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

Fónai Martin

ADAPTÍV JELFELDOLGOZÓ ALGORITMUSOK IMPLEMENTÁCIÓJA

ARM Cortex magú mikrokontrolleren

KONZULENS

Dr. Orosz György

BUDAPEST, 2019

HALLGATÓI NYILATKOZAT

Alulírott **Fónai Martin**, 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, 2019. 12. 15.

...…….

Fónai Martin

Tartalom

Kivonat

Szakdolgozatomban az LMS algoritmus különböző megvalósításainak működését vetettem össze. Ez az algoritmus a digitális jelfeldolgozás jelentős vívmánya, mivel alkalmas rendszerek paraméterbecslésére (identifikáció), ezáltal pedig zajelnyomásra is. Számtalan változata létezik, melyek mindegyike más és más előnyökkel bír.

Népszerűsége és változatossága miatt rengeteg implementációban elkészült, és jelen szakdolgozat keretei közt én is megvalósítottam ARM Cortex-M3 architektúrán. Az összehasonlítás alapját képző függvénycsaládot az ARM készítette.

Az elkészítés során részletesen megismertem egy mikrokontroller sorozatot, mely erre az architektúrára alapszik, majd ezt használva felépítettem egy szoftveres keretrendszert, melyben később meg tudtam valósítani az LMS-t, illetve össze tudtam vetni az ARM optimálisnak mondható megoldásaival. Az elkészítés során ügyeltem arra, hogy minél kevesebb időt várakozzon a processzor, és a számításokat a lehető legegyszerűbb módon végezze el.

A végső cél a két implementáció teljesítményének összevetése volt, melynek során kiderült, hogy a saját készítésű algoritmus futási ideje hozzávetőlegesen összemérhető a gyártóéval.

Abstract

This thesis is an operational comparison of different implementations of the LMS algorithm. This algorithm is a significant achievement in the era of digital signal processing as it is suitable for parameter estimation (identification) of systems and thus for noise canceling. Countless variants are available, each with different benefits.

Due to its popularity and versatility, it has a vast number of implementations, and I have implemented it on ARM Cortex-M3 architecture within this thesis. The function family that is the basis of the comparison was created by ARM.

During the development phase I got acquainted with a microcontroller series based on this architecture, and using this knowledge I have built a software framework, in which I could later implement the LMS and compare it with the optimal solutions of ARM. I have made sure that the processor would be waiting as occasionally as possible and have made the calculations the simplest I possibly could.

The final goal was to compare the performance of the two implementations, which revealed that the runtime of the my algorithm was approximately the same as that of the ARM.

1 Bevezetés

A multimédiás eszközök terjedésével a mindennapi életben is egyre gyakrabban találkozhatunk olyan eszközökkel és alkalmazásokkal, amelyek digitális jelfeldolgozást végeznek. Ezek többségében általános célú számítási egység található, mivel a céleszközök fejlesztése nagyon drága lenne. A nem jelfeldolgozásra tervezett eszközök közül azonban egyre több rendelkezik olyan sajátosságokkal, amelyek mégis lehetővé teszik az ilyen jellegű feladatok elvégzését, illetve az ehhez szükséges algoritmusok hatékony végrehajtását.

Az ehhez hasonló kis teljesítményű DSP (Digitális jelfeldolgozó) eszközök egyik leglátványosabb példái azok a headsetek, amelyek a külső akusztikus zajt csökkentik, így képesek a teljes csend érzetét kelteni akár egy forgalmas utca közepén is. Ahogyan azt látni fogjuk, ennek a jelfeldolgozó algoritmusa nem ütközik a számítási kapacitás korlátaiba, ezért jelen szakdolgozat keretein belül az algoritmus implementálásával próbálkoztam ARM Cortex-M3 architektúrán.

A feladat több olyan megvalósítás összehasonlítása volt, amely megvalósítja a zajcsökkentés alapjait képző identifikációs módszert, az LMS és FxLMS algoritmusokat. Az összehasonlítás alapvető szempontja a teljesítmény volt. Ennek során villamos jelenségek, valamint a digitális jelfeldolgozás ismeretére építek, miközben a rendelkezésre álló mikrokontroller boardot is szakszerűen kell kezelnem. Az értékelés során az alapvető laboreszközök használatában szerzett ismereteimet alkalmazom.

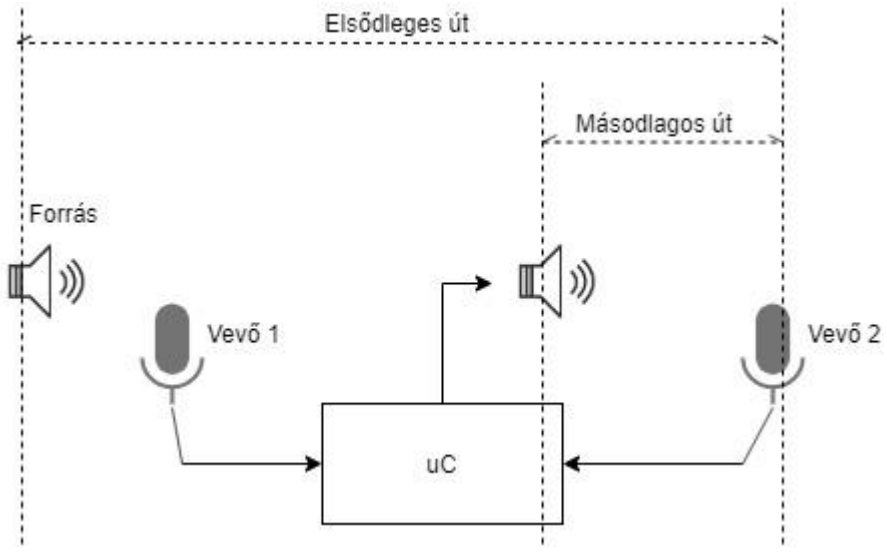
A továbbiakban először szeretném az algoritmus elméletét tárgyalni, majd ismertetem a felhasznált eszközt. Ezek után az algoritmus megvalósításának részleteit fogom bemutatni. Mivel a feladatom része volt egy másik implementációval történő összehasonlítás, ezért az ARM saját könyvtárában rejlő algoritmusokat is szeretném részletezni. Mindenképp összehasonlítom ezek működését, majd bemutatom az egyes algoritmusok működését a mérési eredményeken keresztül, illetve megadok néhány fejlesztési lehetőséget, amelyek a szakdolgozat készítése során merültek fel. A dolgozat végén az általam felhasznált szakirodalom található.

2 A digitális jelfeldolgozás és LMS

Manapság a zajcsökkentő eljárások tömkelege elérhető az egyszerű sávszűréstől kezdve a mesterséges intelligencia alapú megoldásokig[1]. Ezek jellemzően mind digitálisak, tehát mind az észlelés, mind a beavatkozás diszkrét időben történik. Nincs ez máshogy az LMS (Least Mean Square) nevű identifikációs algoritmuson alapuló zajcsökkentéssel sem. Az értelmezés megkönnyítése érdekében ebben a fejezetben összeszedtem a témakör fontosabb alapfogalmait.

2.1 Alapfogalmak[2]

Ahhoz, hogy egy külső (zaj)forrás jelét csökkenteni tudjuk, ismernünk kell a forrásból érkező jelet. Célszerű továbbá ismernünk a hibajelet is, mivel tudnunk kell, hogy sikerült-e kioltanunk a forrás jelét, vagy szükséges valamilyen változtatást eszközölnünk. Egy szóval: szabályoznunk kell, amihez szükséges visszacsatolás. Végezetül képesnek kell lennünk a legjobb belátásunk szerint beavatkozni, így kell egy általunk vezérelt forrás is. Ezek alapján a mérési összeállítás[1] az alábbi ábrán látható:



2-1. ábra: Az eszköz elvi felépítése akusztikus zajcsökkentés esetén

[1] A zajcsökkentés nem feltétlenül akusztikus zajokra értendő, ám ez a legszemléletesebb példa, ezért az ábrán is akusztikus forrásra és vevőre utaló szimbólumokat használtam.

A továbbiakban a külső forrás és a visszacsatolásra használt 2-es számú vevő közti átvitelt **elsődleges útnak**, a beavatkozó forrás és a 2-es számú vevő közti átvitelt **másodlagos útnak** nevezem.

2.2 A folytonos idejű és mintavételezett világ

Az előzőekben láthattuk, hogy az érzékelés és a beavatkozás a folytonos világban történik, ám a mikrokontroller működéséből adódóan mintavételes, ráadásul az értékkészlete is egy diszkrét tartományon mozog. Emiatt felmerül a kérdés, hogy pontosan mi is történik, amikor egy folytonos jelet mintavételezünk, vagy egy diszkrétidejű jelet folytonossá teszünk.

2.2.1 A Shannon-Nyquist-féle mintavételi tétel

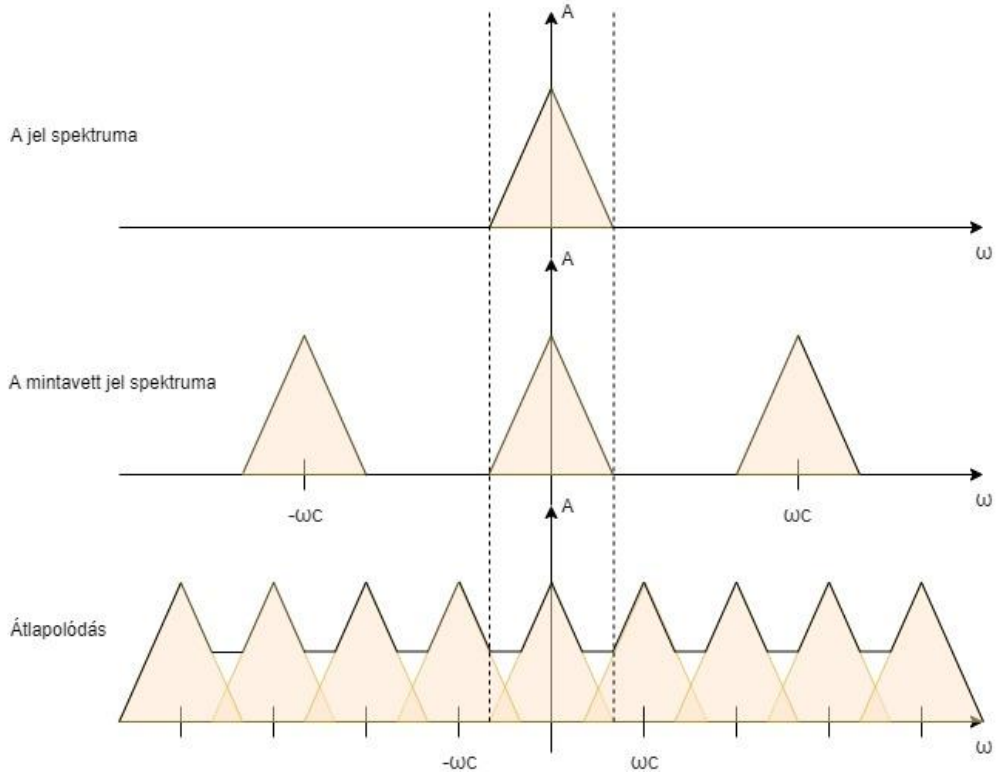
A tétel azon elméleti lehetőséget vizsgálja, amikor a jelet egy úgynevezett Dirac-fésű segítségével mintavételezzük. A Dirac-fésű egy Dirac-impulzus alapú periodikus gerjesztés, mely az alábbi formában írható fel:

$$\Delta_T(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT)$$

Egy tetszőleges folytonos időbeli jelet megszorozva a Dirac-fésűvel, a jel T időnkénti mintái állnak elő impulzusként. A Dirac-fésűt Fourier-sorba fejtve, majd az eredményt transzformálva spektrumként szintén egy Dirac-fésűt kapunk

$$F\{\Delta_T(t)\} = \sum_{n=-\infty}^{\infty} \delta(\omega - n\frac{2\pi}{T}) = \sum_{n=-\infty}^{\infty} \delta(\omega - \omega_s)$$

