Szakdolgozat

Mikrokontrolleres aktív rezgéscsökkentő modul tervezése

Készítette:

Kun Balázs Lajos

Konzulens:

Dr. Sujbert László

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

2010.

Nyilatkozat

Alulírott, ***Kun Balázs Lajos*** a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

....

Kun Balázs Lajos

Tartalomjegyzék

Abstract 4
Összefoglalás 6
1. Bevezetés 8
2. Az aktív rezgéscsillapítás 11
2.1. A modellillesztés feladata 13
2.2. Az LMS algoritmus 15
2.3. Az NLMS algoritmus 16
2.4. Az FxLMS algoritmus 17
2.5. A választott struktúra, és algoritmus bemutatása 18
2.5.1. Matlab szimulációk 21
3. Az aktív rezgéscsökkentő modul megvalósítása 26
3.1. A modul rendszerterve 26
3.2. A mikrokontroller választása 28
3.3. A modul elvi kapcsolási rajza 31
3.4. A modul nyomtatott áramköri terve 36
3.5. Az implementált program 37
4. A mérési eredmények 41
4.1. A rendszer identifikációja 42
4.2. A rezgéscsillapítás eredményei 43
5. Értékelés, a továbbfejlesztés lehetőségei 45
6. Irodalomjegyzék 46
7. Függelék 47
7.1. A Matlab szimuláció forráskódja 47
7.2. A mikrokontroller forráskódja 49

Abstract

Nowadays passive mechanical vibration control is used in many applications, because they are reliable, and their build cost is low. But these vibration control systems have a disadvantage. They can isolate only the vibratory unit from the other ones, and can not cancel the vibration itself.

Active vibration control is a technology, which uses the phenomenon of destructive interference to suppress low-frequency mechanical disturbances with opposite phase vibration. An accelerometer collects the information from the vibranting system, and the vibration control algorithm processes the signal. The algorithm generates a signal for an actuator (which is usually a shaker). The accelerometer and the shaker are situated nearly at the same place of the mechanical system. The result of the destructive vibration and the antivibration is the error signal, which is sensed by the accelerometer. The control algorithm is out to minimize the vibration at the place of the sensor.

The electronic active control systems became possible to realize only by the spreading of digital signal processors, because they can process the complicated algorithm by a few kHz of sample frequency, and 24-32 bits of float scale. Digital signal processors are special, expensive tools for development, and in some cases they are oversized. E.g., when low sampling frequency and a simple algorithm is enough, as it is the case of low frequency vibration control.

This Bachelor Thesis was out to design a modul, which includes a low cost microcontroller, and able to suppress the low frequency vibration.

The designed modul includes an Atmega88-20 microcontroller, which is one of the products of the ATMEL company. The proposed schematics includes AD, DA converters for the signal transformation, and a unit for analog signal conditioning. The embedded software processes the signals at the sampling frequency of 200 Hz. In an experimental system, where the frequency of the vibration was about 30 Hz, the system could reach 11.5 dB supression. It has shown that the modul can successfully complete the passive applications.

Összefoglalás

Manapság is még nagyon sok helyen alkalmaznak passzív, tisztán mechanikus elvű rezgéscsillapítókat, mert alacsony kiépítési költség mellett képesek hosszú távon megbízhatóan működni. Ezeknek a csillapítóknak nagy hátránya azonban, hogy a rezgést végző testet képesek csak elszigetelni a többi egységtől. A rezgést végző egységet csillapítani nem tudják, vagy csak minimális mértékben.

A káros rezgések csillapítására elterjedten használnak aktív, elektronikus szabályozókat. A rezgésről információt a gyorsulásérzékelő gyűjt, az adatok feldolgozása egy rezgéscsillapító algoritmus feladata, ami egy jelfeldolgozó egységen van implementálva. Az algoritmus mindig előállít a gyorsulásérzékelő aktuális jele alapján egy beavatkozó jelet, amelyet általában egy shaker segítségével alakít mechanikus jellé. A gyorsulásérzékelő és a beavatkozó a rendszerben ugyanazon a helyen van elhelyezve, így a káros rezgés, valamint az ellenrezgés szuperpozíciójaként előálló hibajelet a gyorsulásérzékelő újra az algoritmus bemenetére adja, így számítható a beavatkozó jel következő mintája. Az algoritmus célja tehát, hogy a gyorsulásérzékelő helyén a nem kívánt rezgést minimalizálja.

Az elektronikus rezgéscsillapítás jelfeldolgozójaként általában jelprocesszorokat használnak, amelyek akár több kHz mintavételi frekvencia, valamint 24-32 bites lebegőpontos számábrázolás mellett is képesek nagyon bonyolult elnyomó algoritmusok futtatására. Ezek az eszközök azonban drágák, és vannak olyan területek, ahol nincsenek is teljesen kihasználva, valamint beépítésük költsége a termék árában túlzottan érzékelhető lenne. Azokon a területeken tehát, ahol párszor 20 Hz frekvenciájú rezgés szabályozása a cél, ott nem szükséges 500-600 Hz-nél nagyobb mintavételi frekvenciát alkalmazni, valamint elegendő egyszerűbb algoritmus is.

A dolgozat célja olyan néhány eurós mikrokontroller keresése, valamint ehhez az eszközhöz egy olyan modul tervezése és elkészítése volt, amellyel megoldható egy alacsony frekvenciás rezgés csillapítása.

A választás a ATMEL cég által gyártott Atmega88-20PU típusú mikrokontrollerre esett. A mikrokontrollerhez tervezett kapcsolási rajz tartalmazza - többek között az analóg gyorsulásérzékelő jelének kondicionálását megvalósító áramköri elemeket, valamint a szükséges AD, és DA konverziókat elvégző feldolgozó egységeket. A kapcsolási rajz alapján megtervezett, nyomtatott áramköri lapon elvégzett programfejlesztések eredményeként, az elkészült modul 200 Hz mintavételi frekvencia mellett képes volt 11,5 dB-el csökkenteni a nemkívánatos megközelítőleg 30 Hz frekvenciájú rezgést. Ezzel bebizonyította, hogy az elgondolás működőképes, és a passzív rezgéscsillapítás megoldásainak kiegészítője lehet.

1. Bevezetés [1],[2]

Az akusztikus eredetű zajokhoz hasonlóan a mechanikus rezgések is károsak lehetnek. Elég csak olyan balesetekre gondolni, mint például az Egyesült Államokbeli Tacoma tengerszorosban 1940-ben átadott függőhíd összeomlása. A híd a rezonanciakatasztrófának nevezett jelenség áldozata lett. Átadását követően, ha a szél feltámadt, érezhetően lengeni kezdett a szerkezet, mígnem egy szélviharban a keletkező rezgés frekvenciája a híd sajátfrekvenciájával szerencsétlen módon megegyezett. A rezgés a rendszer kis csillapítása miatt, addig erősödött, míg a híd össze nem omlott. Azonban nagyon sok repülőgép, hajó, épület, végezte hasonló módon, esetenként nagyon sok emberéletet követelve, mert nem megfelelően tervezték, vagy mert nem készítették fel őket a keletkező rezgések csillapítására, esetleg kioltására.

Ma is még nagyon sok, tisztán mechanikus megoldást alkalmazó rezgéscsökkenőt használnak szerte a világon. Ezek célja a rezgést folytató rendszer elszigetelése a többi egységtől, hogy azokra ne legyen káros hatással. (Például motorok gumibakos felfüggesztése.) A keletkezett rezgés kioltására, ezek a passzív, mechanikus rendszerek képtelenek. Ami miatt mégis kedveltek, az a nagyon alacsony ár és alacsony telepítési költség. Pár évtizede megjelentek az elektronikus felépítésű, aktív rezgéscsökkentők, amelyek képesek a fenti probléma megoldására. Ez a mikroprocesszorok, mikrokontrollerek és jelprocesszorok számítási kapacitásának növekedése, áruk csökkenése, valamint megbízhatóságuk növekedésének következménye.

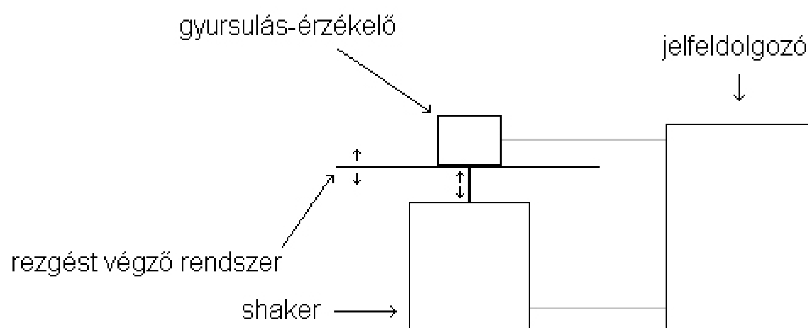
T. Kobori 1997-ben megjelent cikkében *[2]* arra mutat rá, hogy mennyire fontos a magas épületek felkészítése az állandóan jelentkező szél, valamint az esetleges földrengés keltette kilengésekre. Munkájában azt vizsgálta, hogyan lehet az épületet a talapzatánál fogva elmozgatni úgy, hogy az épület lengését csökkentsék. A tetőn elhelyezett gyorsulásérzékelők jelei alapján, egy elektronikus jelfeldolgozó algoritmus adja ki a beavatkozóknak, hogy merre kell mozgatni az épületet.

Napjainkban az aktív zajcsökkentésre, és az aktív rezgéscsillapításra jelprocesszorokat használnak. Ezek az eszközök képesek több 10 kHz mintavételi frekvencia mellett is bonyolult csillapító algoritmusok futtatására 24-32 bites számábrázolással. Ezeket áruk miatt azonban nem lehet minden területen használni, és valahol nincs is szükség az ekkora kapacitásra.

A zajcsökkentés, az akusztikus tér modellezése miatt bonyolultabb algoritmusokat kíván, mint a rezgéscsillapítás. Utóbbi esetén kevesebb változóval írható le a rendszer, kisebb fokszámú szűrőkre van szükség, és alacsony frekvenciás rezgések feldolgozására néhány kHz mintavételi frekvencia is elegendő. Olyan felhasználási területeken, ahol kis frekvenciás rezgéseket kell csillapítani, és esetleg több különálló feldolgozó egységre is szükség van, valamint az áramkör alacsony fogyasztása, és ára fontos, akkor megfelelő választás egy 8 bites mikrokontroller.

A dolgozat célja, egy 8 bites mikrokontrolleren alapuló rezgéscsökkentő modul tervezése, amely egy szenzor jelet képes fogadni, valamint egy beavatkozó jelet meg tud határozni, és ezzel egy alacsony frekvenciás rezgést végző egység valós idejű szabályozását meg tudja valósítani.

A rezgéscsillapító rendszer felépítését tekintve három részre bontható, amint az *1. ábrán* is látható. Az első, a rezgő rendszerről villamos információt szerző gyorsulásérzékelő, a második egy jelfeldolgozó, a harmadik pedig egy beavatkozó, ami az esetek többségében egy shaker.



1. ábra. A rezgéscsillapító rendszer elemei

A mozgó tárgyon elhelyezett gyorsulásérzékelő, a rezgés frekvenciájának és amplitúdójának megfelelő feszültség jelet állít elő, melyet a feldolgozó egységnek továbbít. A feldolgozó egységen implementálva van egy adaptációs és egy szűrő algoritmus. A feldolgozó, olyan jelet állít elő, és ad ki a hozzá illesztett shaker segítségével, hogy a gyorsulásérzékelő helyén a rezgés, és az ellenrezgés összegeként adódó eredő jelet minimalizálja.

Az elv alkalmazható akkor is, ha kiterjedt test rezgését kell csillapítanunk. A kiterjedt, nagy felületű testek nem elég merevek ahhoz, hogy minden tartományuk ugyanabban a fázisban rezegjen. Például, ha egy lemezen transzverzális hullám halad keresztül, akkor egy érzékelő-beavatkozó párossal csak nagyon kis tartomány rezgését tudjuk csillapítani, az egész felületét nem. A megoldást több érzékelő, valamint a hozzájuk tartozó beavatkozó telepítése jelenti. Az így létrejövő MIMO (*multiple-input multiple-output*) rendszerben a különálló szenzorok jelei alapján egy, vagy bemeneti jelek számának megfelelő mennyiségű jelfeldolgozó egység határozza meg a beavatkozók jeleit. Ebben a rendszerben a beavatkozók képesek más-más fázisban és frekvencián működni, ezzel a hozzájuk tartozó érzékelők helyén a rezgést szabályozni, aminek következménye az egész test káros rezgésének csökkenése.

A dolgozatban először bemutatásra kerül az aktív rezgéscsillapító rendszer felépítése, valamint elterjedt rezgéscsillapító algoritmusok működése, és a rendszer identifikációjának folyamata. A megvalósított algoritmus és struktúra bemutatása utáni fejezetben rátérünk a megvalósítás részletes tárgyalására. A mikrokontroller kiválasztását követően, a kapcsolási rajz, valamint a nyomtatott áramköri terv megalkotása következik. A fejezet zárásaként, az implementált program folyamatábrájával, és működésének elemzésével ismerkedünk meg. A dolgozat végén bemutatásra kerül a mérési összeállítás, a vizsgált mechanikus rendszer identifikációja, valamint a rezgéscsillapítás eredménye. Majd levonjuk a következtetéseket, valamint megvizsgáljuk, az esetleges továbbfejlesztési lehetőségeket.

2. Az aktív rezgéscsillapítás [3]

A mechanikai eredetű rezgések csillapítására sokáig kizárólag passzív megoldásokat alkalmaztak. Ezek alapja, hogy a mechanikai rezgést végző egységet rugalmas, rezgéselnyelő anyaggal illesztették a többi, védeni kívánt elemhez. Ebben az esetben nem volt cél, hogy magának a rezgés forrásának az amplitúdóját csökkentsék. Ezek a technikák igen jó hatásfokkal használhatóak ma is, és kedveltek alacsony áruk miatt. (Például autókban, a motor alátámasztásáról rugalmas gumibakok gondoskodnak.)

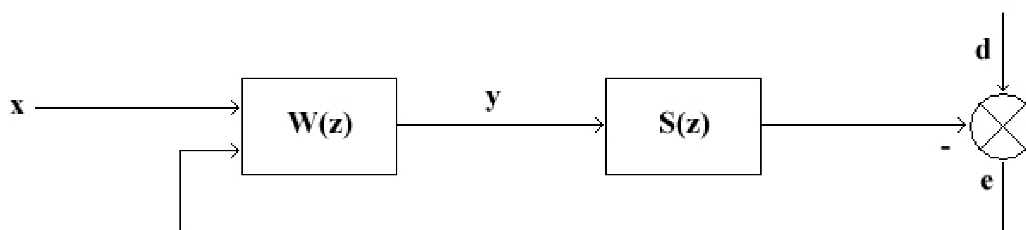
Az aktív rezgéscsillapító rendszerek elmélete régóta ismert, azonban csak néhány évtizede léteznek olyan elektronikus jelfeldolgozó eszközök, amelyekkel valós időben megvalósíthatóak ezek a viszonylag bonyolult algoritmusok. Ezen rendszerek a rezgő test amplitúdóját is nagy mértékben tudják csökkenteni, valamilyen elektronikusan vezérelt beavatkozó felhasználásával, kihasználva a szuperpozíció elvét. A beavatkozó, a káros rezgés frekvenciáján, de ellentétes fázisban működik, így az eredő jel csökkenést mutat. Mivel a mechanikai rendszerek kellően nagy dinamikatartományban lineárisnak tekinthetőek, ezért alkalmazható az előbb említett szuperpozíció elve.

Az aktív rezgéscsillapító rendszer felépítése a *2. ábrán* látható.

2. ábra. A rezgéscsillapító rendszer felépítése

A rendszer működése a következő: A jelfeldolgozó egység a bemenetére kötött szenzor (gyorsulásérzékelő) segítségével tájékozódik a rezgés pillanatnyi amplitúdójáról. Az aktuális érték segítségével a rezgéscsillapító algoritmus meghatározza az ellenrezgés szükséges mértékét.

A kiadott beavatkozó jelet, a shaker segítségével alakíthatjuk újra mechanikus jellé. A megvalósított rendszer nagyon jó rezgéselnyomásra képes, ám rendelkezik hátrányos tulajdonságokkal is. A legjelentősebb ilyen tulajdonsága, hogy mivel az aktív rezgéscsillapító rendszerek visszacsatolt struktúrák, ezért érzékenyek a visszacsatoló ág, az úgynevezett másodlagos út megváltozására. A másodlagos utat a gyorsulásérzékelő, az AD, maga a jelfeldolgozó egység, a DA, a shaker és a beavatkozó–rendszer-érzékelő között lévő mechanikus út határozza meg. Lomhaságából adódóan, leginkább az utóbbi járul hozzá a függvény megváltozásához.



3. ábra. A rezgéscsillapító rendszer szabályozási köre

A *3. ábrán* látható a rezgéscsillapító rendszer visszacsatolt szabályozási körének hatásvázlata.

