

Röviden összefoglalva:

1) szerver socket létrehozása (socket)
2) hálózati cím, port hozzárendelése (bind)
3) passzív állapot kapcsolódáshoz (listen)
4) kliensek csatlakozása (accept)
5) adatok fogadása / küldése (write =send / read = recv)
6) socket lezárása (close)

4.2.2 Kliens oldal létrehozása Pythonban

Hasonlóan az előző fejezethez a 4-7. ábra és 4-8. ábra alapján a kód felépítése a következő:

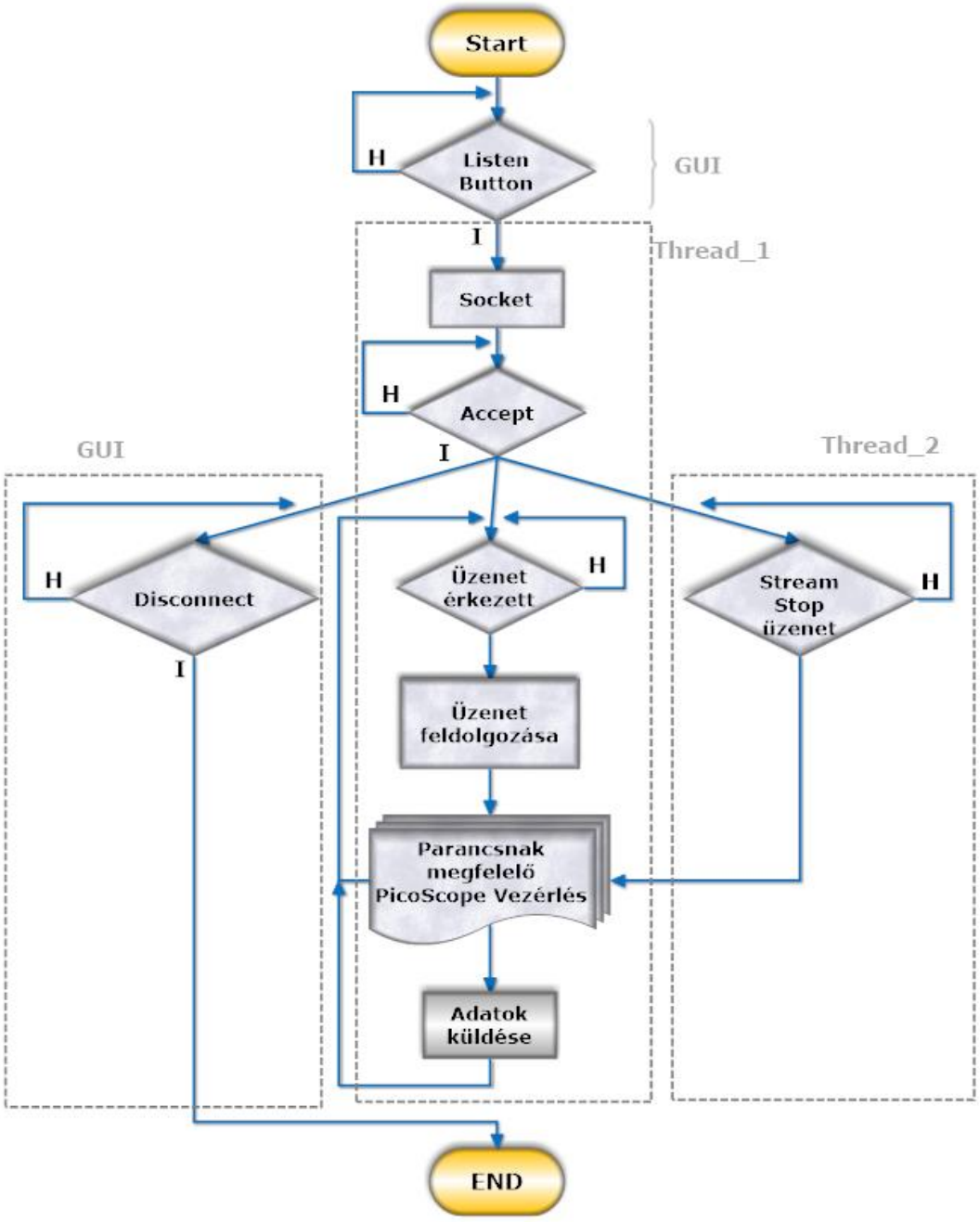
```
import socket

host = socket.gethostname()
port = 12345
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))
s.sendall(b'Hello, world')
data = s.recv(1024)
s.close()
print('Received', repr(data))
```

Röviden összegezve néhány lépésben:

1) új socket létrehozása (socket)
2) kapcsolódás egy szerverhez
3) adatok küldése / fogadása (write =send / read = recv)
4) socket lezárása (close)

5 Szerver oldali szoftverfejlesztés

Előző fejezetben megismerhettük egységében a programcsomag szerver oldali és kliens oldali rész felépítését, és egymással való kapcsolatát, együttes rendszerként való működési elvét. Ebben a fejezetben a szerver oldali programot felépítő részegységek részletes elemzésével és ismertetésével találkozhatunk.



5-1. ábra Szerver oldal folyamatábrája

A 4-2. ábra szerint láthattuk, milyen részekből tevődik össze a program, azok hogyan viszonyulnak egymáshoz. Rögtön leolvashatjuk, hogy szálkezelt programról van szó, ahol párhuzamosan futó eseményvezérelt programstruktúrával állunk szemben. Azt is láthatjuk, hogy milyen kapcsolatban állnak egymással a szálak, a szálak milyen részekből, feladatvégző modulokból tevődnek össze. Azonban magának a program futásának folyamatát leginkább maga a folyamatábra szemlélteti a legjobban.

5.1 Folyamatábra értelmezése

Az 5-1. folyamatábrán három oszlopot láthatunk, amik lényegében a három szálat jelentik. Az első oszlop a főprogram által futtatott GUI szál, míg a középső oszlop a 4-2. ábrán is láthatott Threar_1, a jobb szélső oszlop pedig a Thread_2 szálakat jelentik.

A start esemény a program elindítását jelenti, amikor a 4-1. ábra szerinti kezelői felületet láthatjuk. A következő lépés úgy értelmezhető, hogy a program addig várakozik, amíg rá nem kattintott valaki a Listen gombra. Ha rákattintott a gombra, akkor elindul a Thread_1 szál, ahol felépül a 4.2.1 pontban ismertetett szerver socket és addig várakozunk tovább, amíg a kliens nem csatlakozott. Ha már létrejött a szerver-kliens kapcsolat (Lásd: 4-8. ábrán Session rész) akkor 3 szálon párhuzamosan futva folytatódik a program.

1) Az egyik a GUI szál, ahol lehetőségünk van szétcsatlakoztatni a hálózatot.
2) Második szálon történik lényegében az egész programnak a feladat elvégzései. Az üzenetek fogadása, feldolgozása (Lásd: 4-2. ábrán switch modul), üzenetben szereplő parancs végrehajtása (Lásd: 4-2.ábrán Pico2000 modul), ezáltal PicoScope vezérlése, majd ha szükséges, akkor üzenet küldése a kliensnek.
3) Harmadik szál szintén adatok fogadásával foglalkozik, viszont az üzenetek feldolgozása után csak a streaming mintavételi üzemmód leállítása lehetséges.

5.2 Feldolgozó egység bemutatása

Ez a rész teszi lehetővé, hogy a konfigurálási adatok átvitele után az üzenet értelmezése segítségével a konfigurálás művelete végre is hajtódjon. Az előző pontban láthattuk, hogy amint felépült a kapcsolat, a Thread_1 szálon üzenetek fogadása történik, majd amint beérkezett egy üzenet az feldolgozásra kerül és utána valamilyen feladat-végrehajtási procedúra kezdődik. Ebben az alfejezetben a programunkban található switch modul bemutatására kerül sor.

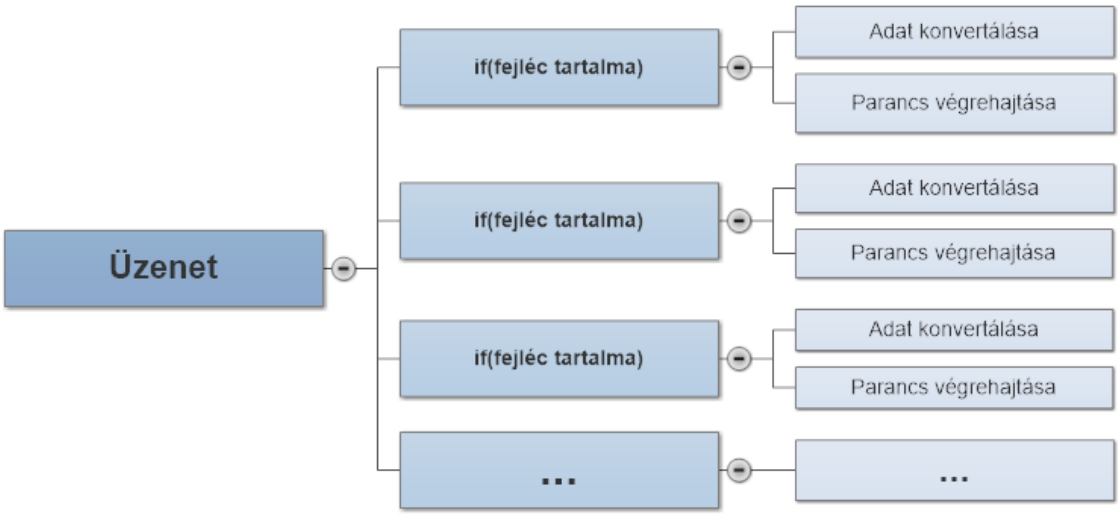
5.2.1 Üzenetfeldolgozó egység felépítése

A switch modul egyetlen egy függvényből áll, ami három bemeneti paramétert vár. Az első az üzenet, második a socket, arra az esetre, ha üzenetet szeretnénk küldeni. Végül pedig a harmadik a GUI ablak példánya, ami ahhoz kellhet, hogy ha valamilyen widget által a kezelői felületen műveletet szeretnénk végrehajtani, akkor az átvett példány segítségével a művelet végrehajtható legyen. Továbbá másik indok, hogy amikor a Thread_2 szálon beérkező üzenet szerint le kell állítani az adatgyűjtést, akkor a főprogramban létrehozott globális változó értékét ezen a paraméteren keresztül el tudjuk érni. A függvénytörzs kódja a következőképpen áll össze:

```
def switch(data, Conn, Form):
    ''' függvénytörzs '''
    (...)
    if data == "Sent_timebaseA":
    (...)
    if data == "Sent_dimmodA":
    (...)
    if data == "start_collect_A_block":
    (...)
```

Mivel a Python programozási nyelv nem támogatja a switch-case szerkezetet, ezért annak logikai felépítését alapul véve állítottam össze a függvényt if kiválasztó parancsok segítségével. A függvénytörzs felépítése először globális paraméterek kezdeti értékadásával kezdődik, amik a PicoScope beállításaiért lesznek felelősek. Utána pedig if szerkezetek sokasága következik.