Ebben megjelent az úgynevezett mintavételi körfrekvencia. Mivel az eredeti jelünket szoroztuk a Dirac-fésűvel, ezért a spektrumát konvolválnunk kell a most kapott spektrummal, melynek eredményeképp úgy fogjuk megkapni a mintavett jel frekvenciatartománybeli képét, hogy az eredetit a mintavételi körfrekvencia összes számszorosára rámásoljuk, majd összeadjuk. Amennyiben az eredeti jel sávkorlátos, és ez a sávkorlát nem nagyobb, mint a mintavételi körfrekvencia fele, akkor az összeadás során a sávszélek nem lapolódnak át, és a jel változatlanul visszanyerhető. A visszanyeréshez egy olyan szűrőre van szükség, melynek a törésponti frekvenciája a mintavételi frekvencia fele.

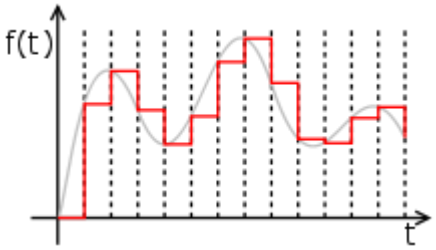


2-2. ábra: Spektrum mintavételezés előtt és után, átlapolódás

2.2.2 Visszaállítás, tartók

Az előzőek tanulsága szerint a visszaállításhoz először a mintákból impulzust kell készíteni, majd ezt egy aluláteresztő szűrővel szűrni. A gyakorlatban viszont lehetetlen impulzust generálni, de még a megközelítése is nehézséget jelent, hiszen vagy a szélessége nő meg túlságosan, vagy a szűrőáramkör számára okoz elviselhetetlenül nagy terhelést. Emiatt a gyakorlatban úgynevezett tartóáramköröket szoktak alkalmazni. Ezek közül is a leggyakrabban (valamint a később alkalmazott eszközben) a nulladrendű tartót alkalmazzák.

Amennyiben a diszkrét jelsorozatra folytonos időben Dirac-fésűként tekintünk, a nulladrendű tartó impulzusválasza a 0 időpillanattól egy mintavételi perióduson át tartó négyszögablak. Egy ilyen tartó a diszkrét jelsorozatból egy lépcsőzetes jelet készít, az alábbi ábrához hasonlóan:



2-3. ábra: Nulladrendű tartó[7]

A nulladrendű tartó amplitúdókarakterisztikája a sinc függvény megfelelően skálázott változata:

$$H_{ZOH}(\omega) = \frac{1}{T}\frac{1-e^{-j\omega T}}{j\omega} = e^{-\frac{j\omega T}{2}}\frac{e^{\frac{j\omega T}{2}}-e^{-\frac{j\omega T}{2}}}{j\omega T} = e^{-\frac{j\omega T}{2}}\frac{e^{j\frac{\omega T}{2}}-e^{-j\frac{\omega T}{2}}}{2j\frac{\omega T}{2}}$$

$$= e^{-\frac{j\omega T}{2}} sinc\left(\frac{\omega T}{2}\right)$$

A sinc jellegű spektrum önmagában egy bizonyos aluláteresztő szűrőként fogható fel, így az eredeti jelet jó közelítéssel visszaállítja.

2.3 Diszkrétidejű szűrők

2.3.1 Véges impulzusválaszú (FIR) szűrők

A FIR szűrők olyan diszkrétidejű rendszerek, amelyeknek minden pólusa zérus, tehát az alábbi alakban írhatók fel:

$$H_{FIR}(z) = \frac{\sum_{i=0}^{n} h_i z^{n-i}}{z^n} = \sum_{i=0}^{n} h_i z^{-i}$$

A második felírásból az is egyértelműen következik, hogy a rendszer impulzusválasza a h_i sorozat, melynek az ***n-1*** index után minden eleme zérus. Emiatt a FIR szűrők válasza tetszőleges gerjesztésre egy véges konvolúcióval megadhatók, feltéve, hogy a gerjesztés előző ***n*** mintáját ismerjük.

$$y_{FIR}[k] = \sum_{i=0}^{n} h_i u[k-i]$$

A továbbiakban ***n***-et a szűrő fokszáma vagy tap-számként fogom említeni.

A FIR szűrők az alábbi előnyökkel rendelkeznek:

- Definíciójukból adódóan stabilak
- Egyszerűen számíthatók

- Nincs bennük visszacsatolás, így a kerekítési hibákat nem halmozzák
- A fázisátvitelük az együtthatók alkalmas megválasztásával lineárissá tehető

Talán a legnagyobb hátrányuknak mondható, hogy azonos átvitel eléréséhez sokkal több szűrőegyütthatóra van szükség, mint egy IIR szűrő esetében, ezért használatuk járhat megnövekedett számítási igénnyel. Bizonyos esetekben viszont sokkal egyszerűbb választásnak mondhatók, például ha hangolni kell a szűrőt, vagy precíziós alkalmazásokban kell használni.

2.3.2 IIR szűrők, exponenciális szűrők[3]

Végtelen impulzusválaszú (IIR) minden olyan szűrő, amelynek az impulzusválasza nem véges, ekvivalensen megfogalmazva tartalmaz nemnulla pólust.

A következőkben bemutatott algoritmus FIR szűrőkre épül ugyan, ám egyéb meggondolásokból adódóan szükségünk lesz felüláteresztő szűrőre, amelynek az együtthatói fordítás előtt meghatározhatók, mivel nem adaptív. Erre exponenciális szűrőt használtam. Az exponenciális felüláteresztő szűrő rendszeregyenlete:

$$y[k] = \alpha(y[k-1] + u[k] - u[k-1])$$

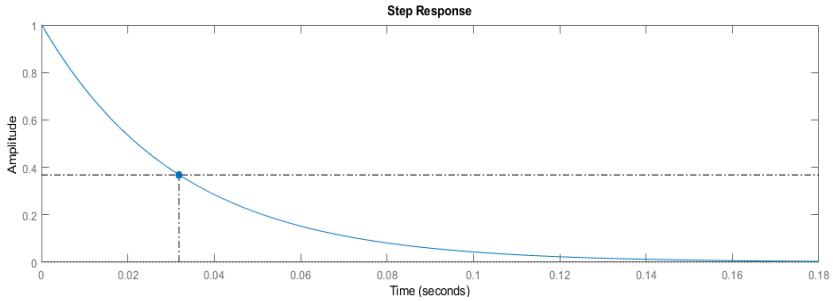
Jól látható, hogy a rendszer ugrásválasza a nulladik ütemben α. A további ütemek az alábbiak szerint adódnak:

$$u[k] - u[k-1] = 0$$
$$y[k] = \alpha y[k-1] = \alpha^k y[0] = \alpha^{k+1}$$

Az együtthatót 1-nél kisebbre választva az ugrásválasz egy folytonos idejű felülátersztő szűrőjét fogja modellezni. Így a töréspontí frekvencia egyszerűen beállítható.

$$\alpha^k := e^{-\frac{t}{\tau}} = e^{-2\pi k T f_c} = \left(e^{-2\pi \frac{f_c}{f_s}}\right)^k \rightarrow \alpha = e^{-2\pi \frac{f_c}{f_s}}$$

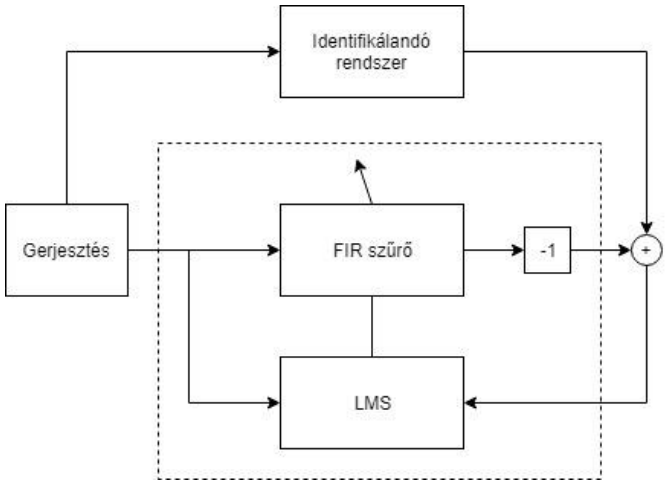
A végleges jelfeldolgozó algoritmusomban is ezt a megközelítést alkalmaztam. A megtervezett szűrő ugrásválaszát MATLAB segítségével ábrázoltam, az alábbi eredménnyel:



2-4. ábra: A szűrő ugrásválasza

2.4 Az LMS algoritmus

Az LMS szűrő egy olyan DSP megoldás, amelyet diszkrétidejű rendszerek identifikációjára fejlesztettek ki. Egy olyan FIR szűrőn alapszik, amelynek az együtthatói egy visszacsatoló ágon keresztül hangolhatók. A visszacsatoló ág az identifikálandó rendszer és az aktuális becslő kimenetéből képez hibát, és a két rendszer közös gerjesztését ismerve frissíti a becslő FIR szűrő együtthatóit. Az összeállítás blokkdiagramja az alábbi ábrán látható:



2-5. ábra: Az LMS szűrő elvi vázlata

Az együtthatók frissítésének a célja az, hogy a kimeneti hiba (innentől $e[k]$) négyzetének várható értéke a lehető legkisebb legyen. Feltételezzük, hogy ez akkor teljesül, ha a lehető legkisebb hibanégyzetre törekszünk.

A minimumkeresés egyik módszere a gradiens ereszkedés. Ez a módszer kihasználja, hogy egy skalárfüggvény gradiensvektor abba az irányba mutat, amelyben az iránymenti derivált a legnagyobb, hossza pedig pontosan ennek a hosszával egyezik meg.

$$\frac{\partial F}{\partial e} = \nabla F \cdot e;\ |e| = 1$$

Amennyiben a gradiensvektorral arányos nagyságú, ellenkező irányú lépéseket teszünk, lokális minimumba érkezünk, feltéve, hogy az arányossági tényezőt jól határoztuk meg, illetve ez a lokális minimum létezik[4].