A jelölések magyarázata:

W(z) : rezgéscsillapító szűrő
S(z) : másodlagos utat reprezentáló átviteli függvény
x : referenciajel
y : a rezgéscsillapító algoritmus kimenete
d : a rezgés elnyomandó komponense
e : hibajel (a maradó rezgés)

A *3. ábrán* látható különbségképzés a valóságban összeadásként megy végbe (szuperpozíció). A jelölés magyarázata, hogy ez a megszokott ábrázolási módja a visszacsatolt szabályozási köröknek. A megvalósítás során, az algoritmus nem y-t, hanem annak *-1* szeresét állítja elő. A szabályzandó mechanikai rendszer átviteli függvénye nagy fokszámú lehet, ezért a meglévő szabályozástechnikai eszközökkel igen nehéz feladat a szabályozó megtervezése. A gyakorlatban, ezért a csillapítani kívánt rendszert modellezik, és adaptív algoritmusok segítségével valósítják meg a rezgéscsökkentést.
A modellezés során elsősorban a másodlagos út viselkedéséről szeretnénk információkat gyűjteni, míg az adaptáció feladata a periodikus rezgés valósidejű elnyomása.

2.1. A modellillesztés feladata [3],[4]

A modellillesztés feladata egy ismeretlen rendszernek megfelelő modell strukturális meghatározása, illetve ezen struktúra paramétereinek megadása. A paraméterbecslés során a valóságos rendszer, és a hozzáillesztett modell kimeneteinek eltérését kívánjuk minimalizálni adott hibakritérium szerint. A modellillesztés, ezen belül két csoportra bontható. Az identifikáció célja, az időben állandónak feltételezett rendszer paramétereinek meghatározása, míg az adaptáció feladata, az időben változó rendszerparaméterek követése. Az identifikáció során nagy pontosságú mérések elvégzésére van lehetőség, ugyanis a rendszert hosszú ideig megfigyelve nagy mennyiségű adat gyűjthető. Az adaptáció során kevésbé fontos a nagyfokú pontosság, és a modellnek a valóságos rendszerhez való pontos illeszkedése. Sokkal inkább fontos, hogy a modell paramétereinek változtatásával, a fizikai rendszer változásait követni tudjuk, és a modell a rendszerrel valós időben együtt tudjon mozogni. Természetesen a fizikai rendszer struktúrája nem minden esetben ismert, és paraméterei sem mindig mérhetők, kizárólag a fizikai rendszer, és a modell kimeneteinek eltérése adhat segítséget a modell paraméterereinek korrigálásához. A modellillesztés egy speciális esete a regressziószámítás feladata, amelynek célja bizonyos változók között fennálló determinisztikus kapcsolat meghatározása. Ezen változók legtöbbször bizonyos rendszerek be- és kimenetei. A valóságos rendszereknél, azonban a bemenetek és a kimenetek között nem csak determinisztikus kapcsolat van, hanem sztochasztikus is,

mivel a kimenet általában zajjal terhelt. A létrehozott modell viszont, csak determinisztikus függvénykapcsolatot valósít meg. A paraméterek korrigálásához egy olyan költségfüggvényt definiálunk, amely a valóságos és modellezett rendszer kimeneteinek különbségétől függ. A cél, hogy a paramétereket úgy módosítsuk, hogy ennek a függvénynek minimális legyen az értéke. A költségfüggvény egy előnyös megválasztása az, ha értéke a kimenetek különbségétől, vagyis a modell hibájától négyzetesen függ, ugyanis, az ilyen költségfüggvény minimum-keresése matematikailag egyszerű, és emellett fizikailag a hibajel teljesítményének minimalizálását végezzük. A hibafüggvény a paraméterek síkja fölötti felület, melynek minimum-keresése után azonnal kiadódik a paraméterek optimális értéke. A modellnek, tehát lesz egy dinamikus, esetlegesen nemlineáris és egy adaptálható része, melyeket célszerű különválasztani egymástól. Az illesztendő modell bemenete, kimenete és paraméterei közötti összefüggés a következő:

$$\hat{y} = \hat{g}(\mathbf{w}, u) = \mathbf{w}^T f(u) = \mathbf{w}^T \mathbf{x}.$$

A jelölések magyarázata:

u : a bemenőjel vektor
$\mathbf{w}$: az állítható paraméterek vektora
$\hat{y}$: a modell kimenete
$\mathbf{x}$: $f(u)$ a modell rögzített része, amely az u bemenőjel vektorból előállítja az $\mathbf{x}$ regressziós vektort, amely ezután az adaptálható rész bemenetét képzi.

A modellillesztéshez használt kritériumfüggvény az átlagos négyzetes hiba alapján:

$$\varepsilon(n) = \mathrm{E}\{ (y(n) - \hat{y}(n))^2 \} = \mathrm{E}\{ y^2(n) \} - 2\mathbf{p}^T\mathbf{w} + \mathbf{w}^T\mathbf{R}\mathbf{w},$$

ahol $\mathbf{p}^T$ a kimenet és a regressziós vektor közötti keresztkorrelációs vektor, illetve **R,** a regressziós vektor autokorrelációs mátrixa. A kritériumfüggvény minimuma, a **w** vektor szerinti differenciálással a következőnek adódik:

$$\partial\varepsilon/\partial\mathbf{w} = -2\mathbf{p} + 2\mathbf{R}\mathbf{w} = 2(\mathbf{R}\mathbf{w} - \mathbf{p}) = 0,$$

melynek megoldása adja az optimum helyét a **w** szűrőegyütthatók terében:

$$\mathbf{w}_{opt} = \mathbf{R}^{-1}\,\mathbf{p}$$

A korrelációs vektorok ismeretének hiányában, illetve azok részleges ismerete esetén a paraméterbeállítás csak iteratív módon végezhető el. Ekkor a statisztikai paraméterek helyett a jelek pillanatnyi értékeinek felhasználásával keressük az optimális paraméterkészletet. Az iteráció egyes lépéseiben meg kell határozni, hogy az adott pontban mennyi a hibafelület gradiense, és a következő lépés paraméterkészletét ennek ismeretében kell meghatározni.

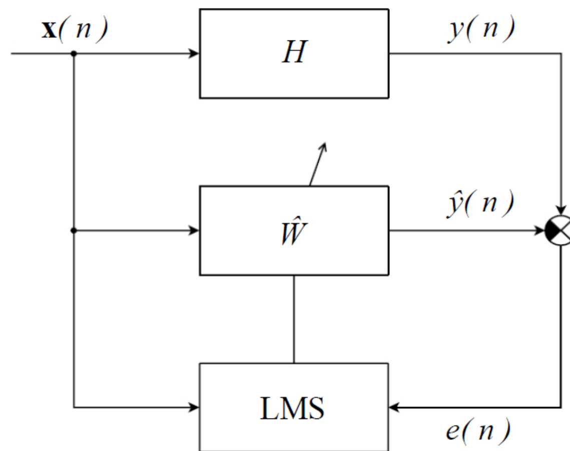
2.2. Az LMS algoritmus [4]

A statisztikai jellemzők szükségességét kiküszöbölhetjük, valamint a bonyolult számításokat is elkerülhetjük, ha a költségfüggvényben az átlagos hiba helyett, csak a pillanatnyi hibát vesszük figyelembe, és a legmeredekebb lejtő módszerével módosítjuk a paramétereket. Ezen az elven működik az LMS algoritmus. A pillanatnyi hibán alapuló becslés miatt a paraméterbeállítás módosítása pontatlan lesz, nem feltétlenül a negatív gradiens irányába történik. Hosszabb időintervallumban, azonban az egyes módosítások pontatlansága kiátlagolódik. A módszer alkalmazásával, viszont a modell a becsült rendszer parametrikus változásait jobban képes követni. Az LMS algoritmus sajátossága, hogy a hibaminimum körül a pillanatnyi hiba kicsi, a hibafelület gradiense nulla körül van. Ezért a paraméterkészlet soha nem állandósul a minimális értéken, azt ugyanis nem is éri el, hanem annak egy kis környezetében lépésről-lépésre változik. Az LMS algoritmusban, ezek alapján, tehát a paramétertérben történő lépések nagyságát (a *bátorsági tényezőt)* célszerű kicsire választani. Ezzel azonban az algoritmus beállási tulajdonságai romlanak. Az LMS algoritmus egy lehetséges blokkdiagramja látható a *4. ábrán.*

Az LMS algoritmusban kiszámításra kerülő rekurzív egyenletek a következők:

$$e(n) = y(n) - \hat{\mathbf{w}}^{\mathrm{T}}(n)\mathbf{x}(n)$$
$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + 2\mu e(n)\mathbf{x}(n)$$

Az LMS algoritmus által használt modell, melynek paraméterei hangolásra kerülnek, egy véges impulzusválaszú *(FIR)* szűrő, melynek impulzusválasza a **w** vektor.



4. ábra. Az LMS algoritmus blokkvázlata

2.3. Az NLMS algoritmus [4]

Ha az LMS algoritmus stabilitási tulajdonságait jobbra, beállási idejét gyorsabbra kívánjuk beállítani egyszerre, akkor nyilvánvalóan a két szempont ellentétes irányban módosítaná a bátorsági tényezőt. A problémára, egy lehetséges megoldást kínál az NLMS algoritmus. Az NLMS, a bátorsági tényezőt a bemeneti jel alapján normálja, így jobb stabilitási tulajdonságokkal, és gyorsabb beállással rendelkezik.

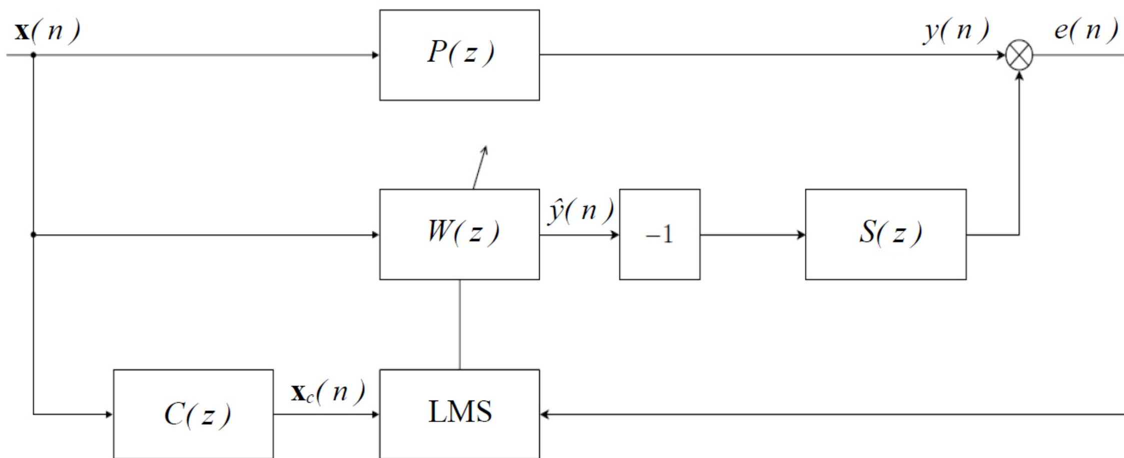
Az NLMS algoritmus rekurzív összefüggései a következők:

$$e(n) = y(n) - \hat{\mathbf{w}}^{\mathrm{T}}(n)\mathbf{x}(n)$$
$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \mu e(n)\mathbf{x}(n) / (a + \mathbf{x}^{T}\mathbf{x})$$

Az *a regularizációs* konstans funkciója az, hogy kis bemeneti érték esetén se válhasson a tört nevezője nullához közelivé, ezzel a lépésköz túl nagy méretűvé.

2.4. Az FxLMS algoritmus [4]

Egyes gyakorlati alkalmazásokban, mint például a rezgéscsillapítás esetében is az LMS algoritmus segítségével adaptált paraméterkészletű szűrő kimenetére egy másik, nem egységnyi átvitelű tag is kapcsolódik. Ezekben az esetekben nem lehet már feltételezni, hogy az adaptív szűrő kimenete az identifikált rendszer kimenetéből azonnal kivonható, és az így előállított különbségi jel azonnal felhasználható az adaptív algoritmusban az adaptív szűrő hangolására. A blokkvázlatot tekintve, az adaptív szűrő kimenete és a különbségképző blokk közé egy átvitel iktatódik, melynek legnagyobb problémája az, hogy a frekvenciatartománybeli szűrésen kívül fázistolást, valamint késleltetést realizál az adaptív szűrő kimenete és a különbségképző között. Így könnyen instabilitási problémát okozhat, ha a fázistolás miatt nem a negatív gradiens irányába módosítjuk a szűrőegyütthatókat. Az ilyen módon a beavatkozó jel útjába iktatott szűrőt nevezik másodlagos útnak. A fejezet elején már esett arról szó, hogy milyen elemek alkotják egy tényleges rendszerben a másodlagos utat. (Esetünkben: gyorsulásérzékelő, AD, jelfeldolgozó, DA, shaker, mechanikai rendszer.) Az úgynevezett másodlagos átvitel hatása akkor eliminálódik, ha az adaptív szűrő az identifikált rendszer mellett a másodlagos átvitel inverzét is modellezi. A gyakorlatban alkalmazható megoldást erre a problémára az FxLMS algoritmus által megvalósított struktúra kínál. E megközelítés szerint a másodlagos átvitel becslőjét kell elhelyezni az LMS algoritmus bemenetén, oly módon, hogy az a referenciajelet szűrje, de az adaptálandó szűrő bemenetére ne legyen hatással. Az *5. ábra* egy ilyen elrendezést mutat.



5. ábra. Az FxLMS algoritmus blokkdiagramja

Az ábrából látható, hogy az LMS algoritmus nem közvetlenül az $\mathbf{x}(n)$ referenciajelet látja, hanem annak $C(z)$ által szűrt változatát. A $C(z)$ olyan *FIR* típusú szűrő, amely az $S(z)$ másodlagos út becslője.

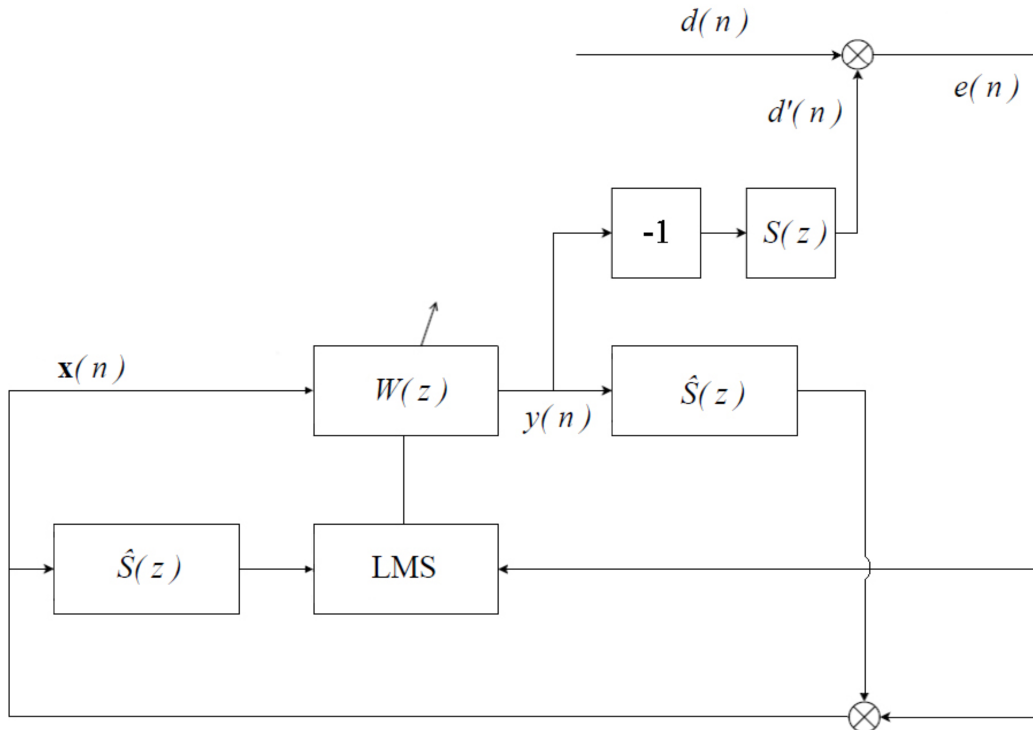
A szűrőegyütthatók adaptálására alkalmazható rekurzív egyenlet:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \mu e(n)\mathbf{x}_c(n)$$

