Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Bozsik Szabolcs Dániel

DINAMIKASZABÁLYZÓ VST AUDIOEFFEKT MEGVALÓSÍTÁSA

KONZULENS

Dr. Rucz Péter

BUDAPEST, 2024

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Bozsik Szabolcs Dániel**, 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(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) 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, 2024. 06. 01.

...…….
Bozsik Szabolcs Dániel

Összefoglaló

Az 1980-as évek óta egyre nagyobb teret nyernek a számítógépalapú megoldások. Nincs ez másként a hangtechnikánál, hangstúdióknál sem, egyre több helyen használnak programokat a fizikai eszközök helyettesítésére, a kevesebb szükséges hely, az egyszerűbb kezelhetőség, a változatosabb lehetőségek és a jóval alacsonyabb ár miatt.

Ezek a programok először csak a fizikai eszközök egyszerűbb kezelését tették lehetővé, majd a technológia fejlődésével megjelentek a rögzítő és keverő, majd az effektező programok. Ezeknek az effekteknek egy csoportja a dinamikaszabályzással foglalkozik, az expanderek a kimeneti dinamikatartományt felnagyítják, míg a kompresszorok ezt összenyomják, szélsőséges esetben limitálják.

Szakdolgozatomban bemutatom a hang és az emberi hallás sajátosságait, bemutatom mi az a dinamika, milyen szabályzó lehetőségek vannak, miért szükségesek, milyen fizikai eszközök léteznek, ezek hogyan fejlődtek az elmúlt közel 100 év alatt, hogyan ültetődött át számítógépes környezetbe a hangfeldolgozás. Ezek után készítek egy egyszerűbb dinamikaszabályzó effektet a VST SDK keretrendszerben, ami a legtöbb fizikai eszköz funkcióit ellátja és azokhoz hasonló paraméterezhetőséggel rendelkezik. A paraméterek változtatásához létrehozok egy hozzá tartozó grafikus vezérlő és visszajelző felületet az egyszerűbb kezelhetőség érdekében. Az így elkészített programot tesztelem és kiértékelem a kapott értékeket. Végül bemutatok pár továbbfejlesztési lehetőséget, amivel plusz funkciókkal lehet bővíteni az elkészült plugint.

Abstract

Since the 1980s, computer-based solutions have been gaining increasing prominence. This is also true in audio technology and recording studios, where more and more places are using software to replace physical devices due to the reduced space requirements, simpler handling, more varied possibilities, and much lower cost.

Initially, these programs only allowed for simpler handling of physical devices, but as technology advanced, recording and mixing solutions, followed by effect processing programs emerged. One group of these effects deals with dynamic control, where expanders enlarge the output dynamic range, while compressors compress it, and in extreme cases, limit it.

In my thesis, I will present the characteristics of sound and human hearing, explain what is the dynamic range, and discuss the various control options available, why they are necessary, and what physical devices exist. I will also cover how these devices have evolved over the past nearly 100 years and how audio processing has been transferred to a computer environment. Following this, I will create a simpler dynamic controller effect in the VST SDK framework that fulfills most of the functions of physical devices and has similar parameterization. To be able to interact with the parameters, I will create a graphical control and feedback interface for easier handling. I will test the program created and evaluate the obtained values. Finally, I will present a few possibilities for further development to add additional functions to the completed plugin.

1 Bevezetés

Az emberiség az ókor óta foglalkozik a hanggal és a hallással valamilyen formában. Elég megnézni egy ókori görög színház felépítését, vagy a mai színházak kialakítását, amik mind azt a célt szolgálták, hogy a nézőközönség minél jobb akusztikai élményben részesülhessen.

A teremakusztikai tervezés mellett az elektroakusztikus átalakítók (mikrofonok, hangfalak, rádiók és televíziók) elterjedésével jóval kiszélesedett a lehetőségek tárháza és megjelentek a különböző hangtechnikai eszközök is, köztük a dinamikaszabályzó eszközök. Ezeket az eszközöket a broadcast (élő közvetítés) átviteli csatornájának, valamint a PA (Public Address) rendszereknek az elektroakusztikus átalakításának korlátai hívták életre. Kezdetben elektroncsöves megoldások kezdtek elterjedni, majd az átalakítók a tranzisztorok megjelenésével egyre elérhetőbbek és kisebbek lettek. Az 1980-as években pedig a számítógépek és a hozzájuk tartozó szoftveres megoldások is fokozatosan megjelentek a stúdiókban. [1]

Ezek a szoftveres megoldások a mai napig próbálják minél jobban modellezni a fizikai eszközöket működésükben és hangzásukban. Ezzel ellentétben igyekeztem egy minél semlegesebb, szinte már mesterséges hangzású effektet létrehozni, ami alapul tud szolgálni más effektek megvalósításához.

Azért esett a választásom egy dinamikaszabályzó kompresszor megvalósítására, mert hobbiként több éve dolgozom mind élő rendezvényeken, mind televíziós adásokon, valamint a hangstúdiózásba is belekóstoltam egy kicsit, így személyesen is sok tapasztalatom van ezeknek az eszközöknek a szükségességében, felépítésében és működésében.

2 A hang és a dinamika

A hallás az emberi érzékelés egyik fajtája és az emberek közötti kommunikáció része. Az emberi hallásról elmondható, hogy nem lineáris, hanem logaritmikus mind a hangmagasság (frekvencia), mind a hangerő (amplitúdó) tekintetében. Ez szemléletesen azt jelenti, hogy egységnyi (érzett) hangosságkülönbséghez kétszerezni kell a fizikai amplitúdó szintet és egységnyi (érzett) hangmagasság különbséghez kétszerezni kell a hang fizikai frekvenciáját. Ebből adódóan a hang jellemzésére is általában logaritmikus léptéket szokás használni. A frekvenciánál jellemzően a megjelenítéshez, mivel az átfogott frekvenciatartomány 3 dekádnak felel meg (20 Hz – 20 kHz, ~10 oktáv, ami tízszer nagyobb, mint a látásé). Viszont, az amplitúdó és hangerő jellemzésére a köznyelvben is a decibel terjedt el, aminek a viszonyítási pontja a hallás küszöbszintje (0 dBSPL, 20 μPa effektív érték 1 kHz frekvencián), felső határa pedig a fájdalomküszöb szintje, nagyjából 130 dBSPL, így az emberi hallás dinamikatartománya több mint 12 nagyságrendnyi teljesítményszintet fog át.

A hang egy közegben terjedő longitudinális hullám. Ezeknek a hullámoknak a rögzítésére először mechanikusan került sor viaszcsövekkel, majd később bakelit lemezekkel.

Az elektromosság és az elektronika megjelenése után kezdtek el mikrofonokat és hangszórókat gyártani, amikkel erősíteni lehetett a hangot, a korábbi szócsövek helyett. A mikrofon a mechanikai mozgást (közegben terjedő hullám) elektromos jellé alakítja, a hangszóró pedig az elektromos jelet hullámmá. A dinamikatartomány ezen elektromos vagy mechanikai hullámok egy adott időablakon belül a legkisebb és a legnagyobb jelszintjének aránya. Ezeket az elektromos jeleket először lemezekre, filmekre rögzítették, majd a technológia fejlődésével elterjedtek a kazetták is. Ezek a rögzítési formák analóg módon tárolták a jeleket és kialakításukból adódóan természetes torzítással rendelkeztek.

Ezzel egyidőben egyre elterjedtebbek lettek a valós idejű adások, először rádió, majd pedig televízió formájában.

Mind a felvételeknél, mind az élő továbbításban egyre fontosabbá vált ennek a dinamikának a szabályozása, mivel sem a mikrofonok, sem a rögzítő eszközök, de még a

lejátszó eszközök sem rendelkeznek végtelen dinamikatartománnyal, de még az emberi beszéd vagy hallás dinamikatartományánál is jóval kisebb ez a sáv.

Ez azért okoz problémát, mert ha a forrás dinamikája lefele lóg ki a vevő tartományából, akkor információ veszik el, zajjá változik, ha pedig felfele, akkor torzul a jel. Ez nem biztos, hogy információvesztéssel jár, viszont az élethű reprodukálhatóságot jelentősen rontja.

Ennek a problémának a kiküszöbölésére kezdtek el dinamikaszabályzó eszközöket gyártani. Ezeket először rádióadókba és vevőkbe építették be, hogy adóoldalon az adott csatorna, a vevőoldalon pedig a csatornák közötti váltás során ne legyenek hirtelen túl nagy jelszintugrások.

Az első, piacon kapható ilyen eszközök láthatóak a 2.1. ábrán. Elsőként a Telefunken U3-as rádióvevő volt kapható az 1930-as évek elején (amit az 1936-os Berlini olimpián használtak) [2], majd 1937-ben megjelent az első rádióadó beépített dinamikaszabályzóval, a Western Electric 110A [3].

2.1. ábra Telefunken U3 (balra) és Western Eletric 110A (jobbra)

A másik előnye a dinamikaszabályzók elterjedésének, hogy a jel kiugró csúcsainak levágása után megnövelhető a jel effektív értéke. Ez két dolgot is hoz magával, az egyik, hogy a hallgatók hangosabbnak érzékelik, így szívesebben hallgatják. Ez a 2020-as évekre odáig nőtt, hogy a mostanában kiadott zenék 95%-nak közel 0 a dinamikatartománya. Ez a hangerőverseny (Loudness war) addig fokozódott, hogy a televíziózásban és a rádiózásban szabványokat és ajánlásokat vezettek be a kijátszott jelek dinamikatartományára és maximális értékére. 1979-ben az EBU Tech. 3205-E szabványt vezették be, ami a mérőeszközök működését határozta meg [4], jelenleg pedig ennek az utódja, az EBU R 128 az irányadó, ami -23 LUFS-ban normalizálja az adások hangerejét [5].

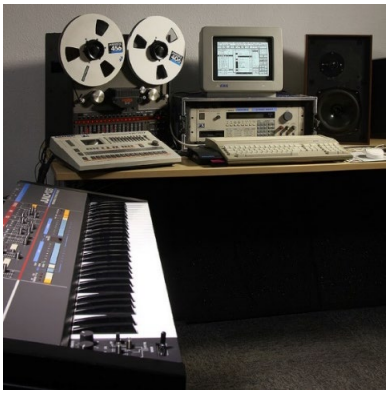
A másik eredménye, hogy azonos adóteljesítmény mellett messzebbről voltak foghatóak az AM (amplitúdó modulált) rádiócsatornák vagy azonos távolsághoz kisebb teljesítményű adó is elegendő volt.

Ezek az első eszközök még elektroncsöveket használtak a szabályzáshoz és a működésükhöz. Az elektronika fejlődésével megjelentek más alkatrészeket használó eszközök. Az 1960-as években megjelentek az optikai csatolókat (OPTO) vagy diódákat használó, majd a ’70-es években térvezérlésű tranzisztorokat (FET) és feszültségvezérelt erősítőket (VCA) tartalmazó dinamikaszabályzók.

A legelterjedtebb kompresszorok láthatóak a 2.2. ábrán. Az OPTO csatolós kompresszorok közül a Teletronix LA-2A (bal fent), diódás kompresszorból a Neve 33609 (bal középen), FET-es kompresszorból a Universal Audio 1176 (bal lent), VCA alapúból pedig a dbx 160 (jobbra) volt a legelterjedtebb.

2.2. ábra Legelterjedtebb kompresszorok

1980-ban megjelentek a stúdiókban a számítógépek és velük együtt a digitális hangfeldolgozás a hozzá szükséges szoftverekkel együtt. Erről láthatunk egy képet a 2.3. ábrán.



2.3. ábra Rekreált 80-as évekbeli otthoni stúdió

3 Dinamikaszabályzók

3.1 Típusok

A dinamikaszabályzók működésük függvényében több különböző kategóriába sorolhatók. Ezeket mutatja be a 3.1. ábra.