Az LMS szűrő hangolása során pontosan ezt fogjuk felhasználni. Legyen a szűrő együtthatóinak halmaza $\underline{w}[k] = [w[0], w[1], \dots, w[N-1]][k]$, az

$$\underline{u}[k] = \begin{bmatrix} u[k] \\ \vdots \\ u[k-N-1] \end{bmatrix}$$

pedig az utolsó N mintát tartalmazza a gerjesztésből. A hiba kifejezése

$$e[k] = y[k] - \underline{w}[k]\underline{u}[k]$$

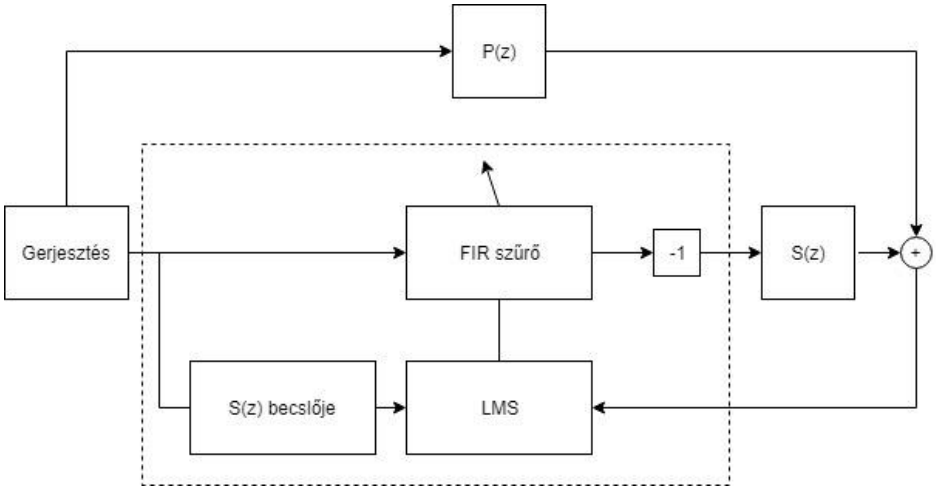
Ekkor a gradiens ereszkedés szerint

$$\underline{w}[k+1] = \underline{w}[k] - \mu \frac{\partial}{\partial \underline{w}[k]} e[k]^2 = \underline{w}[k] - \mu \frac{\partial e[k]}{\partial \underline{w}[k]} \frac{\partial e^2[k]}{\partial e[k]} = \underline{w}[k] + 2\mu e[k]\underline{u}^T[k]$$

Ezzel a hangolással az LMS szűrő a tap-szám és a μ, úgynevezett bátorsági tényező megfelelő megválasztásával képes megközelíteni az identifikálandó függvény átvitelét[5][6].

2.5 FxLMS: az LMS algoritmus kiegészítése

Ha az LMS algoritmust egy olyan feladatban használjuk, ahol a másodagos út átvitele nem egy, hanem valamilyen $S(z)$ átviteli függvény, ebben az esetben pedig a gradiens ereszkedés ennek megfelelően változik, a kifejezésében pedig a gerjesztés szűrt értéke szerepel (ahogy a neve is mutatja: „Filtered x LMS”). Ennek megvalósításához a gerjesztés bemenete és a hangoló közé el kell helyezni a másodlagos út átvitelének becslőjét. A módosított algoritmust az alábbi ábra mutatja:



2-6. ábra: Az FxLMS kiegészítés

A mi esetünkben feltételezhető, hogy elegendő a zajcsökkentés előtt megbecsülni a másodlagos út átvitelét, amihez pedig szintén használható az LMS algoritmus.

3 A felhasznált eszközök bemutatása

3.1 EFM32 mikrokontroller család[7]

A Silicon Laboratories gyártmányú EFM32 chip az ARM Cortex-M3 architektúrán alapul, mely csökkentett utasításkészletű (RISC). A processzor kiegészül 28 különféle perifériával, melyek között analóg jelek kezelésére képesek is szép számmal akadnak. Ezek közül szeretnék megemlíteni néhányat (a teljesség igénye nélkül):

- DMA vezérlő
- NVIC (Beágyazott Vektoros Megszakításkezelő)
- USB, I^2C, UART, USART (Soros kommunikáció)
- GPIO (Általános célú I/O)
- TIMER (Időzítő)
- ADC, DAC, Analóg komparátor, beépített programozható erősítők

Az utasításkészlet a Thumb-2 technológián alapszik, emiatt erősen kibővített, így olyan utasítások is megtalálhatóak benne, amelyek nagyban megkönnyítik a mikrokontroller gépi szintű programozását, illetve lehetővé teszik azokat a feladatokat, melyek egyébként túl számításigényesek lennének. Ilyen parancs például a gépi szintű elágazó utasítás, vagy a kétoperandusú szorzás és akkumulátorhoz adás (MAC).

A mikrokontroller ezen tulajdonságai teszik lehetővé, hogy alapvető DSP feladatokat is el lehessen vele látni.

3.1.1 TIMER áramkör

A mikrokontroller egy általános célú időzítő perifériát kapott, mely képes

- Egy órajelforrás, vagy a többi periféria által a PRS-en (Peripheral Reflex System) adott jelekben éleket számolni
- Adott megadott élszám után megszakítást kezdeményezni, vagy közvetlenül jelezni a többi perifériának PRS-en

- PWM kimenetet biztosítani
- Mindezt 3 csatornán, 16 biten, mindkét irányban, akár egymástól függetlenül
- Tartalmaz belső órajelosztót
- Az egyik csatorna képes a másik két csatorna kimeneti jelére időzíteni
- A csatornák a kisebb sorszámú csatorna túl- és alulcsordulása esetén tudnak lépni, ezzel 32- és 48 bites TIMER egységeket létrehozva

A TIMER-t alapvetően csak a mintavételi ciklus időzítésére kell használnunk, ennek hossza pedig nem indokol 16 bitnél többet. A PRS lehetőséget amiatt nem használom, mivel a mintavételezés elindítása nem okoz jelentős overheadet.

3.1.2 ADC[10]

Az EFM32-es mikrokontrollerek egy darab nyolccsatornás, szukcesszív approximációs alapelven működő A/D átalakítót tartalmaznak. Ez az A/D átalakító

- Tartalmaz belső órajelosztót
- 8-, 10- vagy 12 bites felbontással rendelkezik beállítástól függően
- Referenciafeszültsége több forrás közül választható ki, melyek mindegyike más előnnyel bír (zajszint és feszültségszint kérdése). Egy ilyen forrás lehet az egyik lábon érkező külső referencia
- Képes egyetlen (single), vagy egymás után több (scan) megadott csatornáról mintát venni
- Rendelkezik beépített átlapolásgátló szűrővel, amely adott beállítással megkerülhető
- Programozható offset- és bias kompenzációt tartalmaz
- Része a PRS hálózatnak
- Megszakítást ad beállítástól függően sikeres konverzió, illetve kiolvasás előtti felülíródás esetén
- Két csatorna együtt differenciális jelbemenetet is tud biztosítani

Az ADC-nek két csatornáját alkalmazom, így scan üzemmódban működtetem.

3.1.3 DAC

Az analóg kimenetet egy kétcsatornás D/A átalakító biztosítja, mely

- Rendelkezik belső órajelosztóval
- 12 bites bemeneti adatot vár
- Referenciája választható
- Állítható időközönként frissíti a kimeneti adatot
- A kimenetein megkerülhető aluláteresztő szűrő van
- A kimenet differenciális módban is tud üzemelni
- Programozható offset- és bias kompenzációt tartalmaz
- A csatornák képesek adott frekvenciájú szinuszjelet biztosítani
- A kimenetek rámultiplexálhatók a programozható erősítőre és az ADC-re is
- Része a PRS hálózatnak
- Interrupttal tudja jelezni, ha a kiírás megtörténi, illetve ha egy adatot úgy írt ki, hogy előtte írták újra az adatregisztert (DAC underflow)

A feladat mindkét csatorna használatát megköveteli.

3.1.4 GPIO periféria

A GPIO egy párhuzamos digitális interfész, például részegységek közti kommunikáció megvalósítására.

- 6 portba rendezve, portonként 16 ki- és bemenetet tud kezelni
- A pinek működhetnek
  - Tristate kimenetként
  - Open drain kimenetként
  - Fel- vagy lehúzóellenállással ellátott bemenetként
- Ezek a pinek belsőleg összeköthetők az ADC-vel
- A bemenetnek konfigurált pinek interruptot tudnak biztosítani, illetve fel tudják kelteni a processzort
- A pinek védettek a glitchektől, illetve az elektrosztatikus kisülésektől

- Pinnel lehet PRS impulzust adni

Mint azt a későbbiekben látni fogjuk, ebből mindössze két bemenetre lesz szükségünk.

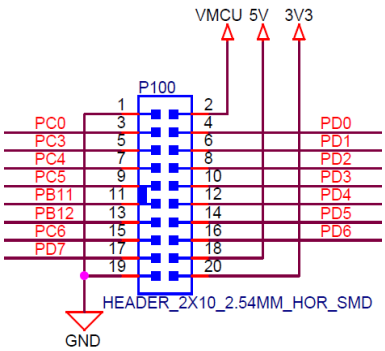
3.2 Az STK3700-as board[11]

A mikrokontrollert a gyártó STK3700-as sorozatszámú boardjára helyezve használom. Ez a board biztosítja, hogy a mikrokontroller pinjeihez gond nélkül hozzá lehessen férni, illetve lehetővé teszi a felhasználó számára, hogy beavatkozzon és betekintsen a működésbe. A board az alábbi interfészekkel rendelkezik:

- Két nyomógomb a megfelelő GPIO pineken
- Kapacitív érintő pad
- LCD kijelző, mely 7 tetszőleges karaktert, 4 darab 7 szegmenses karaktert, illetve egyéb szimbólumokat tud megjeleníteni
- Fémezett falú furatok a GPIO portoknak
- Tüskesor kivezetés az A/D és D/A eszközöknek
- Debug interfész
- Reset gomb
- Energiaforrás kiválasztását lehetővé tevő kapcsoló

Ezek közül az LCD kijelzőt, a gombokat, illetve a tüskesort használom. Az LCD kezelésére a fejlesztőkörnyezet beépített függvényeket tartalmaz, a gombok pedig megadott GPIO vonalakon találhatóak.

A tüskesorral kapcsolatban fontos megemlíteni, hogy a board sematikus ábráján lehet visszakövetni, hogy melyik vonal melyik mikrokontroller pinnel van összeköttetésben. Az ábra vonatkozó részlete itt látható:



3-1. ábra: A tüskesor kiosztása[11]

4 A keretrendszer felépítése

A szoftveres keretrendszer feladata, hogy biztosítsa a felhasználóval történő kommunikációt, a digitális és analóg világ közti átmenetet, illetve a diszkrétidejű jelfeldolgozó algoritmus ütemezését. Ezek a feladatok mind hardvereseményekre alapulnak, melyek megszakításokat generálnak. A feladatunk ezek egységes kezelése.