2.5. A választott struktúra, és algoritmus bemutatása [3]

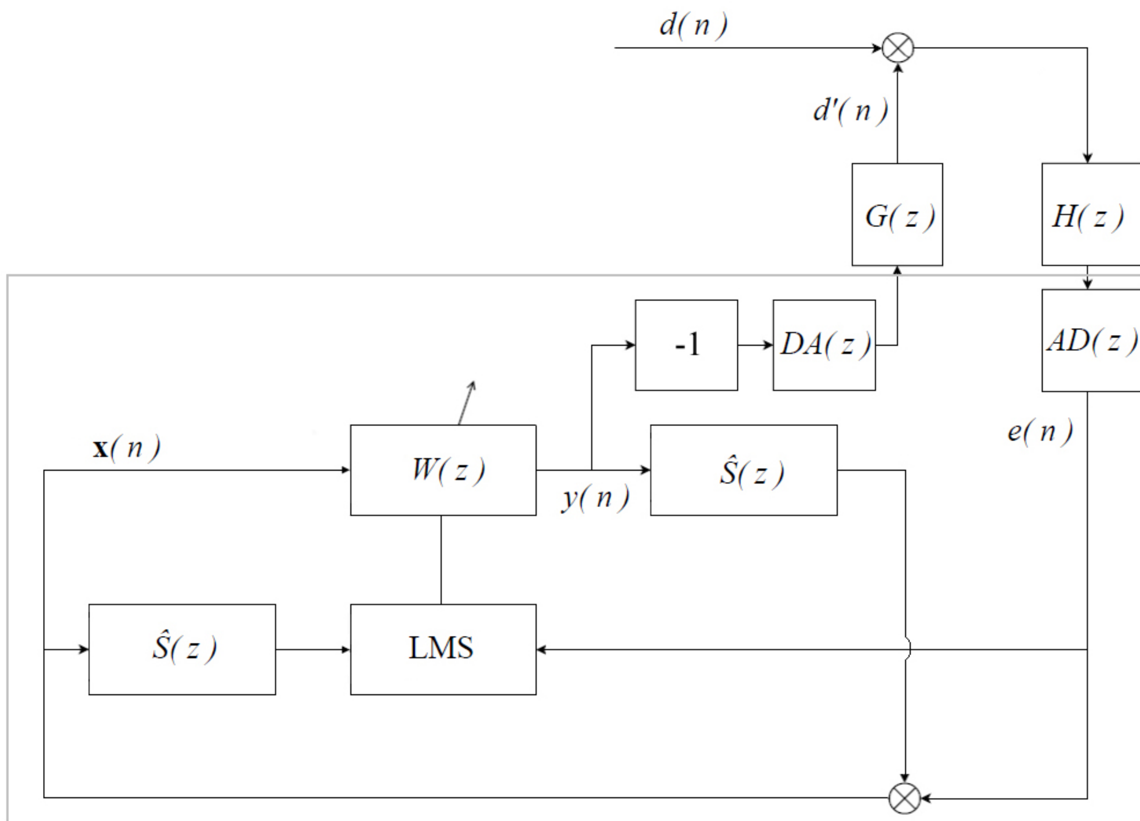
Eddig olyan struktúráról esett csak szó, amelynél felhasználtuk azt, hogy rendelkezünk egy referenciajellel a kioltani kívánt rezgésről. Az ilyen felépítésű rezgéscsillapító rendszereket előrecsatolt rezgéscsillapító rendszereknek nevezzük. Abban az esetben, ha a csillapítandó rendszerben csak a hibajelet tudjuk megfigyelni, akkor az előrecsatolt struktúra nem alkalmazható. A probléma megoldására a rezgéscsillapítás és zajcsökkentés területén egyaránt használt felépítés a visszacsatolt struktúra.

Ebben a modellben a beavatkozószerv irányítójelét, a rezgéscsillapítás célhelyéről származó információkból állítjuk elő. A visszacsatolt struktúra felépítése követhető nyomon a *6. ábrán.*



6. ábra. A visszacsatolt struktúra

A *d(n)* zavarjelet a *d'(n)* beavatkozó jellel kell összeadnunk, hogy kioltsuk a zavaró rezgést. Amennyiben a *6. ábráról* gondolatban lehagyjuk az $\hat{S}(z)$ átvitel, melynek a *y(n)* a bemenete, akkor az **x***(n)* jel az *e(n)* hibajellel egyezik meg, tehát a gyorsulásérzékelő jele, egy az egyben visszacsatolódik a *W(z)* szűrő bemenetére. A közvetlen visszacsatolás a másodlagos út frekvenciafüggő átvitele, valamint késleltetése miatt instabillá teheti a rendszert. Ennek elkerülése érdekében van szükség az *S(z)* becslőjére, az $\hat{S}(z)$ átviteli függvényre. Amennyiben a becslés pontos, úgy **x***(n) = d(n),* tehát a referenciajel megegyezik a kioltandó zajjal. A struktúra ezek alapján, ugyanúgy viselkedik, mint az előrecsatolt változat. A megvalósított rendszer blokkdiagramja látható a *7. ábrán.*



7. ábra. A megvalósított struktúra

Az ábrán jól végigkövethető az *S(z)* átvitel felépítése. A jelölések hasonlóak az eddigiekhez. A *DA(z)* és *AD(z)* blokkok a DA és AD konverterek frekvenciafüggését, késleltetését, valamint az esetleges nemlinearitásokat jelöli. *G(z)* a shaker átvitelét testesíti meg, mivel mechanikus beavatkozóról van szó, ezért leginkább késleltetéssel és frekvenciafüggéssel rendelkezik. *H(z)* a rezgő rendszer, valamint e rendszer és a gyorsulásérzékelő mechanikus viselkedésének átvitele. *d'(n)* a beavatkozó által kiadott, míg *d(n)* a kioltani kívánt mechanikus jel. A két jel szuperpozíciójaként előálló hibajel az algoritmus vezérlője. A téglalap által körülhatárolt rész képzi a modul jelkonverziós és jelfeldolgozó egységének felépítését.

2.5.1. Matlab szimulációk

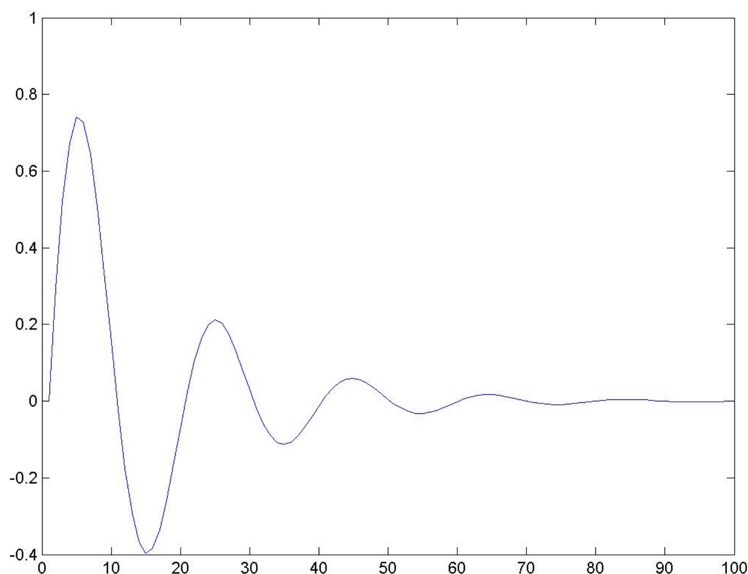
A *2.5.* fejezetben a *7. ábrán* felvázolt algoritmus megvalósítását a program felépítését megfelelően át kell gondolni. A rendszert instabillá teheti, ha nem megfelelő minták összegét számoljuk, vagy nem az aktuális hiba alapján végezzük a koefficiensek tömbjének hangolását. Az is problémát jelenthet, ha nem a megfelelő időben adjuk ki a beavatkozó jelet, vagy nem a megfelelő pillanatba olvassuk vissza a hibát. Ugyanilyen fontos, hogy az egyes futások során, minél pontosabban tartsuk a mintavételi frekvenciát.

A fenti megfontolások alapján, egy lehetséges program felépítés az alábbi:

1. *lépés:* A ciklus elején a mintavétel időzítéséért felelős időzítő nullázása.
2. *lépés:* Az adaptációs szűrés elvégzése az előző ciklusban frissült minták alapján.
3. *lépés:* A 2. lépésben keletkezett beavatkozó jel -1 szeresének a shakerre való kivezetése.
4. *lépés:* Az adaptációs szűrő kimeneti értékével frissítjük az $\hat{S}(z)$ mintatároló tömbjét.
5. *lépés:* A $\hat{S}(z)$ szűrés elvégzése.
6. *lépés:* A gyorsulásérzékelő jelének beolvasása.
7. *lépés:* Az 5. lépésben keletkezett kimeneti érték és a gyorsulásérzékelő mintáinak összeadása.
8. *lépés:* Az adaptációs és $\hat{S}(z)$ szűrők mintatároló tömbjeinek frissítése az előző lépésben keletkezett értékkel.
9. *lépés:* A második $\hat{S}(z)$ szűrés elvégzése.
10. *lépés:* Az előző lépésben keletkezett értékkel frissítjük az LMS algoritmus mintatároló tömbjét.
11. *lépés:* Az LMS algoritmus kiszámítja a következő futáskor érvényes adaptációs koefficienseket.
12. *lépés:* Várakozunk addig, amíg a mintavételért felelős időzítő előre meghatározott értéket (a periódusidőt) el nem éri.

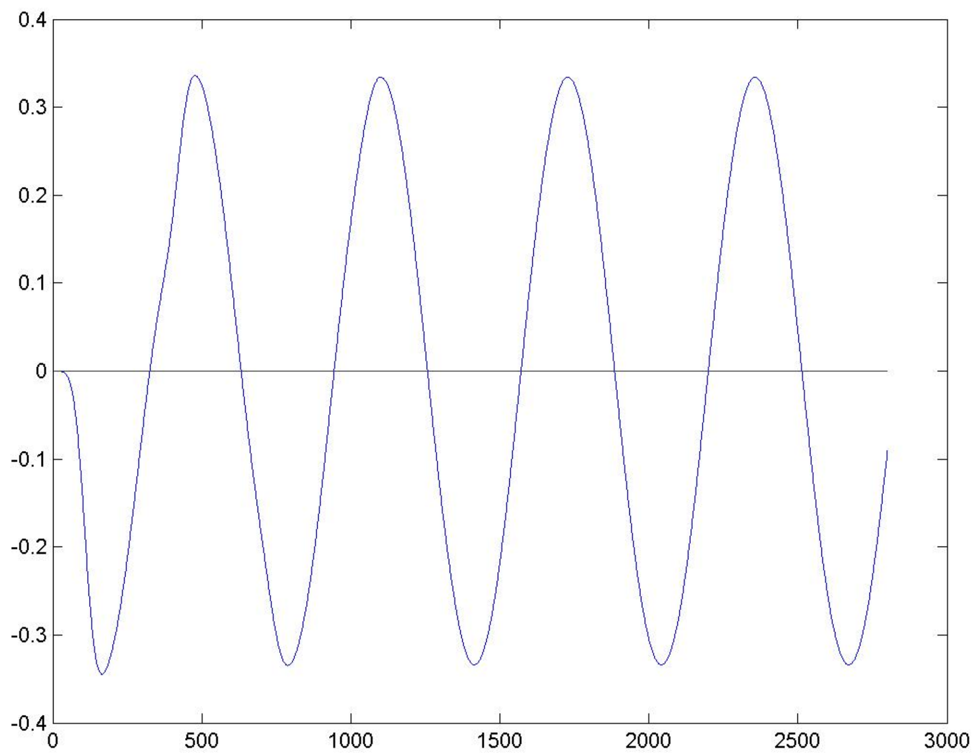
A fenti programmodell tesztelését *Matlab* segítségével végeztem el. 99 tap-es adaptációs, valamint szintén 99 tap-es *Ŝ(z)* szűrőt használtam, és a rendszert első néhány ezer kimeneti, valamint hiba jelét vizsgáltam meg. A szimulációk során egy fiktív rendszert identifikációt használtam, amely a következő paranccsal adódott, és a 8. *ábrán* tekinthető meg:

```
>> x_s=0:1/pi:10*pi;
>> S_Z=sin(x_s).*exp(-0.2*x_s);
```

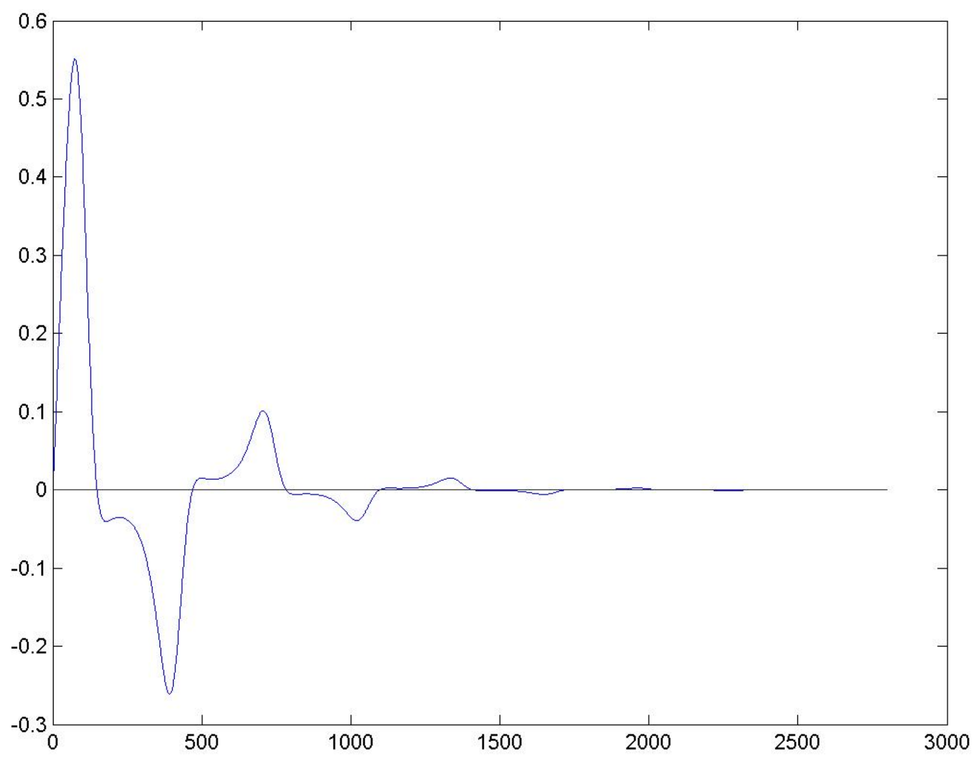


8. ábra. A fiktív rendszer impulzus-válasza

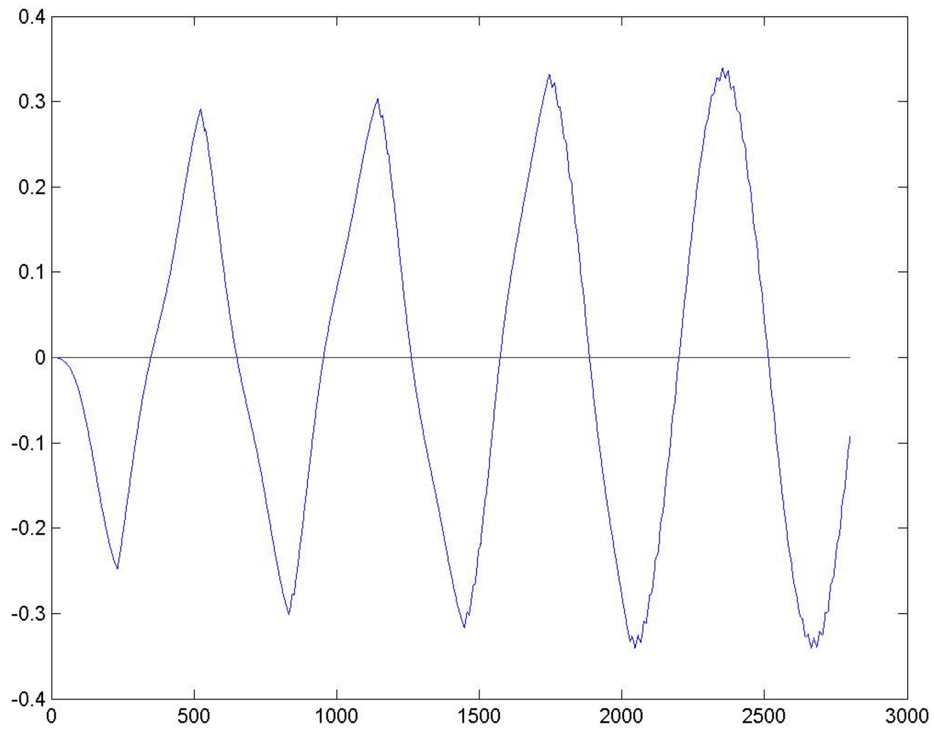
A másodlagos utat modellező impulzus-válasz tehát, egy exponenciális csillapítású szinusz jel, amely a 80. értéke körül már nulla közeli. A szimuláció során kétféle koefficiens számítási módszert vizsgáltam meg. Az első az LMS, amelynél a bátorsági tényező értéke a 99db együttható mellett 10^{-4} -re adódott. A másik módszernél nem a hiba és a bemeneti minta alapján előálló értékkel, hanem egy fix lépésközzel növeljük, vagy csökkentjük az együttható értékét. Az utóbbi a mikrokontrolleren nagy valószínűséggel gyorsabban fut, illetve adott flash méret és mintavételi frekvencia mellett nagyobb együttható számot tesz lehetővé. A második szimuláció során a lépésköz $2{,}5 \times 10^{-5}$ volt. Az LMS algoritmus kimeneti jele a 9. *ábrán*, hibája a *10. ábrán*, míg a második módszer kimeneti jele a *11. ábrán*, a hibája pedig a *12. ábrán* tekinthető meg. A *Matlab* forráskódja a függelékben található.