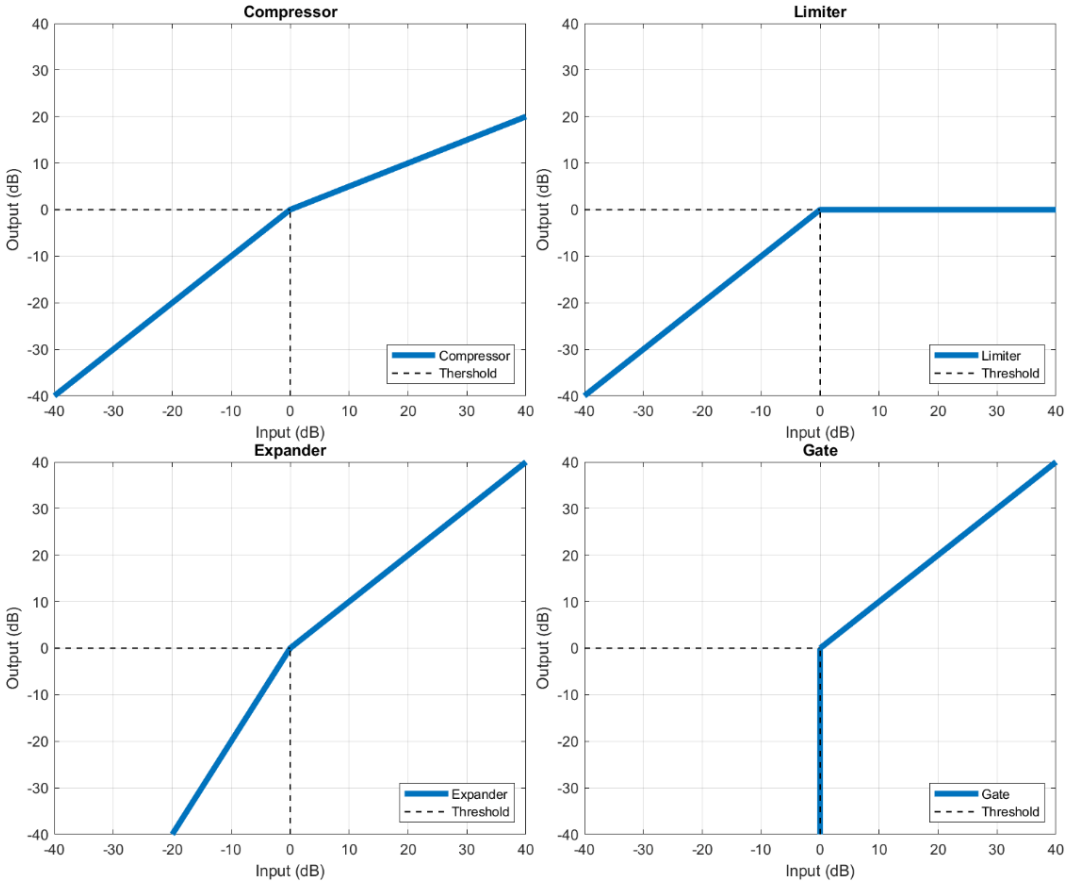
A kompresszor egy olyan dinamikaszabályzó, ami a beállított jelszint fölötti bemenet esetén egy adott aránnyal halkítja a kimenetet. Ezt az arányszámot a be- és kimeneti jelek decibelben számított jelszintjére értelmezzük, ami lineárisan nézve hatványozásként jelenik meg.

Az egyik legegyszerűbb formája a limiter, ami egy beállított jelszint fölé nem engedi a kimenetet. Ez megfeleltethető egy kompresszornak, aminek az aránya ∞:1.

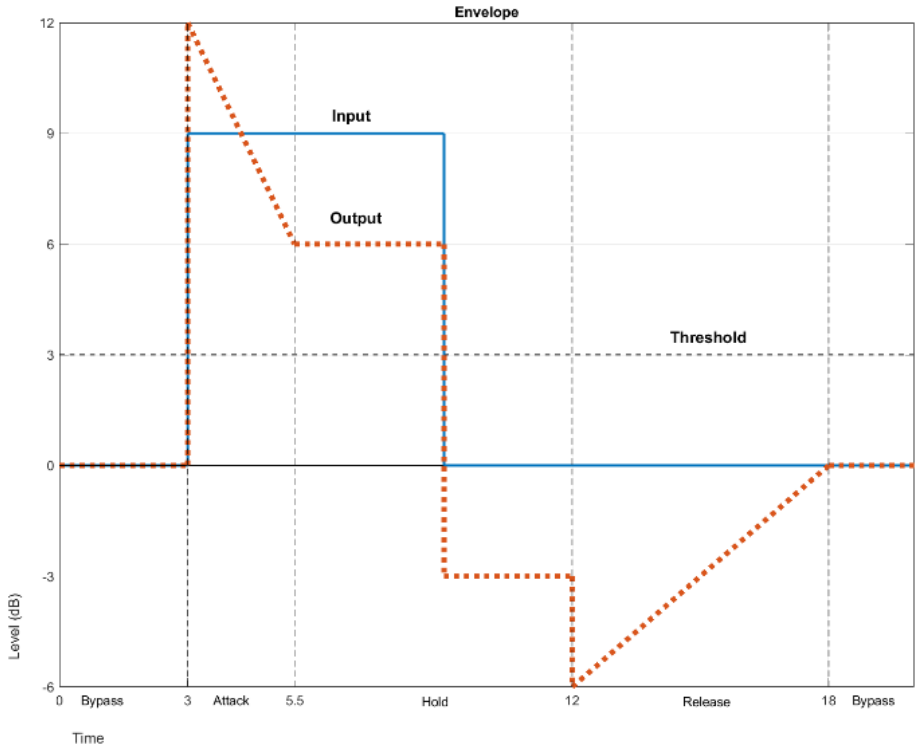
Az expander egy fordított működésű kompresszor. A beállított jelszint alatti bemenet esetén a meghatározott aránnyal halkítja a kimenetet, ezzel növelve a kimeneti dinamikatartományt.

A zajzár (gate) egy fordított működésű limiter. Egy beállított jelszint alatt lecsökkenti a kimeneti jelszintet egy beállítható értékkel. Ez megfeleltethető egy expandernek, aminek az aránya 1:∞

Léteznek még envelope alapú kompresszorok, ahol a jel szabályzása egy megadott görbe szerint történik. A 3.2. ábra mutatja egy lehetséges megvalósítását. Jól látható a bekapcsolási és kikapcsolási tranzienseknél a kiugró tüske a 3.5. ábrával szemben.



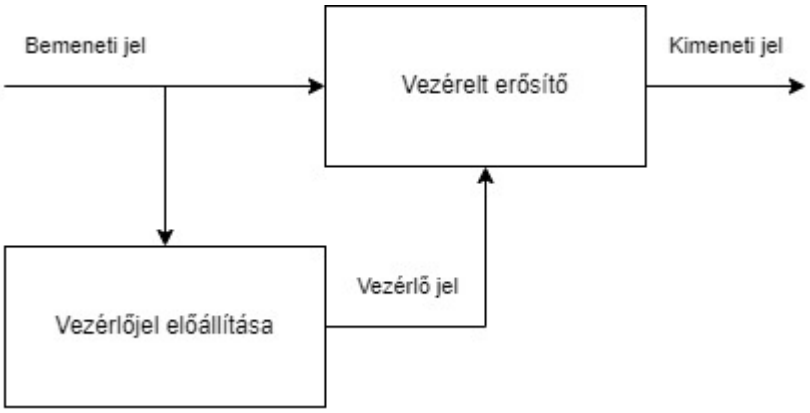
3.1. ábra Kompresszor, Limiter, Expander, Gate átvitelének összehasonlítása



3.2. ábra Envelope szabályzás

3.2 Felépítés

Mint látható, egymáshoz nagyon hasonlóak a különböző működési formák. A legegyszerűbben egy vezérelhető erősítőből és egy vezérlő jelet előállító egységből hozhatók létre, ahogy az a 3.3. ábrán is látható. A vezérlőjel előállítása először a bemeneti jelszint mérésével kezdődik, majd ennek a paramétereknek megfelelő átalakításával áll elő a vezérlő jel.



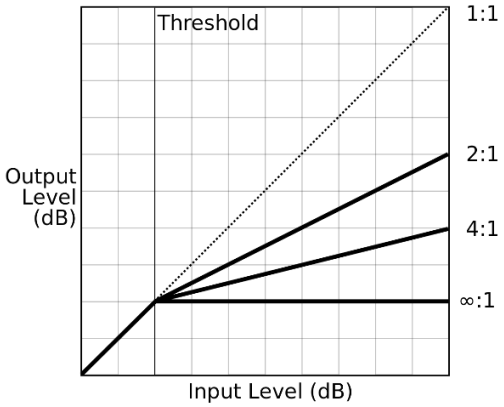
3.3. ábra Dinamikaszabályzó általános blokkdiagramm

3.3 Paraméterek

A dinamikaszabályzók működésüktől és típusuktól függően más-más paraméterekkel rendelkezhetnek. E paramétereknek általános listája:

Threshold: Ez a paraméter kivétel nélkül megtalálható a dinamikaszabályzókon. Az itt beállított érték határozza meg, hogy mikor kezd el működni a szabályzó. Általában dB-ben lehet megadni. Az adott eszköztől függően ez lehet dBU, dBV vagy digitális feldolgozás esetén legtöbbször dBFS.

Ratio: Ez a paraméter határozza meg a szabályozó „erősségét” és irányát. Általában két szám, a bemeneti és kimeneti változás arányaként szemléltetik. Ahogy az a 3.4. ábrán is látható, például 2:1 ratio jelentése, hogy 2 dB bemeneti erősödésre 1 dB-lel emelkedik a kimenet, azaz valamilyen kompresszort valósít meg. Fordítva (1:2) pedig 1 dB bemeneti erősödésre 2 dB-el emelkedik a kimenet, azaz valamilyen expandert valósít meg.

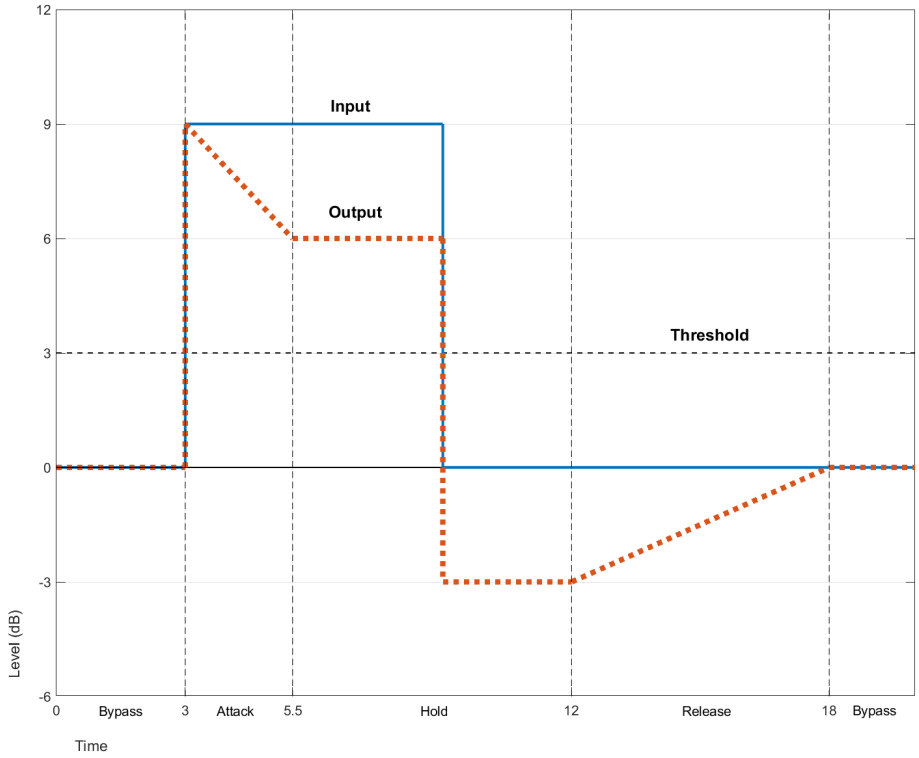


3.4. ábra Különböző erősítési arányok

Attack time: Ez a paraméter határozza meg, hogy a bekapcsolási tranziens mennyi idő alatt fut le. A vezérlőjel ennyi idő alatt éri el nulláról a kívánt értéket, dB-ben mérve lineárisan.

Hold time: Ez a paraméter határozza meg, hogy a bekapcsolás után legalább mennyi ideig marad aktív a vezérlés. A vezérlőjel az attack fázis után ennyi ideig marad konstans, ha csak nem kell rajta növelni.

Release time: Ez a paraméter határozza meg, hogy a kikapcsolási tranziens mennyi idő alatt fut le. A vezérlőjel ennyi idő alatt éri el az aktuális értékéről a nullát, dB-ben mérve lineárisan.

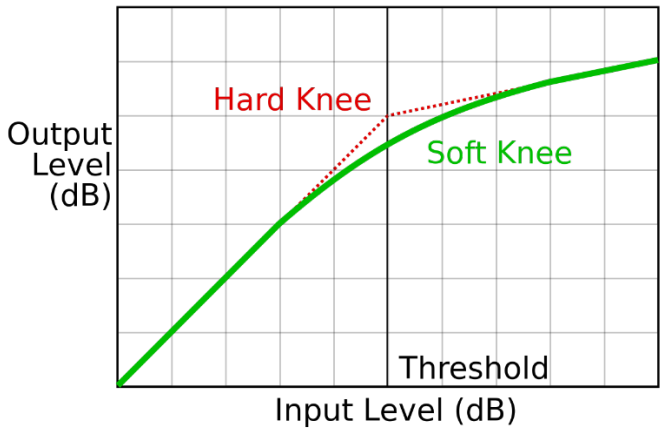


3.5. ábra Attack, Hold és Release szemléltetése

Makeup gain: Más néven kimeneti erősítés. Ez lehet pozitív vagy negatív is. Annak függvényében, hogy kompresszort vagy expandert használunk, általában szükséges a kimeneti jel erősítése vagy gyengítése, ezért a legtöbb eszközbe beépítenek egy erősítőt is, közvetlenül a kimenet elé.