Összegezve elmondható a működésről, hogy először egy kiválasztási, szűrési részen megy át a beérkező adat, majd a fejlécben szereplő parancs szerint történik a feldolgozás és vezérlés végrehajtása. Ennek a fésűs szerkezetnek a felépítését az alábbi ábra szemlélteti:



5-2. ábra Üzenet feldolgozó felépítése

5.2.2 Üzenetfeldolgozó egység működése

Ahhoz, hogy megérthessük, hogyan is működik az üzenetek feldolgozása, érdemes először megvizsgálni magának a megkapott üzeneteknek a felépítését. Üzenet felépítését az alábbi ábra szemlélteti:

5-3. ábra Üzenet felépítése

Az ábrán látható módon egy parancsvezérelt üzenet több részkomponensből tevődik össze. Először egy fejlécnek definiálható üzenetrész érkezik, amit első rétegben egy egyszerű összehasonlító műveletsorozat segítségével ki tudjuk szűrni, hogy melyik blokkban kell továbbfolytatni a műveletsorozatot. A fejléc felépítéséből fakadóan nevezhetnénk akár parancsnak, utasításnak is, ugyanis pontosan tartalmazza, hogy az adat kihez tartozik. Tekintsük meg a következő példát az érthetőség szempontjából:

A kliens oldalon kívánt tevékenységet, például amikor egy mintavételi időalapot szeretnénk blokkos adatgyűjtéshez beállítani, akkor az üzenet a következőképpen fog összeállni:

- Fejléc: `Timebase_Blockmode_A`
- Adat: 1250
- Üzenet vége: —

Ebből láthatóan a fejléc tartalmazza, hogy egy beállításhoz tartozó adat fog érkezni. Pontosabban az A csatornához tartozó blokkos mintavételi üzemmód időalapjához tartozó érték. Ebből következően ennek a parancsnak megfelelő vezérlési blokkot fog végrehajtani az ehhez az üzenethez tartozó feldolgozóegység. Fontos kiemelni, hogy csak string formátumban közlekednek az adatok a hálózaton. Ezért a feldolgozó rész először az adatot számformátumra konvertálja, majd a PicoScope vezérlését tartalmazó Pico2000 modulból kiválasztja a megfelelő vezérlőfüggvényt, és azt a beérkezett adat segítségével, mint függvényparaméter már meg lehet hívni. Végül pedig üzenet vége részre nem minden esetben van szükségünk (alapul véve látható a jelenlegi példa), ugyanis ha tudjuk, hogy a fejléc után csak egy adat fog érkezni, akkor az adat lesz az üzenet vége.

5.3 PicoScope vezérlő interfész

Ez a rész az elkészült mérőrendszeren belül a PicoScope 2207A típusú oszcilloszkóppal közvetlen kapcsolatban áll. Ezen keresztül lehet vezérelni a mérőműszert a klienstől kapott parancsoknak megfelelően. Itt történik minden olyan funkció, beállítás, a mérésindítástól a jelgeneráláson keresztül minden, amit a kliens oldali kezelői felületen be lehet állítani. Ebben a modulban (4-2. ábra szerint Pico2000) lett megírva minden olyan függvény, ami a PicoScope konfigurálásához, vezérléséhez szükséges. Az 5-1. ábra szerint a parancsnak megfelelő PicoScope vezérlés rész ide helyezhető el. Ez a modul a kommunikál a PicoScope-pal a 3-2. ábra szerint a 3.2.1.-es fejezetben kifejtett könyvtári függvények segítségével.

Minden egyes szükséges könyvtári függvényt, ami a szoftvercsomagban szereplő feladatok elvégzéséhez kellhetek, egy-egy általam írt függvény segítségével tettem a többi programrész számára kezelhetővé. Ezekkel a függvényekkel építettem fel a Pic2000 modult.

Egy csatornabeállító függvény hívása:

```
Pico2000.channel(ChA,ChAen,channelAtype,channelArange,0)
```

Egy időalap beállításához szükséges függvény felépítése:

```
def Timebase(timebase, NoSamples, segmentIndex):
    c_timebase = c_uint32(timebase)
    c_noSamples = c_int32(NoSamples)
    c_timeIntervalNanoseconds = c_int32()
    c_oversample = c_int16(0)
    c_maxSamples = c_int32()
    c_segmentIndex = c_uint16(segmentIndex)
    PICO_STATUS = mydll.ps2000aGetTimebase(c_handle, c_timebase,
c_noSamples, byref(c_timeIntervalNanoseconds), c_oversample,
byref(c_maxSamples), c_segmentIndex)
```

5.4 Adatgyűjtő üzemmódok

Már a 3.1.2. pontban megismerhettük, hogy milyen mintavételi üzemmódokkal is rendelkezik az oszcilloszkóp. Ezek közül a blokkos és folyamatos mintavételi üzemmódok készültek el. Blokkos üzemmód előnye, hogy ha tudjuk, hogy mennyi adatot szeretnénk gyűjteni vagy mennyi ideig szeretnénk gyűjteni, akkor az időalap és mintavett adatok számából már el is lehet végezni a mérést. Azonban, ha folyamatos mérésre lenne szükségünk anélkül, hogy tudnánk mennyi adatot szeretnénk begyűjteni, akkor arra is lehetőséget ad a mérőrendszer.

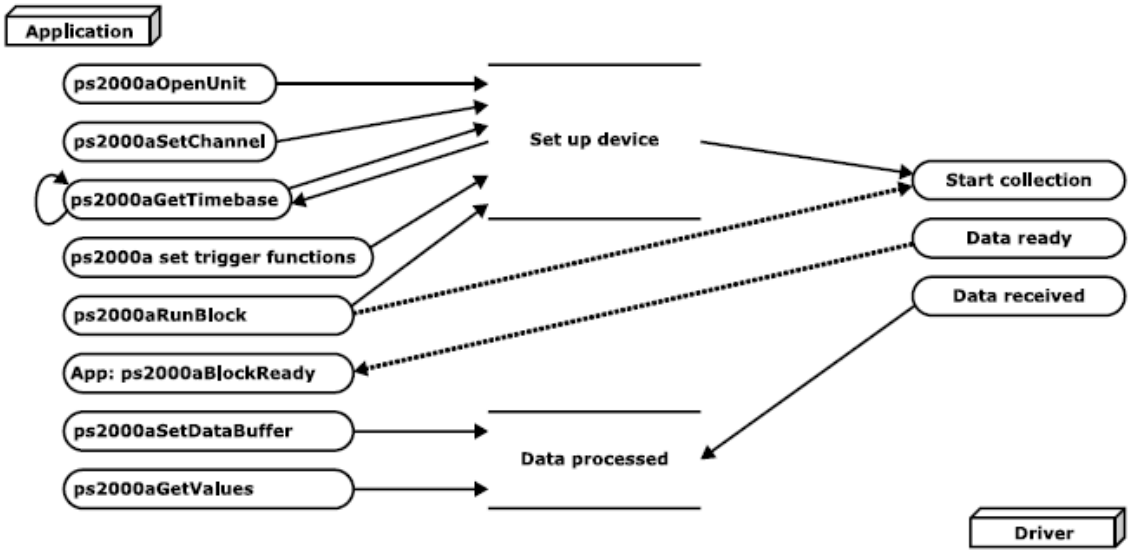
Általánosan egy adatgyűjtés indítása a következő lépéseket tartalmazza:

1) PicoScope csatlakoztatása, megnyitása
2) Csatornák beállítása
3) Trigger beállítása
4) Gyűjtés indítása
5) Várunk, amíg az oszcilloszkóp kész a gyűjtéssel
6) Gyűjtés leállítása
7) Adatok kimásolása további felhasználáshoz
8) PicoScope bezárása

Ezeket a lépéseket a Pico2000 modulban megírt vezérlőfüggvények valósítják meg. Azonban elhelyezés szempontjából ezek a vezérlőfüggvények a kódon belül a 4-2. ábra szerint a switch modulban helyezkednek el, a szűrés után kiválasztott résznél. Így ez a két szerkezeti elem szorosan egymáshoz kapcsolódva működik.

5.4.1 Blokkos adatgyűjtés

A blokkos adatgyűjtés felépítése tömören és szemléletesen a következő ábrán látható:



5-4. ábra Blokkos mintavételi üzemmód felépítése [5]

Az általános adatgyűjtési felépítéshez hasonlóan a blokkos adatgyűjtés is a PicoScope megnyitásával kezdődik. A függvény először azonosítja az oszcilloszkópot, majd a számítógépünkkel az API-n keresztül a PicoScope-hoz csatlakozik. Kimeneti paraméterként megkapjuk az oszcilloszkóp egyedi azonosítóját[25], amivel a későbbi függvényhívásokhoz kelleni fog. Visszatérési értékként pedig, mint a többi függvény is egy státuszbittel tér vissza, amit egy táblázat segítségével értelmezve megtudhatjuk, ha hiba történt, illetve annak okát. A függvény kódrészlete a következő:

```
def open():
     (...)
    PICO_STATUS = c_byte()
    PICO_STATUS = mydll.ps2000aOpenUnit(byref(c_handle),
serialNullTermStr)
```

Miután azonosítottuk az eszközt, és csatlakoztunk is, következő lépésként a csatornák beállítása következik. Először meg kell adni, hogy melyik csatornán szeretnénk mérést végezni. Utána a csatolás típusának megadása következik. Végül pedig, hogy milyen legyen a bemeneti feszültségtartomány, illetve ha szükséges, akkor

[25] handle

analóg ofszet feszültség is beállítható. Csatornaállító függvénynek a kódolt formája a következő:

```
def channel(channel, enabled, Type, Range, analogOffset):
     (...)
    PICO_STATUS = mydll.ps2000aSetChannel(c_handle, c_channel, c_enabled,
c_type, c_range, c_analogOffset)
```

Harmadik lépésben a konfigurálás részhez tartozóan még be kell állítani az időalapot. Fontosabb bemeneti paraméterek még a mintavett adatok darabszámának megadása, illetve a mintavételi idő megadása, ami két mintavett érték között eltelt idő. Ennek az értéknek a megadása nem magától értetődő, mivel a függvényparaméterként megadott érték nem egyezik meg az idő dimenziójában megadott számmal. Erre a gyártó által közé tett programozói útmutatóban [4] találhatunk egy táblázatot, ami bemutatja, hogy milyen képleteket kell alkalmazni bizonyos mintavételi időközök esetén.