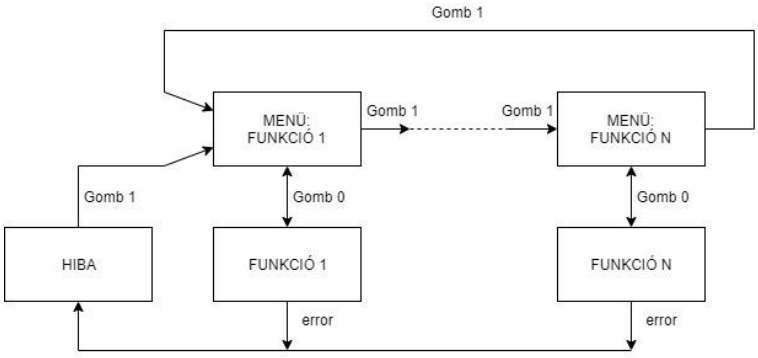
4.1 Felhasználói inputok, állapotgép

Az STK3700-as board viszonylag limitált lehetőséget ad a felhasználói bemenetek kezelésére. Szükség esetén ezt egy egyszerű bővítőhardverrel széleskörűbbé lehetne tenni, ám ehhez a feladathoz a rendelkezésre álló két nyomógomb is elegendőnek bizonyul.

Ahhoz, hogy az összes funkciót ugyanazon programban meg tudjuk valósítani, szükséges egy, a funkció kiválasztására, elindítására és leállítására képes, valamint a hibák kezelésére alkalmas állapotgép. Ennek működését a kevés felhasználói input eléggé megszabja, az alábbi alapelvek szabják meg a struktúráját:

- Minden funkcióhoz tartozzon egy menü állapot, amelyben a funkció neve jelenik meg a kijelzőn! A menüt lehessen körkörösen léptetni az egyik gombbal!
- A másik gomb lenyomása tudja a menüből elindítani és leállítani a funkciót!
- Amennyiben a programban bárhol hiba keletkezik, az léptessen egy olyan állapotba, ahol egyik funkció sem fut, a kijelzőn pedig megjelennek a felmerült hibák.

Az állapotgép egyszerűsített változata az alábbi ábrán látható:



4-1. ábra: A program állapotgépének egyszerűsített vázlata

A funkciók jelen esetben:

- A másodlagos út identifikációja saját implementációval
- Adaptív zajszűrés saját implementációja
- A másodlagos út identifikációja CMSIS könyvtár használatával
- Adaptív zajszűrés CMSIS könyvtár használatával

A HIBA állapot feladata, hogy egy funkció rendellenes működése esetén leállítsa azt, és tájékoztassa a felhasználót a leállás okáról. A hibák keletkezését és kezelését a későbbiekben részletesen tárgyalom.

4.2 Programszervezés, periféria állapot flagek

Programszervezési eljárásnak a megszakításokkal kiegészített ciklikus módszert[12]választottam. A struktúrából adódik, hogy a fő ciklusban egy elágazó utasítás biztosítja, hogy az állapotgép megfelelő része hajtódjon végre. A fő ciklus alapvetően nem szorul időzítésre, ám azok az állapotok igen, amelyek jelfeldolgozást végeznek.

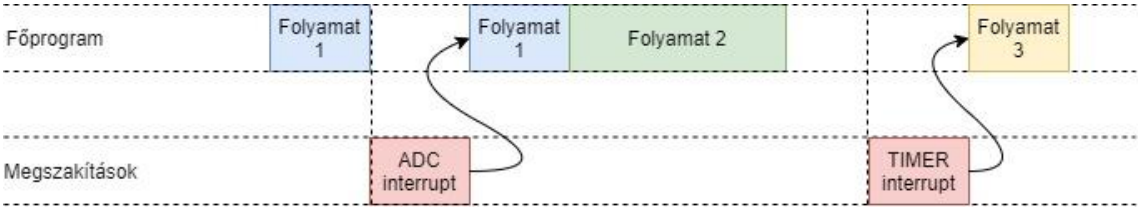
Többek között ez a probléma teremtette meg az igényt arra, hogy valamilyen módon fel lehessen függeszteni a főprogram futását, amíg bizonyos hardveresemények be nem következnek. A megoldás a blokkoló ciklus, illetve egy bináris állapotok tárolására használt változó, a „periféria állapot regiszter”. A változó ebben a realizációban az alábbi bithelyeket tartalmazza:

0.: **Mintavételi ciklus engedélyezve:** értéke 1, ha a következő mintavétel elkezdődhet
1.: **Szöveggörgetés engedélyezve:** értéke 1, ha a tájékoztató szöveg tovább gördíthető a kijelzőn
2.: **ADC érték forrása:** értéke jelzi, hogy melyik csatornáról fogja olvasni az ADC a következő adatot (0, ha a CH0-ról).
3.: **ADC CH0 érték valid**
4.: **ADC CH1 érték valid**
5.: **A mintavételi ciklus befejeződött**

A főprogramot ezek után a következő kódrészlettel lehet felfüggeszteni:

```
while(peripheral_state_flags & (FLAG1 | … | FLAGN));
//várakozás a flagekre
__asm__ volatile (”nop\n”);
peripheral_state_flags &= ~( FLAG1 | … | FLAGN);
//flagek törlése
```

A blokkolás pontosan akkor szűnik meg, ha a flagek közül legalább az egyik bebillen. A kódrészlet közepén látható assembly utasítás a processzort egy gépi ciklusnak megfelelő várakozásra kényszeríti, ellenben a fordítót megakadályozza abban, hogy a blokkoló ciklus elé helyezzen olyan utasítást, amelynek mindenképp később kell bekövetkeznie. Valamelyest letisztultabb megoldás lenne a szekvenciák függvénybe gyűjtése, ám a hívó utasítások sokkal nagyobb overheadet okoznak.



4-2. ábra: A folyamatok megszakítási- és blokkolási sémája

4.3 Hibakezelés, hibaflagek

A program bizonyos helyzetekben képes a saját hibás működésének felismerésére. Amennyiben ilyenbe ütközik, azonnal a HIBA állapotba lép, és jelzi a felhasználónak a probléma jellegét.

A készítés során az alábbi hibák figyelését implementáltam:

- „**watchdog” hiba:** amennyiben a program lefagy, tehát egy bizonyos időnél többet tölt egy ciklus végrehajtásával, abban az esetben ez a hiba jelentkezik. Az észlelés alapja, hogy a mintavételi ciklus kezdetét jelző flaget egy ideje nem törölte semmi. Ha ez nagyon sok megszakítási cikluson keresztül nem következik be, akkor sejthetjük, hogy a program valahol lefagyott. Célszerű ezt a hibát a kritikus blokkoló ciklusokban figyelni, és jelentkezése esetén a ciklusból azonnal kilépni.
- **időkeret hiba:** ha egy mintavételi ciklus nem ér véget a következő időzítő megszakításig, akkor a szinkronizáció felborul. Ez a hiba ilyenkor jelentkezik, megakadályozva a további szabálytalan működést. A hiba észleléséhez a mintvételi cilus befejezését jelző flaget használtam.

- **ADC felülírási hiba:** az ADC segítségével két csatornáról olvasunk adatot. Ha az egyik eredményét nem kezeljük elég gyorsan, azt felülírhatja a másik. Ez a hiba jelzi, ha ez megtörtént. A hiba észleléséhez az ADC megfelelő overflow regiszterét használom.

A hibákat a periféria állapot flagekhez hasonlóan tárolom, gyakorlatilag ugyanabban a változóban. Ennek az az előnye, hogy egyetlen bitművelettel lehet egyszerre hiba- és állapotflaget ellenőrizni, amely nagyban meggyorsítja a feltételek kiértékelését.

A HIBA állapotban a felmerült hibáknak megfelelően futószöveg jelenik meg. A HIBA állapotot az egyik beviteli gomb segítségével lehet nyugtázni, ekkor az állapotgép az egyik (1-es sorszámú) funkcióba ugrik.

4.4 Adattárolás, cirkuláris bufferek

Mivel az algoritmus FIR szűrőket tartalmaz, viszonylag sok korábbi adatot kell megőrizni. Amennyiben azt szeretnénk, hogy az ezeket tároló buffer indexelése szigorúan összefüggjön az elemek beérkezési sorrendjével (tehát a legfrissebb elem a 0. helyen legyen, a második legfrissebb az 1. helyen stb.), akkor minden egyes minta után az egész tömböt el kéne mozgatni egy mezővel, ami nagyon nagy számítási teljesítményt igényel. Ennek kézenfekvő, a gyakorlatban nagyon sokszor használt megoldása az, ha az indexelést modulo N végezzük (N a tömb hossza), és az új mintát mindig a legrégebbi helyére tesszük. A kapott struktúrát (címzési módja miatt) cirkuláris buffernek nevezzük.

A legegyszerűbb programszintű megvalósítása valóban a modulo N aritmetika, azonban a modulo osztás egy költséges művelet, így több egyszerűsítési lehetőség is van:

- A tömb megkettőzése. Ebben az esetben minden elemet duplán kell bemásolni, és nagyobb memóriaterületet is foglal.
- Az indexelés a tömb végéig, majd a tömb elejétől, különválasztva. Itt az indexműveletek kevésbé letisztultak, és a ciklust is érdemes kettéválasztani.

A kettő között (a mi esetünkben) lényeges különbséget nem találtam, így az utóbbi módszernél maradtam. A buffer megvalósítását az alábbi kódrészlettel szemlélteltem:

```
for(i = 0; (i < num_of_datas_to_process) && (i < array_length); i++) {
//addig haladunk, amíg el nem érjük a tömb végét, vagy fel nem dolgozzuk a
kellő számú adatot
     process(array[i]);
}for(; (i < num_of_datas_to_process); i++) {
//a feldolgozás befejezése
     process(array[i - array_length]);
}
```

A kód esetében feltételeztem, hogy a feldolgozandó adatok száma nem haladja meg a tömb hosszát. Az adatokat 32 biten, 15 bites törtrésszel tároltam el. A nagyobb felbontás nem okoz overheadet, mert a processzor akkumulátora is 32 bites.

4.5 A CMU beállítása

A CMU-ban a nagypontosságú külső oszcillátort engedélyeztem és választottam. Az órajelet a maximális 48 MHz-ben állapítottam meg. Az órajelet engedélyeztem a megfelelő perifériák számára.

Ezt az órajelet a perifériák 4-gyel leosztva kapják meg. Mint azt később látni fogjuk, a 12 MHz is elegendően gyors a perifériáknak. Ahol pedig túl gyors, ott nem kell túl nagy osztást beállítani.