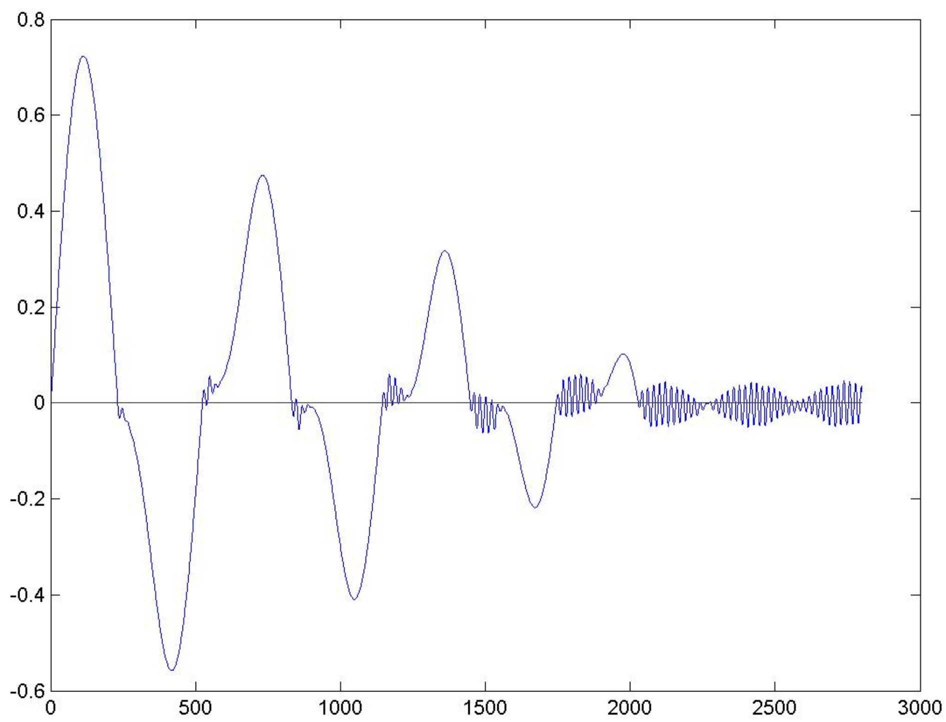
9. ábra. Az LMS algoritmus kimeneti jele



10. ábra. Az LMS algoritmus hibája



11. ábra. A fix lépésközzel működő algoritmus kimeneti jele



12. ábra. A fix lépésközzel működő algoritmus hibája

Mind a kimeneti jelek, mind a hibák alapján a különbség a két számítási mód között látható. A változtatható lépésköznek köszönhetően az LMS algoritmus sokkal gyorsabban beáll, és kisebb maradó hibával rendelkezik, mint a fix lépésközzel működő társa. A fix lépésköz miatt a második szimuláció kimeneti jele hosszabb idő alatt állt be, és a hiba nagyobb értéken minimalizálódott. A második módszerrel, tehát nagyjából 28 dB-es elnyomás valósítható csak meg ezekkel a paraméterekkel, azonban az elsőhöz képest gyorsabb futást tesz lehetővé. A szimulációk során az algoritmusokban használt bátorsági tényező, illetve lépésköz, úgy lett beállítva, hogy a minták 80 %-a körül, már minimális hibával rendelkezzenek. Azt, hogy az LMS, vagy az úgynevezett hiba-előjeles számítási módot fogja megvalósítani a mikrokontroller, az a későbbiekben még további vizsgálódás tárgyát képzi.

3. Az aktív rezgéscsökkentő modul megvalósítása

Ebben a fejezetben bemutatásra kerül az aktív rezgéscsillapító modul rendszerterve, a mikrokontroller kiválasztásának szempontjai, majd ennek az eszköznek a felépítése. A modul elvi kapcsolási rajzának felvázolása után, az elkészült nyomtatott áramköri terv és ennek eredménye kerül ismertetésre. Ezt követően rátérünk a megvalósított identifikációs és rezgés elnyomó program felépítésére, valamint részletes bemutatásra.

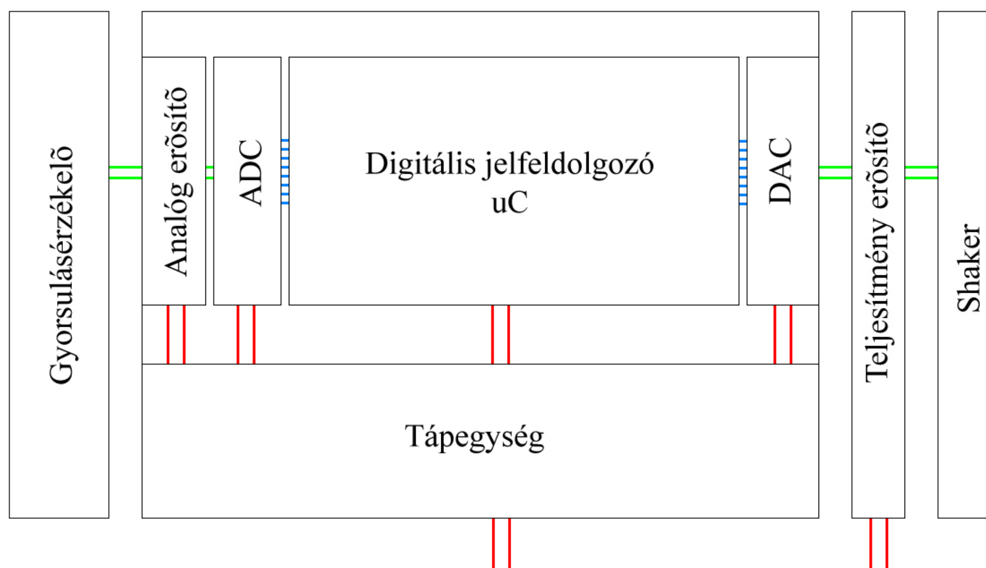
3.1. A modul rendszerterve

A rezgéscsillapító modul tervezésének főbb szempontjai a következők voltak: alacsony fogyasztás, egyszerűség, kis költség, minimális, könnyen beszerezhető, nagy számban gyártott alkatrészek. Kompromisszumot kellett tehát kötni, hogy melyik elemet, egységet érdemes a modulra tervezni, és melyeket nem.

Az integrált gyorsulásérzékelők mai kínálatát áttekintve széles termékpaletta tárul elénk. Létezik analóg kimenetű, de találunk nagyon sokféle kommunikációra képes digitális szenzort is, amelyekben minden szükséges jelkondicionáló elem is megtalálható. Elkülöníthetünk *X, Y, Z* koordinátájú mozgást érzékelő típusokat, de olyat is találhatunk, ami mind három irányban aktív. Az *Analog Devices* oldalán közzétett adatok alapján nagykereskedelmi áruk *4-500 $* körül mozog. A legolcsóbbak, az egy irányban érzékelők, a legdrágábbak, a három aktív iránnyal, és nagy érzékelési tartománnyal rendelkező típusok. A diszkrét, vezetékkel csatlakozó érzékelők néhány kivétellel csak analóg kimenetel rendelkeznek, ezekből is széles választékot találunk az erre szakosodott cégek termékei között. Nagy előnyük, hogy a jelfeldolgozó egységtől távol telepíthető, ezzel a többi elektronikus eszközt megóvva a fellépő vibrációktól. Többségük robusztus fém kialakítású, így az integrált társaikkal ellentétben sokkal megbízhatóbbak, azonban a kedvező tulajdonságaik az árukban érzékelhető, ugyanis *500 $* alatt nem kaphatóak.

A mobilitásuk, robusztusságuk, valamint analóg, szabadon konvertálható kimenetük miatt, és mert a laborban több típus is található, ezért a modul egy külső gyorsulásérzékelőt fog használni. A megfontolások alapján a *Brüel & Kjaer* cég által gyártott *BK4399A* típusú csavarral rögzíthető szenzorra esett a választás. Az analóg kimenetű gyorsulásérzékelőknek felépítésükből adódóan (tartalmaznak aktív erősítőfokozatot) szükségük van tápfeszültségre a működéshez, valamint mivel kis jelszintet produkálnak, ezért még egy erősítő fokozatot is igényelnek. Egy műveleti erősítővel, valamint szűrőfokozattal megépített egyszerű erősítő kis alkatrészigénye és minimális költsége miatt helyet kaphatott a tervezett kártyán. A gyorsulásérzékelőtől származó analóg jel feldolgozásához elengedhetetlen egy AD konverter áramkör. A feldolgozó egységből kijövő digitális beavatkozó jel shakerre való továbbításához szükséges továbbá egy DA átalakító. A DAC-ból kijövő beavatkozó jel teljesítménye kevés ahhoz, hogy közvetlenül képes legyen meghajtani a shaker-t, ezért még egy végerősítőre is szükség van. A teljesítmény erősítő, azonban fogyasztása és helyigénye miatt nem került beépítésre a modulon. A beavatkozó a korábban már említett *Brüel & Kjaer* cég egy másik terméke.

A modul tápellátásának is egyszerűnek kellett lennie, ezért aszimmetrikus tápfeszültség bemenettel rendelkezik, amiből a kártyán található DC-DC konverter IC-k állítják elő a különböző referencia, illetve tápfeszültségeket az aktív elemek számára. A 13. *ábrán* látható az előbbi megfontolások után adódó rendszerterv.



13. ábra. A megvalósított rendszer

A gyorsulásérzékelő jele kívülről van tehát hozzávezetve a modulhoz. Az analóg erősítőnek több feladatot kell ellátnia egyszerre. Első feladata, egyenfeszültséggel ellátni a szenzort, hogy működni tudjon, másrészt jelét szabályozhatóan erősítenie kell, hogy az AD átalakítót mindig optimálisan vezérelje ki. Az AD által feldolgozott gyorsulásérzékelőtől származó jel a digitális jelfeldolgozó bemeneti jele. A beavatkozó jel, pedig a jelfeldolgozón futó algoritmus kimeneti jele, amit a DA konverter alakít újra analóg értékké. A modulhoz csatlakozó teljesítmény erősítő feladata a beavatkozó jel illesztése a shaker-hez. Külső tápellátást maga a modul és a teljesítmény erősítő igényelnek.

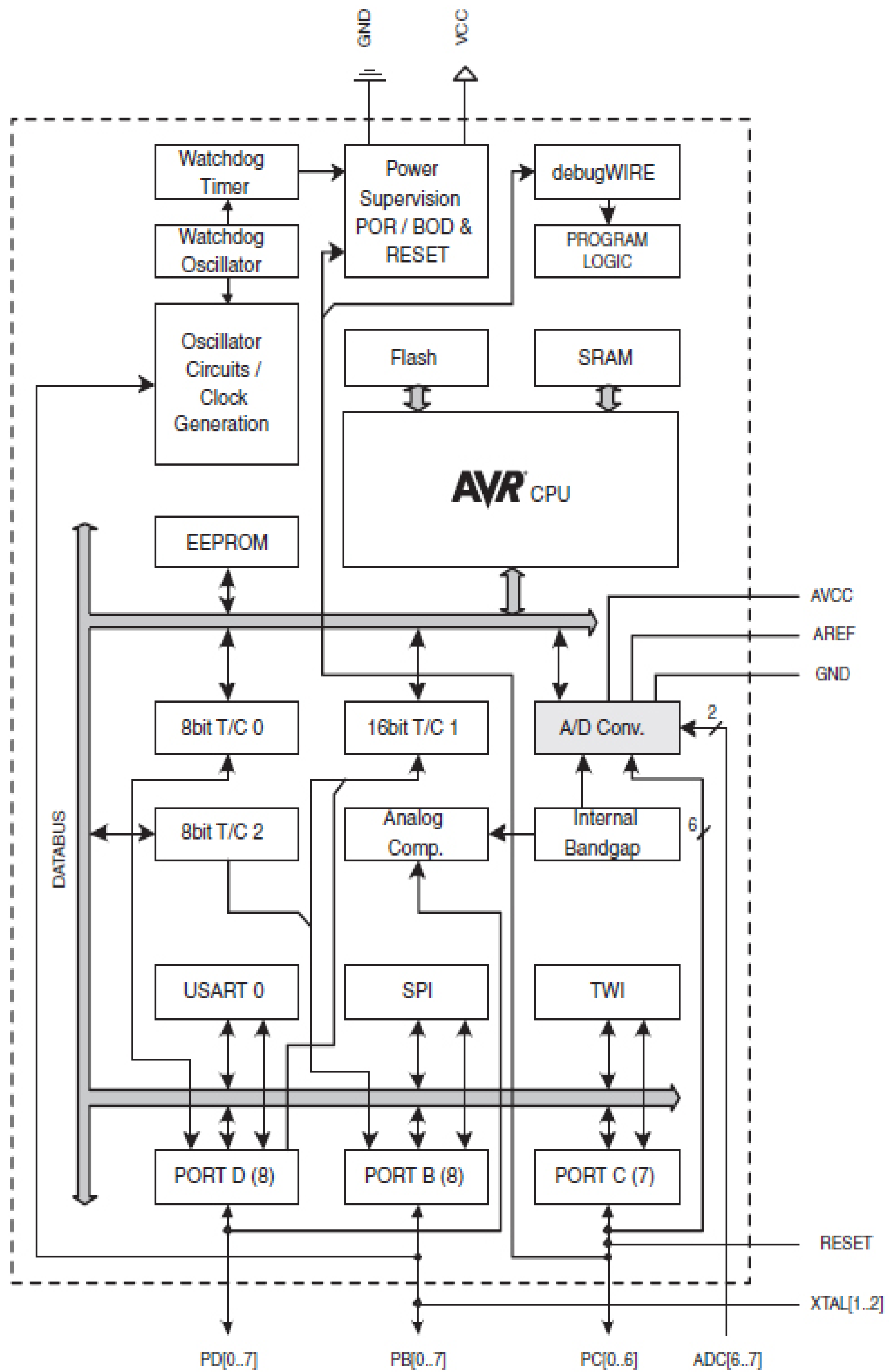
3.2. A mikrokontroller választása [5]

A mikrokontroller kiválasztásakor is alapvetően a rendszertervben leírt szempontok domináltak. Ha a 8-16-32 bites mikrokontroller családokat összehasonlítjuk, akkor két tulajdonságban a mai napig is verhetetlenek az egyik csoport. Egy 8 bites mikrokontrollert egy-két euró körül megkapunk, és emellett nagyon kis fogyasztással rendelkezik. Ugyanakkor, némely típusok rendelkeznek már hardveres szorzó egységgel, ami a fejlesztett algoritmus futásidejét nagyban csökkentheti. Sajnos a 8 bitesekre nem jellemző a DAC, mint beépített periféria, ez inkább a következő családtól van jelen. Viszont a 8-10bites AD átalakító már majdnem mindegyikben megtalálható, ami szükségtelenné teszi egy különálló és költséges ADC áramkör alkalmazását. Mivel az algoritmus feltételezhetően csak párszor 1000 műveletet fog elvégezni két mintavétel között, így szerényebb program, illetve adatmemória is elegendő. A program futási idejének csökkentése céljából, olyan típusra van szükség, ami képes viszonylag magas órajel frekvencián működni. Az elmondottak alapján a választás az *ATMEL* által gyártott *Atmega88-20PU*-as mikrokontrollerre esett.