1 GS/s maximum sampling rate models:

timebase (n)	sample interval formula	sample interval examples
0	2^n / 1,000,000,000	1 ns*
1		2 ns
2		4 ns
3 to $2^{32}-1$	(n – 2) / 125,000,000	3 => 8 ns … $2^{32}-1$ => ~ 34 s

5-5. ábra Mintavételi időközök számításai [5]

A táblázatban szereplő képletek segítségével már ki lehet számítani a szükséges paramétereket. A képleteket átalakítva úgy, hogy a hiányzó paraméter a mintavételi időköz legyen és végeredményként a függvénynek szükséges bemeneti paramétert kapjuk, a következő képleteket kapjuk, ahol t jelenti a mintavett időköz értékét másodpercben:

Mintavételi időköz (t)	Képletek
1 ns … 4 ns	$n = log_2(10^9 * t)$
8 ns … 34 s	$n = 125 * 10^6 * t + 2$

A beállítást végrehajtó kódrészlet:

```
def Timebase(timebase, NoSamples, segmentIndex):
 (...)
    PICO_STATUS = mydll.ps2000aGetTimebase(c_handle, c_timebase,
c_noSamples, byref(c_timeIntervalNanoseconds), c_oversample,
byref(c_maxSamples), c_segmentIndex)
```

Negyedik lépésben szükséges megadni, hogy mennyi memóriaszegmenst szeretnénk használni. Alapértelmezettként egy szegmens lefoglalása kerül sor.

Ötödik lépésként a triggerek beállítása következik. Lehetőségünk van egy feszültségküszöb beállításra, amit túllépve elkezdődik a mintavételezés vagy akár trigger jelre várakozás nélkül gyakorlatilag azonnal is el lehet indítani a mérést.

Eddig tartott a konfigurálási rész. Hatodik lépésben már az adatgyűjtés indítása következik:

```
PICO_STATUS = mydll.ps2000aRunBlock(c_handle, c_noOfPreTriggerSamples,
                                    c_noOfPostTriggerSamples,
                                    c_timebase, c_oversample,
                                    byref(c_timeIndisposedMs),
                                    c_segmentIndex, c_lpReady,
                                    byref(c_pParameter))
```

Hetedik lépésként írni kell egy olyan függvényt, ami akkor tér vissza, ha már az oszcilloszkóp befejezte az adatok gyűjtését. Ez a függvény a következőképpen áll össze:

```
def ready():
    c_ready = c_int16(0)
    PICO_STATUS = mydll.ps2000aIsReady(c_handle, byref(c_ready))
    print "c_ready : ",c_ready
    while(c_ready.value == 0):
        PICO_STATUS = mydll.ps2000aIsReady(c_handle, byref(c_ready))
        print "Collecting"
    if(c_ready != 0):
        print "c_ready : ",c_ready
        print "Block Finished collecting"
```

A gyártó már előre megírta nekünk a `ps2000aIsReady` függvényt, ami ready paraméterével nulla, ha még gyűjt a PicoScope és egyes, ha már befejezte.

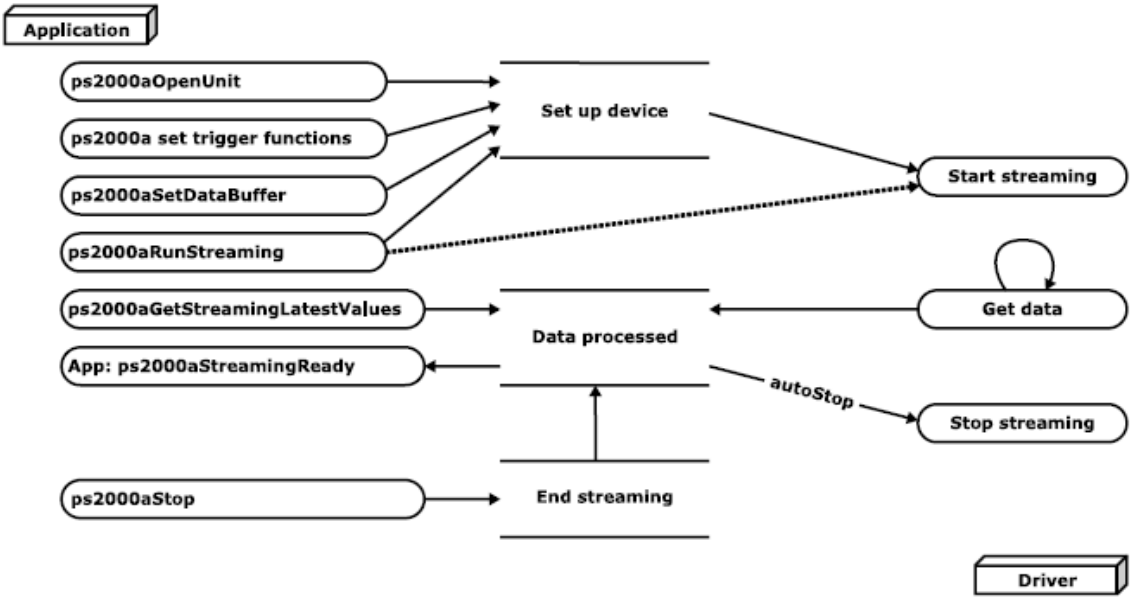
Nyolcadik lépésben meg kell adni, hogy a PicoScope belső memóriájába gyűjtött adatokat a számítógépen belül hova másolja át. Itt meg kell adni egy buffernek a memóriacímét, a memória terület nagyságát. Így ez a terület lefoglalásra kerül az adatok számára.

Kilencedik lépésben az adatok kimásolása történik a lefoglalt memóriaterületről egy általunk kiválasztott tömbbe, listába vagy ki is írhatjuk külön fájlba.

Tízedik lépésben már a begyűjtött adatok rendelkezésünkre állnak, szabadon felhasználhatóak, így már csak be kell zárni a PicoScope-ot.

5.4.2 Folyamatos adatgyűjtés

Folyamatos adatgyűjtés felépítésének szemléletesebb megértése érdekében a következő ábra segíthet:



5-6. ábra Folyamatos adatgyűjtés felépítése [5]

Hasonlóan a blokkos üzemmódhoz, az első lépés a PicoScope megnyitása, csatornák beállítása, trigger értékek beállítása, memóriaterület lefoglalása utána az adatgyűjtés kezdődik. A legnagyobb eltérés konfigurálási rész során a mintavételi időköz beállítása, ugyanis jelen üzemmód esetén egy külön paraméter segítségével lehet megadni, hogy mekkora legyen a mintavételi időközöket. Egy másik paraméter segítségével pedig be lehet állítani a mintavételi idő mértékegységét.

A folyamatos mintavételi üzemmód indítására szolgáló függvény:

```
PICO_STATUS = mydll.ps2000aRunStreaming(c_handle,
                                        byref(c_sampleInterval),
                                        c_sampleIntervalTimeUnits,
                                        c_maxPreTriggerSamples,
                                        c_maxPostTriggerSamples,
                                        c_autoStop,
                                        c_downSampleRatio,
                                        c_downSampleRatioMode,
                                        c_overviewBufferSize)
```

Az eddig bemutatott részek szinte teljes mértékben megegyeznek a blokkos üzemmódban bemutatottakkal. A legnagyobb kihívást azonban az úgynevezett CallBack függvény megírása okozta. Ennek a függvénynek a feladata, folyamatosan számon

tartani a begyűjtött adatokat, amiket bizonyos időközönként a számítógépen lefoglalt memóriaterületről kimásol egy adatgyűjtéstől független tárolóba. Ez a függvény a következőképpen épül fel:

```
def CallBack2(c_buffer, c_bufferLth):
    global Buff
    Buff = []
    CMPFUNC = WINFUNCTYPE(None ,c_int16, c_int32, c_uint32, c_int16,
c_uint32, c_int16, c_int16, c_void_p)
    def ps2000aStreamingReady(c_handle, c_noOfSamples, c_startIndex,
c_overflow, c_triggerAt, c_triggered, c_autoStop, c_pParameter):
        del Buff[0:len(Buff)]
        Buff.extend([c_buffer.__getitem__(i) for i in
range(c_startIndex,(c_startIndex + c_noOfSamples))])
        return None
    global my_StreamReady
    my_StreamReady = CMPFUNC(ps2000aStreamingReady)
```

Lényegében ez a függvény a lefoglalt memóriaterületről kimásolja a már begyűjtött adatokat. A `c_startIndex` a kimásolandó adatok memóriaterületének a kezdőértékét adja meg, míg `c_noOfSamples` a kimásolandó adatok számát tartalmazza.. A kimásolt adatok a létrehozott `Buff` listába tölti. Ezt a függvényt egy másik függvény időközönként hívja meg, ami már szintén gyártó által készen adott függvény.

```
def GSLV2(Streamingmode_A_stop):
    c_lpPs2000AReady = my_StreamReady
    c_pParameter = c_void_p()
        time.sleep(1)
        PICO_STATUS = mydll.ps2000aGetStreamingLatestValues(c_handle,
c_lpPs2000AReady, c_pParameter)
```

Ahogyan meghívja a CallBack függvényt, értéket ad a **`ps2000aStreamingReady`** függvény bemeneti paramétereinek. Így ciklikusan le tudjuk menteni a begyűjtött adatokat egy általunk biztosan eltárolt helyre. Azért szükséges lementeni az adatokat, mert a PicoScope számára lefoglalt számítógépes memóriaterület folyamatosan felülíródik, ha megtelt adattal a gyűjtés során.

5.5 Jelgenerátor működése

Kétféle jelgenerálási lehetőségünk is van. Az egyik előre megadott listából kiválasztható jelalakok generálása, a másik pedig tetszőlegesen programozható jelalak generálása.