4.6 A TIMER beállítása, megszakítás

A TIMER-t default beállításokkal inicializáltam. Kezdőértéknek 0-t állítottam be. A top értéket úgy határoztam meg, hogy a TIMER periódusideje megegyezzen a mintavételi idővel. Ez az alábbi módon számítható:

$$N = \frac{T_s}{T_{clk}} - 1 = \frac{f_{clk}}{f_s} - 1$$

Ezeken kívül engedélyeztem a megszakítást a TIMER0 túlcsordulása esetén, illetve az NVIC számára jeleztem, hogy a TIMER részéről számítani kell megszakítási kérésekre. A megszakítási rutin az alábbiakat végzi el:

- Ha valahány egymás utáni megszakításon keresztül igaz, hogy nem törölték a mintavételi ciklust engedélyező flaget, akkor watchdog hiba jelentkezik.
- Ha az előző mintavételi ciklus még nem fejeződött be, időkeret hiba jelentkezik.

- Minden megszakításnál bebillen a mintavételi ciklust engedélyező flag.
- Minden N'-edik megszakításnál bebillen a szöveg görgetését engedélyező flag.

  $N' = \frac{f_s}{f_{szöveg}} - 1$, a görgetésre vonatkozó top érték.

- A megszakítási flag törlődik

4.7 A GPIO beállítása, megszakítás

A programban két digitális bemenetet használok: a két gombot. Ezeket bementként adom meg a GPIO egységnek, illetve az ezekhez tartozó megszakításokat is inicializálom.

A megszakítási rutinok az állapotgépet léptetik. Az állapotok közti átmenetek során szükség esetén változókat is kezel, például törli a periféria állapot flageket, vagy kiüríti a buffereket.

4.8 Az ADC beállítása, megszakítás

Az ADC teljes (12 bites) felbontással, scan üzemmódban üzemel, a 0. és 1. csatornát olvassa hívásonként egyszer. A referenciának a belső 2.5V-os bandgap referenciát választottam a pontossága miatt[9]. Mivel az energiafogyasztás jelenleg nem szempont, az ADC-t végig bemelegítve tartom. Bemeneti szűrőt nem alkalmazok.

Az interrupt függvény az ADC esetében sok flaget billent feltételesen, ezért ügyelni kell rá, hogy elég gyorsan le tudjon futni, és az ADC ne tudja felülírni a másik csatornáról érkezett mintát. Emiatt az órajelet negyedelnem kellett az ADC belső órajel-osztójával.

A megszakítási rutin az alábbi feladatokat végzi el:

- A forrást jelző flagnek megfelelő változóba tölti a beolvasott értéket.
- A megfelelő valid flaget 1-be állítja.
- A forrást jelző flag állapotát megváltoztatja.
- Ha az ADC-ben történt felülírás, akkor a megfelelő hiba jelentkezik.
- Törlődik a megszakítási flag.

4.9 A DAC beállítása

A DAC beállításai a legtöbb helyen megegyeznek az ADC beállításaival. Az egyetlen kivétel, hogy a DAC nem generál megszakításokat, mivel feltételezem, hogy egy mintavételi ciklus elegendő a kimenet stabilizálódásához.

5 Az LMS algoritmus megvalósítása

A megvalósítás során törekedtem arra, hogy a lehető legkevesebb késleltetést okozza a jelfeldolgozás, hogy a szabályozási kör minél kevesebb holtidőt tartalmazzon. Ez úgy oldható meg, ha az A/D átalakítások és a számítások párhuzamosan történnek, és az ezt követő ciklusban állítjuk be a kimenetet. Utóbbira azért van szükség, hogy a kimeneti változások minél inkább szinkronban maradjanak az időzítéssel.

5.1 A zajszűrés algoritmusa

Az LMS algorimus realizációjához szükségünk van a zavarjel (x) legfrissebb annyi mintájára, hogy mindkét két FIR szűrő összes tapjével tudjunk számolni. Ez $\max(N, M)$ darab, ahol N az adaptív szűrő fokszáma, M pedig a kimeneti (másodlagos út) átvitel becslőjének ($\hat{S}$) fokszáma. Amikor a k-adik ütemre adott választ számítjuk, a zavarjel (x) és a hibajel (e) k-adik mintája még nem áll rendelkezésre. Mivel az A/D átalakítások jelentős időbe kerülnek, ezért olyan algoritmust kell készíteni, amely az A/D átalakítás kezdete és vége közt minden, éppen akkor lehetséges számítást elvégez, így az átalakítás overheadje elkerülhető. Tekintsük most ezeket a számításokat!

A számításokhoz alábbi jelölést vezetem be: egy q változó befejezetlen alakját jelölje q^1. Jelen esetben ez nem keverhető össze a hatványozással.

5.1.1 A hibajel szűrése a kimeneti átvitel becslőjével

Ehhez a művelethez egy konvolúciót kell elvégezni, x M darab legfrissebb mintája és a szűrőegyütthatók között. Ez az alábbiak szerint írható fel:

$$x_s[k] = \sum_{i=0}^{M-1} h_s[i]x[k-i] = h_s[0]x[k] + \sum_{i=1}^{M-1} h_s[i]x[k-i] = h_s[0]x[k] + x_s^1[k]$$

Ebből a második tag a k-adik minták rendelkezésre állása nélkül is kiszámítható.

5.1.2 A szűrő hangolása

A k-adik ütembeli szűrőegyütthatók meghatározásához szükség van a hibajel szűrt alakjának x_s legfrissebb N mintájára, illetve a hibajel k-adik ütembeli értékére. Ezek birtokában számíthatjuk az új szűrőegyütthatókat:

$$W[k] = W[k-1] + \mu e[k]\left[x_s[k], \dots, x_s[k-N+1]\right]$$
$$= W[k-1] + \mu e[k]\left[0, x_s[k-1], \dots, x_s[k-N+1]\right]$$
$$+ \ \mu e[k][x_s[k], 0, \dots, 0] = W^1[k] + \ \mu e[k][x_s[k], 0, \dots, 0]$$

Ebből az első két tag az $x[k]$ ismerete nélkül számítható, így a szűrő összes együtthatója beállítható a 0-adik kivételével.

5.1.3 A válasz számítása

A válaszhoz az eredeti zavarjelet kell szűrni a k-adik ütemre hangolt W-vel. Ez szintén egy konvolúció, az alábbi összefüggés szerint:

$$y[k] = \sum_{i=0}^{M-1} h_{W[k]}[i]x[k-i] = h_{W[k]}[0]x[k] + \sum_{i=1}^{M-1} h_{W[k]}[i]x[k-i]$$
$$= h_{W[k]}[0]x[k] + \sum_{i=1}^{M-1} h_{W^1[k]}[i]x[k-i] = h_{W[k]}[0]x[k] + y^1[k]$$

Mint azt láttuk, a $h_{W[k]}[0]$, azaz a $W[k]$ sorvektor 0. eleme kivételével az összes meghatározható $x_s[k]$, és ezáltal $x[k]$ nélkül, ezért e konvolúció nagy része is elvégezhető a mintavételezés előtt.

5.1.4 Offset megszüntetése exponenciális IIR szűrővel

A szabályozási kör statikus karakterisztikája nemlineáris, mivel az átalakítók valamekkora offsetet visznek bele. Ezek szabályozási körben hagyása eredményezheti annak helytelen működését.

A jelenség megakadályozásához a bemeneteket szűrni kell. Erre a célra egy exponenciális szűrőt alkalmaztam. A választás oka a kicsi számítási igény volt. A szűrő rendszeregyenlete az alábbi:

$$y[k] = \alpha(y[k-1] + u[k] - u[k-1])$$

$$\alpha = e^{-\frac{\omega_c}{f_s}} = e^{-2\pi\frac{f_c}{f_s}}$$

A szűrő válaszának számítása nagyon kevés és olcsó művelettel megoldható, így ennek számítását közvetlenül a jelbeolvasást követően elvégzem. A megtervezett szűrőt a korábbiakban már bemutattam, törésponti frekvenciája hozzávetőleg 5 Hz.

5.1.5 Értékelés, az algoritmus összeállítása

Látható, hogy a műveletek jelentős hányada elvégezhető bizonyos ismeretlenek hiányában. Az egyes számítások előfeltételeit egy táblázatban foglaltam össze, melyben az oszlopok az előfeltételt, a sorok pedig az adott változó számítását jelentik.

	$x_s^1[k]$	$x_s[k]$	$W^1[k]$	$W[k]$	$y^1[k]$	$y[k]$	$e[k]$	$x[k]$
$x_s^1[k]$	0	0	0	0	0	0	0	0
$x_s[k]$	1	0	0	0	0	0	0	1
$W^1[k]$	0	0	0	0	0	0	1	0
$W[k]$	1	1	1	0	0	0	1	1
$y^1[k]$	0	0	0	0	0	0	1	0
$y[k]$	1	1	1	0	1	0	1	1

5-1. táblázat: Az előfeltételek táblázata

Fontos kiemelni, hogy ha a táblázat egy oszlopa tartalmaz egy előfeltételt, akkor tartalmaznia kell az adott előfeltétel összes előfeltételét is.

Jelen esetben a két mintavételre szoruló jel közül a hibajel sokkal több és számításigényesebb műveletben szerepel, mint a zavarjel, ezért azt célszerű hamarabb bekérni, hogy a két mintavételezés közti időt minél inkább kitöltse. Ezért az első lépésünk a hibajel mintavételezése lesz.

Az első számítás, amit elvégezhetünk, az $x_s^1[k]$, mivel ehhez egyik minta se kell. A hibajel mintájának elkészültekor azt begyűjthetjük, és elkezdhetjük a zavarjel mintavételezését.

Ennek elkészültéig a $W^1[k]$, majd az $y^1[k]$ számítása lehetséges, így ezt is elvégezhetük. Amint a minta rendelkezésre áll, az $x_s[k]$, a $W[k]$, végül pedig az $y[k]$ is képezhető. Így az algoritmus az alábbi módon írható le:

1. A legfrissebb $y[k]$ kimenetre küldése
2. $e[k]$ és $x[k]$ mintavételezésének kezdeményezése
3. $x_s^1[k]$ számítása
4. Várakozás, amíg a hibajel nem érvényes
5. $W^1[k]$ számítása
6. $y^1[k]$ számítása
7. Várakozás, amíg a zavarjel nem érvényes
8. $x_s[k]$ befejezése
9. $W[k]$ befejezése
10. $y[k]$ befejezése

5.2 A másodlagos út identifikációja

Az identifikációhoz szintén tudjuk használni az LMS algoritmust. Ebben az esetben az elv az, hogy a FIR szűrőt úgy próbáljuk hangolni, hogy a vizsgálójelre adott válasza minél jobban megközelítse a másodlagos úton egyszer már végig haladt vizsgálójelet.

5.2.1 Zaj generálása

Ahhoz, hogy a kimeneti átvitelt becsülni tudjuk, szükségünk van egy olyan vizsgálójelre, amelynek a spektruma minél inkább egyenletes. Ezt a kritériumot egy fehér zaj kielégíti.

Vegyük észre, hogy a fehér zaj értékkészlete annyi elemből áll, ahány lehetséges kimenete a D/A átalakítónak van. Ez a mi esetünkben 12 biten írható le, tehát a

legkézenfekvőbb periódushossz a 4096 ütem. A célunk az 5kHz-es mintavételi frekvencia, ahol ez 819,2ms-os periódusidőt eredményez.