Mérési forma: A legtöbb eszközön lehet választani, hogy a bemeneti jelszintet RMS vagy Peak mérés alapján határozza meg. Sokszor az RMS mérés időablakát is lehet változtatni igaz, csak két érték között. Ez a két érték jellemzően 125 ms és 1s szokott lenni. Elő szokott még fordulni 35 ms is, de ez már annyira rövid időablak, hogy impulzuszerű jelek is megjelennek rajta, így jó közelítéssel peak mérés valósul meg. Ennek pontos ellentéte miatt nem érdemes túl hosszú időablakot se hagyni, mert minél összemérhetőbb az idő a vizsgált jel hosszával, annál inkább közelít egy konstans értékhez a kapott RMS.

Soft knee: A threshold és a ratio egy töréspontos görbét határoz meg a működéshez. Ahhoz, hogy az emberi fül számára kellemesebb lehessen a működés, ezt a töréspontot ki lehet simítani, így folyamatosabb átmenetet létrehozva a két állapot között. Ez a két különböző görbe látszik az 3.6. ábrán

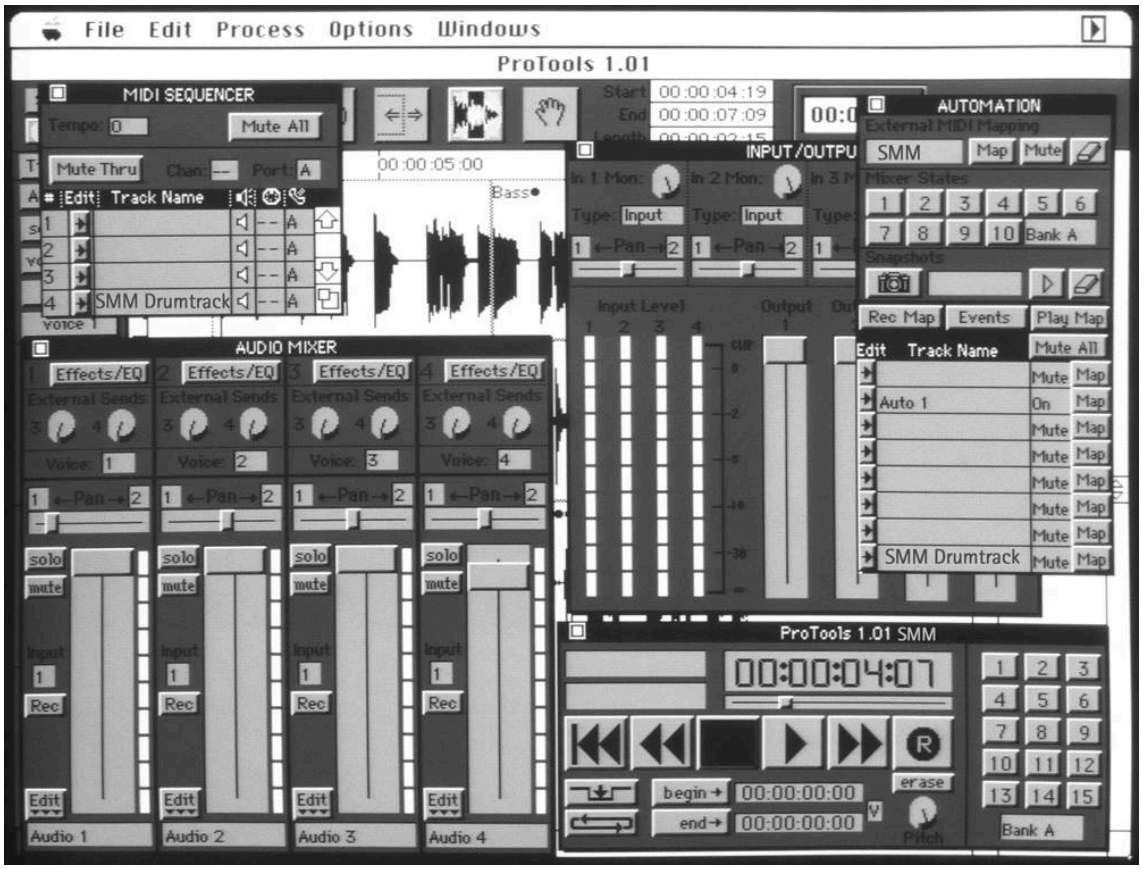


3.6. ábra Soft Knee

Dual / Master-Slave / Link: Általában kétcsatornás fizikai eszközökre jellemző, de előfordul többcsatornás eszközökön is. Dual állásban a különálló csatornák feldolgozása egymástól függetlenül történik. Azonban sztereó jelek feldolgozásakor ez az oldalak közti egyensúlyt el tudja tolni valamelyik irányba. Master-Slave vagy linkelt (összekapcsolt) állapotban több kimeneti vezérelt erősítőt azonos vezérlőjellel látunk el, így minden jel egységesen hangosodik vagy halkul, megtartva a sztereó hatást.

4 Szoftverek, pluginek

A számítógépek stúdiókban való elterjedésével megjelentek az eszközöket digitalizáló programok is. Ezeket digitális audio munkaállomásoknak (Digital Audio Workstation, röviden DAW) nevezzük. Ezek a programok kezdetben digitális fizikai eszközökkel voltak összekötve és így alkottak egy rendszert. Idővel egyre több funkciót implementáltak a programokba, így ma már bármilyen más eszköz nélkül lehet hangfelvételeket készíteni, szerkeszteni. Az első DAW, ami a nagy stúdiókban is elterjedt a Digidesign cég 1991-ben kiadott Pro Tools szoftvere volt. Ennek a programnak a kezelőfelülete látható a 4.1. ábrán.



4.1. ábra Pro Tools 1.01 kezelőfelülete

1992-ben a német Steinberg cég kiadta Cubase Audio nevű DAW szoftverét, ami 1996-ban frissült a pluginek támogatását jelentő VST (Virtual Studio Technology) szabvánnyal. Ezzel lehetővé vált külső programokat integrálni a DAW-ba, ezzel bővítve a funkciókat, lehetőségeket. Ennek a szabványnak a továbbfejlesztett változatát, a VST 3-at a mai napig használják a legnagyobb cégek. Ezeket a pluginokat nem csak az audio

szerkesztőkben, hanem akár lejátszókban, vagy fizikai eszközökben is lehet használni. A szabvány tartalmaz olyan jól meghatározott interfészeket, ami minden plugin működéséhez szükségesek. Ilyenek például a ki- és bemeneti audio bufferek kezelése, paramétetek, üzenetek továbbítása a grafikus felület, az audio processor és a host program között, MIDI jelek kezelése.

A cég publikusan elérhetővé tette a VST 3 pluginok fejlesztéséhez készített környezetet (Software Development Kit, röviden SDK) GitHubon [6]. Ennek segítségével bárki tud több host programmal kompatibilis plugineket fejleszteni, anélkül, hogy egy-egy programnak a saját megoldását kéne megtanulnia, vagy a szabványnak megfelelően létrehoznia az alapvető funkciók, részfeladatok biztosításához szükséges kódokat.

Az SDK tartalmaz egy API-t (Application Programming Interface), ami megkönnyíti a host programmal történő integrációt. Tartalmaz előre megírt osztályokat, amelyek segítik a plugin implementációját. Találhatóak benne még wrapperek, én ezeket nem használtam. Emellett tartalmaz példa plugineket, amelyek sokat segítettek a plugin implemetációja közben.

Az SDK mellett a cég közzétett egy projekt generátor programot is, ami az alapvető információk kitöltése után létrehoz egy projektet a szükséges header és cpp fájlokkal [7].

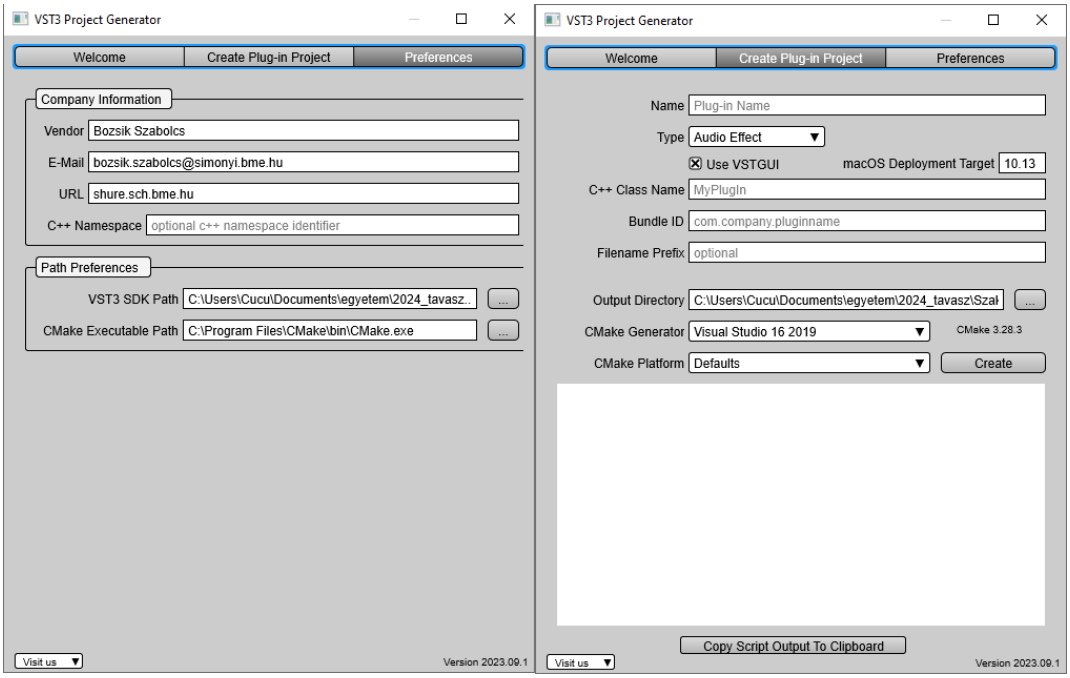
A 4.2. ábra mutatja a generátor program felületét. A preferences részen meg kell adni a készítő cég adatait, ami esetemben a saját adataim voltak. Itt kell még megadni a telepített SDK és a CMake program elérési útvonalát.

A plugin létrehozásához meg kell adni, hogy mi legyen a plugin neve, milyen típusú lesz a plugin (audio effect vagy instrument) és hogy szeretnénk-e használni a VSTGUI-t. Az audio effect alapvetően rendelkezik hang be- és kimenettel, és a kapott hangmintán hajt végre valamilyen változtatást. Az instrument (VSTi) pedig kimeneti hangot generál önállóan (például egy szinusz generátor) vagy MIDI üzenetek alapján (például szintetizátor).

Ezek után lehet megadni, ha szeretnénk egyedi C++ osztály nevet vagy fájlnév előtagot, én ezeket üresen hagytam. Itt kell még megadni a Bundle ID-t, ami egy teljesen egyedi, az adott pluginhez kapcsolt azonosító.

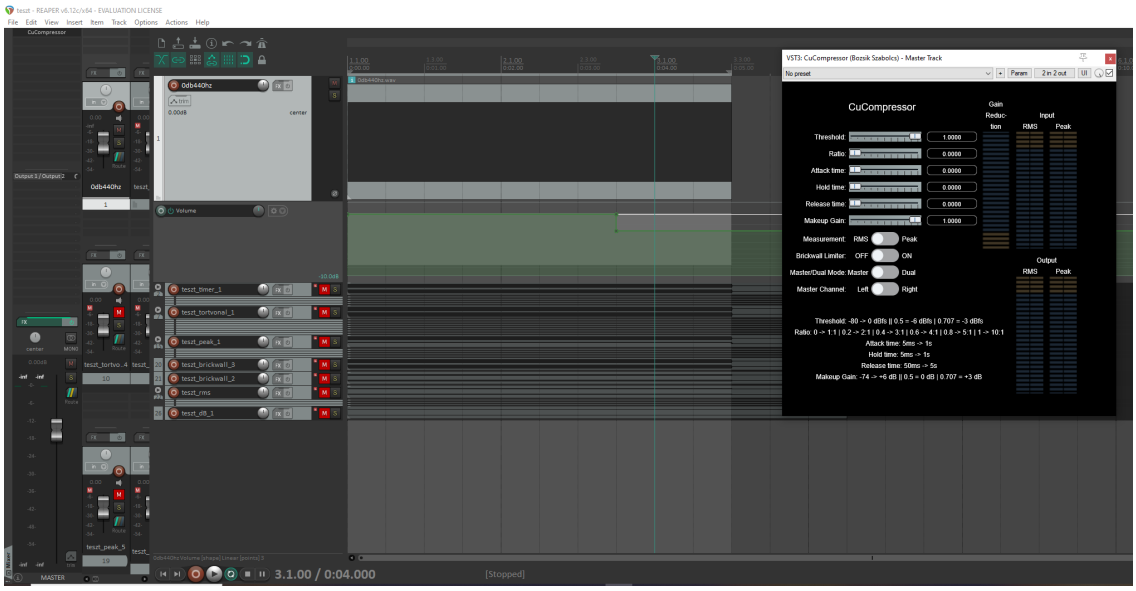
Utolsó lépésként meg kell jelölni a helyet, ahova a program létrehozza a plugint, valamint a CMake programnak szükséges változókat, hogy melyik fejlesztő környezetet használjuk és milyen platformon. A plugint Microsoft Visual Studio 16 2019-ben írtam, a platformot alapbeállításon hagytam.