Az alkalmazás számára fontos tulajdonságai az Atmega88-nak:

- *20 MHz* maximális működési frekvencia
- *7-8 mW* fogyasztás (a maximális frekvencián)
- *8 kbyte* program memória
- *1 kbyte SRAM*
- *512 byte EEPROM*
- 32 db 8 bites általános célú regiszter
- 3 db időzítő egység (2 db 8 bites és 1 db 16 bites)
- 10 bites szukcesszív approximációs analóg-digitális átalakító, amelynek 13-260 us a konverziós ideje
- 23 db programozható IO kivezetés
- kb. 2 euro/db ár

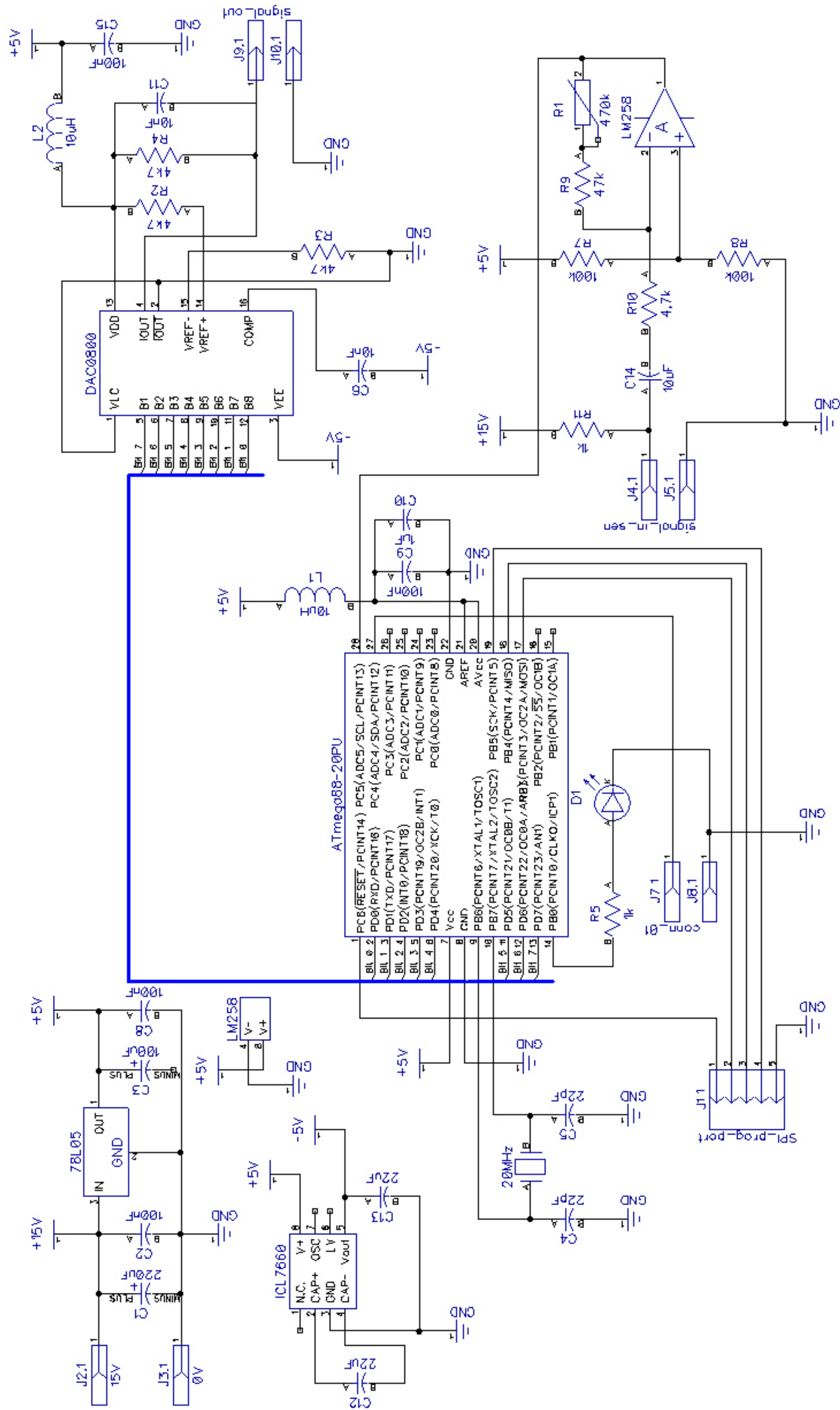
A mikrokontroller belső felépítése a 14. *ábrán* látható. Az áramkör RISC architektúrával rendelkezik, 131 utasítást ismer, melyek között megtalálható a szorzás művelete is. Az alkalmazás számára fontos egységek a következők: Az algoritmus futása során a számításokat végző aritmetikai, logikai egység. A programot, illetve a változókat tároló flash, illetve SRAM. A gyorsulásérzékelőtől származó jel konverzióját végzi majd a belső AD blokk. A másodlagos út identifikálása során a paramétereket szükséges lesz elmenteni, erre a célra a mikrokontroller belső EEPROM egységét fogjuk használni. Az algoritmus ciklikus működéséhez és az adatkonverziók vezérléséhez elengedhetetlen a pontos időzítés, amelyet az egyik belső 16 bites Timer egység hivatott megoldani. Végül a beavatkozó jelet, a DA konverterhez szükséges, a bitszámnak megfelelő mennyiségű általános IO lábakon elvezetni.



14. ábra. Az Atmega88 belső felépítése

3.3. A modul elvi kapcsolási rajza [5],[6],[7],[8]

A kapcsolási rajz elkészítésére a *DipTrace* nevű program ingyenes változatának, *Schematic* beépülő modulját használtam. A fejezet elején felvázolt, majd a mikrokontroller választása alapján körvonalazódott, hogy milyen egyéb áramkörökre van szükség a mikrokontrolleren kívül. Az AD konverzió elvégzésére a mikrokontroller belső konverterét használjuk fel, ezzel szükségtelenné téve a külső AD konverter beépítését. Mivel a mikrokontrollerben található AD konverter unipoláris, ezért hogy szimmetrikusan lehessen kivezérelni, szükség van még egy fél referencia feszültségre történő eltolásra. Tehát a rendszertervben felvázolt analóg jelkondicionáló áramkörnek még egy feladatot el kell látnia. A gyorsulásérzékelő táplálására, jelének erősítésére és az említett eltolásra célszerű egy műveleti erősítős struktúrát használni. Mivel a választott mikrokontroller nem tartalmaz DA konverter, ezért elkerülhetetlen a különálló áramkör alkalmazása. A DA kiválasztásánál elsősorban el kellett dönteni, milyen bitszámú, illetve soros, vagy párhuzamos interfésszel rendelkező legyen. Mivel a műveleteket 8 biten fogjuk végezni, és a kontroller a láb csoportjait is 8 bitbe szervezve kezeli, ezért egy megfelelő választás a 8bites párhuzamos bemenetű DA konverter. A program a futása során ennek a megfontolásnak köszönhetően egyszerre egy port művelettel ki tudja adni a beavatkozó jelet. Amennyiben soros bemenetű DA került volna beépítésre, akkor egymás után kellett volna elküldeni a minta és az esetleges vezérlő biteket, ami a port műveletek sokszorozódásához vezetett volna, ezzel is lassítva az algoritmust. A választott DA, a *DAC0800* típusú áramkör, ami az előbb említett pozitív tulajdonságokkal rendelkezik, és egy olcsó áramkörnek mondható. A megtervezett kapcsolási rajz a *15. ábrán* tekinthető meg.



15. ábra. A modul elvi kapcsolási rajza

A modul +15V-os tápfeszültségről működik, mert a gyorsulásérzékelő belső áramkörének +10 - 12V-os feszültség mellett, 4mA áramra van szüksége a működéshez. A C1-es 220μF-os elektrolit kondenzátor pufferelési feladatot lát el, míg a C2-es 100nF-os kerámiakondenzátor az esetlegesen a tápfeszültségen lévő nagyfrekvenciás zavarokat hivatott csökkenteni.

A +15V-os bemeneti feszültségből a *78L05*-ös szabványos feszültségstabilizáló áramkör állítja elő a kártyán található többi aktív elem számára a +5V-os táp, illetve referencia feszültséget. A stabilizáló áramkör kimenetén lévő C3-as jelű, 100μF-os és C8-as 100nF-os kondenzátorok hasonló feladatot látnak el, mint a C1-C2-es társaik.

A *DAC0800* +5V, -5V-os tápfeszültséget kíván, ezért a legegyszerűbb megoldásnak az kínálkozott, hogy egy feszültség konverter áramkör segítségével állítjuk elő a szükséges -5V-ot. Az *ICL7660* típusú IC ennek a célnak megfelel. 50mA a maximális kimeneti árama, amely bőségesen elég a *DAC0800* által felvett néhány mA számára. Az *ICL7660*-as áramkör két kondenzátort használ a tápfeszültségének invertálására, amelyek a kapcsolási rajzon a C12, C13-as jelzésű elektrolit kondenzátorok.

A gyorsulásérzékelő működési feszültségen történő 4mA-es áramellátásáról, az R11-es 1kΩ-os ellenállás gondoskodik. Mivel a 4mA-es áram hatására ezen az ellenálláson 4V esik, ezért a gyorsulásérzékelő 15V – 4V = 11V-os feszültséget kap a paneltől, ami kielégítő számára.

A C14-ből és R10-ből álló szűrő feladata a gyorsulásérzékelő egyenfeszültségének leválasztása. C14 értéke 10μF, még R10-é 4,7kΩ, tehát a töréspont frekvencia $f = 1 / (R10*C14*2*\pi) = 3{,}39$ Hz. Az alkalmazott műveleti erősítő a kis zajú *LM258*-as, melynek tápellátása aszimmetrikus +5V, hogy túl nagy erősítése beállítsa esetén megakadályozzuk az AD konvertert túlvezérlését. A műveleti erősítő invertáló kapcsolásban működik, a program fog majd a későbbiekben gondoskodni a jel megfordításáról. Az erősítő nem invertáló bemenete az R7-R8-ból álló ellenállásosztó segítségével van a tápfeszültség felére emelve, így kimenetén szimmetrikusan vezérli majd az AD átalakítót. Az erősítő visszacsatoló ágában elhelyezett R9-es ellenállás és R1-es trimmer segítségével valósítható meg a szenzor jelének szabályozható erősítése. Ha R1 a legkisebb értékre van állítva, akkor a fokozat erősítése A = R9/R10 = 47kΩ/4,7kΩ = 10, ha a trimmer maximális értéken van, akkor pedig A = (R9+R1)/R10 = (47kΩ + 470kΩ)/4,7kΩ = 110. Látható tehát, hogy viszonylag széles határok között változtatható a gyorsulásérzékelő jelének erősítése. Az erősítőfokozat kimeneti jele a mikrokontroller 28-as lábára, az 5. analóg csatornára kerül.

A modul tervének középpontjában az *Atmega88-20*-as mikrokontroller áll. A 20 MHz-es kvarc kristály a 9-10-es lábakhoz kapcsolódik. A D port kivezetéseihez van illesztve, a külső DA konverter áramkör. A program könnyebb kezelhetősége végett, a port MSB bitje megegyezik a konverter bemeneti MSB bitjével, tehát a portra minden további módosítás nélkül kiadható az algoritmus által meghatározott beavatkozó jel értéke. Az áramkör 14-es lábára egy LED került, ezzel meg lehet jeleníteni a program működését, és az esetleges hibák javítását is megkönnyíti. A mikrokontroller 27-es lábára csatlakozik egy általános felhasználású ki/bemenet. Ennek a kivezetésnek a célja, hogy vagy az identifikáció során használt referencia jelet bevezesse az áramkörbe, és ezzel egy másik analóg csatornán tudja azt a modul feldolgozni, vagy egy nyomógombot felhelyezve a programot vezérelni lehessen futása közben, vagy esetlegesen egy második LED segítségével a program állapotát lehessen megjeleníteni. A mikrokontroller programozására használt ISP kommunikációs vezetékeket külön csatlakozóval vezetjük el a kártyáról. Az *Atmega88*-as logikai tápellátása közvetlenül a +5V-os táprészhez csatlakozik, míg az analóg chip-rész különálló egysége az L1 induktivitáson keresztül kap feszültséget. Az L1 feladata, a nagyfrekvencián működő mikrokontroller által a tápfeszültségen megjelenő zajok fojtása. A C9-es kerámia, és C10-es tantál anyagú kondenzátorok is segítenek ebben a zavarszűrésben. C10 feladata inkább a pufferelés, még a C9-é a nagyfrekvenciás komponensek földre vezetése. Az AD konverter a számára nélkülözhetetlen referencia feszültséget is ebből az L1 által zavarmentesített részből nyeri.

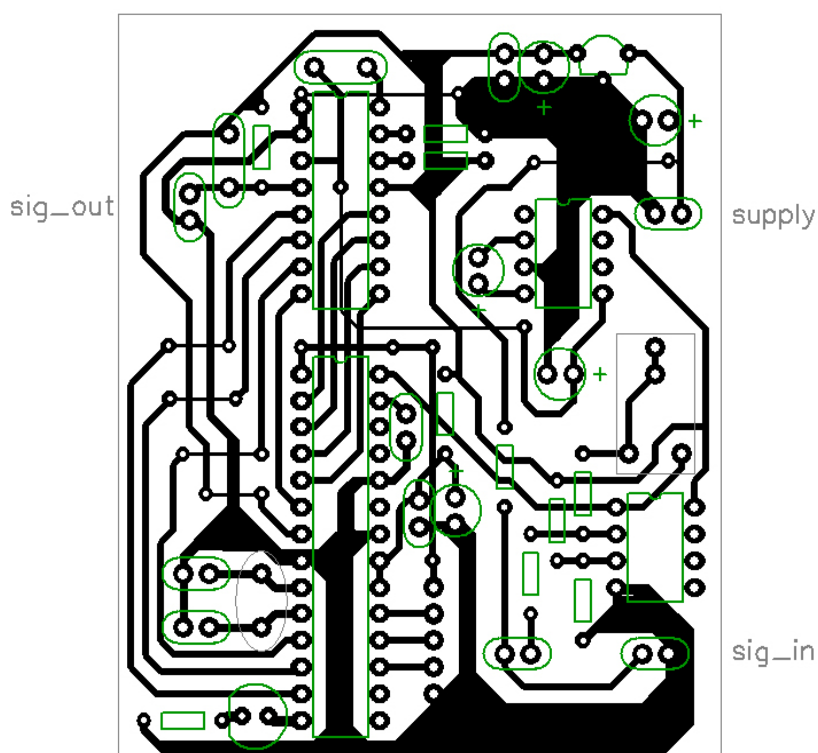
A *DAC0800* a -5V-os feszültségre közvetlenül kapcsolódik, míg a +5V-os részhez, L2-n keresztül és C15-el párhuzamosan a korábban említett nagyfrekvenciás zavarok bejutását megakadályozva. A *DAC0800* referencia feszültségeit a föld, és a +5V segítségével az R2, R3 ellenállás állítja elő. A konverter áram kimenetéből R4-es 4,7kΩ-os ellenállás segítségével nyerjük a beavatkozó feszültség jelét. Az R4, C11-ből álló alul áteresztő szűrő feladata egyrészt a konverter sávszélességének csökkentése (a négyszögjelesítés hatásának csökkentése), valamint az esetlegesen a nem egy időben történő bitváltozások hatására létrejövő, rövid idejű tüskék hatásának eliminálása. A szűrő törésponti frekvenciája $f = 1/(R4*C11*2*\pi) = 1/(4{,}7k\Omega*10nF*2*\pi) = 3386{,}3$ Hz, ami az alkalmazás szempontjából nem jelent csillapítást.

A modul alkatrészigénye:

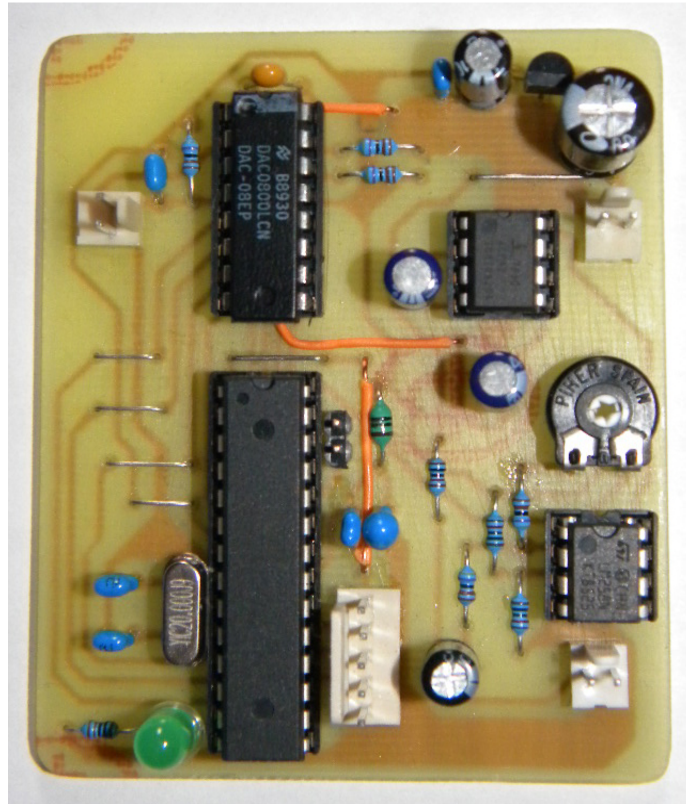
Atmega88-20PU	mikrokontroller	1db
LM258	műveleti erősítő	1db
ICL7660	DC-DC konverter	1db
78L05	+5V stabilizátor	1db
LED	5mm zöld	1db
20MHz	kvarc kristály	1db
22pF/50V	Kerámia	2db
10nF/50V	Kerámia	2db
100nF/50V	Kerámia	4db
1uF/16V	TA ELKO	1db
10uF/25V	AL ELKO	1db
22uF/16V	AL ELKO	2db
100uF/16V	AL ELKO	1db
220uF/25V	AL ELKO	1db
1kΩ	0,75W	2db
4,7kΩ	0,75W	4db
47kΩ	0,75W	1db
100kΩ	0,75W	2db
470kΩ	trimmer	1db
10uH/50mA	induktivitás	2db
2 pólusú NSL	csatlakozó	3db
5 pólusú NSL	csatlakozó	1db

3.4. *A modul nyomtatott áramköri terve [5],[6],[7],[8]*

A modul nyomtatott áramköri tervének elkészítéséhez a korábban már említett *DipTrace* nevű program egy másik, *PCBLayout* modulját használtam. A tervezett áramkör egyoldalas, de szükséges volt a túloldalon 9 db átkötést létrehozni a vezetősávok elhelyezkedése miatt. A kisfrekvenciás jelek és az áramkör relatív kis mérete miatt a vezetékek hosszának jelkésleltetése nem számottevő. Az egyes hidegítő kondenzátorok a cél áramkörhöz a lehető legközelebb lettek elhelyezve. Az elkészült terv négy funkcionálisan jól elhatárolható részre van felosztva. A *DAC0800* és a passzív kiszolgáló áramköri elemei mellett található a modul tápellátásáért felelős *78L05*, *ICL7660*-re épülő blokk. A mikrokontroller, valamint a működéséhez elengedhetetlen elemek mellett található az analóg jelkondicionálást elvégző műveleti erősítős egység. Az integrált áramkörök esetleges cseréjének megkönnyítése végett mindegyikük DIP tokozású, és a lapon foglalatokba lettek elhelyezve. Az elkészült terv a *16. ábrán,* míg az áramkör készre szerelve a *17. ábrán* látható.