A fehér zajhoz szükség van egy megbízható véletlenszám-generátorra. Úgy döntöttem, hogy mivel semmiképp nem szeretnék nagy számítási igényű véletlenszám-generálást végezni, ezért megvalósítok egy egyszerű, de hatékony megoldást. Választásom a Galois LFSR algoritmusra esett, mivel képes a 0 kivételével a teljes értékkészletet lefedni.

5.2.2 Galois LFSR

Az algoritmus/eszköz egy többszörösen visszacsatolt shiftregiszter. A visszacsatolások struktúráját általában polinomiális formában szokták megadni, de egy regiszterben tárolt adatsorként fogok rá tekinteni, hogy minél közelebb maradjunk a tényleges megvalósítás elvéhez.

Legyen $LFSR[i]$ a shiftregiszter i-edik eleme, a shiftregiszter hossza (és az MSB bitindexe) pedig N. Ezen kívül szükségünk van egy szintén N bites G regiszterre. Ekkor a shiftregiszter léptetési szabálya az alábbi:

$$LFSR[i+1] \leftarrow LFSR[i] \oplus (G[i+1]LFSR[N])$$

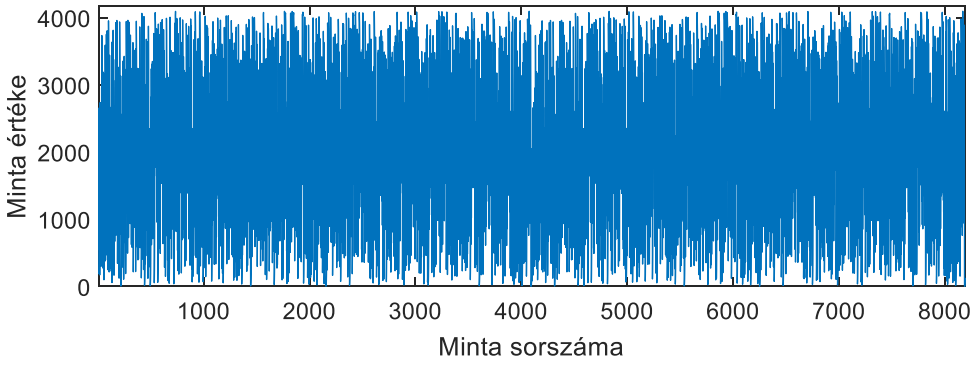
$$LFSR[1] \leftarrow G[1]LFSR[N]$$

Ez a struktúra a G tartalmának megfelelő megválasztásával $2^N - 1$ hosszú ciklusokat állít elő, melyben a csak 0-kat tartalmazó állapotvektor kivételével az összes lehetőség előfordul pontosan egyszer. A mi esetünkben $G = 110010100000$.

A léptetés megvalósítása kimondottan egyszerű, mivel az XOR műveletek bitenkénti XOR-rá vonhatók össze. A léptetés C kódja:

```
if(LFSR & 1) { //a kilépő bit ellenőrzése
     LFSR = (LFSR >> 1) ^ G;
}else{
     LFSR = LFSR >> 1;
}
```

Az algoritmus működését MATLAB-ban vizsgáltam, a megadott struktúra segítségével az alábbi adatsort generálta:



5-1. ábra: Galois LFSR által generált zaj

Az adatsor vizsgálata során megállapítottam, hogy a periódushossz valóban 4095, így a zajgeneráláshoz ezt a struktúrát használtam.

A saját véletlenszám-generálást egyébként az indokolta, hogy az stdlib könyvtárban lévő megoldás úgynevezett LCG-t (Lineáris kongruencia generátor) alkalmaz, amely sokkal költségesebb műveletekből áll.

5.2.3 Az identifikációs algoritmus összeállítása

Az LMS algoritmushoz képest nagyon fontos eltérés, hogy csak egy adatot olvasunk be, a gerjesztés ugyanis a mikrokontrolleren belül jön létre. Emiatt ez az algoritmus lényegesen egyszerűbb lesz.

A lépéseket itt aszerint lehet csoportosítani, hogy szükség van-e a hibajelre az elvégzésükhöz. Ebből a szempontból a becslő hangolása, illetve az ezzel történő szűrés a kitüntetett művelet.

5.3 Összefoglalás

Megfelelő megfontolásokkal olyan algoritmust hoztam létre, mely a mintavételezés szigorú kereteit tartja, ugyanakkor minden számítást a lehető legkorábbi időpontban elvégez.

6 Az ARM DSP könyvtára[13]

Az ARM a saját fejlesztésű architektúráinak teljes ismeretében kidolgozott egy olyan réteget, amely elválasztja a fejlesztőt a különböző gyártók hardvereinek sajátosságaitól azzal a céllal, hogy az új projektek fejlesztése minél gördülékenyebben menjen. Ennek során rengeteg széleskörűen használt függvényt implementáltak, köztük gyakorlatilag minden alapvető DSP megoldást, így az LMS szűrőt is.

Mivel a gyártó az algoritmusok esetében is az ARM, erősen optimalizált algoritmusokra számíthatunk, és ez az esetek többségében teljesül is. Pontosan ez az, ami miatt a CMSIS DSP könyvtárban található függvények ideális összehasonlítási alapot képeznek a saját megvalósítás teszteléséhez. A következőkben azokat a függvényeket és adatstruktúrákat fogom részletesen bemutatni, amelyek az általam implementált algoritmusok bonyolultabb lépéseit tudják helyettesíteni.

Fontos megemlíteni, hogy ezek a függvények több különböző fixpontos formátumot is támogatnak, jellemzően a q15-öt és q31-et (1.15 és 1.31). Ezek között a lényeges különbség az, hogy a q31 képes túlcsordulni, mialatt a q15 telítéses viselkedéssel bír. Munkám során q15-öt használtam.

6.1 FIR szűrők

A CMSIS DSP könyvtára többek között a FIR szűrők esetében is tartalmaz könnyedén implementálható megoldást. A megoldás alapja egy struktúra, melynek előkészítésére és felhasználására a könyvtár függvényeket biztosít:

```
arm_fir_instance_q15
```

Ahhoz, hogy a struktúra ismerje a FIR szűrő paramétereit, és legyen olyan memóriaterület, ahova a korábbi bemeneteket menteni tudja, inicializálni kell, mely az alábbi függvény segítségével lehetséges:

```
arm_status arm_fir_init_q15    (    arm_fir_instance_q15 *     S,
uint16_t   numTaps,
const q15_t *   pCoeffs,
q15_t *    pState,
uint32_t   blockSize
)
```

A függvény rendre paraméterként várja az inicializálandó struktúrát, a szűrő tap-számát, az együtthatókat tartalmazó tömböt, egy olyan memóriaterületet, ahol a korábbi bemeneteket tárolhatja, illetve azt, hogy hány adatot dolgozzon fel egyszerre. A függvény q15-ös változatában a tap-szám csak páros lehet, legalább 4. Amennyiben ettől eltérő fokú FIR-szűrőnk van, azt csak úgy lehet bevinni, ha a fokszámot megfelelően bővítjük, a nem használt tap-ek erősítésének pedig 0-t választunk. A függvény visszatérési értéke jelzi számunkra, hogy sikerült-e a létrehozás

```
ARM_MATH_SUCCESS //sikeres létrehozás
ARM_MATH_ARGUMENT_ERROR //a tap-szám nem megfelelő
```

Az inicializált szűrőstruktúra segítségével az adatfeldolgozást ez a függvény végzi:

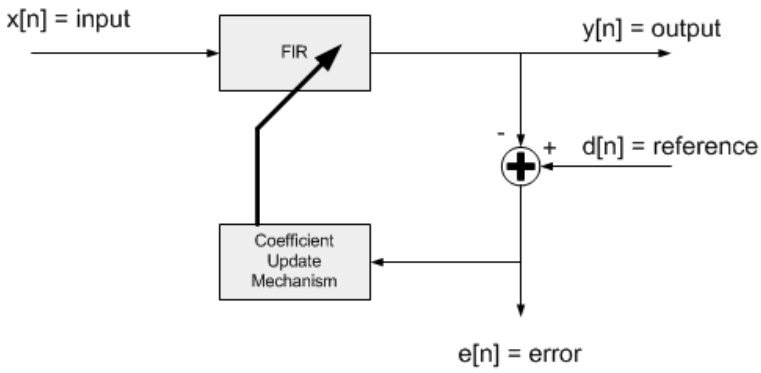
```
void arm_fir_q15     (    const arm_fir_instance_q15 *   S,
const q15_t *   pSrc,
q15_t *    pDst,
uint32_t   blockSize
)
```

A függvény a struktúrát pointerként veszi át, ahogy az adatforrást és az adat célhelyét is. Az egyszerre feldolgozandó adatok számát itt is meg lehet adni. A függvénynek van egy arm_fir_fast_q15 nevű változata is, amely sokkal gyorsabb, ám nincs védve a túlcsordulástól.

6.2 Az LMS algoritmus megvalósítása

Az LMS szűrő hasonlóan egy struktúra és az ezt kezelő függvények segítségével működik, ám ezek valamennyivel bonyolultabbak, mint a FIR szűrő megvalósítása.

Ahhoz, hogy alaposabban megértsük, miképp is működnek ezek a függvények, látnunk kell az egyes jelek szerepét, illetve hogy ezek hova csatlakoznak a szabályzási körön belül. Az ARM dokumentációjában található jelmagyarázat ezt viszonylag jól bemutatja.



6-1. ábra: Az ARM LMS-szűrőjének jelmagyarázata[14]

Az ábra tanulsága szerint nekünk a gerjesztést és az identifikálandó rendszer válaszát kell biztosítanunk, a hibajelet és az LMS válaszát pedig a függvény biztosítja. Lássuk ennek tükrében a struktúrát létrehozó függvényt:

```
void arm_lms_init_q15     (     arm_lms_instance_q15 *    S,
uint16_t   numTaps,
q15_t *    pCoeffs,
q15_t *    pState,
q15_t      mu,
uint32_t   blockSize,
uint32_t   postShift
)
```

Ez a függvény az inicializálandó struktúrát várja, a belső FIR szűrő tap-számát, memóriaterületet az együtthatók számára, egy állapotbuffert, a bátorsági tényezőt, az egyszerre feldolgozandó adatok számát, illetve egy postShift nevű értéket. Utóbbinak az a szerepe, hogy a szűrőegyütthatók látszólag kiléphessenek a [-1, 1) tartományból, amelybe egyébként a számábrázolás kényszeríti őket. Ezt azáltal éri el, hogy a szűrő kimenetét utólagosan bitshifteli a beállított értékkel.

Miután a struktúra kezdeti értékeit beállítottuk, az arm_lms_q15 függvény hívásával léptethetjük az algoritmust, melynek függvénytörzse itt látható:

```
void arm_lms_q15     (     const arm_lms_instance_q15 *   S,
const q15_t *   pSrc,
q15_t *    pRef,
q15_t *    pOut,
q15_t *    pErr,
uint32_t   blockSize
)
```

Ez a függvény szintén a struktúrát várja, ezen kívül pedig a különböző bemeneteket és kimeneteket (sorrendben: közös gerjesztés, identifikálandó rendszer

válasza, LMS kimenete, a keletkező hibajel). A blockSize paraméter az egyszerre feldolgozandó adatok számát adja meg.