Ha minden adat ki van töltve, a Create gomb megnyomásával a program létrehozza a plugint, a szükséges fájlokat és megnyitja a fejlesztőkörnyezetet, benne a projektfájllal. A generátor létrehozza a plugin két fő osztályát és a hozzájuk tartozó header fájlokat, a processort, ami az audio minták feldolgozásáért felelős valamint a controller osztályt, ami pedig a paraméterezhetőségért, a grafikus kezelőfelület létrehozásáért és a felhasználóval való interakciók megvalósításáért felel. Ezek a létrehozás után teljesen üres vázak, megvannak bennük a host program által hívható függvények fejlécei és némi magyarázat ezekhez, de semmilyen feldolgozást, paraméterezést vagy grafikus felületet nem tartalmaznak.



4.2. ábra VST generátor felülete

A plugin készítése közbeni hibakereséshez és a végső teszteléshez a Cockos cég Reaper nevű DAW programját használtam korábbi tapasztalataim miatt. Ennek a kezelőfelületét mutatja a 4.3. ábra.



4.3. ábra Reaper kezelőfelülete

5 Audio jelfeldolgozás megvalósítása

5.1 Tervezés

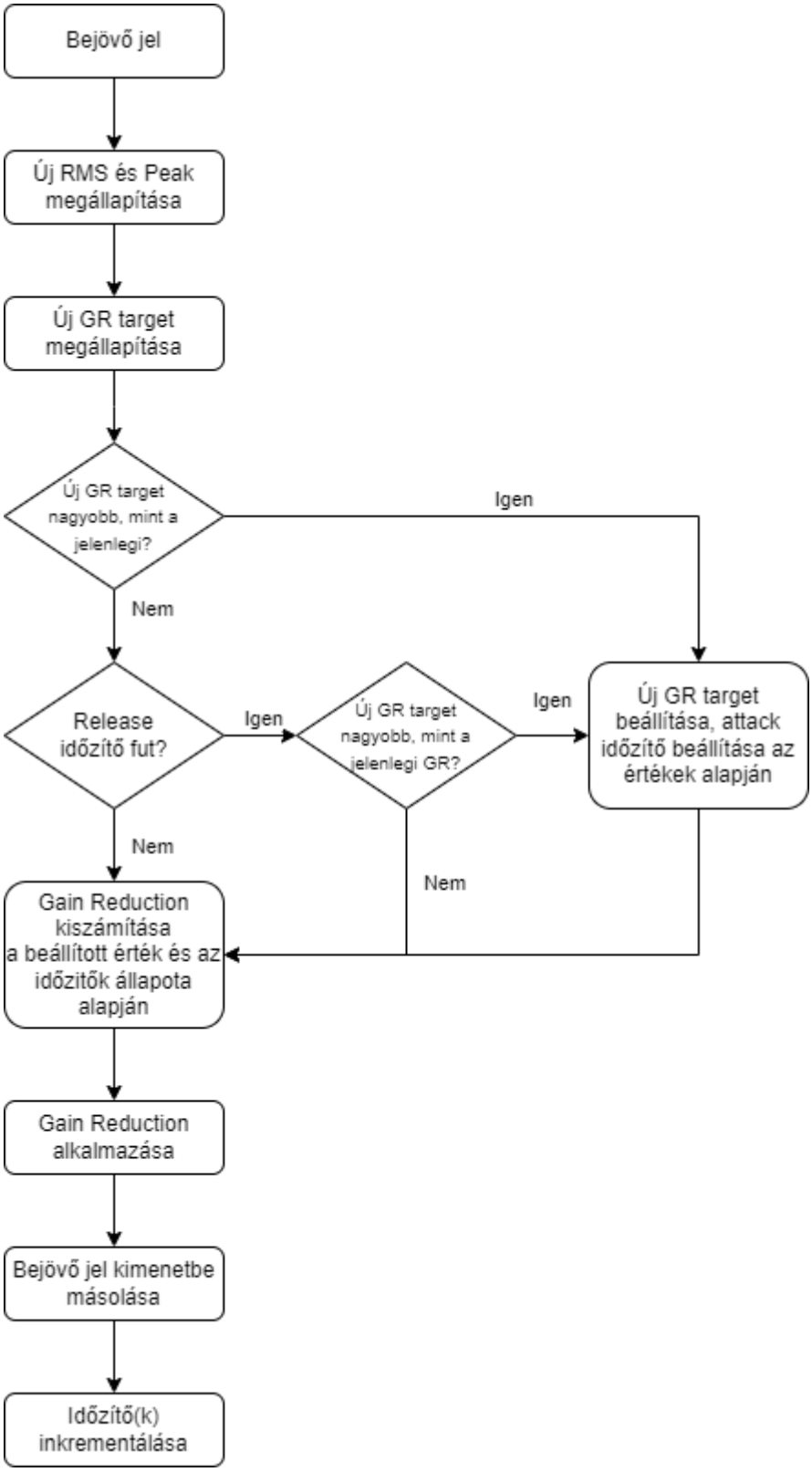
Első lépésként a plugin működésének és funkcionalitásának meghatározása volt szükséges. Egy kompresszor megvalósítására esett a választás, mert korábban sokat dolgoztam ilyen eszközökkel, így jól ismertem a működésüket, viselkedésüket. A pluginnak tudnia kellett a következő paramétereket kezelni:

- Threshold
- Ratio
- Attack
- Hold
- Release
- Makeup gain
- RMS vagy Peak mérés
- Dual és Master-Slave működés választható master csatornával
- Brickwall limiter

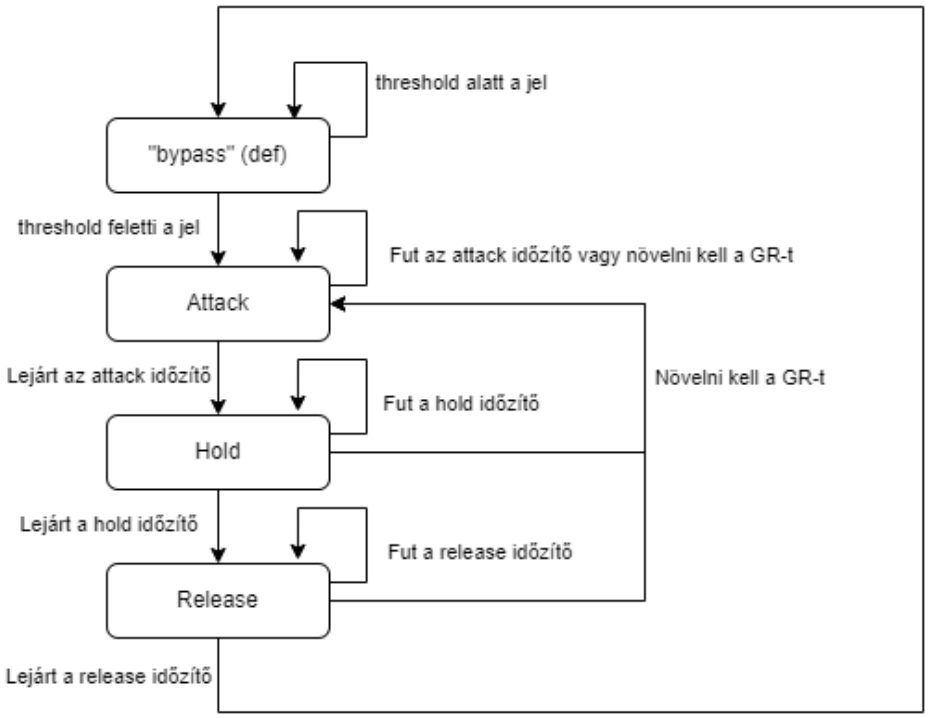
A paraméterek azonosítása után megterveztem a plugin audio feldolgozási folyamatábráját, ami az 5.1. ábrán látható és a plugin állapotgépét, amit az 5.2. ábra mutat.

Megfigyelhető, hogy külön választottam a gain reduction targetet és az aktuális gain reductiont. Erre azért volt szükség, mert a target egy elérni kívánt célérték, ami ugrásszerűen is változhat, és ebből az időzítők állása alapján számítom ki az aktuális értéket, így biztosítva, hogy ne ugorjon a jel, hanem folyamatos átmenetet képezzen bármilyen változtatás hatására.

Ezeket a feldolgozásokat a processor osztály valósítja meg.



5.1. ábra Audio processzálás folyamatábrája



5.2. ábra Plugin állapotgépe

5.2 Megvalósítás

5.2.1 Audio kezelése

Ahhoz, hogy a plugin tudjon audiot fogadni és küldeni magából, létre kell hozni egy be- és egy kimeneti buszt. Ezt az inicializáláskor tesszük meg, amikor példányosodik az osztályunk. Ennek a kódja látható az 5.3. ábrán.

```
tresult PLUGIN_API CuCompressorProcessor::initialize (FUnknown* context)
{
    // Here the Plug-in will be instantiated

    //---always initialize the parent-------
    tresult result = AudioEffect::initialize (context);
    // if everything Ok, continue
    if (result != kResultOk)
    {
        return result;
    }

    //--- create Audio IO ------
    addAudioInput (STR16 ("Stereo In"), Steinberg::Vst::SpeakerArr::kStereo);
    addAudioOutput (STR16 ("Stereo Out"), Steinberg::Vst::SpeakerArr::kStereo);

    /* If you don't need an event bus, you can remove the next line */
    addEventInput (STR16 ("Event In"), 1);
```

5.3. ábra Audio ki- és bemenetek létrehozása

A létrehozáskor elnevezzük őket és megadjuk, hogy milyen típusúak. Esetünkben egy egyszerű sztereó típust választottam, de lehet választani monó vagy akár 5.1 csatornás buszokat is. Az audio csatornák száma alapvetően meghatározza, hogy a későbbiekben hogyan dolgozzuk fel azokat, így fontos az elején rögzíteni ezt. Sok pluginból érhető el külön monó és sztereó változat ezért.

Ezek után a process nevű függvényben történik a hangminták tényleges feldolgozása. Ez a bufferek beolvasásával kezdődik, ez látható az 5.4. ábrán. Először beolvassuk, hogy hány csatornánk van. A getChannelBuffersPointer függvénnyel lekérjük, hogy a be- és kimeneti buffereink melyik memóriacímen találhatóak. Ez egy void** típusú változót fog visszaadni, mivel van lehetőség 32 és 64 bites mintákkal is dolgozni. Mivel a canProcessSampleSize függvényben nem tettem elérhetővé a 64 bites mintákat, így biztos csak Sample32 típussal fogok dolgozni, emiatt rögtön a beolvasásakor át is kasztolom ilyen típusba. Az így kapott pointerek egy kétdimenziós tömbre mutatnak, aminek kettő sora és minta darabszámú oszlopa van, emiatt a minták darabszámát is elmentjük. Utolsó lépésként létrehozzuk és inicializáljuk a kettő futóváltozót.

```
int32 numChannels = data.inputs[0].numChannels;

//---get audio buffers----------------
uint32 sampleFramesSize = getSampleFramesSizeInBytes(processSetup, data.numSamples);
Sample32** in = (Sample32**)getChannelBuffersPointer(processSetup, data.inputs[0]);
Sample32** out = (Sample32**)getChannelBuffersPointer(processSetup, data.outputs[0]);
int32 samples = data.numSamples;
int32 j = -1;
int32 i = 0;

while (++j < samples)
{

    for (i = 0; i < numChannels; i++)
    {
```

5.4. ábra Audio beolvasása

A feldolgozás során egy while ciklussal végigmegyünk a mintákon, miközben egy beágyazott for ciklus végigmegy mindegyik csatornán.A minták -1 és 1 közé normalizált számok, ezek a szélsőértékek jelentik a 0 dBFS-t.

5.2.2 RMS és peak mérés, dB számítás

Miután a bejövő jel rendelkezésünkre áll, meg kell állapítani az aktuális peak és RMS értékeket, majd ezeket decibelben is eltárolni, a későbbi felhasználás miatt.

Mivel több helyen szükség lesz a normalizált és decibelben mért értékek közötti átváltásra, így létrehoztam erre 2 különálló függvényt, aminek a kódja az 5.5. ábrán látható.

