EFOP-3.4.3-16-2016-00014

JEGYZET

Jelfeldolgozás és számítógépes méréstechnika

Készítette: Kelemen András

SZTE JGYPK

Informatika Alkalmazásai Tanszék

Lektorálta:Tóth Attila

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával. Projekt azonosító: EFOP-3.4.3-16-2016-00014.

Alprojekt azonosító: AP2 – Komplex képzés- és szolgáltatásfejlesztés

Altéma azonosító: AP2_JGYPK5 Magyar és idegen nyelvű képzések oktatási innovációja az MTMI területen és tanártovábbképzés

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

EFOP-3.4.3-16-2016-00014

Kelemen András

Jelfeldolgozás és számítógépes méréstechnika

Szegedi Tudományegyetem

Szeged 2020

Tanulási eredmények

Azoknak az előírt szakmai kompetenciáknak, kompetencia elemeknek *(tudás, képesség* stb.,*KKK 7. pont*) a felsorolása, amelyek kialakításához a tantárgyelem jellemzően, érdemben hozzájárul:

Tudás	**Képesség**	**Attitűd**	**Autonómia/felelősség**
Ismeri a digitális jelfeldolgozás alapfogalmait.	Képes az analóg jelek digitalizálásához megfelelő paraméterekkel rendelkező analóg-digitális átalakítót választani.	Törekszik az elemi jelfeldolgozási műveletek precíz megoldására.	Az oktató által kijelölt feladatokat képes megoldani, munkájának önálló ellenőrzésére.
Tisztában van a mintavételi tétel jelentőségével.	Jól alkalmazza a mintavételi tételt, adott feladatnál helyes mintavételi frekvenciát alkalmaz.	Nyitott a jelfeldolgozási feladatoknál az előzetes paraméterek precíz meghatározására.	Önállóan ellenőrzi, és szükség esetén javítja az előzetes kalkulációit.
Ismeri az időalapú és a frekvenciaalapú jelfeldolgozási módszereket.	Képes eldönteni, hogy az adott jelfeldolgozási feladathoz milyen jelfeldolgozási módszert válasszon.	Nyitott a különböző jelfeldolgozási algoritmusok megismerésére, alkalmazására.	A jelfeldolgozási műveletek során kapott eredményeket szisztematikusan ellenőrzi, a hibákat javítja.
Ismeri a digitális szűrők tervezésének módszereit,	Képes digitális szűrők szoftveres implementálására.	Törekszik a szoftveres implementációnál a precíz programozásra.	A szoftveres implementációknál széleskörűen tesztel, javítja a hibákat.
Ismeri a különböző mikrokontrollerek programozási lehetőségeit	Használja a mikrokontrollereket méréstechnikai feladatoknál.	Programjaiban precízen használja a mikrokontrollerek erőforrásait.	Önállóan ellenőrzi, és szükség esetén javítja munkáját. Precízen teszteli programjait.

Tartalomjegyzék

Bevezetés

Digitális jelfeldolgozásnak (angolul Digital Signal Processing – DSP) nevezzük azt a folyamatot, amikor egy valós fizikai mennyiséget (pl. a levegő nyomásváltozásait, hőmérsékletváltozást, képet, stb.) valamilyen módon digitális technikával dolgozunk fel. Ez a módszer napjaink technikájának egyik alappillére. Gondoljunk csak bele, hogy digitális fényképező gépeink, video kameráink vannak. A mindennapi kommunikációban mobil telefont, internetet használunk, melyek az információt digitális formában tárolják, és digitális hálózatokon keresztül juttatják el a célig. A digitális jelfeldolgozás jelen van napjaink orvos diagnosztikai eszközeiben is a bonyolult CT (Computer Tomography) berendezéstől kezdve az otthon használatos automata vérnyomásmérőig bezárólag szinte mindenütt. Hasonlóképpen jelen van az iparban a közlekedésben mindenütt, ahol mérni, szabályozni, vezérelni kell valamit. Ennek oka, hogy a mikroelektronikai és informatikai fejlődés nagyon olcsóvá és egyszerűvé tette az információ feldolgozását.

Összefoglalva elmondhatjuk, hogy a digitális jelfeldolgozás igazi interdiszciplináris tudomány, mert egyrészt felhasználja a matematika, a műszaki informatika, az elektronika, a méréstechnika által már elért eredményeket. Másrészt megoldási módszereket, eljárásokat kínál más tudományágak részére. Jellégét és alkalmazott módszereit tekintve a mérnöki tudomány. Felsorolunk néhány olyan diszciplínát, melynek sikeres művelése a digitális jelfeldolgozás fogalmai, módszerei nélkül nem lehetséges:

- Távközlés technológia.
- Képfeldolgozás
- Orvosi méréstechnika, diagnosztika
- Hang és videotechnika
- Ipari folyamatirányítás, vezérlés
- Radar és szonár technika