5.5.1 Beépített jelek generálása

Lényegében ez a funkció összesen csak egy függvényből tevődik össze. Ez a következőképpen néz ki:

```
PICO_STATUS =
mydll.ps2000aSetSigGenBuiltIn(c_handle,c_offsetVoltage,c_pkToPk,c_wave
Type,c_startFrequency,c_stopFrequency,c_increment,c_dwellTime,c_sweepT
ype, c_operation,c_shots,c_sweeps,c_triggerType,c_triggerSource,
c_extInThreshold)
```

Először be kell állítani, hogy milyen ofszet feszültsége legyen a generálandó jelnek, amit microVolt dimenzióban kell megadni. Másodikként következik az amplitúdó értékének a beállítása szintén microVolt dimenzióban. Utána következhet a jelalak kiválasztása, ami lehet szinusz, négyszög, háromszög, felfutó vagy lefutó éllel rendelkező fűrészjel, sin(x)/x, Gauss, illetve fél szinuszokból álló jel. Ezután következik a frekvencia megadása. A kódból is látható, hogy két frekvenciaértéket vár bemeneti paraméternek. Az a kettő frekvencia normál beállítások között megegyezik, azonban ha sweep jelet szeretnénk beállítani, akkor a kezdeti és végső frekvencia értékeket jelentik. Ezek után még további beállítási lehetőségeink vannak. Például sweep üzemmód esetén egy adott frekvencián eltöltött időtartam beállítása (dwellTime), illetve frekvencialépték mértéke is beállítható (increment).

5.5.2 Tetszőlegesen programozható jelalak generálása

Kicsivel bonyolultabb beállításokkal rendelkezik a tetszőleges jelalak generátor. Az oszcilloszkóp DDS[26]-t használ az általunk megadott jelalak legenerálásához. Az áramkör alapja egy 32 bites fázisakkumulátor, ami indukálja a jel éppen aktuális helyeit. A fázisakkumulátor felső bitjeit egy bufferben tárolja indexek formájában, ami maga a jelalak, és az alsó bitek pedig egy indexnek részértékeit teszi ki. Ennek köszönhetően nagy felbontású és kis frekvenciás jelek generálására egyaránt lehetséges.

A tetszőlegesen programozott jelalak generálásához a szükséges függvény a következő:

[26] Direct digital synthesizer = Direkt Digitális Szintézer

```
PICO_STATUS =
mydll.ps2000aSetSigGenArbitrary(c_handle,c_offsetVoltage, c_pkToPk,
c_startDeltaPhase, c_stopDeltaPhase, c_deltaPhaseIncrement,
c_dwellCount, byref(c_arbitraryWaveform), c_arbitraryWaveformSize,
c_sweepType, c_operation, c_indexMode, c_shots, c_sweeps,
c_triggerType, c_triggerSource, c_extInThreshold)
```

Láthatóan hasonló a beállítása, mint a beépített jelalak generátorénak, azonban vannak különbségek. Először az ofszet feszültséget, majd az amplitúdót kell megadni. A jel frekvenciáját azonban a következő képlet segítségével lehet kiszámolni:

$$F_{out} = F_{DDS} * \frac{\Delta\varphi}{N_{PhaseAcc}} * \frac{N_{awgBuff}}{N_{arb}}$$

$\boldsymbol{F_{out}}$= Kimeneti jel frekvenciája [Hz]

$\boldsymbol{F_{DDS}}$= DDS számláló frekvenciája = 20MHz, tehát a DDS periódus 50ns

$\Delta\boldsymbol{\varphi}$ = Delta fázis, azaz a keresett érték, amit meg kell adni a konfiguráláshoz (`c_startDeltaPhase, c_stopDeltaPhase`)

$\boldsymbol{N_{PhaseAcc}}$= Fázisakkumulátor mérete = 2^{32}

$\boldsymbol{N_{awgBuff}}$= Tetszőleges jelalak generátor[27] buffer mérete=8192=2^{13}[db]

$\boldsymbol{N_{arb}}$= Tetszőleges jelalak mérete [db]

Ezt a képletet átalakítva megkaphatjuk azt az alakját, amikor a keresett paraméter a delta fázis és a változtatható paraméter a kimenő jel frekvenciája, amit a felhasználó választhat meg. Az átalakított képlet a következő:

$$\Delta\varphi = \frac{F_{out} * N_{PhaseAcc} * N_{arb}}{F_{DDS} * N_{awgBuff}}$$

Így már a felhasználó által elég a kívánt frekvenciát megadni, mivel ennek a képletnek a segítségével már ki lehet számítani a jel generálásához szükséges delta fázist. Hasonlóan a beépített jelgenerátorhoz, ha sweep jelet szeretnék, akkor a kezdeti és végső frekvencia éréke megadható, különben a két érték megegyezik konstans frekvenciájú jel generálásához. Ezután már következhet a `c_arbitraryWaveform` paraméter megadása, ami magának a jelalak pontjainak a feszültségértékeit tartalmazza egyenlő időközökre szétosztva.

[27] AWG = arbitrary waveform generator

A jelalak pontjainak feszültségértékeit a következő képlet segítségével lehet kiszámítani:

$$U_{out} = 1\mu V * \frac{U_{pp}}{2} * \frac{SV}{32767} + U_{off}$$

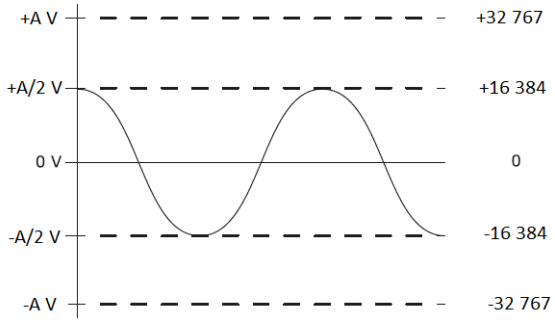
$\boldsymbol{U_{out}}$ = Kimeneti jelalakban szereplő minta feszültségértéke, mely maximum 2V lehet [V]

$\boldsymbol{U_{pp}}$ = `c_pkToPk` paraméterként megadott peak to peak érték [µV]

$\boldsymbol{SV}$ = Sample Value, azaz a megadott minta értéke, ami egy egész szám [-32767, +32767] intervallumon belül

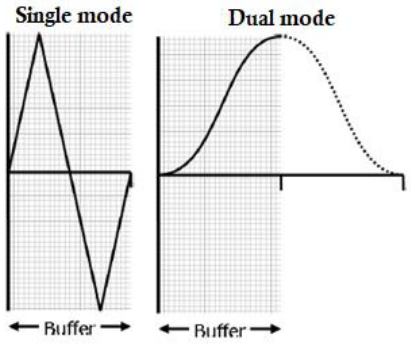
$\boldsymbol{U_{off}}$ = `c_offsetVoltage` paraméterként megadott ofszet feszültség [µV]

A képletben szereplő SV paraméter megadásához szemléletes magyarázatot ad a következő ábra, ahol A megegyezik U_{out} értékével.



5-7. ábra Tetszőleges jelalak [5]

A többi paraméter beállítása hasonló a beépített jelalak generátoréhoz, egyetlen különbség még az `c_indexMode` beállítás. A paraméter által lehetséges beállítási módokat a következő ábra szemlélteti:



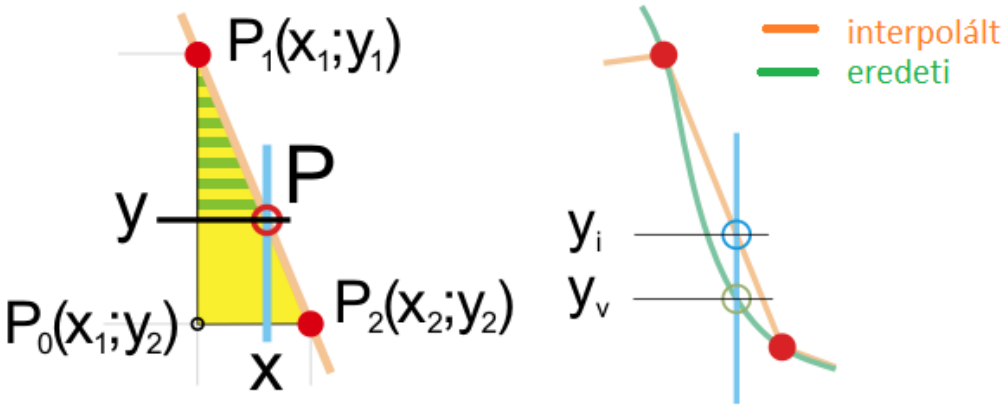
5-8. ábra Index mode beállítások [5]

Az ábrán láthatóak alapján singel mode beaállításban a megadott jelalak jelenik meg a kimeneten. Dual mode esetén pedig a beprogramozott jelalak a tükörképével együtt jelenik meg a kimeneten.

Annak érdekében, hogy a kimeneti jel a lehető legpontosabb legyen érdemes kihasználni a generátor adta lehetőségeket. Így ha a felhasználó nem adja meg mind a 8192 minta számát, akkor egy általam megírt kódrészlet lineáris interpoláció segítségével számítja ki a minták közötti értékeket – lásd: 5-9. ábra –, majd azokat le is generálja. A kódrészlet a következőképpen néz ki:

```
x = np.linspace(0, (len(arbitraryWaveform)-1),
num=len(arbitraryWaveform), endpoint=True)
f = interp1d(x, arbitraryWaveform)
xnew = np.linspace(0, (len(arbitraryWaveform)-1), num=8192,
endpoint=True)
newarray = []
for i in f(xnew):
    newarray.append(int(round(i)))
```