Fontos felismerni, hogy a függvény nem a hibajelet várja, ám néhány alkalmazásban elsősorban a hibajel áll elő. Ez esetben azt a trükköt lehet alkalmazni, hogy referenciajelnek a hibajel és a kimenet összegét adjuk meg. Ilyenkor a hibajel a megfelelő helyen, a különbségképzés után keletkezik.

7 Az algoritmusok tesztelése és összehasonlításuk

7.1 Tesztelési módok, értékelés szempontrendszere

Mivel a tesztelések során nem állt rendelkezésemre megbízható próbaáramkör, ezért úgy határoztam, hogy az identifikálni kívánt átviteleket a programon belül, diszkrétidőben valósítom meg, viszont az analóg perifériák tesztelése érdekében nem a programon belül, hanem az ADC-n és a DAC-n keresztül csatolok vissza.

A diszkrét átvitelek identifikálásának végül lett egy olyan következménye, hogy tetszőleges átvitelt megpróbálhattam megközelíteni az LMS segítségével, köztük olyanokat is, amelyek Kirchhoff-típusú áramkörrel csak elméletben realizálhatók (pl. ideális lyukszűrő).

Végül úgy láttam jónak, ha egy lyukszűrőt közelítek az algoritmussal, ugyanis a tökéletes elnyomáshoz nagyon pontosan el kell találni az együtthatókat. Az elnyomás mértéke a lyukszűrő vágási frekvenciáján jellemzi a közelítés minőségét.

Az LMS tesztelésére használt lyukszűrő átviteli függvénye az alábbi:

$$H_{notch}(z) = \frac{5}{8}\frac{1 - 0.5z^{-1} + z^{-2}}{1 + 2.5z^{-1} + 5z^{-2}}$$

Az elnyomás azon a frekvencián fog jelentkezni, ahol a számláló nulla, tehát

$$\left(e^{j\theta}\right)^2 - \frac{1}{2}e^{j\theta} + 1 = 0$$

$$e^{j\theta} = \frac{\frac{1}{2} \pm j\sqrt{\frac{15}{4}}}{2}$$

Ez előállhat, ugyanis a jobb oldali kifejezés hossza valóban 1. Ebben az esetben a diszkrét körfrekvencia

$$\theta_{1,2} = arg\left(\frac{\frac{1}{2} \pm j\sqrt{\frac{15}{4}}}{2}\right) = arctg\left(\pm\sqrt{15}\right) = \pm 1.3181 rad$$

Az FxLMS algoritmus tesztelésekor egy másik átvitelt kellett használnom a másodlagos útra, ugyanis a szűrő az elsődleges- és másodlagos átvitel hányadosaként áll elő, emiatt a másodlagos átvitel zérusaiból pólusok lesznek. Mivel ezek az egységkörön helyezkednek el, a kapott átvitelnek nem lesz lecsengése, ezt pedig FIR szűrővel megközelíteni lehetetlen. Emiatt az átvitelt kis mértékben módosítanom kellett.

A másik szempont a hányados viszonylagos egyszerűsége volt, így az elsődleges átvitel pólusai megegyeznek a másodlagoséval. Ezen megfontolások mentén az alábbi átvitelek adódtak:

$$P(z) = \frac{5}{64}\frac{1 - z^{-1} + 0.5z^{-2}}{1 + 2.5z^{-1} + 5z^{-2}}$$

$$S(z) = \frac{5}{8}\frac{1 - 0.5z^{-1} + 0.5z^{-2}}{1 + 2.5z^{-1} + 5z^{-2}}$$

$$W_0(z) = \frac{1}{8}\frac{1 - z^{-1} + 0.5z^{-2}}{1 - 0.5z^{-1} + 0.5z^{-2}}$$

A két átvitel szimulációjakor arra is kell figyelni, hogy feltétlenül az elsődleges átvitel késsen többet, ellenkező esetben a becsülendő rendszer akauzális lenne.

A két megvalósítás összehasonlításának kizárólag futási hatékonyság szempontjából van értelme, ugyanis ugyanazokat az összefüggéseket használják.

Erre az egyik alapvető teljesítménymutató az, hogy hogyan függenek össze a szűrőparaméterek a lépésszámmal, maximális frekvenciával stb.

A lépésszámnak két összetevője van: a szoftverkeret órajelszáma ($K_{FRAMEWORK}$), illetve a szűrőegyüttható̋k újrakalibrálásából és a kimenet számításából adódó órajelszámszám (K_{FILTER}). A kettőnek az összegeként adódik a teljes órajelszám.

$$K = K_{FRAMEWORK} + K_{FILTER}$$

Ezen felbontásnak azért van értelme, mert a szűrőszámítások a tap-számmal várhatólag lineárisan változnak, és ennek a változásnak a mértéke az, ami igazán szemléletesen mutatja az algoritmus hatékonyságát. A változás meredekségét (k_{FILTER}) bevezetve, és m darab szűrőt feltételezve $N_1, N_2, \dots, N_m$ tap-számokkal, a teljes órajelszámra az alábbi összefüggést kapjuk:

$$K(N_1, N_2, \dots, N_m) = K_{FRAMEWORK} + \sum_{i=1}^{m} k_{FILTER,i} N_i$$

Ezek után számítsuk ennek a meredekségét!

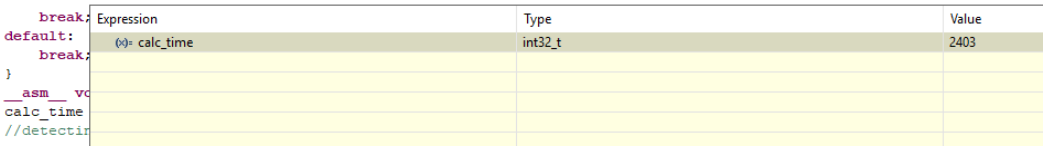
$$k_{FILTER,i} = \frac{\partial K}{\partial N_i} = \frac{K(N_1, N_2, \dots, N_i', \dots, N_m) - K(N_1, N_2, \dots, N_i, \dots, N_m)}{N_i' - N_i}$$

Emiatt ha meg tudjuk mérni a teljes órajelszámot, akkor az összes együtthatóra és az alap lépésszámra is tudunk következtetni. Ez pedig úgy lehetséges, ha lekérdezzük a CORE időzítő egyik regiszterét az alábbi címzéssel:

```
DWT-> CYCCNT
```

Ez az időzítő az indítás óta eltelt órajelek számát tárolja, így a futás elején és végén kiolvasva és különbséget képezve megkapjuk a kívánt értékeket.

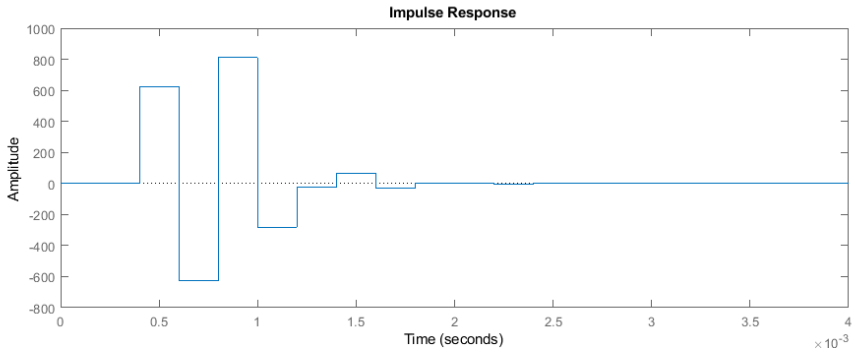
A kiolvasásra kiváló lehetőséget nyújt a Simplicity Studio trace funkciója.



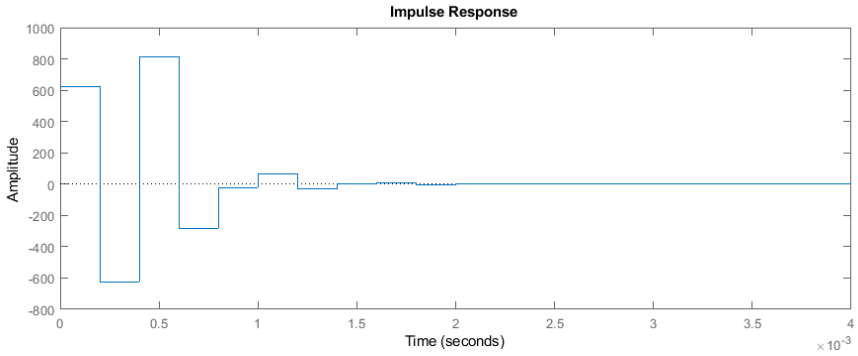
7-1. ábra: A Simplicity Studio trace funkciója

7.2 A saját algoritmus tesztelése

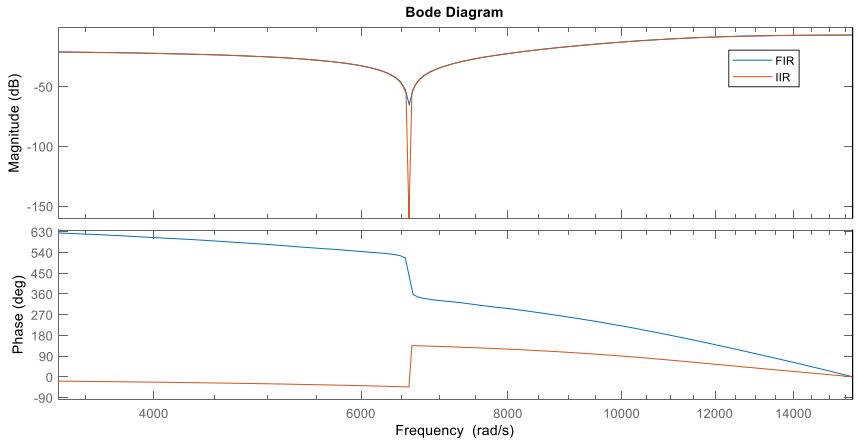
Elsőként az identifikációt teszteltem a már bemutatott lyukszűrő segítségével, 50 tap-es szűrővel. A lyukszűrő és a létrejött FIR szűrő átvitelei az alábbi ábrákon láthatók:



7-2. ábra: A becslő FIR szűrő impulzusválasza



7-3. ábra: A becsült lyukszűrő impulzusválasza



7-4. ábra: A becslő (FIR) és a becsült (IIR) szűrő Bode-diagramjai

A becslő szűrő elnyomása a lyukszűrő vágásánál -65 dB, ami megfelelőnek mondható. A két impulzusválasz közötti két mintányi késleltetést az okozza, hogy a kimeneten egy mintavételi ciklussal később jelennek meg az adatok, és ezeket is a generált zaj frissítése előtt számítom. Ez okozza a két átvitel fázisdiagramja közti eltérést is.