A kiindulási egyenlet a decibel alapegyenlete volt, ami a következő:

$$dB_{fs} = 10\log_{10}\left(\frac{X_n^2}{FS^2}\right) = 20\log_{10}\left(\frac{X_n}{FS}\right) \xrightarrow{FS=1} 20\log_{10}(X_n)$$

```
float NormalizedTodBfs(float normValue)
{
    return 20.f* log10f(normValue);
}

float dBfsToNormalized(float dB)
{
    return expf(logf(10.f) * dB / 20.f);
}
```

5.5. ábra Normalizált és dBFS közötti átváltás

A két mért érték közül a peak mérés megvalósítása az egyszerűbb, mivel itt csak az aktuális bemeneti értékre, és az előző peak értékre van szükségünk. Ha az aktuális érték nagyobb, mint az eddigi peak érték, akkor elmentjük azt, ha pedig kisebb, akkor az eddigi peak értékét csökkentjük exponenciálisan egy konstanssal. Ennek a kódját mutatja az 5.6. ábra. A konstans értékét úgy választottam meg, hogy nagyjából 125 milliszekundum alatt érje el az eredeti érték 5%-át, hasonlóan, mint ahogy az RMS mérésnél is a gyors, 125 milliszekundumos időablakot választottam.

```
if (abs(in[i][j]) > fPeak[i])
{
    fPeak[i] = abs(in[i][j]);
    fPeakdB[i] = NormalizedTodBfs(fPeak[i]);
}
else
{
    fPeak[i] = fPeak[i] * kPeakExp;
    fPeakdB[i] = NormalizedTodBfs(fPeak[i]);
}
```

5.6. ábra Peak számítás

Az RMS (Root Mean Square, négyzetes közép) számítás már valamivel bonyolultabb. Először értelmezni kell a jelszintet, ami a jel energiatartalmával kapcsolatos, és a jel egy T hosszú szakaszából számítható, az alábbi képlettel [8]:

$$E = \int_{t=0}^{T} p^2(t)dt$$

A jel teljesítményét ennek az energiának az egységnyi időre vetítésével definiáljuk, majd diszkrét időben $p[k] = p(k\Delta t)$ összefüggéssel közelítjük:

$$P = \frac{E}{T} = \frac{1}{T}\int_{t=0}^{T} p^2(t)dt \approx \frac{1}{N\Delta t}\sum_{k=0}^{N-1} p^2[k]\Delta t = \frac{1}{N}\sum_{k=0}^{N-1} p^2[k] = p_{RMS}^2$$

Az így kapott P jelteljesítmény megegyezik az N mintából számolt RMS értékének négyzetével. Ez az érték az adott időpillanat előtt minden értéket figyelembe vesz, nekünk viszont csak egy adott hosszúságú időablak alapján számított értékre van szükségünk, ehhez definiáljuk a jel futó RMS szintjét, az alábbi csúszóablakos átlagolással:

$$p_{RMS}^2(t) = \frac{1}{T}\int_{t-T}^{t} p^2(\tau)d\tau$$

Ebből látható, hogy az aktuális értékét az előző T időablaknyi N darabszámú minta befolyásolja. Ezek azonban a VST működése miatt (ha csak nem mentjük el a mintákat egy független memóriaterületre) nem biztos, hogy rendelkezésünkre fognak állni, amikor szükség van rájuk. Ennek kiküszöbölésére az előző mintákat nem négyszög, hanem azzal azonos területű, végtelen mély exponenciális csúszóablakkal súlyozzuk. Az így adódó új definíció:

$$p_{RMS}^2(t) \approx \frac{1}{T}\int_{-\infty}^{t} p^2(\tau)e^{-\frac{t-\tau}{T}}d\tau$$

Az integrálást szummázással közelítjük, így az RMS számításához használt képlet az alábbi lesz:

$$p_{RMS}^2[k] = \frac{1}{N\Delta t}\sum_{i=-\infty}^{k} p^2[i]e^{-\frac{(k-i)\Delta t}{T}}\Delta t = \frac{1}{N}\sum_{i=-\infty}^{k} p^2[i]\alpha^{k-i} = \frac{1}{N}p^2[k] + \alpha p_{RMS}^2[k-1]$$

Látható, hogy ennél a módszernél csak az aktuális jelre és az előző RMS értékre van szükségünk, hogy ki tudjuk számolni az új RMS értéket. Intuitíven látszik, hogy ahhoz, hogy egy konstans jel RMS értéke konstans legyen, $\frac{1}{N} + \alpha = 1$ egyenletnek teljesülnie kell, azaz $\frac{1}{N}$ helyettesíthető $1 - \alpha$ kifejezéssel. Ennek az egyenletnek a kódbeli

megfeleltetése az 5.7. ábrán látható. Az egyenletben található α konstans kiszámítása az alábbi módon történik:

$$\alpha = e^{-1/N} \quad \text{ahol} \quad N = \frac{T}{\Delta t} \quad \text{és} \quad \Delta t = \frac{1}{fs} \Rightarrow N = T * fs$$

A konstans kiszámítása a plugin példányosításakor történik, mivel korábban nem áll rendelkezésre a mintavételi frekvencia, később pedig felesleges processzor ciklusokat használna fel, ha újra és újra ki kellene számítani. A konstans kiszámítását az 5.8. ábra mutatja.

```
fRMS[i] = ((1.0 - kRMS_Alpha) * in[i][j] * in[i][j] + kRMS_Alpha * fRMS[i]); //calculate new RMS
fRMSdB[i] = NormalizedTodBfs(fRMS[i])/2.f;
```

5.7. ábra RMS kiszámítása

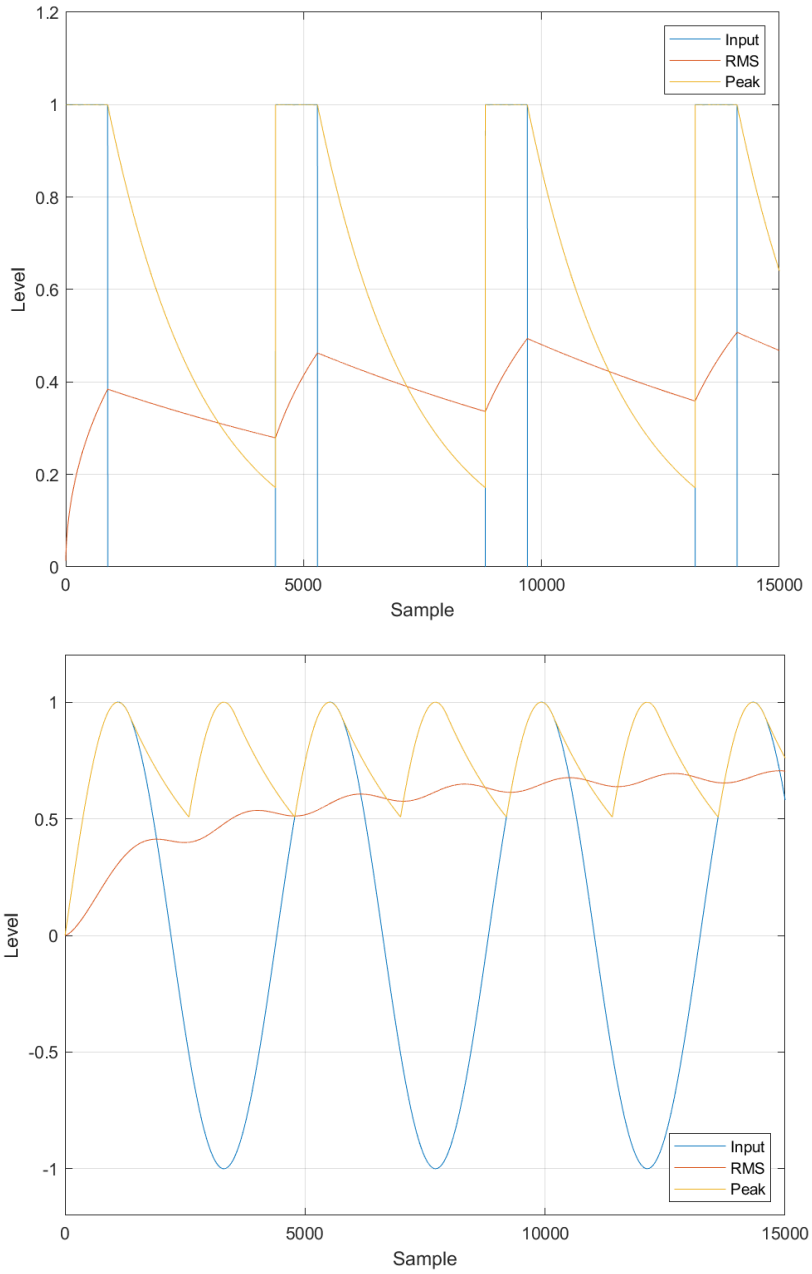
```
kSampleRate = this->processSetup.sampleRate;
kSampleTime = 1000.0 / kSampleRate;
kRMS_N = kSampleRate * 0.125; //125 ms for fast RMS
kRMS_Alpha = expf(-(1.0 / kRMS_N));
```

5.8. ábra RMS számításhoz szükséges konstans kiszámolása

Az új RMS megállapítása után ezt az értéket elmentjük decibelben is. Látható az 5.7. ábra kódjában egy kettővel történő osztás is a kapott decibel érték elmentése előtt.

Erre azért van szükség, mivel az 5.5. ábrán látható módon számolom a decibel értékeket, ehhez viszont az eredeti egyenletből kiemeltem egy négyzetre emelést. Mivel az RMS számításának a végeredménye az RMS értékének négyzete, így ez nem lenne kiemelhető, viszont meglátásom szerint egy kettővel történő osztás gyorsabb, mint egy külön függvényt írni erre vagy az RMS számításaként kapott értékből négyzetgyököt vonni.

Az 5.9. ábrán felül látható egy négyszögjel, alul pedig egy szinuszjel és a hozzájuk számolt RMS és peak értékek. Jól látható, hogy a peak értéke felsimul a bemenő jelformára, míg az RMS értéke nem.



5.9. ábra RMS és peak ábrázolása

5.2.3 Gain Reduction Target megállapítása

Miután rendelkezésünkre állnak a bemeneti jel értékei és a paraméterek, ki kell számítani, hogy mennyi gain reductionre lenne szükségünk. Ennek az általános képlete az alábbi:

$$\mathrm{GR}_{\mathrm{target}} = (\mathrm{jel}_{\mathrm{dB}} - \mathrm{Threshold}_{\mathrm{dB}})\left(1 - \frac{1}{\mathrm{Ratio}}\right)$$

A jel helyére a kiválasztott mérőjelnek az értékét kell behelyettesíteni decibelben (RMS vagy peak), a ratio helyére pedig az aránynak az értékét (például 4:1 arány esetén 4-et). Ez a számítási módszer biztosítja azt, hogy a jelnek csak a threshold feletti részét

vesszük figyelembe és az arány inverzének megfelelő mértéket vesszük el, vagyis 4:1 aránynál ¼-et akarunk megtartani, szóval ¾-et kell elvenni.

Az így kapott targetre két feltételt is ellenőrizni kell, az egyik, hogy ha negatív az érték, akkor nullával kell egyenlővé tenni. Erre azért van szükség, mert a negatív érték jelentése, hogy hangosítani kéne a jelen, ami egy másik működési formának felelne meg. A másik, hogy nem túl nagy-e, ez akkor fordulhat elő, ha túl gyorsan változik valamelyik bemeneti vezérlő paraméter. Itt az értéket 100 dB-ben maximalizálom, ami már jó közelítéssel a teljes csend.

Az ellenőrzött értéket két esetben kell elmenteni mint új target, ha ez nagyobb, mint a mostani, vagy release fázisban nagyobb, mint az aktuális gain reduction. Ha ezek közül egyik sem teljesül, akkor nem változtatjuk meg az eddigi targetet az újra.

5.2.4 Aktuális Gain Redution

Az aktuális Gain Reduction mutatja azt az értéket decibelben, amennyivel az éppen feldolgozott mintát csökkenteni kell. Ez a plugin aktuális processzálási állapotától függően más-más számítást jelent. Az 5.2. ábra alapján 4 állapot lehet, ezek között esetszétválasztással döntöm el, hogy melyik számítást használom. Ennek a kódja az 5.10. ábrán látható. Bypass esetén nincs processzálás, így a szükséges változtatás mértéke is nulla. Hold esetén nem változik a vezérlőjel, így a Gain Reduction sem változik. Attack esetén lineárisan próbáljuk elérni a target értéket, ezt az attack time paraméter (pAttackTime) és az attack timer állapota (tAttackTime) alapján tudom megvalósítani. Release esetén pedig pont ellentétesen, az aktuális target értéktől kiindulva próbálom elérni a nulla értéket lineárisan, amit szintén az időzítő és a paraméter felhasználásával tudok elérni.

```
switch (processor)
{
case MyCompanyName::CuCompressorProcessor::bypass:
    kGainReduction[i] = 0;
    break;
case MyCompanyName::CuCompressorProcessor::attack:
    kGainReduction[i] = pGainReductionTarget[i] * (tAttackTime / pAttackTime);
    break;
case MyCompanyName::CuCompressorProcessor::hold:
    kGainReduction[i] = kGainReduction[i];
    break;
case MyCompanyName::CuCompressorProcessor::release:
    kGainReduction[i] = pGainReductionTarget[i] * (1.0 - (tReleaseTime / pReleaseTime));
    break;
default:
    break;
}
```

5.10. ábra Gain Reduction kiszámítása

5.2.5 Kimeneti audio előállítása

Mielőtt a bemenetre alkalmaznánk az előállított vezérlőjelünket, még néhány feltételt meg kell néznünk. Először megnézzük, hogy Dual vagy Master-Slave üzemmódban van a kompresszor. Ha össze van kapcsolva a két oldal, akkor a kiválasztott oldalnak a vezérlőjelével felülírjuk a kapott vezérlőjelünket.