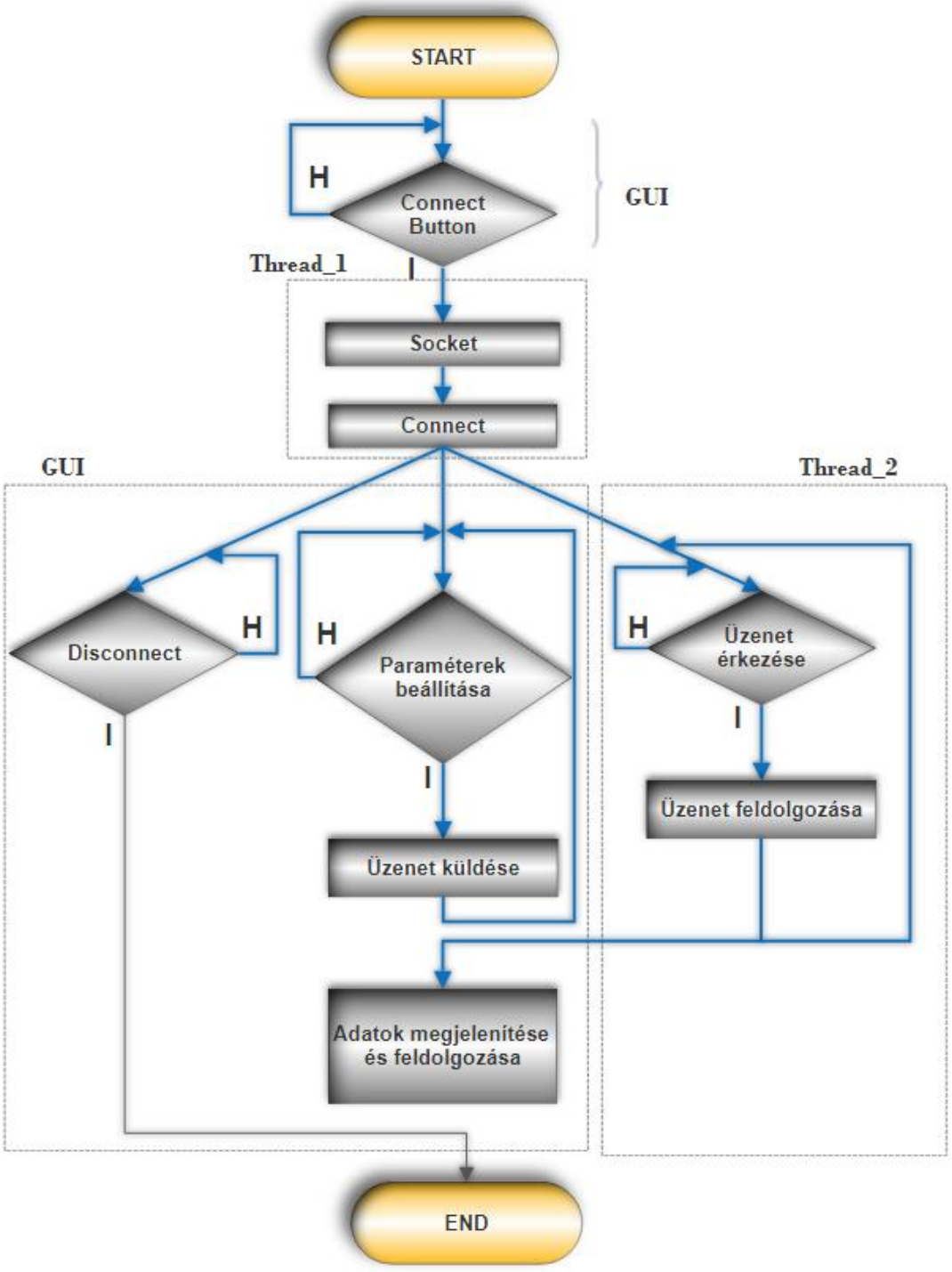
Lineáris interpoláció magyarázatára szolgáló ábra:



5-9. ábra Lineáris interpoláció [11]

6 Kliens oldali szoftverfejlesztés

Ebben a fejezetnem a szerver oldali szoftverfejlesztési fejezethez hasonlóan a folyamatábra segítségével az egyes részkomponensek részletes bemutatása következik.



6-1. ábra Kliens oldali folyamatábra

6.1 Folyamatábra értelmezése

A program a grafikus kezelői felület indításával kezdődik. Miután megnyitottuk az alkalmazást a 4-3. ábra szerinti kezelői felülettel találkozhatunk. A funkciók eléréséhez először csatlakozni kell a szerverhez a connect gomb segítségével. Ezt a folyamatábra úgy reprezentálja, hogy a program várakozik, amíg nem csatlakozunk a szerverhez. Miután rákattintottunk a csatlakozás gombra elindul a 4-4. ábra szerinti programstruktúrán is észrevehető Thread_1 szál. Ezen a szálon először felépül a kliens, majd csatlakozik a szerverhez a 4.2.2 alfejezetben leírtak alapján. Miután a szerver-kliens kapcsolat létrejött, elindul egy harmadik szál a Thread_2.

1) Ezen a szálon üzenetek érkezését várjuk. Ha érkezett egy üzenet, akkor azt szűrjük, majd értelmezzük[28]. Amennyiben begyűjtött adatok érkeztek, azokat beletöltjük egy listába.

2) Az így előállt lista tartalmát meg lehet jeleníteni vagy fel lehet dolgozni. A Thread_2 szál mellett párhuzamosan dolgozik még a fő szál (a grafikus kezelő felületet futtató szál). A GUI-t futtató szálon egyrészt lehetőségünk van szétkapcsolni a hálózati kapcsolatot és bezárni a kezelői felületet, illetve választhatjuk a PicoScope konfigurálását, jel generálását, vagy mérés indítását.[29]. Miután beállítottunk egy paramétert, annak értékét üzenet formájában elküldjük a szervernek, ahol a kívánt funkció végrehajtásra kerül.

6.2 Konfigurálási lehetőségek kliens oldalon

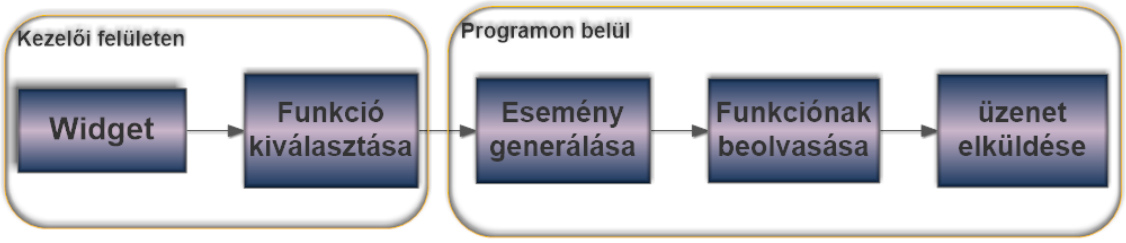
Az 5. fejezetben ismertetésre kerültek a PicoScope funkciói, beleértve az adatgyűjtést és a jelgenerálást. Magának a feladat végrehajtásának elvégzése a szerver oldalon, a mérőműszerhez csatlakozva történik. Azonban a feladat végrehajtásának kiindulása, a vezérlés gyökere a kliens oldalon történik a felhasználó által. A felhasználó a kliens oldali kezelői felületen keresztül tudja közölni a mérőrendszernek a végrehajtandó feladatokat. Gyakorlatilag úgy értelmezhető, mintha olyan kommunikációs felület lenne a felhasználó és a mérőműszer között, ami egyben

[28] 6-1. ábrán üzenetek feldolgozása

[29] 6-1. ábrán paraméterek beállítása

fordítóként is üzemel. A felhasználó szándékát lefordítja a mérőeszköz nyelvére, míg a mérőeszköz visszajelzéseit a felhasználó által értelmezhető formára fordít.

Az 5.2.2 alfejezetben említett üzenet feldolgozási része a kliens oldalról indul. Az üzenetek létrehozása pedig a kezelői felületen elhelyezett gombok, legördülő menük, összefoglaló néven widgetek által generált eseményeken belül történik. A szerver oldalnak küldendő üzenet elkészítéséig a kezelői felületen kiválasztott funkciótól egészen az üzenet elkészítésig a folyamatot az alábbi ábra szemlélteti:



6-2. ábra Konfiguráló üzenet készítés folyamata

Az ábrán is látható módon egy példával illusztrálva egy legördülő menüből az A csatornának AC vagy DC csatolási típusának kiválasztása közül az egyikre kattintva – funkció kiválasztása, majd ez által esemény generálása – a meghívódó függvény először beolvassa a kezelői felületen kiválasztott értéket[30]. Példával élve a kiválasztott AC csatolási típust. A programon belül azonban ez a művelet nevezhető műszer nyelvére való fordításnak, ugyanis a legördülő menü listájából kiválasztott érték listabeli indexe kerül beolvasásra. Ez a művelet egy nagy segítség a programozó számára, ugyanis a Python nem támogatja a C nyelven használatos typedef vagy enum típusokat. A PicoScope-hoz kapott könyvtári függvények viszont C programozási nyelven készült és a legtöbb függvény használja valamelyiket. Azonban végiggondolva az enum működését, rá lehet jönni, hogy tulajdonképpen egy szótár szerű listának feleltethető meg, ugyanis a parancs által összefogott listát, mint egy számsort tárol (A lista első eleme nulla, a második elem 1, stb.). Ezt mind végiggondolva észre lehet venni, hogy a legördülő menü listája gyakorlatilag egy C programozási nyelvben ismert enum típusnak is megfeleltethető. Tehát miután beolvasásra került a kiválasztott lista elemének az indexe, következhet az üzenetkészítési rész, ahol az 5-3. ábrának

[30] 6-2. ábrán Funkciónak beolvasása

megfelelően először a fejléc kerül elküldésre, ami tartalmazza pontosan beazonosíthatóan, hogy kitől érkezett az üzenet és milyen céllal. Miután a példa szerint elküldésre került, hogy az A csatorna csatolási típusának beállítási paramétere az üzenet, következhet a paraméter elküldése. Programozástechnikailag ügyelve arra, hogy először string konvertálást kell végrehajtani.

Így összefoglalva a kliens oldal lényegében a kezelői felületen látható kezelői szervek funkcióinak működéséhez szükséges feladatok vannak megírva.

6.3 Mérési adatok megjelenítése

A 6-1. ábrán látható folyamatábrán megfigyelhető, először a szervertől kapott üzenetek feldolgozására kerül sor a 4-4. ábra szerinti programstruktúrából kiolvasva a Thread_2 szálon. Majd a GUI szálán a begyűjtött adatok megjelenítésével folytatódik a program.

6.3.1 Üzenet feldolgozása

A szervertől beérkező üzenetekkel végrehajtott feldolgozási mechanizmus elvi felépítése gyakorlatilag egy az egyben megegyezik a szerver oldaliéval, amit az 5.2. alfejezetben már bemutatásra került.

Először itt is az üzenetek szűrése történik az 5-2. ábra szerint ismertetett fésűs szerkezet alapján. Erre azért van szükség, mivel a szerver oldaltól kaphatunk válasz üzeneteket is. Azért, hogy biztosak legyünk, hogy a szerver oldal aktív, kommunikál a kliens oldallal és végre is hajtotta a kitűzött feladatot. Viszont érkezhetnek begyűjtött adatok is.