Az LMS algoritmus általagosan az alábbi lépésszámokkal futott le:

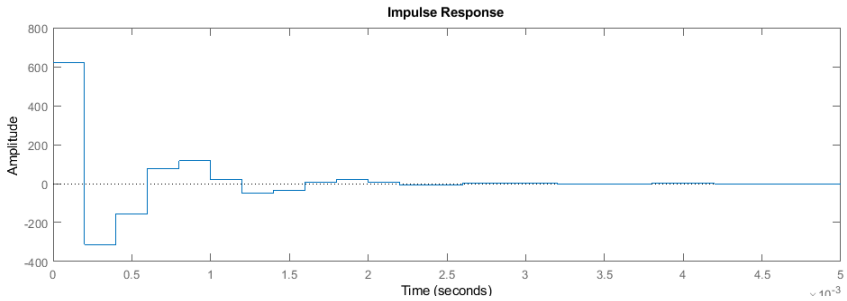
- 30 tap: **1670**
- 50 tap: **2225**

Ebből az alábbi értékek adódnak az érzékenységre és a keretrendszerek alap lépésszámaira:

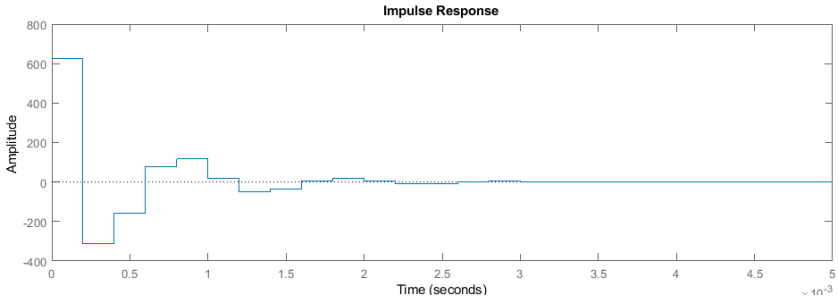
$$k_s = \frac{\partial K}{\partial N_s} = 27.75$$

$$K_{FRAMEWORK} = 837.5$$

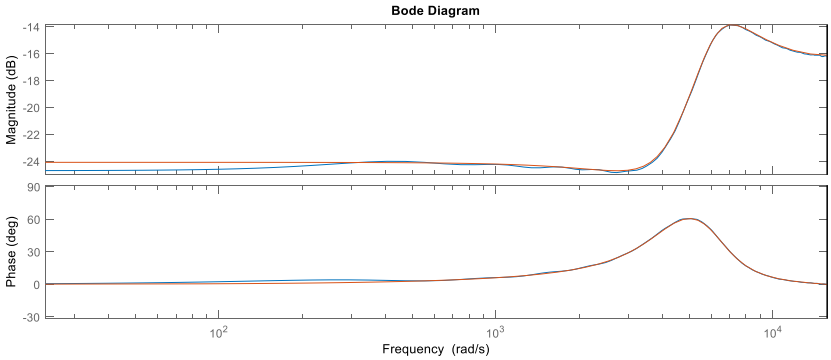
Ezek után az FxLMS tesztelése következett. Mivel nem állt rendelkezésemre külső zajforrás, ezért a Galois LFSR alapú zajgeneráló függvényemet használtam gerjesztésként. Mind a másodlagos út átvitelének becslője, mind a hangolt szűrő 50-ed fokú. A kapott impulzusválaszok és Bode-diagramok az alábbi ábrákon láthatók:



7-5. ábra: A becslő FIR szűrő impulzusválasza (FxLMS)



7-6. ábra: Az elsődleges és másodlagos út átvitelének hányadosának impulzusválasza



7-7. ábra: A két átvitel összehasonlítása Bode-diagramon

Az LMS-sel ellentétben itt nem látunk eltolódást időben, mivel az elsődleges- és másodlagos átvitel a mintavételezések miatt pontosan ugyanannyit késleltet, így ez a becslő szűrőben nem jelenik meg.

Ezután az FxLMS futási idejét vizsgáltam, mere vonatkozó méréseim az alábbiak szerint alakultak:

- $N_s = 30, N_f = 30$: **2057**
- $N_s = 30, N_f = 50$: **2589**
- $N_s = 50, N_f = 50$: **2752**

Ezekből a már ismertetett módon számíthatóak az érzékenységek, illetve a keretrendszer lépésszáma:

$$k_s = \frac{\partial K}{\partial N_s} = 8.15$$

$$k_f = \frac{\partial K}{\partial N_f} = 34.75$$

$$K_{FRAMEWORK} = 740$$

Az eredmény a várakozásainknak megfelel, mivel hozzávetőlegesen ugyanolyan érzékenységeket kaptunk. A másodlagos átvitel becslőjére vonatkozó érzékenység azért ilyen kicsi, mert ott nem kell az együtthatókat hangolni, csupán egy konvolúciót kell elvégezni.

7.3 Az ARM implementáció tesztelése

Az ARM esetében az LMS-alapú identifikációt valósítottam meg. Szerettem volna bemutatni a képességeit, ám az eredmények közt nem volt észrevehető különbség. Ez egyébként nem is meglepő, hiszen teljesen ugyanakkora törtrészű számábrázolást használok, a működés alapelvei pedig megegyeznek.

Az ARM algoritmusa esetében is meghatároztam a lépésszámokat, melyek az alábbira adódtak:

30 tap: **2023**

50 tap: **2702**

Ebből kiszámítottam az érzékenységet és az offsetet.

$$k_s = \frac{\partial K}{\partial N_s} = 33.95$$

$$K_{FRAMEWORK} = 1044.5$$

7.4 Az algoritmusok értékelése

A kapott mérőszámokat az alábbi táblázatban foglaltam össze:

	LMS	ARM LMS	FxLMS
$K_{FRAMEWORK}$	**837.5**	**1044.5**	**740**
k_s	**27.75**	**33.95**	**8.15**
k_f	-	-	**34.75**

7-1. táblázat: Az algoritmusok lépésszámainak összehasonlítása

Értékelésük során az derült ki, hogy a saját algoritmusom hozzávetőleg azonos számítási igényű, mint az ARM hasonló célú függvényei. Ez először meglepő lehet, ám gondoljunk arra is, hogy

- Az ARM megoldása tartalmaz függvényhívásokat
- Paraméteként veszi át a tap-számot, így bizonyos aritmetikai műveleteket kénytelen futásidőben végrehajtani
- Egyes lépésekkel kapcsolatban nincs sok jelentős lehetőség az optimalizációra

A saját algoritmusom előnye, hogy az ADC szabadon tud mintavételezni, ameddig az algoritmus végzi a számításokat, ám összetett és nehéz karbantartani. A törtrész hossza az ARM implementációjában nem változtatható, ellenben a lehető legoptimálisabb. Van viszont a gyári LMS-nek még egy nagy előnye: a q15-ös változat mindenhol védve van a túlcsordulás ellen.

8 Összefoglalás, kitekintés

Szakdolgozat-írásom alatt elmélyedtem az LMS szűrők elméletében, alaposan megismertem a Silicon Laboratories EFM32GG sorozatú mikrokontrollerét, illetve betekintést nyertem az erre írt ARM könyvtárba. Saját algoritmusom megírásához a programszervezési módokkal és eseményvezérelt programozással kapcsolatos ismereteim segítettek, a munkát pedig a Simplicity Studio 2.4-es verziója alatt végeztem. Az eredmények szemléletes ábrázolásához és elemzéséhez a R2018b MATLAB -ot használtam, beleértve a Control System Toolboxot[15] és Simulinket is.

A téma sok fejlesztési lehetőséget felvet. Egyrészt a jövőben meg lehetne vizsgálni az algoritmuscsalád többi elemét is (pl. Normalizált LMS), másrészt mivel a szabályozás diszkrét időben történik, a körben lehet elég nagy holtidő, illetve a szakasz modelljét elég pontosan ismerjük az előzetes LMS identifikáció miatt, az FxLMS-t ki lehetne egészíteni egy Smith-prediktorral.

Ezen kívül érdemes lenne egy konkrét akusztikus feladatban bemutatni a működést. Ez a céljaim között is szerepelt, ám (ahogyan arra már utaltam) a megvalósítás nehézségekbe ütközött, mivel sehogyan sem tudtam megbízhatóvá tenni a meghajtott áramkör működését. A vegyes eredmények miatt döntöttem végül úgy, hogy ezt a törekvésemet nem foglalom a szakdolgozatom kereteibe.

9 Irodalomjegyzék

[1] Lubna Badri: Development of Neural Networks for Noise Reduction
https://www.semanticscholar.org/paper/Development-of-Neural-Networks-for-Noise-Reduction-Badri/9503c47050f4390bdf54d9c2d476e364dfdba131

[2] Fodor György: *Hálózatok és Rendszerek (Műegyetemi Kiadó)*

[3] Eva Ostertagová, Oskar Ostertag: *The Simple Exponential Smoothing Model*
https://www.researchgate.net/publication/256088917_The_Simple_Exponential_Smoothing_Model

[4] Prof. Yaron Singer: *AM 221: Advanced Optimization*
https://people.seas.harvard.edu/~yaron/AM221-S16/lecture_notes/AM221_lecture9.pdf

[5] Sujbert László, Balogh Tibor (BME MIT): *Adaptív szűrők vizsgálata*
https://www.mit.bme.hu/system/files/oktatas/targyak/10227/inflab-3m.pdf

[6] Sen M. Kuo and Dennis R. Morgan: *Active Noise Control: A Tutorial Review*
https://www.researchgate.net/publication/2985088_Active_noise_control_A_tutorial_review

[7] Ismeretlen szerző: *Nulladrendű tartó válasza*
https://commons.wikimedia.org/wiki/File:Zeroorderhold.signal.svg

[8] Silicon Laboratories: *EFM32GG Reference manual*
https://www.silabs.com/documents/public/reference-manuals/EFM32GG-RM.pdf

[9] Silicon Laboratories: *EFM32GG Datasheet*
https://www.silabs.com/documents/public/data-sheets/efm32g-datasheet.pdf

[10] Silicon Laboratories: *AN0021 Datasheet*
https://www.silabs.com/documents/public/application-notes/AN0021.pdf

[11] Silicon Laboratories: *EFM32GG-BRD2200A-A03 schematic*
https://www.silabs.com/documents/public/schematic-files/EFM32GG-BRD2200A-A03-schematic.pdf

[12] Iivo Raitahila: *Software Architectures in Embedded Systems*
https://www.cs.helsinki.fi/u/iivorait/Software_Architectures_in_Embedded_Systems.pdf

[13] ARM: *CMSIS DSP függvénydokumentáció*
http://www.keil.com/pack/doc/CMSIS/DSP/html/modules.html

[14] ARM: *CMSIS LMS könyvtárak*
http://www.keil.com/pack/doc/CMSIS/DSP/html/group__LMS.html

[15] MathWorks: *Using the Control System Toolbox*
https://edoras.sdsu.edu/doc/matlab/pdf_doc/control/usingcontrol.pdf