Ezek után, ha a brickwall limiter be van kapcsolva, akkor a vezérlőjeltől függetlenül a beállított thresholdnál elvágjuk a jelet, az ennél nagyobb értékeket felülírjuk a kimenetben. Ha ez nincs bekapcsolva, akkor a vezérlőjel alkalmazásával a kimeneti bufferbe másoljuk a bemenetet.

Végül a kimeneten alkalmazzuk a makeup gaint.

5.3 Processor felépítése

A host program a processornak a process függvényét hívja meg, amikor rendelkezésre áll a következő buffernyi minta, és fel kéne azokat dolgozni.

Első lépésként beolvasom, hogy milyen bemeneti paraméterek változtak és ezeket egy switch-case szerkezettel feldolgozom. Ennek egy részlete látható az 5.11. ábrán. A switch argumentuma a paramtéter ID-ja, amit a controllerben létrehoztam. A kód felépítése, hogy ha sikerül beolvasnom a paramQueue-ból (ami tartalmazza, hogy milyen paraméterek változtak) egy pontot a getPoint függvénnyel, akkor kezdem feldolgozni. A getPoint függvénynek meg kell adni, hogy az adott paraméterből hanyadikat olvassa be, mivel egy bufferen belül többször is változhat az értéke, ilyenkor több pontot vesz fel a host. Meglátásom szerint elég a másodpercenként nagyjából 90 frissítés a paramétereknek, így én mindig csak az adott paraméterből az utolsó hozzáadott pontot dolgozom fel. A függvény visszaadja még, hogy az adott pont, hány minta eltéréssel lett létrehozva, illetve mi lett a konkrét értéke. Mivel ezek az értékek egy 0 és 1 közé normalizált számok, így fel kell dolgoznom, hogy az adott paraméternél az adott érték mit jelent.

Az ábrán 4 paraméter feldolgozása látható. Az első a release time változása. Itt mivel úgy választottam meg a paramétert, hogy 50 ms és 5 s között lehet állítani, így a kapott normalizált értéket megszorzom 5000-rel, kivéve, ha 0,01 alatt van (ami megfelel 50 ms-nak), mert akkor közvetlenül 50 ms-re állítom az értékét. A második a Gain

reduction értéke, ezt nem kell feldolgoznom, mert ezt egy csak olvasható paraméterként vettem fel a grafikus visszajelzés miatt.

A harmadik a makeup gain, itt -74 és +6 dB közé választottam az állíthatóságot. A kapott értéket, ha kisebb, mint 0,0001 akkor egyenlővé teszem -74 dB-lel, egyébként pedig a korábban ismertetett függvényekkel kiszámítom dB-ben és hozzáadok 6 dB-t. Erre azért van szükség, mert 1-nek az átszámított értéke a 0 dB, és ezt szeretném eltolni 6 dB-lel. A 0,0001 pontosan -80 dB-nek felel meg, így ezt eltolva -74 és +6 közötti tartományt kapok.

A negyedik paraméter a measurement type, hogy RMS vagy peak mérés alapján szeretnénk vezérelni a kompresszort. Ez egy egyszerű bool változó, így ha 0,5 felett van az érték, akkor igaz, egyébként hamis.

```
case pReleaseTimeId:
    if (paramQueue->getPoint(numPoints - 1, sampleOffset, value) == kResultTrue)
    {
        if (value > 0.01)
        {
            pReleaseTime = value * 5000;
        }
        else
        {
            pReleaseTime = 50;
        }
    }
    break;
case kGainReductionId:
    break;
case pMakeUpGainId:
    if (paramQueue->getPoint(numPoints - 1, sampleOffset, value) == kResultTrue)
    {
        if (value < 0.0001)
        {
            pMakeUpGain = -74;
        }
        else
        {
            pMakeUpGain = NormalizedTodBfs(value) + 6;
        }
    }
    break;
case pMeasureTypeId:
    if (paramQueue->getPoint(numPoints - 1, sampleOffset, value) == kResultTrue)
    {
        if (value < 0.5)
        {
            pMeasureType = false;
        }
        else
        {
            pMeasureType = true;
        }
    }
    break;
```

5.11. ábra Paraméter változások beolvasása

A paraméterek beolvasása és feldolgozása után az 5.2 fejezetben ismertetettek szerint feldolgozom az audio jelet.

Miután egy adott időpillanathoz tartozó minden mintát feldolgoztam, megvizsgálom és beállítom a kompresszor következő állapotát. Ennek a megvalósítása látható az 5.12. ábrán. Az állapotok egyszerűbb követhetőségéhez létrehoztam egy enum típust, amibe felvettem a kompresszor 4 állapotát. A kódban alapvetően az 5.2. ábra bal oldalán látható egyenes ágon haladok végig. Az időzítők növeléséhez szükségem van egy minta idejére milliszekundumban, amit az 5.8. ábrán látható módon számítok ki. Az állapotgépen látható még, hogy bármelyik fázisból át kell lépni attack fázisba, ha meg kell növelni a gain reductiont. Ezt rögtön a target kiszámítása után megteszem, hogy a feldolgozás már az új értékekkel tudjon megtörténni.

```
if (processor == attack && tAttackTime < pAttackTime)
{
    tAttackTime = tAttackTime + kSampleTime;
}
else if (processor == attack && tAttackTime >= pAttackTime)
{
    processor = hold;
}
else if (processor == hold && tHoldTime < pHoldTime)
{
    tHoldTime = tHoldTime + kSampleTime;
}
else if (processor == hold && tHoldTime >= pHoldTime)
{
    processor = release;
}
else if (processor == release && tReleaseTime < pReleaseTime)
{
    tReleaseTime = tReleaseTime + kSampleTime;
}
else if (processor == release && tReleaseTime >= pReleaseTime)
{
    kGainReduction[0] = 0;
    kGainReduction[1] = 0;
    pGainReductionTarget[0] = 0;
    pGainReductionTarget[1] = 0;
    tAttackTime = 0;
    tHoldTime = 0;
    tReleaseTime = 0;
    processor = bypass;

}
```

5.12. ábra Állapotgép megvalósítása

Az összes audio minta feldolgozása után elküldöm a hostnak a kimeneti paramétereimnek az új értékét, amit ezek után a host elküld a controllernek, ami megjeleníti azt a grafikus felületen. Ennek a kódja látható az 5.13. ábrán. Először elmentem a kimeneti paraméterek helyére mutató pointert. Ha ez sikeres, akkor ide létre kell hozni minden paraméternek egy listát. Ezt az addParameterData függvénnyel lehet megtenni, aminek az argumentuma az adott paraméternek az ID-ja, valamint egy index, amin a függvény visszaadja, hogy a kimeneti paraméterek között ez hanyadik. Ha ez is sikeres, akkor a paraméter listájának a végéhez hozzáadok egy adatpontot az addPoint

függvénnyel, aminek az argumentumában meg kell adni, hogy hány minta eltolással legyen érvényes az új érték, én itt mindig nullát adok át, mivel nem szükséges egy feldolgozáson belül eltolnom több pontot. Át kell még adni a paraméter értékét, valamint egy indexet, amiben a függvény visszaadja, hogy ez hányadik adatpont a listában.

Miután minden kimeneti paramétert átadtam a host programnak, visszatér a process függvény kResultOK értékkel, hogy a feldolgozás sikeres. Ezek után kezdődik a feldolgozás újra az új mintákkal.

```
//---3) Write outputs parameter changes-----------
IParameterChanges* outParamChanges = data.outputParameterChanges;
// the new values will be send to the host
// the host will send it back in sync to our controller for updating our editor
if (outParamChanges)
{
    int32 index = 0;
    int32 index2 = 0;
    IParamValueQueue* paramQueue = outParamChanges->addParameterData(kRatioActualId, index);
    if (paramQueue)
    {
        paramQueue->addPoint(0, kRatioActual, index2);
    }
    paramQueue = outParamChanges->addParameterData(kGainReductionId, index);
    if (paramQueue)
    {
        paramQueue->addPoint(0, dBfsToNormalized(-kGainReduction[0]), index2);
    }
```

5.13. ábra Kimeneti paraméterek frissítése

6 Vezérlés, grafikus kezelőfelület

Az elkészített kompresszornak rendelkeznie kell valamilyen felülettel, amiről a különböző paraméterek beállíthatók.

A vezérlés megvalósítására a controller osztályt használjuk. Itt tudjuk definiálni azokat a paramétereket, amiken keresztül a grafikus felület és a processor egymással kommunikálni tud. Itt tudunk létrehozni egyedi paramétereket, felülírni a hozzájuk tartozó függvényeket. Valamint itt tudjuk meghatározni, hogy melyik fájl alapján hozza létre a grafikus felületet.

```
//---Gain Reduction parameter---
stepCount = 0;
defaultVal = 0;
flags = ParameterInfo::kIsReadOnly;
tag = kGainReductionId;
parameters.addParameter(STR16("Gain Reduction"), nullptr, stepCount, defaultVal, flags, tag);

//---Makeup Gain parameter---
stepCount = 0;
defaultVal = 0.5;
flags = ParameterInfo::kCanAutomate;
tag = pMakeUpGainId;
parameters.addParameter(STR16("Makeup Gain"), nullptr, stepCount, defaultVal, flags, tag);

//---Measurement Type parameter---
stepCount = 1;
defaultVal = 0;
flags = ParameterInfo::kNoFlags;
tag = pMeasureTypeId;
parameters.addParameter(STR16("Measurement Type"), nullptr, stepCount, defaultVal, flags, tag);
```

6.1. ábra Néhány paraméter létrehozása

Paramétereket az addParameter függvénnyel tudunk létrehozni, ez látszik a 6.1. ábrán. Itt meg kell adni a paraméter nevét és mennyiségi egységét, amit én üresen hagytam. Ez után meg kell adni, hogy hány lépésben lehet állítani. 0 esetén bármire be lehet állítani, ha pedig bármi más számot adunk meg, akkor annyi egyenlő részre osztja szét a tartományt. Meg lehet adni egy alapértéket, bármilyen állítás nélkül ezzel az értékkel jön létre a controller és benne a paraméter.

A flagekben lehet állítani a paraméter tulajdonságait. A programban én 3 flaget használtam:

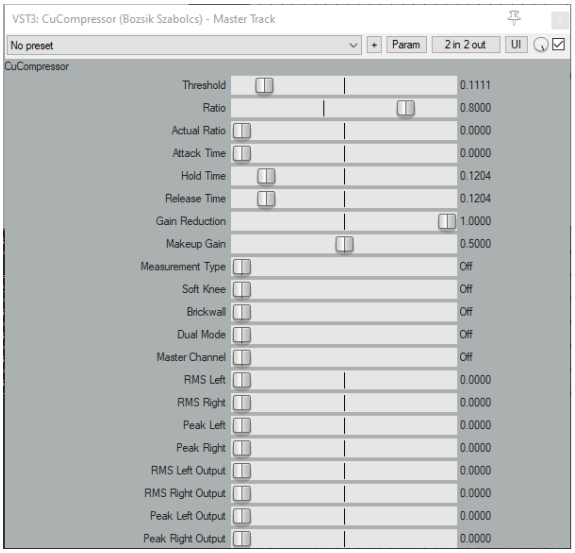
kNoFlags: Nem állítunk be semmilyen flaget.

kCanAutomate: A host programban létre lehet hozni hozzá automatizációt, ami alapján a feldolgozás során változik az érték. Ilyenkor a host programban létre lehet hozni egy speciális automation track-et, ahol meg lehet adni, hogy a paraméter melyik időpillanatban, hogyan változzon, milyen értéket vegyen fel.

kIsReadOnly: A paraméternek nem lehet állítani az értékét a felületről, csak kiolvasásra szolgál. Ilyen típusúak a kivezérlésjelzők.