Miután kiszűrtük, hogy milyen típusú üzenet érkezett, annak megfelelően az üzenet feldolgozása következik. A pontosabb megértés érdekében először tekintsük röviden át, hogyan is áll össze az üzenet a szerver oldalon. Ehhez a korábban megismert 3.1.2. alfejezetben ismertetett kétféle adatgyűjtési esetekre bontanám szét logikailag az üzenetküldési részét.

1) Blokkos mintavételi mód esetén a szerver oldali switch modulban megírt adatgyűjtés befejezése után a begyűjtött adatok átkerülnek egy bufferbe, ami a Pico2000 modulban megírt `CollData()` függvény visszatérési értékeként áll rendelkezésünkre. Ezután már ennek a függvénynek a rendelkezésével el tudjuk indítani az üzenetküldést. Az üzent küldése a

már jól ismert 5-3. ábra szerint történik, annyi különlegességgel, hogy az adatok átküldése a buffer feldarabolásával kezdődik, majd az adatok egyesével kerülnek továbbításra. Az üzenetküldés kódrészletei a következők:

a. Fejléc: `conn.sendall("Block_A_buffer")`

b. Adat:
```
Buff = Pico2000.CollData(bufferA, bufferLthA)
for i in range(len(Buff)):
    conn.sendall(str(Buff[i]))
    conn.sendall(",")
```

c. Üzenet vége: `conn.sendall("send_finished")`

2) Folyamatos adatgyűjtés esetén hasonló az üzenetkészítés procedúrája, mint a blokkos üzemmód esetén. Lényegében csak a fejléc tartalmában különböznek. Ebben az esetben a `StreamBuffer()` függvény szolgáltatja visszatérési értékként a begyűjtött adatokat.

Miután már ismerjük, hogy milyen módon épül fel az üzenet, tekintsük meg, milyen műveletsorozatok révén kerül az adat kicsomagolása az üzenetből:

1) Először begyűjtjük egy helyi bufferbe az érkező adatokat
2) Ezután a jelöléseknél, ami egy vessző, szétdaraboljuk a buffert
3) Majd az így kapott adathalmazt még egy szűrésen átengedjük, hogy biztosan ne kerüljön hibás adat a végső listába.
4) Végül pedig egész számmá konvertálva beletöltjük egy listába, amit a kliens oldalon már a függvények elérnek.

6.3.2 Megjelenítés

Miután a kliens oldalon is rendelkezésünkre állnak a begyűjtött adatok, így azokat már meg lehet jeleníteni. Erre a célra a 3.2.2. alfejezetben már ismertetett matplotlib modult használtam. Ezáltal az előre megírt függvények segítségével Matlab szerűen lehet grafikonok megjelenítését elvégezni. A mért pontok kijelzésének mértékegységhelyes kijelzéséhez tudni kell, hogy a mért jel digitalizálása, majd mintavett értékek begyűjtése során a jel pontjainak feszültségtengelyen felvett éréke megkapható 32768-cal való leosztás után. Az időtengely helyes értékeinek megadásához tudni kell, hogy a mintavételt milyen mintavételi időközök beállításával

indítottuk el. Ezek után az időtengelyt ezzel a számmal való szorzás után már meg is kapjuk.

Legvégül már csak a grafikus felületbe való beintegrálás szükséges a céloknál kitűzött kompakt tulajdonsággal bíró kezelői felület létrehozása érdekében. Ehhez szintén a matplotlib csomag nyújtott segítséget, ugyanis a Qt4-et támogatva a következő részeket beimportálva a programunkba a `figure.add_subplot(111)` parancs segítségével már a megjelenített grafikon a kezelői felületen fog megjelenni. Importált modulok kódrészletei a következőek:

```
from matplotlib.backends.backend_qt4agg import FigureCanvasQTAgg as
FigureCanvas
from matplotlib.backends.backend_qt4agg import NavigationToolbar2QT as
NavigationToolbar
from matplotlib.figure import Figure
```

6.4 Adatfeldolgozási lehetőségek

6.4.1 Adatfeldolgozás lehetővé tétele

Adatok utólagos feldolgozásához az elsődlegesen legfontosabb feltétel a mérés során begyűjtött, mintavett adatok eltárolása, illetve hozzáférhetővé tétele. Legelőször az adatok a PicoScope-ból a szerver oldalon jelennek meg. Ott a 6.3.1 alfejezetben bemutatottak alapján az adatok elérhetőségét szolgáló függvények a következőek:

1) Blokkos beolvasás esetén:

```
def CollData(c_buffer, c_bufferLth):
    a=[c_buffer.__getitem__(i) for i in range(c_bufferLth.value)]
    return a
```

2) Folytonos mintavételezésnél, ahol a `Buff` az 5.4.2. alfejezetben kifejtett `CallBack` függvényben szereplő globális bufferrel egyezik meg:

```
def StreamBuffer():
    return Buff
```

A kliens oldalon a 6.3.1 alfejezetben bemutatottak alapján ugyan ezeket az értékeket kapjuk meg, amiket egy `pyarr` nevű listában tárolunk el. A lista globálisan lett létrehozva, így bárki, aki a szoftver továbbfejlesztésén gondolkodik, ennek a listának az elemeit felhasználva rögtön meg tud írni egy olyan jelfeldolgozó függvényt, aminek bemerő paramétere az átvett lista. A mérési adatokat tartalmazó lista létrehozása a következő:

```
BuffData = s.myreceive()
global pyarr
pyarr = []
dlist = BuffData.split(',')
for v in range(len(dlist)):
    if dlist[v] == "send_finished":
        break
    elif dlist[v] == '':
        break
(...)
    elif dlist[v] == "-":
        break
    else:
        pyarr.append(float(dlist[v]))
```

A 6.3.1 alfejezetben említettek szerint itt is látható, hogy a beérkezett adatokból kivágjuk a válaszként szolgáló vesszőket, majd egy szűrésen keresztül feltöltjük a listát. Későbbi továbbfejleszthetőségekre előrelátva, megírtam egy függvényt, ami bemeneti paraméterként megkapja a `pyarr` listát és a 4-3. ábrán látható Measure rész felülről a harmadik gombjára kattintva meghívja ezt a függvényt. Ezáltal a későbbiek során megírt algoritmus lefutásra kerül. A megírt kódrészlet a következő:

```
def SignalProcess(self, SignalBuffer):
    LocalArray = SignalBuffer
    print "Please fill the missing part of the code"
    # jelfeldolgozó algoritmus

def pushButton_2_event(self):
    self.SignalProcess(pyarr)
```

6.4.2 Adatfeldolgozás demonstrálása

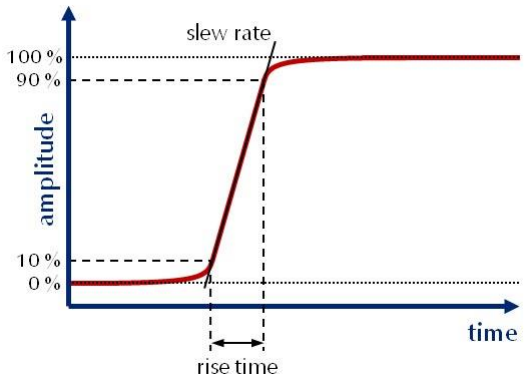
Az adatok utólagos feldolgozásának lehetővé tételét két egyszerű feldolgozáson keresztül fogom demonstrálni. Az első példában a mért jel Fourier transzformáltját kiszámoló adatfeldolgozó rész segítségével egyszerűen szemléltetni lehet a funkció működési mechanizmusát. Második példában egy felfutó él számító algoritmuson keresztül kerül demonstrálásra az adatfeldolgozás. Hangsúlyozandó, hogy a szakdolgozatban szereplő feladatkiírás alapján csak az adatok utólagos feldolgozását lehetővé tevő funkció bemutatása a cél, nem pedig egy tökéletes jelfeldolgozó algoritmus fejlesztése, megvalósítása. Választhattam volna kész jelfeldolgozó algoritmust a demonstráláshoz, azonban érdekesebb kihívásnak találtam egyedi algoritmuson keresztül bemutatni a feladatot.

Az első példában szereplő kódrészlet:

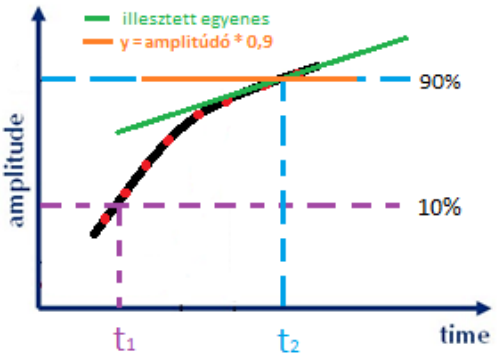
```
def FFT(self, SignalBuffer):
    sp = np.fft.fft(SignalBuffer)
    print sp

def pushButton_1_event(self):
    var = self.FFT(pyarr)
    print var
```

Látható, hogy a `pushButton_1` gomb megnyomására aktiválódott `pushButton_1_event()` meghívja az adatfeldolgozást végző függvényt, ami jelen példában az `FFT()`. Ez a függvény pedig kiszámolja, majd kiírja a begyűjtött adatok Fourier transzformáltjait.

Második példaként egy felfutó él mérést valósítottam meg. Az algoritmus a rendelkezésre kapott adatok alapján kiszámolja a jel amplitúdójának 10%-a és 90%-a között eltelt időt.