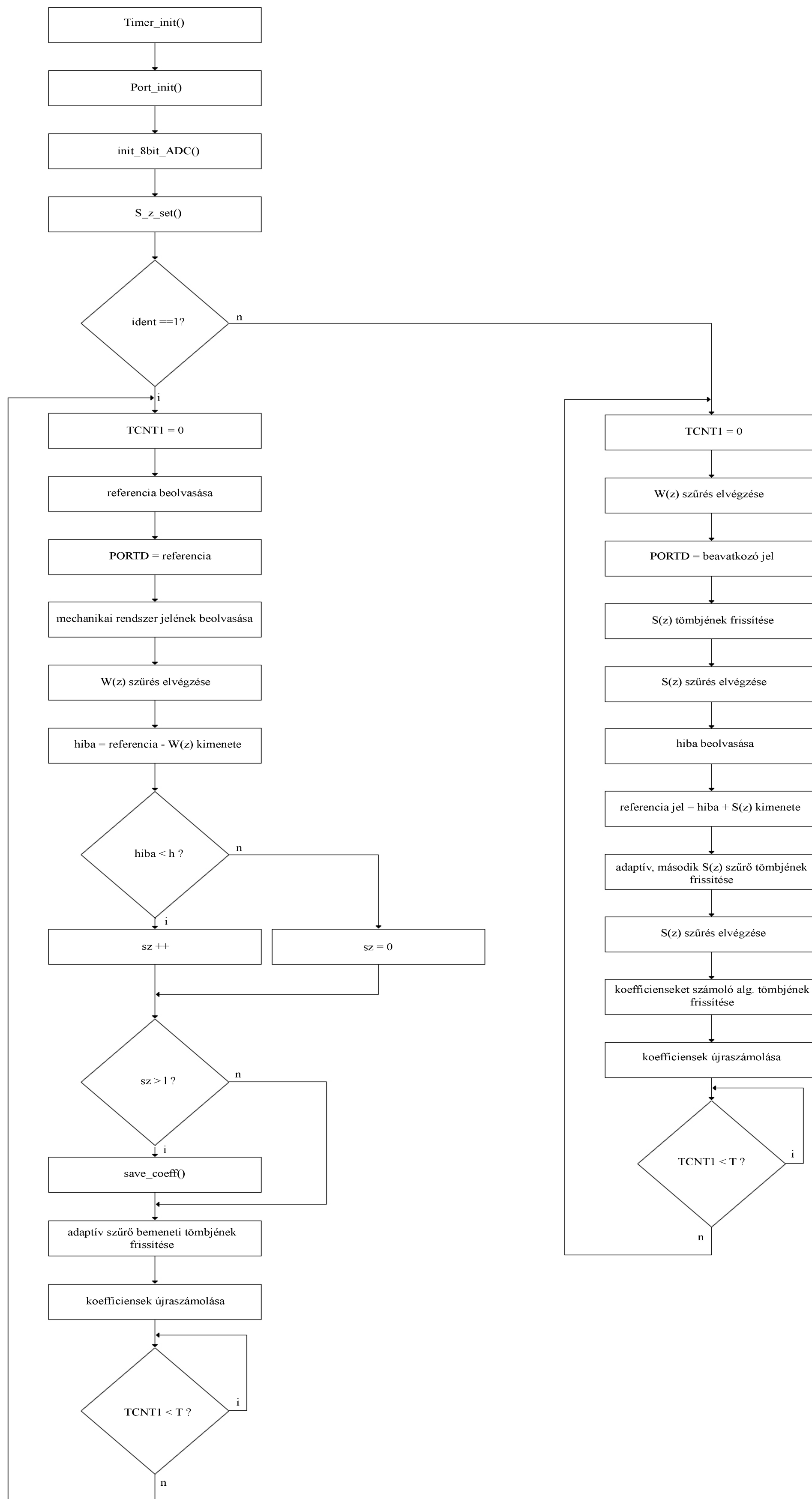
16. ábra. A modul nyomtatott áramköri terve



17. ábra. A készre szerelt rezgéscsillapító modul

3.5. Az implementált program

A 2.5.1 fejezetben bemutatott program vázat módosítani kellett, hogy az adatkonverziókat, időzítéseket és alapvető periféria inicializálási feladatokat is el tudja végezni. A végleges, mikrokontrolleren futó program a függelékben található, míg folyamatábrája a *18. ábrán* látható.



18. ábra. Az implementált program folyamatábrája

A *main* függvény elsőként a legpontosabb 16 bites Timer1 inicializálását végző *Timer_init()* nevű egységet hívja meg, amely a mintavétel időzítését fogja majd végezni. A *Timer_init()* a Timer1 órajelét állítja be a rendszerórajel 1/8 -ára, ezáltal az 20MHz/8 = 2,5MHz frekvencián fog járni, és ezzel várhatóan a mintavétel pontatlansága 0,4µs lesz.

A következő lépésként az általános IO lábak inicializálását megvalósító *port_init()* függvény kerül hívásra. A mikrokontroller digitális kimenetei egyrészt a D porton lévő DA konverter, másrészt a B port 0-s kivezetésén elhelyezkedő LED.

Szükséges még a belső AD konverter felparaméterezése, amelyről az *init_8bitADC()* függvény gondoskodik. Az *init_8bitADC()* először engedélyezi a konvertert, majd beállítja a rendszerórajel 1/32-re való osztását, mellyel a későbbiekben nagyjából 22,5µs konverziós időt tesz majd lehetővé. Utolsó lépésként beállítja referencia feszültségként az erre a célra szolgáló mikrokontroller kivezetést, valamint az eredmény regiszter balra rendezéséről is gondoskodik.

Az *S_z()* függvény az előzetes identifikáció eredményeként adódó koefficiensekkel tölti fel az *S(z)* szűrő együtthatóinak tömbjét.

A program ezután két részre ágazik egy előre beállított változó segítségével, melynek következménye, hogy csak azt a rész fogja a fordító gépi kóddá alakítani, amelyik aktív, a másikat eldobja ezzel is csökkentve a program méretét.

Amennyiben identifikációs módban van a program, úgy az alábbiakban jár el. Elsőként a Timer1-et nullázza, így mérve az adott ciklus periódusidejét a későbbiekben. A mikrokontroller 4-es lábára adott, analóg referenciajelet *x_ref* változóba olvassa a *read_8bitADC()* függvény, amely a korábban beállított értékek segítségével egy adott csatornán képes egy konverziót lefolytatni. A következő lépésben a D portra kiadjuk a korábban beolvasott referencia jelet, majd az 5-ös analóg csatornáról visszaolvassuk a rendszer válaszát. Az adaptációs szűrés elvégzését követően meghatározzuk a hibát a rendszer válasza és a modellezett jel különbsége alapján. Amennyiben a hiba egy előre beállított *h* szint alá csökken, akkor megnöveljük az *sz* nevű változót, amely a stabil működés hosszát számolja, amennyiben ez az érték elér egy szintén előre beállított konstanst, akkor az identifikáció eredményét elfogadja, és a koefficienseket eltároljuk a *save_coeff()* függvény segítségével, amely az EEPROM-ba menti sorban az egyes együtthatókat. Azonban, ha a hiba még nem minimalizálódott kellőképpen, akkor folytatjuk az adaptációt, amely előtt frissítjük annak bemeneti tömbjét. A ciklus végén megvizsgáljuk a Timer1-et, amely ha kisebb, mint az előre beállított periódusidő, akkor várakozunk, majd a ciklust elölről kezdjük.

Ha rezgéselnyomó módban van a program, akkor szintén a Timer1 nullázásával kezdi a ciklust, majd az adaptációs szűrés elvégzését követően kiadja a számított beavatkozó jelet. A másodlagos utat reprezentáló szűrés elvégzését követően beolvashatóvá válik a hibajel, és e két jel összegeként számítható a referenciajel értéke. A referenciajel aktuális értékével frissítjük az adaptációs és a második *S(z)* szűrő bemeneti mintatároló tömbjét. Az *S(z)* szűrés újbóli elvégzése után a számított kimeneti jel segítségével frissíthető a koefficiensek újraszámolásáról gondoskodó algoritmus mintatároló tömbje. Az együtthatók aktualizálását követően az identifikációs módban leírtak szerint, a Timer1 kiértékelésével zárul a ciklus.

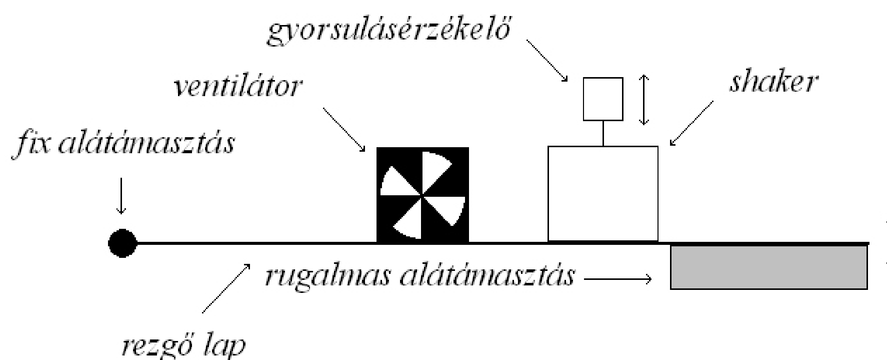
Az adaptációs algoritmus, valamint a szűrők működése során a mintatároló tömbök elemeire való hivatkozás nem indirekt, nem mutatók segítségével történik, hanem direkt módon, ezzel is csökkentve a program méretét, valamint a ciklus is gyorsabb a paraméterátadások kihagyása miatt.

A mikrokontroller 8 kbyte-os adatmemóriája 38 együtthatós másodlagos utat modellező szűrőt, valamint szintén 38 együtthatós adaptív szűrést és hangolást tesz lehetővé. A konkrét alkalmazás során, ha szükséges a nagyobb adaptív szűrőszám, akkor a többi szűrő együtthatóinak csökkentésével a probléma megoldható, és fordítva.

A fenti 38-38 együtthatóval az elnyomó ciklus maximálisan 400 Hz mintavételi frekvenciát tesz lehetővé, ha az alkalmazás kívánja, akkor természetesen ez is lehet kisebb.

4. A mérési eredmények

A méréseket a *19. ábrán* látható rendszeren végeztem. A káros rezgést egy lapon elhelyezett aszimmetrikus forgó résszel rendelkező ventilátor hozta létre. A ventilátor fordulatszámának, a rezgés amplitúdójának változtatásáról egy hálózatról működő, szabályozható kimeneti feszültséggel rendelkező toroid transzformátor gondoskodott. A lap bal oldala egy fix ponthoz volt rögzítve, míg a jobb oldala egy puha anyaggal volt alátámasztva. A ventilátor keltette rezgés következtében a lap jobb oldala függőleges irányú mozgásra volt képes. A shaker a rendszerben ezen az oldalon foglalt helyet, a szenzor pedig ezen az eszközön volt elhelyezve. A shaker feladata, hogy a szenzor minél kisebb függőleges irányú kitéréssel rendelkezzen. Ha megfelelően működik a csillapítás, akkor a gyorsulásérzékelő csak minimális mértékben mozog majd.



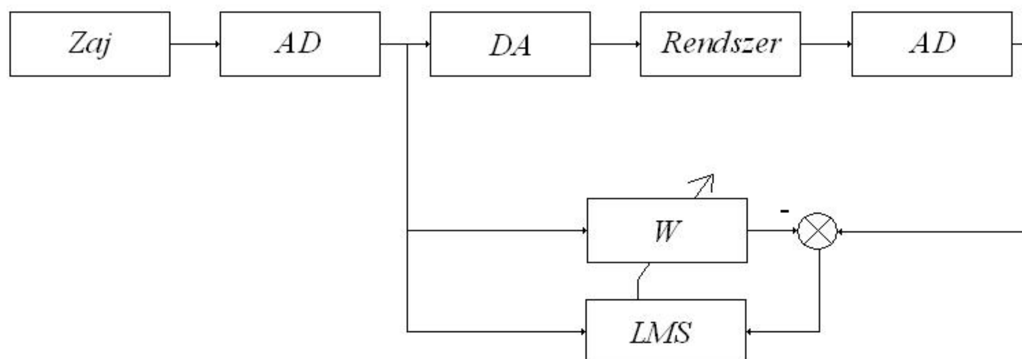
19. ábra. A csillapítani kívánt rendszer felépítése

A mechanikai rendszeren végzett előzetes identifikációk során körvonalazódott, hogy milyen mintavételi frekvencia és másodlagos szűrő hossz szükséges a megfelelő modellezéshez. Amennyiben a mintavételi frekvencia túl nagy, akkor a hosszabb modellező szűrő szükséges, ugyanannak a rendszernek a leírásához. Amennyiben a szűrő hossza adott frekvencián túl rövid, akkor nem megfelelően modellezi a rendszert, ezáltal instabillá téve a teljes struktúrát. Az a feltevés is beigazolódott, amely szerint a hiba-előjeles algoritmus több együtthatót enged meg.

A fenti megfontolások alapján, a mintavételi frekvencia 200 Hz, az adaptációs és *S(z)* szűrök, pedig egyaránt 38 együtthatót tartalmaznak. (A 200 Hz Timer1 szerinti periódusideje 12500.) A teljes elnyomó program memóriaigénye ezek alapján 96 %-a a mikrokontroller belső flash egységének.

4.1. A rendszer identifikációja

A rendszer identifikációjához 50 Hz frekvenciájú sávkorlátozott zajt használtam. A zaj a modul 4-es analóg bemenetére került, a DA konverter kimeneti jel pedig egy teljesítmény erősítő segítségével hajtotta mag a shakert. A gyorsulásérzékelőtől származó jel a modul erre alkalmas analóg jelfeldolgozó egységére lett kötve. Az identifikációs struktúra látható a *20. ábrán.*

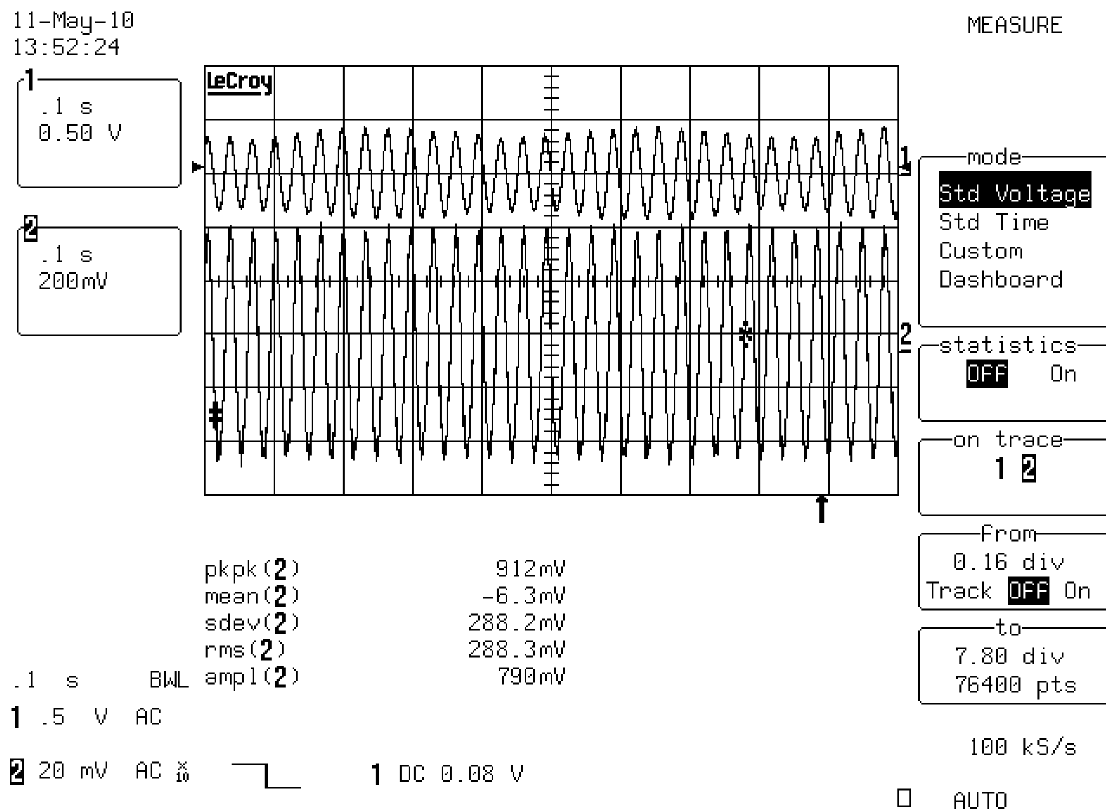


20. ábra. Az identifikációs struktúra

A *20. ábrán* jól látható, hogy nem csak a rendszert modellezi a *W* szűrő, hanem az *AD* és a *DA* átvitelét is.