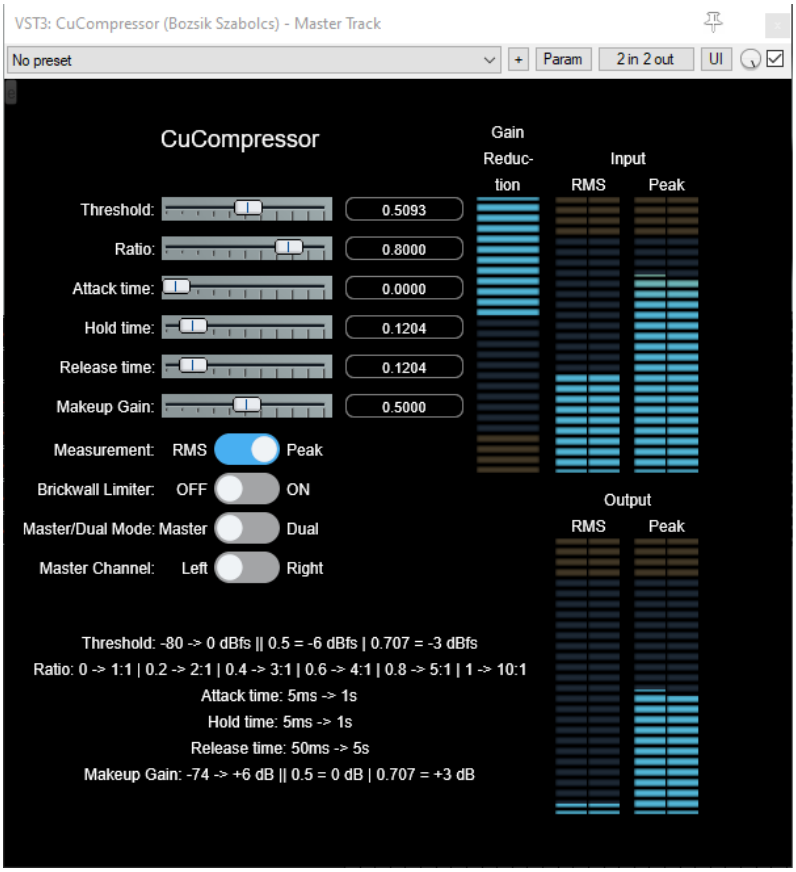
Végül pedig meg kell adni a paraméterhez tartozó ID-t, ami alapján mind a processzorban, mind a controllerben be lehet azonosítani, hogy melyik paraméterről van szó. Az ID-k egyszerű kezelhetősége érdekében létrehoztam egy paramids header fájlt, amiben egy enumban elhelyeztem a paraméterek neveit és a hozzájuk tartozó ID-kat. Ezek után már lehet a nevek alapján hivatkozni rájuk.

Vannak olyan szoftverek, amelyek ezek után a paramétereket egyszerű formában önmaguktól is ki tudják rajzolni egy grafikus felületre (ez látható a 6.2. ábrán), de nem mindegyik.



6.2. ábra Reaper által generált GUI

A GUI létrehozásához az SDK beépített szerkesztővel is rendelkezik, de szövegesen is el lehet készíteni a felületet leíró XML fájlt. Ebben kell megadni a háttér méretétől és színétől kezdve minden egyes szövegdoboz, kezelőfelület méretét, pozícióját, típusát. A host program a controller createView függvényét hívja meg, amikor szüksége van a grafikus felületre, ekkor a VSTGUI feldolgozza az XML fájlt és létrehozza a grafikus felületet. Ez az általam elkészített felület látható a 6.3. ábrán.



6.3. ábra Elkészített kezelőfelület

Bal oldalt láthatók a különböző paraméterekhez tartozó kezelőegységek egy-egy egyszerű tolópotméter formájában. A két állásban állítható paraméterekhez egy kapcsolót hoztam létre, amivel a két állás között lehet váltani. A paraméterek alatt a felesleges üres részt a paraméterek szélsőértékeivel és egy-egy fontosabb érték megadásával töltöttem ki. Jobb oldalra kerültek a különböző kivezérlésjelzők. A Gain Reduction jelzi az éppen alkalmazott jelcsökkentés mértékét. Emellett a bemenet és a kimenet is kapott egy-egy sztereó RMS és Peak értéket mutató kivezérlés jelzőt.

Minden grafikus elemhez szükséges legalább 1 bitmap, amit a program be tud tölteni. A tolópotméterek „CSlider” típusúak, itt külön meg kell adni a háttér valamint a fogantyú képét. A kapcsolók „CHorizontalSwitch” típusúak, itt egy képet kell megadni, és a lehetséges állások számától függően és 1 darab alkép magasságától függően mindig a kép megfelelő darabját rajzolja ki. A kivezérlésjelzők „CVuMeter” típusúak. Itt meg kell adni a teljesen bekapcsolt és a teljesen kikapcsolt állapot képét, valamint hogy hány darabban váltson a kettő között. A program ez alapján egymásra rajzolja a 2 képet. A tolópotméter és a kivezérlés jelző képeihez az SDK-hoz mellékelt példaprogramokban található bitmapeket használtam fel. A kapcsolóhoz egy egyszerű képet alakítottam át a

programnak megfelelő formára. A Gain Reduction képe a kivezérlés jelzőé 180°-kal elforgatva.

6.4. ábra A felhasznált bitmapek

A kommunikációt a processor, a controller és a GUI között a host program biztosítja egy event bus-on keresztül. Minden változtatás egy adatpontot hoz létre ebben a listában az adott paraméter ID-jával.

A processorban minden egyes adatcsomag feldolgozása előtt beolvasom ezt a listát és frissítem a paraméterek értékeit, feldolgozás után frissítem a csak olvasható értékeket (kivezérlésjelzők), ezek utána szinkronizálódnak a felülettel.

Az általam használt Reaper program átlagosan 512 minta méretű csomagokkal dolgozik, ez 48 kHz-es mintavételi frekvencia mellett körülbelül 90 frissítést jelent másodpercenként, ami több mint az átlagos monitor 60 vagy 75 Hz-es frissítési rátája, így folyamatosnak tűnik a megjelenítés. Sokkal több mintát tartalmazó vagy alacsonyabb mintavételi frekvencia esetén azonban akár már nagyobb eltérések is lehetnek egy-egy csomagon belül. Ilyenkor a paraméterekhez tartozó sample offset értéket is érdemes lehet feldolgozni, ám én ezt nem használtam a plugin megvalósítása során.

7 Tesztelés, eredmények kiértékelése

A plugin tesztelése 3 külön részre bontható, ezek a bemeneti RMS és Peak mérése, a paraméterek által megadott időzítések és a kimeneti kompresszált jel tesztelése. A mérések elvégzéséhez 4 mérőjelet hoztam létre az Adobe Audition program segítségével, mindegyiket 44.1 kHz-es mintavételezési frekvenciával, 16 bit mélységgel.

A tesztelés eredményét is ebben a programban értékeltem ki, a beépített Amplitude Statistics funkciójával. A mért adatok közül a Peak Amplitude és Total RMS értékeket használtam fel. Az RMS méréséhez tartozik néhány beállítás, a 0 dB értéket a full scale négyszögjel RMS értékéhez választottam (másik lehetőség a full scale szinusz), az ablakszélességet pedig 125 ms-ra állítottam, hasonlóan, mint az elkészített pluginben. A 4 mérőjel amit létrehoztam az 1 kHz-es szinusz, négyszög és háromszög jel, valamint egy fehérzaj. A mérőjeleket a létrehozás után visszaellenőriztem, és úgy állítottam be a jelszintjüket, hogy mindegyik 0 dB peak értékkel rendelkezzen.

A mérőjeleket, a plugin által létrehozott kimeneti jeleket elektronikus mellékletként csatolom a szakdolgozatomhoz, az elkészített pluginnel együtt.

7.1 RMS és Peak mérés

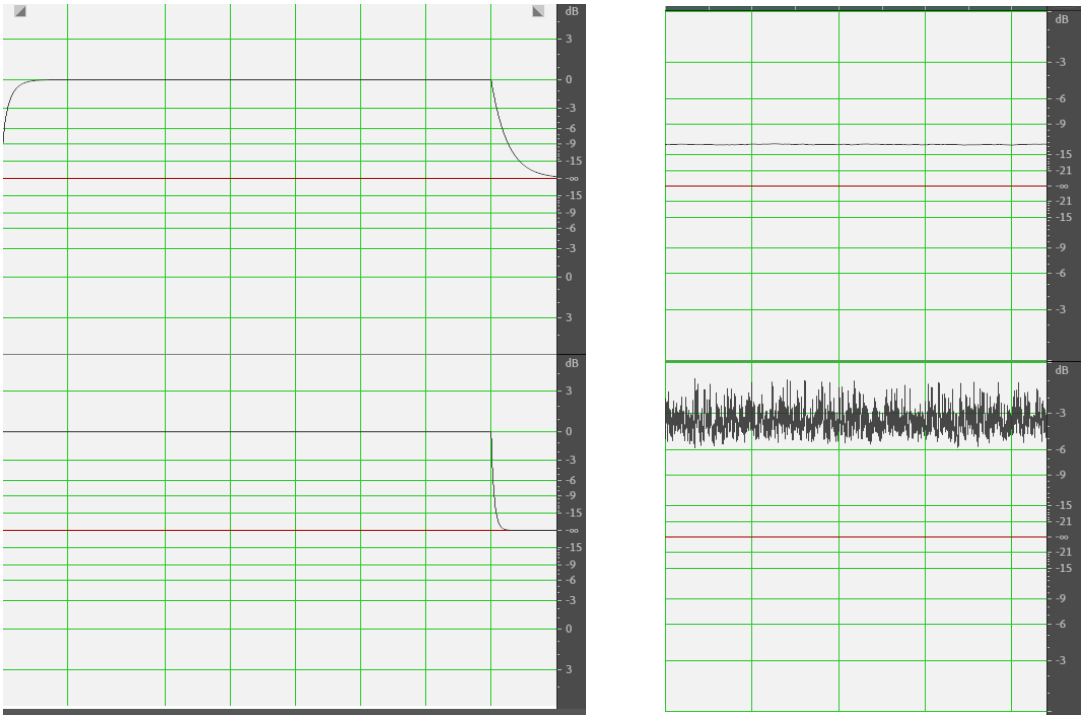
A plugint úgy módosítottam, hogy a kimeneti hangsáv bal oldalán a mért RMS érték, a jobb oldalán pedig a mért Peak érték legyen. Az így kapott fájlokat újra kiértékelve a bal oldali csatorna peak értéke az eredeti jel RMS-e, a jobb oldali csatorna peak értéke pedig a peak értéke. A kapott értékek összehasonlítását mutatja a 7.1. táblázat.

	Szinusz	Szinusz plugin	Négyszög	Ngyszög plugin	Háromszög	Háromszög plugin	Fehérzaj	Fehérzaj plugin
Peak Amplitude:	0,00 dB	0,00 dB	0,00 dB	0,00 dB	0,00 dB	0,00 dB	0,00 dB	0,00 dB
RMS Amplitude:	-3,01 dB	-3,01 dB	0,00 dB	0,00 dB	-4,75 dB	-4,75 dB	-12,48 dB	-12,25 dB

7.1. táblázat RMS és csúcsértékmérés eredményei

Jól látható, hogy a periodikus jeleknél a plugin pontosan kiszámolja az értékeket. Eltérés egyedül a fehérzaj RMS értékénél látszik. Ez egy véletlenszerű jel, így az RMS számítás implementációjától függően kicsit eltérhetnek az értékek, valamint a plugin által mért RMS értékekből a peak érték használatával a legnagyobbat választottam ki, nem pedig az egésznek az átlagát. A 7.1. ábra mutatja a négyszögjel plugin által mért RMS (felül) és peak (alul) értékét. Jól látható, hogy bekapcsoláskor a peak azonnal felugrik,

míg az RMS exponenciálisan nő a kívánt érték eléréséig, majd kikapcsolás után mind a két érték exponenciálisan csökken nulláig, közte pedig közel állandó. Az ábrán szintén látható a fehérzaj értékeinek egy kiragadott részlete. Megfigyelhető rajta, hogy az RMS értéken csak egy minimális változás látható, a peak értéke pedig egészen tüskés.