6-3. ábra felfutó él [9]



6-4. ábra felfutó él

Ez eleinte egyszerűnek hangzik, azonban végiggondolva nem annyira az, mivel diszkrét értékekről van szó, ezért ha a jel amplitúdójának 10%-a vagy 90%-a nem érint egy pontot sem, akkor valamilyen közelítéssel kell élni. Lehetne lineáris interpolációval közelíteni a pontosabb értékek kiszámítása érdekében, azonban ugyanúgy fenn áll a veszély, hogy nem érint egy konkrét pontot sem. Ezért az általam választott kiszámítási módszer két egyenes által metszett pont kiszámításával történik. Az egyik egyenes $y = a_1$, (ahol a_1 az amplitúdó 90%-a) míg a másik egyenes a 6-4. ábrán zöld látható illesztett egyenes. Ennek a két egyenesnek a szemléltetése a 6-4. ábrán látható. A két egyenes által metszett pont időtengelyen felvett értéke t_2 (lásd: 6-4. ábra). Hasonlóan történik t_1 értékének kiszámítása is. Ezek ismeretében már a felfutó él értéke a következő képlet alapján számítható ki: $t_{rise\ time} = t_2 - t_1$

7 Tesztelés

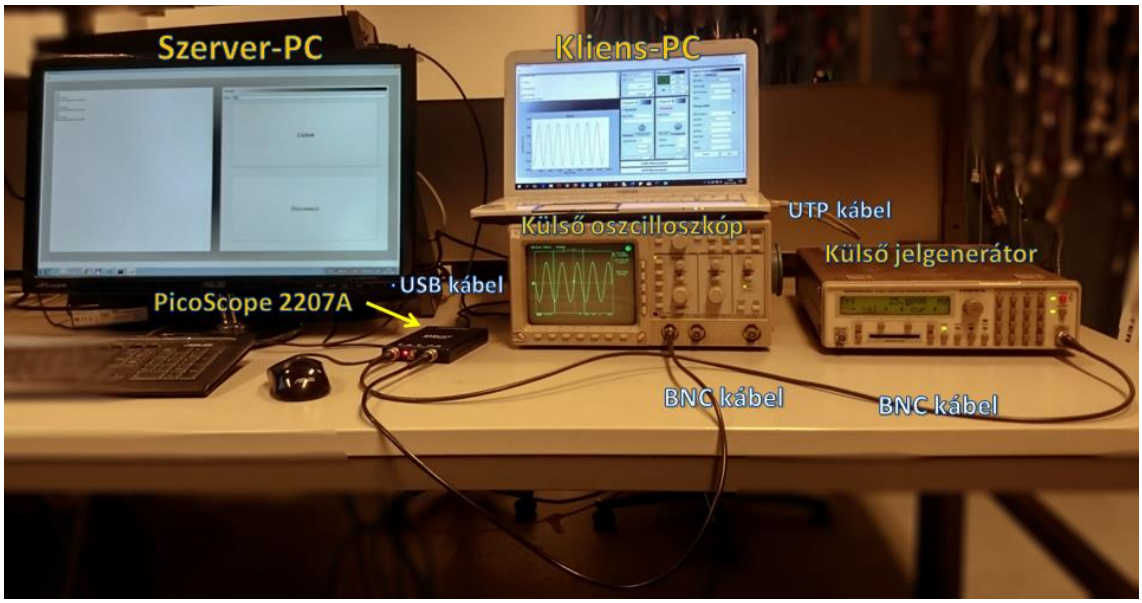
7.1 Mérési eredmények

Mint minden fejlesztési folyamat először a tervezéssel indul, majd a rendszer elkészítése követi. Miután összeállt a rendszer, minden alkotóelemével együtt, akkor végül a tesztelési és kiértékelési fázis következik.

7.1.1 Tesztelés

Az elkészült mérőrendszer tesztelése logikailag két fázisra bontható. Először a szükséges funkciók helyes működésének ellenőrzése történik. Ebben a fázisban meg kell bizonyosodni, hogy minden feladatrész a specifikációnak megfelelően működik. Amennyiben valamelyik kritikus funkció nem működik, nem indul el, vagy nem a specifikációnak megfelelően végzi a feladatát, akkor itt még lehetőség van azok kijavítására. Miután ezek a lépések megtörténtek, tudni lehet, hogy a rendszer működik, még ha kompromisszumosan is, de az előre kitűzött célnak megfelelően végzi a feladatait.

Ezt követi a hibakeresésnek nevezhető fázis, amikor már nem a működőképességét vizsgáljuk a rendszernek, hanem sokkal inkább az elkészült termék határainak feszegetése történik. Például előfordulhat, hogy nincsen teljesen tökéletesen minden hibaforrás lehetősége kiküszöbölve, így amikor meg kell adni egy egész számot paraméter beállítás céljából akkor, ha egy karakter kerül begépelésre szám helyett, lehet, hogy hibás működést fog eredményezni. Ettől függetlenül a rendszer egy előre megadott specifikáció szerint működhet helyesen. Értelem szerűen a cél minél biztonságosabb és megbízhatóbb rendszer tervezése, főleg az iparban, ahol komoly feltételeknek és minőségbiztosítási előírásoknak, szabványoknak kell megfelelnie egy új terméknek a piacra kerülése előtt. Ezekre a műveletekre, szakdolgozatom témája nem terjed ki. Értelem szerűen az idő nem is engedné, hogy évekig tartó több felhasználó által visszajelzett tapasztalatok alapján teljesen hibamentesre javítsam ki a mérőrendszert. Azonban az első fázis elvégzése, a helyes működés ellenőrzése elengedhetetlen részét képzi a mérőrendszerem fejlesztésének. Tesztelés szempontjából előállított mérési elrendezést az alábbi ábrán láthatjuk:



7-1. ábra Tesztelési elrendezés

A képen látható mérés éppen egy folyamatos mintavételi üzemmódban elindított adatgyűjtés. Az adatgyűjtés mellett párhuzamosan jelgenerálás is látható.

1) **Szerver** oldali elrendezés:

   A kép baloldalán látható a szerver oldali számítógép, amin a szerver oldali alkalmazás fut és UTP kábelen keresztül össze van kötve a hálózattal. Ez a számítógép USB kábelen keresztül csatlakozik az adatgyűjtést végző mérőműszerhez, a PicoScope 2207A típusú PC-s oszcilloszkóphoz. A PicoScope AWG kimenete BNC kábel segítségével össze van kötve a kép közepén látható külső oszcilloszkóppal, míg az A bemeneti csatornájára a kép jobboldalán található külső jelgenerátor csatlakozik.

2) **Kliens** oldali elrendezés:

   A kép közepén látható laptop a kliens-PC, amin a kliens oldali kezelői felület fut. Ez a számítógép UTP kábel segítségével össze van kötve a hálózattal, amin keresztül létrejött a kommunikáció a két oldal között.

7.1.2 Tesztelés folyamata és kiértékelés

Miután összeállt a mérési elrendezés, először elindítottam a szerver oldali számítógépen az alkalmazást. Ezután beállítottam egy tetszőleges portot, majd rákattintottam a listen gombra. Következő lépésként elindítottam a külső jelgenerátort,

ami a rendszerterv szerint a mérendő rendszert reprezentálja. Ezekkel a műveletekkel a szerver oldal be is lett állítva, mérésre készen áll. Következhetett a kliens oldal előkészítése. Először itt is elindítottam a kliens oldali PC-s alkalmazást, majd a host mezőbe beleírtam a szerver-PC IP címét, illetve a kiválasztott tetszőleges portot is a port mezőbe. Itt fontos megemlíteni, hogy mind a két oldalon ugyan annak a portnak kell szerepelnie. Végül pedig rákattintottam a connect gombra, majd a kezelői felület kijelzőjén megjelent a hálózati chat, ami annyit jelent, hogy mind a szerver, mind a kliens egy üdvözlő üzenetet küld, amiből tudhatjuk, hogy a kapcsolat felépült és működik a hálózati kommunikáció a két fél között. Miután felépült a kapcsolat beállítottam egy 100 Hz-es 1 V_{pp} amplitúdójú 0 V ofszet feszültségű szinusz jelet, végül pedig rákattintva a start gombra, ezáltal legenerálta a kívánt jelet. A generált jelalak a 7–1. ábrán is látható módon, a külső oszcilloszkóp képernyőjén megjelent, ezzel láthatóvá téve a jelgenerátor működését. A pontos tesztelés érdekében kipróbáltam, hogy ha változtatok a paramétereken, akkor a kimeneten is megváltozik-e a jelalak, amit szintén a külső oszcilloszkóp képernyőn tudtam megvizsgálni.

A jelgenerátor kipróbálása után következhetett az adatgyűjtés tesztelése. Először kiválasztottam az Enabled mezőre kattintva az A csatornát, majd beállítottam neki 2V bemeneti feszültségtartományt. Következő lépésként a Streaming mode fülre kattintva beállítottam 100 µV mintavételi időközt. Végül pedig a start gombra kattintva elindítottam az adatgyűjtést. A jelgenerátor (mint mérendő rendszer) által generált jel a Kliens-PC képernyőjén a 7-1. ábrán is látható módon megjelent. Legvégül elvégeztem néhány mérést blokkos adatgyűjtési üzemmódra is, és végig próbáltam az összes funkciót.

Összességében megállapítható, hogy a teszt sikeres volt és az elkészített mérőrendszer specifikációnak megfelelően működik.

Elvégeztem egy pár szélsőséges mérést is, amikből kiderült, hogy nem tökéletes a működés, azonban ezek már a továbbfejlesztési feladatokhoz tartoznak.

7.2 Továbbfejlesztési lehetőségek

Előző pontokban bemutatottak alapján látható, hogy az általam elkészített mérőrendszer még nem feltétlen jelenti azt, hogy teljesen hibamentes és minden felhasználó számára kiválóan használható vagy a témában elképzelhető minden

lehetőséget magába foglal az elkészült rendszer. Ennek következtében lehetőség van a rendszer továbbfejlesztésére.

A mérőrendszer szoftvere nyitott további hibavédő eljárások kifejlesztésére. Például hibás adatok bevitelének kiküszöbölése, a hálózaton történő biztonságosabb kommunikálás megvalósítása, illetve véletlen helytelen gombok megnyomásának, aktiválásának vagy deaktiválásának kiküszöbölése.

Továbbfejlesztési szempontból második kategóriába sorolhatnám logikai felépítés szerint a már megírt részek frissítését. Például a hálózaton lévő kommunikáció fejlesztésével gyorsabb küldést, feldolgozást lehetővé tenni. A folyamatos adatgyűjtésből fakadó kijelzésen is van még lehetőség pontosabb, gyorsabb megjelenítéshez.

Hibavédő eljárások fejlesztése mellett további funkciók elkészítése is lehetséges. Gondolhatunk itt az apróbb finomhangolásokra, mint például jelgenerátorhoz előre megírt jelalakok generálása mellett további jelalakok generálási lehetősége. Például multi szinusz generálása egy gombnyomásra. Esetleg web kamera funkció elkészítése. Azonban lehetőség van a rendszert egy mesterséges intelligenciával megírt okos mérőrendszer továbbfejlesztésére is.