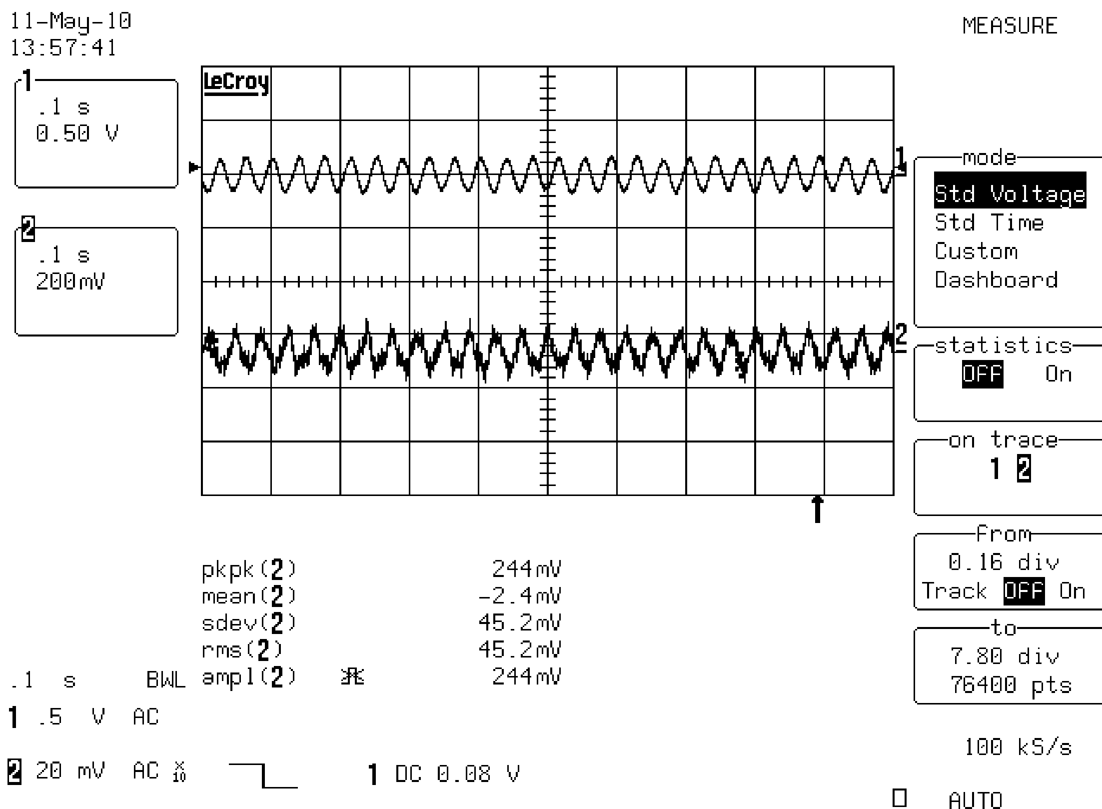
4.2. A rezgéscsillapítás eredményei

Az előző pontban meghatározott másodlagos utat modellező együtthatók segítségével a 3. fejezetben leírt elnyomó algoritmus, az alábbi eredményeket hozta. A korábban bemutatott rezgő rendszer frekvenciája megközelítőleg 30 Hz volt, míg az elnyomó algoritmus nélküli amplitúdója 912 mV. A rendszer csillapítás nélküli viselkedése a *21. ábrán* látható. Az ábrán a gyorsulásérzékelő jele, az oszcilloszkóp 2-es, míg a beavatkozó jel az 1-es csatornán látható.



21. ábra. A rendszer rezgéselnyomás nélkül

A beavatkozó jel az elnyomó algoritmus bekapcsolásával csökkenni kezdett, majd a *22. ábrán* látható értéket vette fel. A káros rezgés amplitúdója is beállt az implementált algoritmus számára elfogadható minimumra. A csillapított jelben látható a *Matlab* szimuláció során megfigyelt, az alaprezgésnél nagyobb frekvenciájú zaj. Az elnyomó algoritmus, tehát a 912 mV-os rezgést 244 mV-ra csillapította, ami 11,5 dB-nek felel meg.



22. ábra. A rendszer az elnyomó algoritmus bekapcsolásával

A *Matlab* szimuláció során, ez az algoritmus 28 dB elnyomást produkált, azonban akkor nem vettük figyelembe a modellillesztés pontatlanságát, valamint a 8 bites egész számok ábrázolása miatti hibát és a fix lépésközt. A rezgéselnyomó modul, azonban jól használható a passzív rezgéscsillapítók kiegészítőjeként.

5. Értékelés, a továbbfejlesztés lehetőségei

A kitűzött cél egy választott mikrokontroller felhasználásával olyan modul tervezése, elkészítése és software fejlesztése volt, amely egy alacsony frekvenciás rezgést végző egység szabályzását képes ellátni. Az *Atmega88-20* típusú mikrokontrollerre épülő modul a mérési eredmények alapján 11,5 dB csillapítást valósított meg egy nagyjából 30 Hz frekvenciájú rezgés esetén, ezzel bebizonyította, hogy ebben a formában a mechanikus rezgéscsillapító megoldások kiegészítője lehet.

Több ilyen modul elkészítésével és a hozzá tartozó érzékelő – beavatkozó segítségével érdemes lenne megvizsgálni, hogyan viselkednek együtt, mint különálló rendszerek kiterjedt, rugalmas anyagból készült rezgő test csillapításában.

Érdemes lenne megvizsgálni továbbá az alkalmazott mikrokontrollernél nagyobb memóriával rendelkező eszközt. Ugyanis a megvalósítás során a mintavételi frekvenciát 200 Hz-re kellett választani, mert a vizsgált mechanikus rendszert csak így lehetett modellezni a programmemória által biztosított 38 együtthatós szűrök segítségével. Nagyjából 12 kbyte memória elegendő lehet 50 együtthatós szűrők implementálására, ezzel valószínűleg tovább növelhető az elnyomás mértéke. Nagyobb kapacitású mikrokontroller segítségével továbbá megvizsgálható a nagyobb, esetleg 16 bites dupla pontos együttható ábrázolás viselkedése, amely szintén javíthat az elnyomás mértékén.

6. Irodalomjegyzék:

[1] Wikipedia, Machining vibrations
http://en.wikipedia.org/wiki/Machining_vibrations

[2] T. Kobori, „Structural Control of Buildings Under Earthquakes And Strong Winds”,Proceedings of the Active '97, The International EAA Symposium on Active Control of Sound and Vibration, Aug. 1997,
Budapest, Hungary, pp. 905-916.

[3] Sujbert László, „Periodikus zavarhatások csökkentésének aktív módszerei”,
BME PhD, 1997.

[4] Sujbert László, Balogh Tibor, „Adaptív szűrők vizsgálata”,
BME mérési segédlet, 2008.

[5] ATMEL, Atmega88-20PU, Atmel Corporation, 2007.
http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf

[6] NATIONAL SEMICONDUCTOR, DAC0800, National Semiconductor, 1999.
http://www.national.com/ds/DA/DAC0800.pdf

[7] MAXIM, ICL7660, Maxim Integrated Products, 1994.
http://datasheets.maxim-ic.com/en/ds/ICL7660-MAX1044.pdf

[8] NATIONAL SEMICONDUCTOR, LM258, National Semiconductor, 2005.
http://www.national.com/ds/LM/LM158.pdf

7. Függelék

7.1. A Matlab szimuláció forráskódja

```
x = -12*pi:0.01:31.3*pi;     % bemeneti minta
X = sin(x);
w = zeros(99);               % adaptív koefficiensek
in = zeros(99);              % bemeneti minták tömbje
s_sys_in = zeros(99);        % mechanikai rendszer mintatároló
s_ident_1 = zeros(99);       % 1. S(z) mintatároló
s_ident_2 = zeros(99);       % 2. S(z) mintatároló
lms_buffer = zeros(99);      % lms mintatároló
output = zeros(2800);        % megfigyelt kimenőjel tároló
error_s = zeros(2800);       % megfigyelt hibajel tároló
x_s=0:1/pi:10*pi;            % mechanikai rendszer identifikáció
S_Z=sin(x_s).*exp(-0.2*x_s);
figure(1);
plot(x_s, S_Z);              % mechanikai rendszer identif. megjelenítése
mu = 1/10000;                % mu értékadása



for t=1:1:2800

    % adaptív szűrése elvégzése

    sum_1 = 0;
    for i_1=1:1:99
        sum_1 = sum_1 + in(i_1)*w(i_1);
    end

    % beavatkozó jel kiadása

    DA = -sum_1;

    % mechanikai rendszer reakciója

    for i_2=99:(-1):2
        s_sys_in(i_2) = s_sys_in(i_2-1);
    end

    s_sys_in(1) = DA;

    sum_2 = 0;
    for i_3=1:1:99
        sum_2 = sum_2 + s_sys_in(i_3)*S_Z(i_3);
    end

    % a mechanikai rendszer találkozik a kioltandó rezgéssel

    sum_2 = sum_2 + X(t);


    % S(z) szűrés elvégzése

    for i_4=99:(-1):2
        s_ident_1(i_4) = s_ident_1(i_4-1);
    end
```

```
    s_ident_1(1) = sum_1;

    sum_3 = 0;
    for i_5=1:1:99
        sum_3 = sum_3 + s_ident_1(i_5)*S_Z(i_5);
    end

    % referencia jel meghatározása

    sum = sum_3 + sum_2;

    error = sum_2;

    % 2. S(z) szűrés elvégzése

    for i_6=99:(-1):2
        in(i_6) = in(i_6-1);
    end

    in(1) = sum;

    sum_4 = 0;
    for i_8=1:1:99
        sum_4 = sum_4 + in(i_8)*S_Z(i_8);
    end

    % lms buffer frissítés

    for i_9=99:(-1):2
        lms_buffer(i_9) = lms_buffer(i_9-1);
    end

    lms_buffer(1) = sum_4;

    % a kétféle algoritmus elvégzése

    for i_7=1:1:99
 % 1 %  w(i_7) = w(i_7) + mu*error*lms_buffer(i_7);

 % 2 %  if error*lms_buffer(i_7) > 0
 %          w(i_7) = w(i_7) + 0.000025;
 %      end
 %      if error*lms_buffer(i_7) < 0
 %          w(i_7) = w(i_7) - 0.000025;
 %      end
    end

    % DA, és hiba elmentése

    output(t) = DA;
    error_s(t) = error;
end

figure(2);
plot(output);
figure(3);
plot(error_s);
```

7.2. A mikrokontroller forráskódja

```
#include <avr/io.h>

// a led ki/be kapcsolása

#define led_off         PORTB &= ~(1<<PINB0)
#define led_on          PORTB |=  (1<<PINB0)

#define m 0             // hiba komparálási szint
#define h 5             // hiba identifikáció során elfogadható értéke
#define l 50            // a hibának elfogadható értéken, eddig kell maradnia
#define ident 0         // ident = 1, ha identifikáció fut, 0 ha elnyomás

// i változók az adaptációs szűrő bemeneti mintáinak tömbje
// c változók az adaptációs szűrő koefficiensei

volatile int8_t i1,i2,i3,i4,i5,i6,i7,i8,i9,i10,i11,i12,i13,i14,i15,i16,i17,i18,i19,i20;
volatile int8_t c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17,c18,c19,c20;

volatile int8_t i21,i22,i23,i24,i25,i26,i27,i28,i29,i30,i31,i32,i33,i34,i35,i36,i37,i38;
volatile int8_t c21,c22,c23,c24,c25,c26,c27,c28,c29,c30,c31,c32,c33,c34,c35,c36,c37,c38;

// y változók az S(z) szűrő bemeneti mintáinak tömbje
// s változók az S(z) szűrő koefficiensei

volatile int8_t y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20;
volatile int8_t s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11,s12,s13,s14,s15,s16,s17,s18,s19,s20;

volatile int8_t y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,y32,y33,y34,y35,y36,y37,y38;
volatile int8_t s21,s22,s23,s24,s25,s26,s27,s28,s29,s30,s31,s32,s33,s34,s35,s36,s37,s38;

// b változók a koefficiens számoló algoritmus bemeneti mintáinak tömbje

volatile int8_t b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13,b14,b15,b16,b17,b18,b19;
volatile int8_t b20,b21,b22,b23,b24,b25,b26,b27,b28,b29,b30,b31,b32,b33,b34,b35,b36,b37,b38;

// függvény az AD konverter konfigurálásához

void init_8bitADC()
{
        ADCSRA =        (1<<ADEN)
                |       (1<<ADPS2)
                |       (0<<ADPS1)
                |       (1<<ADPS0);

        ADMUX |= (0<<REFS0)|(1<<ADLAR);    // Vref mint ref, balra rendezett ADC
}

// minta beolvasására függvény

int8_t read_8bitADC(int8_t ch)
{
        ADMUX = (ADMUX & 0b11110000) | ch;     // ADC csatorna
        ADCSRA |= (1<<ADSC);                   // konverzió elindítás
        while (ADCSRA & (1<<ADSC));            // átalakítás
        return (ADCH);                         // ADC értékének visszaadása, (csak a felső 8bit)
}
```

// koefficiensek kimentéséről gondoskodó függvény

```
void eeprom_write (uint8_t add, int8_t data)
{

        while(EECR & (1<<EEPE))
        EEARH = 0;
        EEARL = add;
        EEDR = data;
        EECR |= (1<<EEMPE);
        EECR |= (1<<EEPE);

}
```