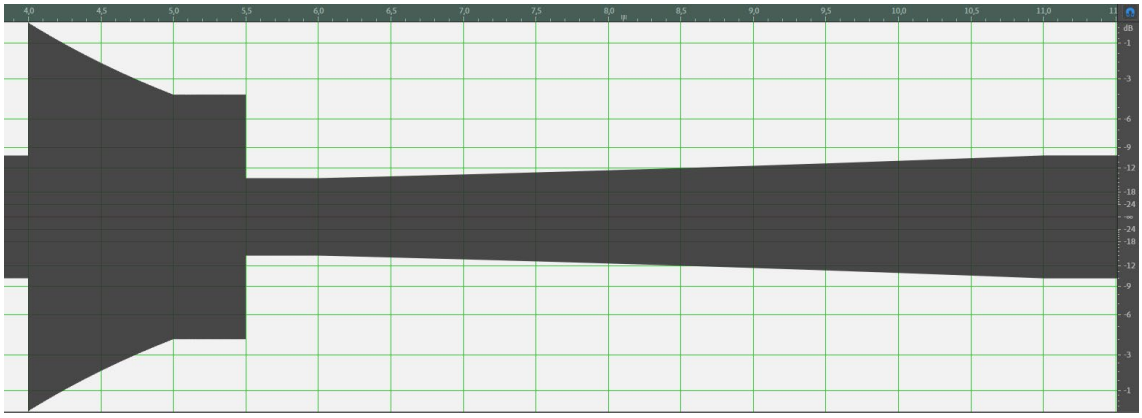


7.1. ábra Négyszögjel (balra) és fehérzaj (jobbra) RMS (felül) és peak (alul) értékei

7.2 Időzítések

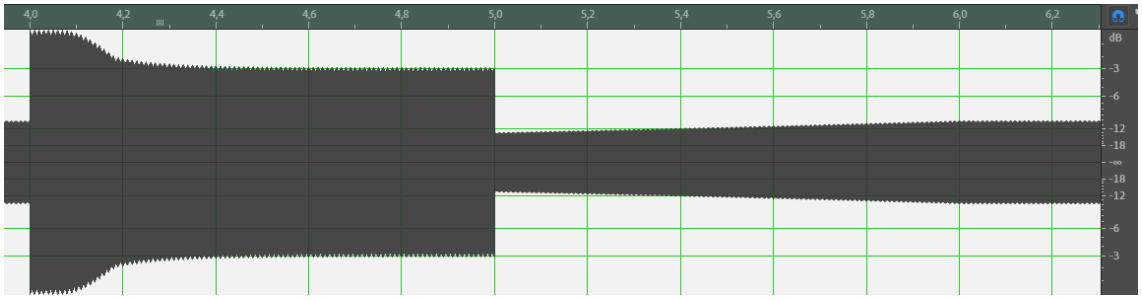
Az időzítések teszteléséhez készítettem egy automatizációt a mérőjel hangerejére, ami 4 másodpercig -10 dB, 1,5 másodpercig 0 dB, majd megint -10 dB értékű. A plugin paramétereit a következőképpen állítottam be: Threshold: -6 dB, Ratio: 3:1, Attack és Hold 1 másodperc, release 5 másodperc, makeup gain 0 dB, peak mérés.

A 7.2. ábra mutatja a szinusz jelből kapott kimenetet a fenti beállítások mellett. Jól látható rajta az elvárt működés, miszerint 4 másodpercnél felugrik a jel, 1 másodperc alatt lecsökken a kompresszált kimenetre. A bemeneti jel lecsökkenésével a kimenet is lecsökken, viszont a hold időzítő miatt, még 0,5 másodpercig megmarad a csökkentett jelszint, majd 5 másodperc alatt fokozatosan eléri a kimeneti jelszint a bemenetit.



7.2. ábra Időzítés megvalósulása szinusz jelen

Egy másik paraméter beállítás a következő: Threshold: -3 dB, Ratio: 10:1, Attack 50 milliszekundum, Hold 5 milliszekundum, release 1 másodperc, makeup gain 0 dB, RMS mérés. Az automatizáció pedig 4 másodpercig -10 dB, 1 másodpercig 0 dB, majd megint -10 dB értékű. A 7.3. ábra mutatja ezen beállítások mellett a háromszögjelre adott kimenetét. Látható a képen, hogy az RMS átlagoló hatása révén a jel megugrását követően nem kezdődik el azonnal a kompresszálás, kell egy kis idő mire eléri az RMS a threshold értékét, és utána minden egyes egyre nagyobb RMS érték hatására újraindul az attack időzítés, egyre meredekebben, így egy közel exponenciális letörés jelenik meg a jelen. Ez a release fázist nem kimondottan befolyásolja.



7.3. ábra Időzítés megvalósulása háromszög jelen

7.3 Kimeneti jel tesztelése

A plugin paramétereit a következő féleképpen állítottam be: Threshold: -6 dB, Ratio: 3:1, Attack és Hold 5 ms, release 1 másodperc, makeup gain 0 dB. A szinuszjelet létrehoztam mind RMS, mind peak méréssel is, majd az Audition programmal meghatároztam a kimeneti jel RMS és peak értékeit a bekapcsolási tranziens után, állandósult állapotban. Ezek az értékek a 7.2. táblázatban láthatóak.

	Szinusz	Szinusz RMS	Szinusz peak
Peak Amplitude:	0,00 dB	-2,04 dB	-4,03 dB
RMS Amplitude:	-3,01 dB	-5,05 dB	-7,05 dB

7.2. táblázat Kimeneti jel értékei

Matematikailag kiszámítva is ezeket az értékeket kell kapnunk, RMS mérés esetén 3 dB-lel lépi meg a jel a thresholdot, azaz 1 dB-el kell hangosabbnak lennie a kimenetnek, mint a -6 dB, ami -5 dB RMS. Peak mérés esetén 6 dB-lel lépi meg a jel a thresholdot és így 2 dB-lel kell hangosabbnak lennie a kimenetnek -6 dB-nél, ami -4 dB peak.

A valós felhasználást egy zeneszám használatával prezentálnám, a fentivel megegyező beállításokkal, peak méréssel. A kompresszálás után beállítottam +3 dB makeup gaint is. Az így kapott értékek láthatóak a 7.3. táblázatban.

	Eredeti		Kompresszált		+3 dB makeup gain	
	Left	Right	Left	Right	Left	Right
Peak Amplitude:	0,00 dB	-0,02 dB	-0,49 dB	-0,38 dB	0,00 dB	0,00 dB
Total RMS Amplitude:	-12,40 dB	-12,57 dB	-13,91 dB	-14,08 dB	-10,90 dB	-11,07 dB
Maximum RMS Amplitude:	-5,44 dB	-5,53 dB	-6,66 dB	-6,75 dB	-3,65 dB	-3,74 dB
Dynamic Range:	73,61 dB	72,93 dB	69,78 dB	69,00 dB	65,23 dB	65,27 dB

7.3. táblázat Zeneszám kompresszálása

Jól látható a kompresszálás hatására csökkent peak, RMS és dinamikatartomány értékek, majd a makeup gain hatására a peak és RMS nőtt, viszont a dinamikatartomány tovább csökkent, feltehetően voltak benne 0 dB-nél nagyobb jelek, amik le lettek vágva.

8 Fejlesztési lehetőségek

Az elkészített plugin használható, működik, de mindig azonosíthatók továbbfejlesztési lehetőségek, esetünkben leginkább két irányban, funkcionalitásban és kinézetben.

8.1 Kezelőfelület

Az elkészített GUI ugyan használható, de közel sem annyira szép, mint egy-egy cég által gyártott plugin felülete. Teljesen saját bitmapek gyártásával sokat lehetne javítani rajta. Emellett jelenleg a 0 és 1 közé normalizált értékeket jeleníti meg, létre lehetne hozni teljesen egyedi paramétereket, amelyeket rögtön dB-ben vagy milliszekundumban lehetne megadni. A ratio beállításához létre lehetne hozni egy kapcsolót, amivel a tényleges értékek között lehet váltani. Erre látható kettő példa a 8.1. ábrán.

8.1. ábra Waves plugin GUI

A paramétereket időnként egyszerű elképzelni, hogy hogyan is néznek ki, hogyan befolyásolják a működést. A threshold és a ratio esetében ez nem feltétlen van így, emiatt érdemes lehet egy grafikonon ábrázolni, hogy a jelenlegi beállítások mellett hogy viszonyul egymáshoz a bemenet és a kimenet jelszintje, és éppen mekkora a bemeneti és

kimeneti jelszint. Ennek az ábrázolásnak egy lehetséges megvalósítása látszik a 8.2. ábrán.

8.2. ábra Waves plugin threshold és ratio ábrázolása

8.2 Paraméterek

Az elkészített plugin sok alapvető funkcióval rendelkezik, azonban vannak olyanok, amelyek hasznosak tudnak lenni és nincsenek jelenleg implementálva.

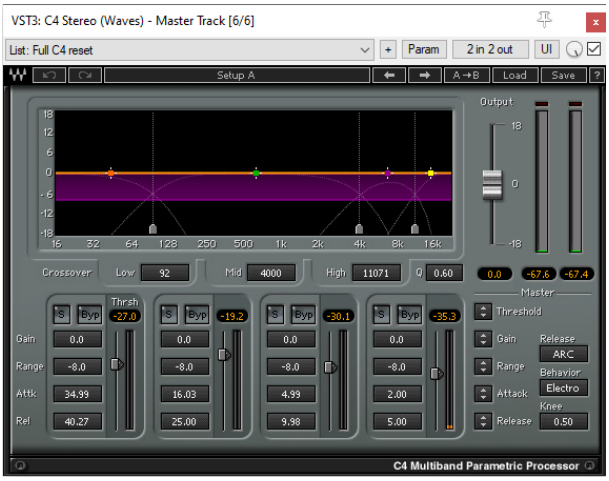
Az egyik ilyen lehetőség a soft knee beépítése, hogy a ratio a töréspontos megvalósítás helyett folyamatos átmenetet képezzen.

Egy másik egyszerű fejlesztési lehetőség az RMS számítás sebességének a változtatása, akár csak 2 állás között is. (Fast 125ms, slow 1s)

8.3 Funkciók

A jelenleg megvalósított kompresszor jó kiindulási pontot ad más dinamikaszabályzók megvalósításához.

Ilyen lehet egy de-esser megvalósítása, aminél a vezérlő jelet a bemenet egy sávszűrt részéből állítjuk elő, ami kimondottan az emberi beszéd „S” és „SZ” hangjaira érzékeny. Több szűrő beiktatásával a vezérlőjelhez és akár a kimeneti erősítő(k) bemenetéhez is egy multiband azaz többsávos kompresszort tudunk létrehozni. Egy ilyen többsávos kompresszor kezelőfelülete látható a 8.3. ábrán. Ezeknek a megvalósításához egy lehetséges megoldás a DSPfilters csomag, ami többfajta szűrő létrehozásához is használható. A megvalósítás egy lehetséges módját mutatja be Kostyál Domonkos Többsávos VST kompresszoreffekt megvalósítása című szakdolgozata [9].



8.3. ábra Waves többsávos kompresszor

Egy külső bemenet hozzáadásával és a vezérlőjel ebből történő előállításával létre lehet hozni egy duckert, aminek tipikus alkalmazása a háttérzene halkítása, ha a mikrofonba beszélnek az emberek.

A program paraméterfüggőségének megváltoztatásával pedig létre lehet hozni egy zajzárt vagy expandert is külön pluginként, beépített funkcióként, vagy akár egymás mellett a kettőt, amikor is először egy zajzáron, aztán pedig a kompresszoron megy keresztül a jel egy pluginen belül.

Köszönetnyilvánítás

Szeretnék köszönetet mondani konzulensemnek, Dr. Rucz Péternek, hogy végigsegített a szakdolgozat megírásának rögös útján, és mindig válaszolt a kérdéseimre.

Köszönettel tartozom Kiss Dávidnak, aki az össze helyesírási hibámra felhívta a figyelmemet.

És végül, de nem utolsó sorban köszönöm Bélteky Zsuzsannának, hogy mindig tartotta bennem a lelket, és mindig szívesen volt a gumikacsám, ha nem jöttem rá egy hiba forrására.

Irodalomjegyzék

[1] musictechstudent.co.uk: *History and Development of Compression* https://musictechstudent.co.uk/music_technology_/history-and-development-of-compression/#:~:text=Timeline%20of%20Compression (hozzáfésér dátuma: 2024 május 24.)

[2] akirumusic.com: *History and Types of Compressors,* https://akirumusic.com/history-and-types-of-compressors/#:~:text=and%20soft%20sounds.-,The%20First%20Compressor,the%20precursor%20to%20the%20V72. (hozzáfésér dátuma: 2024 május 24.)

[3] Western Electric: *Program amplifier 110A*, https://www.worldradiohistory.com/Archive-Catalogs/Western-Electric/Western-Electric-110A-Program-Amp-1937.pdf (hozzáfésér dátuma: 2024 május 24.)

[4] EBU: *THE EBU STANDARD PEAK-PROGRAMME METER FOR THE CONTROL OF INTERNATIONAL TRANSMISSIONS,* https://tech.ebu.ch/docs/tech/tech3205.pdf (hozzáfésér dátuma: 2024 május 24.)

[5] EBU: *R 128 LOUDNESS NORMALISATION AND PERMITTED MAXIMUM LEVEL OF AUDIO SIGNALS,* https://tech.ebu.ch/docs/r/r128.pdf (hozzáfésér dátuma: 2024 május 24.)

[6] Steinberg: *VST3SDK* https://github.com/steinbergmedia/vst3sdk (hozzáférés dátuma: 2024. május 24.)

[7] Steinberg: *VST 3 Project Generator* https://github.com/steinbergmedia/vst3projectgenerator (hozzáférés dátuma: 2024. május 25.)

[8] Fiala P., Rucz P.: *Stúdiótechnika Laboratórium, Hangjelek digitális feldolgozása az akusztikai gyakorlatban mérési segédlet*

[9] Kostyál D.: *TÖBBSÁVOS DINAMIKASZABÁLYOZÓ EFFEKT MEGVALÓSÍTÁSA VST KÖRNYEZETBEN* (BME-VIK szakdolgozat, 2021.) https://diplomaterv.vik.bme.hu/hu/Theses/Tobbsavos-VST-kompresszoreffekt-megvalositasa (hozzáférés dátuma: 2024. május 29.)