A megírt programcsomag kódolástechnikai felépítéséből fakadóan továbbfejlesztési lehetőségek komplex és egymásba ágyazott részek gyökerestől való átírása nélkül is lehetséges. A legtöbb funkcióhoz egy-egy előre megírt függvény segítségével lehet csatlakozni, így csak egy minimális csatlakozó interfész megírásával lehetőség van akár egy fejlettebb különálló szoftvert megírva csatlakozni az általam elkészített programhoz.

7.3 Javaslatok

Az elkészült mérőrendszer továbbfejlesztésén már én magam is elkezdtem dolgozni. Többen jelezték számomra, hogy mennyi lehetőséget látnak ebben a témában, és ha megvalósulnának, akkor mennyire hasznos lenne számukra.

Az egyik javaslat úgy szólt, hogy nagy kényelmet okozna a felhasználó számára a mérőrendszer, ha meg lehetne valósítani, hogy otthonról vezérelhető teljes értékű oszcilloszkópként viselkedjen. Ennek következtében, ha a felhasználó bemegy az egyetemi laborba elvégzi a szükséges mérését, viszont a mérési összeállítást nem szedi

szét, akkor ha otthon eszébe jut, hogy még mit kellene módosítani a mérésen, akkor minden aggodalom nélkül megtehetné azt. Ennek megvalósításához lényegében csak a szükséges mérési algoritmusok rendszerbe való integrálása hiányzik.

Egy másik továbbfejlesztési javaslat audio jelek mérésénél hasznosnak bizonyulhat. Bizonyos audio jelek mérését úgy végzik, hogy bemennek a laborba, összeállítják a mérési elrendezést és kézzel elindítják a mérést. Azonban nagyon precíz méréshez az emberi személyes jelenléte, mint zavarás megjelenhet a mérésben. Ennek következtében az általam elkészített mérőrendszert fel lehetne használni ilyen esetekre is. Például a laborban összeállított mérés után laboron kívülről is vezérelni lehetne a mérést. Ennek elkészítéséhez is csak egy-két helyen történő fejlesztés szükséges. Főleg a méréshez szükséges algoritmusok megírása, például átviteli függvény mérés funkció elkészítése a nagyobb feladat.

Végül pedig főleg labor demonstrációs feladatokhoz hasznos funkció lehet egy web kamera integrálása a szoftverbe, aminek köszönhetően a mérésvezető akár távolról is be tudja mutatni a szükséges lépéseket.

Látható, hogy rengeteg felhasználási lehetőség áll rendelkezésre az elkészült kísérleti mérőrendszer kapcsán, mind ipari, mind oktatási, demonstrálási célok kapcsán.

8 Konklúzió

8.1 Tapasztalatok

Egy érdekes kihívás volt számomra az egész féléves munka elkészítése. A feladat elkészítése során megismerkedhettem magával a szoftverfejlesztési lépésekkel, nehézségekkel. Ezen belül is magával a Python nyelvvel, ezzel is tágítva a programozási ismereteimet. A Python elsőre szokatlan volt a szintaxisa, illetve a fejlesztői környezet.

Feladatom során egy újabb programozási nyelv elsajátítása mellett programozástechnikai ismereteimet is fejleszthettem. Egyrészt szálkezelés téren szerezhettem új tapasztalatokat, ezáltal teret kaptam szélesebb spektrumban gondolkodni, több processzoron párhuzamosan végzett feladatok világában elmélyedni. Másrészt grafikus kezelői felület elkészítésének módját is megtanulhattam.

Nem utolsó sorban pedig felfrissíthettem korábbi tanulmányaim során C++ programozás keretein belül tanult objektumorientált programozási mechanizmusokat. Velejáróan átismételhettem az osztályok felépítését, az öröklés működését és még sok más programozási területet.

A PicoScope-hoz mellékelt gyári könyvtári függvények miatt fel kellett frissíteni az első félévben tanult C programozási nyelvet. Továbbá betekintést nyerhettem Python és C programnyelvek kapcsolatába.

A szoftverfejlesztési ismeretek bővítésén túl előkerültek egyéb tanulmányok során tanult ismeretek is. Például a jelgeneráláshoz használt lineáris interpoláció vagy a hálózat felépítésénél előforduló TCP/IP hálózati protokoll világa. Ezek által az elméleti ismereteimet gyakorlatba is ültethettem.

Látható, hogy ez a feladat elég széles mérnöki területeket magába foglal kezdve a szoftverfejlesztéstől a jelfeldolgozáson keresztül rendszerszintű komplex gondolkodásmódot igénylő átlátásig.

8.2 Kihívások

A feladat szépségén és tanulságain túl azonban akadtak megoldandó problémák és nehézségek is bőven. Ezek közül párat szeretnék megemlíteni.

Legelőször kihívás volt számomra egy teljesen új programozás nyelv elsajátítása a vele járó új szintaktika, és minden rá jellemző logikai gondolkodásmóddal együtt. Tanulmányaim során már megismerkedhettem más nyelveken keresztül a programozás rejtelmeivel, azonban minden nyelv egy kicsit különbözik egymástól.

Továbbá meg kellett mérkőznöm a már elsajátított programozási tudásomon felül új technikák megismerésével. Ideértendő maga a grafikus programfejlesztés, a többprocesszoros, párhuzamos szálkezelési mechanizmusok.

A szoftverfejlesztésen belül maradva talán a legnagyobb nehézséget a Python és a C nyelv közötti áttérés megvalósítása jelentette. Például ami C nyelven enum segítségével lett megírva, azt milyen módon lehet Python nyelven helyettesíteni, hogy kompatibilisek legyenek. Illetve a korábban megszokott programozási logikákat keresztülhúzva a Python által nem támogatott switch-case szerkezet hiánya is bőséges fejtörést okozott.

Rendkívül nagy odafigyelést és koncentrációt igényelt a program fejlesztése miután már több ezer soros kóddá hízott, nagyon nehéz volt megtalálni a hibákat. Azért is okozott kellemetlen pillanatokat, mivel egyes elkövetett figyelmetlenségek nem feltétlen okoztak olyan hibákat, amiknek következtében nem indult el a program, hanem szemmel láthatóan helytelen működést okozott. Például, amikor a Pico2000 modult fejlesztettem, akkor a kódban `c_uint32()` helyett `c_uint16()` írásával nem működött megfelelően a program.

Hosszas órák eltöltését igényelte még magának a gyártó által közzétett programozói segédletnek a tanulmányozása, értelmezése is. Ennek oka a számomra nehezen értelmezhető, rendkívül szűkszavúra sikeredett specifikáció okozhatta. Ennek következtében rengeteg időt emésztett fel az egyes kérdőjeles funkciók egyesével való végigpróbálása.

8.3 Összegzés

Összességében elmondható, hogy elkészült egy olyan kísérleti automata mérőrendszer, amely alkalmas jelgenerálásra, és a felhasználó igényeit kiszolgálva több adatgyűjtési üzemmódot is tartalmaz. A mérési eredmények a kezelői felületbe integrálva megjeleníthetőek. Továbbfejleszthetőség szempontjából megvalósult egy mérés folyamán begyűjtött adatok utólagos feldolgozhatóságára lehetőséget biztosító

interfész. A mérőrendszer lelkét alkotó PicoScope 2207A típusú PC-s oszcilloszkóp a kívánt funkcióknak megfelelően hálózaton keresztül konfigurálható.

Mindezek a funkciók, amikre képes a mérőrendszer a felhasználó számára kényelmesen egy grafikus kezelői felületen keresztül vezérelhetőek. Ráadásul a komfortot tovább fokozva egy mérés elvégezhető Interneten keresztül történő vezérlés révén is.

A feladat elején kitűzött céloknak megfelelően a mérőrendszer kompakt, ugyanis elegendő egyetlen egy kezelői felület elindítása, ahonnan az összes funkció elérhető. Mobilis, ugyanis a mérést végző PicoScope fizikailag kis méretű, akár egy zsebben is elfér. Illetve praktikus, mivel nem szükséges a mérés helyszínén tartózkodni a mérés folyamata alatt.

Lehetőség van az elkészült mérőrendszer továbbfejlesztésére is. A későbbiek során megvalósítható akár egy ipari felhasználású intelligens mérőrendszer.

9 Irodalomjegyzék

[1] „Műszerek és mérések,” [Online]. Elérhető: https://www.muszeroldal.hu/measurenotes/muszeripar.pdf. [Hozzáférés dátuma: 26 11 2017].

[2] „beágyazott rendszerek kommunikációja,” 19 11 2017. [Online]. Elérhető: http://szabilinux.hu/konya/konyv/2fejezet/2fbereko.htm. [Hozzáférés dátuma: 27 11 2017].

[3] „GPIB jövője,” 23 10 2017. [Online]. Elérhető: https://elektro-net.hu/rendszerintegrator/1637-gpib-jovoje. [Hozzáférés dátuma: 27 11 2017].

[4] P. Technology, „Programmer's Guide,” Pico Technology, www.picotech.com, 2011.

[5] „A python programozási nyelv,” [Online]. Elérhető: http://nyelvek.inf.elte.hu/leirasok/Python/index.php?chapter=2. [Hozzáférés dátuma: 30 11 2017].

[6] „PyQt Class Reference,” [Online]. Elérhető: http://pyqt.sourceforge.net/Docs/PyQt4/classes.html. [Hozzáférés dátuma: 1 12 2017].

[7] „SlidePlayer,” [Online]. Elérhető: http://slideplayer.hu/slide/2112645/. [Hozzáférés dátuma: 5 12 2017].

[8] M. Dr. Herdon, Z. Magó és G. Dr. Kovács, „www.szily.hu,” 2007. [Online]. Elérhető: https://www.szily.hu/docs/osztalyhir/Szamitogep-halozatok.pdf. [Hozzáférés dátuma: 5 12 2017].

[9] „www.gamry.com,” [Online]. Elérhető: https://www.gamry.com/application-notes/instrumentation/understanding-specs-of-potentiostat/. [Hozzáférés dátuma: 5 12 2017].

[10] P. Technology, „Data Sheet,” Pico Technology, www.picotech.com, 2000.

[11] „SlidePlayer,” [Online]. Elérhető: http://slideplayer.hu/slide/2091463/. [Hozzáférés dátuma: 1 12 2017].

[12] „Wikipedia,” [Online]. Elérhető: https://upload.wikimedia.org/wikipedia/commons/6/67/Linear_interpolation_2.png. [Hozzáférés dátuma: 3 12 2017].