// S(z) koefficiensei

```
void S_z_set(void)
{
                s1=0xFF;
                s2=0xFF;
                s3=0x14;
                s4=0x14;
                s5=0x14;
                s6=0x14;
                s7=0x13;
                s8=0x13;
                s9=0x12;
                s10=0x11;
                s11=0x10;
                s12=0x08;
                s13=0x06;
                s14=0x04;
                s15=0x04;
                s16=0x04;
                s17=0x02;
                s18=0x01;
                s19=0x02;
                s20=0x01;
                s21=0x02;
                s22=0x02;
                s23=0x01;
                s24=0x02;
                s25=0x02;
                s26=0x01;
                s27=0x02;
                s28=0x01;
                s29=0x02;
                s30=0x01;
                s31=0xFF;
                s32=0x01;
                s33=0x02;
                s34=0x00;
                s35=0x01;
                s36=0x00;
                s37=0x01;
                s38=0x00;
}
```

```
// portokat inicializáló függvény

void port_init(void)
{
        DDRB = (1<<PINB0);              // led kimenet

        DDRD =          (1<<PIND0)|     // DAC kimenete
                        (1<<PIND1)|
                        (1<<PIND2)|
                        (1<<PIND3)|
                        (1<<PIND4)|
                        (1<<PIND5)|
                        (1<<PIND6)|
                        (1<<PIND7);
}

// identifikáció során a koefficienseket a hexa 30 címtől kezdődően kimeneti az EEPROM-ba

void save_coeff(void)
{
        eeprom_write(0x30, c1);
        eeprom_write(0x31, c2);
        eeprom_write(0x32, c3);
        eeprom_write(0x33, c4);
        eeprom_write(0x34, c5);
        eeprom_write(0x35, c6);
        eeprom_write(0x36, c7);
        eeprom_write(0x37, c8);
        eeprom_write(0x38, c9);
        eeprom_write(0x39, c10);
        eeprom_write(0x3A, c11);
        eeprom_write(0x3B, c12);
        eeprom_write(0x3C, c13);
        eeprom_write(0x3D, c14);
        eeprom_write(0x3E, c15);
        eeprom_write(0x3F, c16);
        eeprom_write(0x40, c17);
        eeprom_write(0x41, c18);
        eeprom_write(0x42, c19);
        eeprom_write(0x43, c20);
        eeprom_write(0x44, c21);
        eeprom_write(0x45, c22);
        eeprom_write(0x46, c23);
        eeprom_write(0x47, c24);
        eeprom_write(0x48, c25);
        eeprom_write(0x49, c26);
        eeprom_write(0x4A, c27);
        eeprom_write(0x4B, c28);
        eeprom_write(0x4C, c29);
        eeprom_write(0x4D, c30);
        eeprom_write(0x4E, c31);
        eeprom_write(0x4F, c32);
        eeprom_write(0x50, c33);
        eeprom_write(0x51, c34);
        eeprom_write(0x52, c35);
        eeprom_write(0x53, c36);
        eeprom_write(0x54, c37);
        eeprom_write(0x55, c38);
}
```

```
// Timer1 inicializálása, az alap clk/8-ra, ezzel egy egység 0,4us

void init_timer_1(void)
{

	TCCR1B =	(0<<CS12) |
			(0<<CS11) |
			(1<<CS10);
}

// fő függvény

void main(void)
{

		int16_t y_adap;				// adaptív szűrő kimenete
		int16_t y_s;				// S(z) szűrő kimenete
		int8_t y_adap_8, y_s_8, x_ref, x_rend;	// adaptív, S(z) szűrő 8bites kimenete
		int8_t error_null, temp;		// hiba értéke, ideiglenes változó
		uint8_t sz;				// hiba beállásának hossza
		uint8_t t1;

		// változóknak kezdőérték adása (amennyiben szükséges)

		sz = 0;
		t1 = 0;
		c1=c2=c3=c4=c5=c6=c7=c8=c9=c10=c11=c12=c13=c14=c15=c16=c17=c18=c19=c20=0;
		c21=c22=c23=c24=c25=c26=c27=c28=c29=c30=c31=c32=c33=c34=c35=c36=c37=c38=0;

		// inicializáló függvények hívása, S(z) koefficienseinek felparaméterezése

		init_timer_1();
		port_init();
		init_8bitADC();
		S_z_set();

		// identifikáló mód:

		while(ident == 1)
		{
			TCNT1 = 0;				// ciklus elején timer1 nullázása

			x_ref = read_8bitADC(4) – 128;		// referencia jel beolvasása

			PORTD = x_ref + 128;			// referencia jel kiadása a DA-ra

			x_rend = read_8bitADC(5) – 128;		// rendszer válaszának beolvasása

			// adaptív szűrés elvégzése

			y_adap = i1*c1 + i2*c2 + i3*c3 + i4*c4 + i5*c5 + i6*c6 + i7*c7 + i8*c8 + i9*c9 +
i10*c10 + i11*c11 + i12*c12 + i13*c13 + i14*c14 + i15*c15 + i16*c16 + i17*c17 + i18*c18 + i19*c19 + i20*c20 +
i21*c21 + i22*c22 + i23*c23 + i24*c24 + i25*c25 + i26*c26 + i27*c27 + i28*c28 + i29*c29 + i30*c30 + i31*c31 +
i32*c32 + i33*c33 + i34*c34 + i35*c35 + i36*c36 + i37*c37 + i38*c38;

			y_adap_8 = y_adap>>8;

			// hiba jel számítása a rendszer válasza, és az adaptív szűrő kimenete alapján

			error_null = x_rend - y_adap_8;
```

```
// ha a hiba elfogadható szintre csökkent, akkor növeljük a számlálót,
// ha nem akkor nullázzuk

if((error_null < h)&(error_null > -h))
{           sz++;       }
else
{           sz=0;       }

// ha a hiba az előre meghatározott ideig a hibahatáron belül marad, és a futás során még
// nem érte el ezt az állapotot, akkor elmentjük a koefficienseket az EEPROM-ba

if((sz>l)&(t1==0))
{
            led_on;
            t1 = 1;
            save_coeff();
}

// adaptív szűrő, és a koefficienseket számoló algoritmus tömbjének frissítése

i38 = i37;
i37 = i36;
i36 = i35;
i35 = i34;
i34 = i33;
i33 = i32;
i32 = i31;
i31 = i30;
i30 = i29;
i29 = i28;
i28 = i27;
i27 = i26;
i26 = i25;
i25 = i24;
i24 = i23;
i23 = i22;
i22 = i21;
i21 = i20;
i20 = i19;
i19 = i18;
i18 = i17;
i17 = i16;
i16 = i15;
i15 = i14;
i14 = i13;
i13 = i12;
i12 = i11;
i11 = i10;
i10 = i9;
i9 = i8;
i8 = i7;
i7 = i6;
i6 = i5;
i5 = i4;
i4 = i3;
i3 = i2;
i2 = i1;
i1 = x_ref;
```

```
// ha még nem mentettük el a koefficienseket,
// akkor az aktuális hiba alapján módosítjuk őket

if(t1 == 0)
{
        temp = error_null*i1>>8;
        if(temp >   m)      {c1++;}
        if(temp <  -m)      {c1--;}

        temp = error_null*i2>>8;
        if(temp >   m)      {c2++;}
        if(temp <  -m)      {c2--;}

        temp = error_null*i3>>8;
        if(temp >   m)      {c3++;}
        if(temp <  -m)      {c3--;}

        temp = error_null*i4>>8;
        if(temp >   m)      {c4++;}
        if(temp <  -m)      {c4--;}

        temp = error_null*i5>>8;
        if(temp >   m)      {c5++;}
        if(temp <  -m)      {c5--;}

        temp = error_null*i6>>8;
        if(temp >   m)      {c6++;}
        if(temp <  -m)      {c6--;}

        temp = error_null*i7>>8;
        if(temp >   m)      {c7++;}
        if(temp <  -m)      {c7--;}

        temp = error_null*i8>>8;
        if(temp >   m)      {c8++;}
        if(temp <  -m)      {c8--;}

        temp = error_null*i9>>8;
        if(temp >   m)      {c9++;}
        if(temp <  -m)      {c9--;}

        temp = error_null*i10>>8;
        if(temp >   m)      {c10++;}
        if(temp <  -m)      {c10--;}

        temp = error_null*i11>>8;
        if(temp >   m)      {c11++;}
        if(temp <  -m)      {c11--;}

        temp = error_null*i12>>8;
        if(temp >   m)      {c12++;}
        if(temp <  -m)      {c12--;}

        temp = error_null*i13>>8;
        if(temp >   m)      {c13++;}
        if(temp <  -m)      {c13--;}

        temp = error_null*i14>>8;
        if(temp >   m)      {c14++;}
        if(temp <  -m)      {c14--;}

        temp = error_null*i15>>8;
        if(temp >   m)      {c15++;}
        if(temp <  -m)      {c15–;}
```

```
temp = error_null*i16>>8;
if(temp >   m)        {c16++;}
if(temp <  -m)        {c16--;}

temp = error_null*i17>>8;
if(temp >   m)        {c17++;}
if(temp <  -m)        {c17--;}

temp = error_null*i18>>8;
if(temp >   m)        {c18++;}
if(temp <  -m)        {c18--;}

temp = error_null*i19>>8;
if(temp >   m)        {c19++;}
if(temp <  -m)        {c19--;}

temp = error_null*i20>>8;
if(temp >   m)        {c20++;}
if(temp <  -m)        {c20--;}

temp = error_null*i21>>8;
if(temp >   m)        {c21++;}
if(temp <  -m)        {c21--;}

temp = error_null*i22>>8;
if(temp >   m)        {c22++;}
if(temp <  -m)        {c22--;}

temp = error_null*i23>>8;
if(temp >   m)        {c23++;}
if(temp <  -m)        {c23--;}

temp = error_null*i24>>8;
if(temp >   m)        {c24++;}
if(temp <  -m)        {c24--;}

temp = error_null*i25>>8;
if(temp >   m)        {c25++;}
if(temp <  -m)        {c25--;}

temp = error_null*i26>>8;
if(temp >   m)        {c26++;}
if(temp <  -m)        {c26--;}

temp = error_null*i27>>8;
if(temp >   m)        {c27++;}
if(temp <  -m)        {c27--;}

temp = error_null*i28>>8;
if(temp >   m)        {c28++;}
if(temp <  -m)        {c28--;}

temp = error_null*i29>>8;
if(temp >   m)        {c29++;}
if(temp <  -m)        {c29--;}

temp = error_null*i30>>8;
if(temp >   m)        {c30++;}
if(temp <  -m)        {c30--;}

temp = error_null*i31>>8;
if(temp >   m)        {c31++;}
if(temp <  -m)        {c31--;}
```

```
temp = error_null*i32>>8;
if(temp >   m)        {c32++;}
if(temp <  -m)        {c32--;}

temp = error_null*i33>>8;
if(temp >   m)        {c33++;}
if(temp <  -m)        {c33--;}

temp = error_null*i34>>8;
if(temp >   m)        {c34++;}
if(temp <  -m)        {c34--;}

temp = error_null*i35>>8;
if(temp >   m)        {c35++;}
if(temp <  -m)        {c35--;}

temp = error_null*i36>>8;
if(temp >   m)        {c36++;}
if(temp <  -m)        {c36--;}

temp = error_null*i37>>8;
if(temp >   m)        {c37++;}
if(temp <  -m)        {c37--;}

temp = error_null*i38>>8;
if(temp >   m)        {c38++;}
if(temp <  -m)        {c38--;}


}


// ha a ciklus gyorsan eljutna erre a pontra, akkor Timer-es késleltetés

while(TCNT1 < 12500) {;}   // fs = 200Hz


}


// rezgéselnyomó mód


while(ident == 0)
{
        TCNT1 = 0;                    // ciklus elején a Timer1 nullázása

        // adaptív szűrés elvégzése

        y_adap = i1*c1 + i2*c2 + i3*c3 + i4*c4 + i5*c5 + i6*c6 + i7*c7 + i8*c8 + i9*c9 +
i10*c10 + i11*c11 + i12*c12 + i13*c13 + i14*c14 + i15*c15 + i16*c16 + i17*c17 + i18*c18 + i19*c19 + i20*c20 +
i21*c21 + i22*c22 + i23*c23 + i24*c24 + i25*c25 + i26*c26 + i27*c27 + i28*c28 + i29*c29 + i30*c30 + i31*c31 +
i32*c32 + i33*c33 + i34*c34 + i35*c35 + i36*c36 + i37*c37 + i38*c38;

        y_adap_8 = y_adap>>8;


        // beavatkozó jel invertálása, és kivezetése a DA konverterre

        PORTD = 255 - (y_adap_8 + 128);
```

```
// S(z) bemeneti tömbjének frissítése

y38 = y37;
y37 = y36;
y36 = y35;
y35 = y34;
y34 = y33;
y33 = y32;
y32 = y31;
y31 = y30;
y30 = y29;
y29 = y28;
y28 = y27;
y27 = y26;
y26 = y25;
y25 = y24;
y24 = y23;
y23 = y22;
y22 = y21;
y21 = y20;
y20 = y19;
y19 = y18;
y18 = y17;
y17 = y16;
y16 = y15;
y15 = y14;
y14 = y13;
y13 = y12;
y12 = y11;
y11 = y10;
y10 = y9;
y9 = y8;
y8 = y7;
y7 = y6;
y6 = y5;
y5 = y4;
y4 = y3;
y3 = y2;
y2 = y1;
y1 = y_adap_8;

// S(z) szűrés elvégzése

y_s = y1*s1 + y2*s2 + y3*s3 + y4*s4 + y5*s5 + y6*s6 + y7*s7 + y8*s8 + y9*s9 +
y10*s10 + y11*s11 + y12*s12 + y13*s13 + y14*s14 + y15*s15 + y16*s16 + y17*s17 + y18*s18 + y19*s19 + y20*s20
+ y21*s21 + y22*s22 + y23*s23 + y24*s24 + y25*s25 + y26*s26 + y27*s27 + y28*s28 + y29*s29 + y30*s30 +
y31*s31 + y32*s32 + y33*s33 + y34*s34 + y35*s35 + y36*s36 + y37*s37 + y38*s38;


y_s_8 = y_s>>8;

// hiba beolvasása a gyorsulásérzékelőről

error_null = read_8bitADC(5) – 128;


// referencia jel számítása

x_ref = error_null + y_s_8;
```

// adaptív szűrő, és a második S(z) tömbjének frissítése

```
i38 = i37;
i37 = i36;
i36 = i35;
i35 = i34;
i34 = i33;
i33 = i32;
i32 = i31;
i31 = i30;
i30 = i29;
i29 = i28;
i28 = i27;
i27 = i26;
i26 = i25;
i25 = i24;
i24 = i23;
i23 = i22;
i22 = i21;
i21 = i20;
i20 = i19;
i19 = i18;
i18 = i17;
i17 = i16;
i16 = i15;
i15 = i14;
i14 = i13;
i13 = i12;
i12 = i11;
i11 = i10;
i10 = i9;
i9 = i8;
i8 = i7;
i7 = i6;
i6 = i5;
i5 = i4;
i4 = i3;
i3 = i2;
i2 = i1;
i1 = x_ref;
```

// a második S(z) szűrés elvégzése

```
y_s = i1*s1 + i2*s2 + i3*s3 + i4*s4 + i5*s5 + i6*s6 + i7*s7 + i8*s8 + i9*s9 + i10*s10 +
i11*s11 + i12*s12 + i13*s13 + i14*s14 + i15*s15 + i16*s16 + i17*s17 + i18*s18 + i19*s19 + i20*s20 + i21*s21 +
i22*s22 + i23*s23 + i24*s24 + i25*s25 + i26*s26 + i27*s27 + i28*s28 + i29*s29 + i30*s30 + i31*s31 + i32*s32 +
i33*s33 + i34*s34 + i35*s35 + i36*s36 + i37*s37 + i38*s38;

y_s_8 = y_s>>8;
```

// a koefficienseket számoló algoritmus mintatároló tömbjének frissítése

```
b38 = b37;
b37 = b36;
b36 = b35;
b35 = b34;
b34 = b33;
b33 = b32;
b32 = b31;
b31 = b30;
b30 = b29;
b29 = b28;
b28 = b27;
b27 = b26;
b26 = b25;
b25 = b24;
b24 = b23;
```

```
b23 = b22;
b22 = b21;
b21 = b20;
b20 = b19;
b19 = b18;
b18 = b17;
b17 = b16;
b16 = b15;
b15 = b14;
b14 = b13;
b13 = b12;
b12 = b11;
b11 = b10;
b10 = b9;
b9 = b8;
b8 = b7;
b7 = b6;
b6 = b5;
b5 = b4;
b4 = b3;
b3 = b2;
b2 = b1;
b1 = y_s_8;

// koefficiensek számolása

temp = error_null*b1>>8;
if(temp >   m)        {c1++;}
if(temp <  -m)        {c1--;}

temp = error_null*b2>>8;
if(temp >   m)        {c2++;}
if(temp <  -m)        {c2--;}

temp = error_null*b3>>8;
if(temp >   m)        {c3++;}
if(temp <  -m)        {c3--;}

temp = error_null*b4>>8;
if(temp >   m)        {c4++;}
if(temp <  -m)        {c4--;}

temp = error_null*b5>>8;
if(temp >   m)        {c5++;}
if(temp <  -m)        {c5--;}

temp = error_null*b6>>8;
if(temp >   m)        {c6++;}
if(temp <  -m)        {c6--;}

temp = error_null*b7>>8;
if(temp >   m)        {c7++;}
if(temp <  -m)        {c7--;}

temp = error_null*b8>>8;
if(temp >   m)        {c8++;}
if(temp <  -m)        {c8--;}

temp = error_null*b9>>8;
if(temp >   m)        {c9++;}
if(temp <  -m)        {c9--;}

temp = error_null*b10>>8;
if(temp >   m)        {c10++;}
if(temp <  -m)        {c10--;}
```

```
temp = error_null*b11>>8;
if(temp >   m)        {c11++;}
if(temp <  -m)        {c11--;}

temp = error_null*b12>>8;
if(temp >   m)        {c12++;}
if(temp <  -m)        {c12--;}

temp = error_null*b13>>8;
if(temp >   m)        {c13++;}
if(temp <  -m)        {c13--;}

temp = error_null*b14>>8;
if(temp >   m)        {c14++;}
if(temp <  -m)        {c14--;}

temp = error_null*b15>>8;
if(temp >   m)        {c15++;}
if(temp <  -m)        {c15--;}

temp = error_null*b16>>8;
if(temp >   m)        {c16++;}
if(temp <  -m)        {c16--;}

temp = error_null*b17>>8;
if(temp >   m)        {c17++;}
if(temp <  -m)        {c17--;}

temp = error_null*b18>>8;
if(temp >   m)        {c18++;}
if(temp <  -m)        {c18--;}

temp = error_null*b19>>8;
if(temp >   m)        {c19++;}
if(temp <  -m)        {c19--;}

temp = error_null*b20>>8;
if(temp >   m)        {c20++;}
if(temp <  -m)        {c20--;}

temp = error_null*b21>>8;
if(temp >   m)        {c21++;}
if(temp <  -m)        {c21--;}

temp = error_null*b22>>8;
if(temp >   m)        {c22++;}
if(temp <  -m)        {c22--;}

temp = error_null*b23>>8;
if(temp >   m)        {c23++;}
if(temp <  -m)        {c23--;}

temp = error_null*b24>>8;
if(temp >   m)        {c24++;}
if(temp <  -m)        {c24--;}
temp = error_null*b25>>8;
if(temp >   m)        {c25++;}
if(temp <  -m)        {c25--;}

temp = error_null*b26>>8;
if(temp >   m)        {c26++;}
if(temp <  -m)        {c26--;}

temp = error_null*b27>>8;
if(temp >   m)        {c27++;}
if(temp <  -m)        {c27--;}
```

```
                temp = error_null*b28>>8;
                if(temp >   m)          {c28++;}
                if(temp <  -m)          {c28--;}

                temp = error_null*b29>>8;
                if(temp >   m)          {c29++;}
                if(temp <  -m)          {c29--;}

                temp = error_null*b30>>8;
                if(temp >   m)          {c30++;}
                if(temp <  -m)          {c30--;}

                temp = error_null*b31>>8;
                if(temp >   m)          {c31++;}
                if(temp <  -m)          {c31--;}

                temp = error_null*b32>>8;
                if(temp >   m)          {c32++;}
                if(temp <  -m)          {c32--;}

                temp = error_null*b33>>8;
                if(temp >   m)          {c33++;}
                if(temp <  -m)          {c33--;}

                temp = error_null*b34>>8;
                if(temp >   m)          {c34++;}
                if(temp <  -m)          {c34--;}

                temp = error_null*b35>>8;
                if(temp >   m)          {c35++;}
                if(temp <  -m)          {c35--;}

                temp = error_null*b36>>8;
                if(temp >   m)          {c36++;}
                if(temp <  -m)          {c36--;}

                temp = error_null*b37>>8;
                if(temp >   m)          {c37++;}
                if(temp <  -m)          {c37--;}

                temp = error_null*b38>>8;
                if(temp >   m)          {c38++;}
                if(temp <  -m)          {c38–;}

                // ha a ciklus gyorsan eljutna erre a pontra, akkor Timer-es késleltetés

                while(TCNT1 < 12500) {;}   // fs = 200Hz

        }

}
```