A digitális jelfeldolgozáshoz szorosan kapcsolódik a számítógépes méréstechnika. A számítógép feladata a méréstechnikában az információ gyűjtése, megjelenítése, tárolás és feldolgozása. Ebben a könyvben számítógépen személyi számítógépet (asztali PC, laptop) értünk. A személyi számítógépek a jelenlegi hardver felépítésük miatt (az estleges mikrofon bemenet kivételével) közvetlenül nem használhatóak a méréstechnikában. Ezért az adott fizikai jel (pl. hőmérséklet, nyomás, potenciál változás, stb) tényleges mérését végző

műszernek képesnek kell lennie a számítógéppel történő digitális kommunikációra. Ennek következtében a mérőműszerek különböző mikrokontrollereket használnak a valós fizikai jelek digitalizálására, és a személyi számítógéppel történő kommunikációra.

Ennek a könyvnek a célja, hogy bevezetést nyújtson a számítógépes méréstechnika és a digitális jelfeldolgozás világába. A könyv első 5 fejezete összefoglalja a digitális jelfeldolgozás alapfogalmait, alapvető módszereit, algoritmusait. A 6-9 fejezetek pedig a számítógépes méréstechnika világába vezet be. A könyv írásakor a gyakorlati szempontok vezettek, ezért szinte minden a könyvben előforduló módszert, algoritmust a valóságban is működő programkóddal illusztráltam. A programkódok C valamint C# nyelven készültek. A C# nyelven íródott kódok értelemszerűen minden olyan gépen futnak, ahol a .NET framework telepítve van. A C nyelvű kódok viszont a Silocon Labs C8051F120-as mikrokontrollerére lettek írva. A könyvben ez a kontroller a mérőműszer lelke. A könyvben foglaltak megértéséhez szükség van a trigonometrikus és exponenciális függvények, a komplex számok, valamint a differenciál és integrálszámítás alapfogalmainak ismeretére. Kell még némi elektronikai, valamint programozási és számítógép architektúra ismeret is.

A könyvben minden egyes fejezet végén kérdések és feladatok vannak. Ezek megoldásához általában elég az adott fejezet átolvasása és megértése. Ott, ahol a megoldáshoz programot kell írni, ott a megoldást is közlöm.

Minden kedves olvasónak jó tanulást kívánok.

Szeged,

2020 augusztus Kelemen András

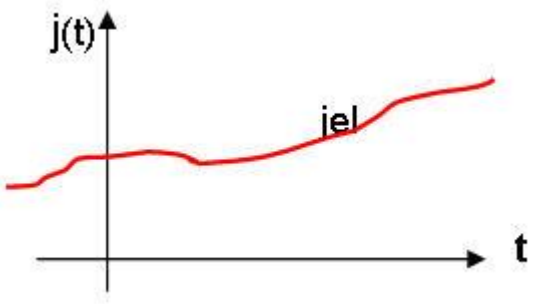
1. A jelfeldolgozásról általában

1.1 Jelek fogalma, osztályozása

Jelnek nevezzük azokat a fizikai mennyiségeket, amelyek egy fizikai rendszer állapotát írják le a valamely más fizikai mennyiség (pl. hely, idő, hőmérséklet) függvényeként. A jel viselkedését általában egyetlen független változó - mely egyben a jel értelmezési tartománya- függvényében vizsgáljuk. A jel értékkészletét a jel amplitúdójának nevezzük. A jeleket értelmezési tartományuk és az értékkészletük alapján osztályozzuk.

Analóg jelek

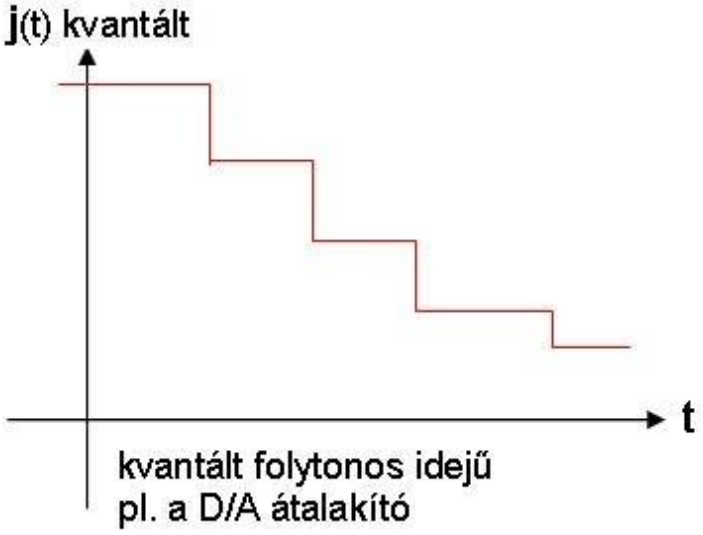
Az értékkészletükben és értelmezési tartományukban is folytonos jeleket analóg jeleknek nevezzük (1.1. ábra).



1.1. ábra: Analóg jel

Diszkrét amplitúdójú jelek

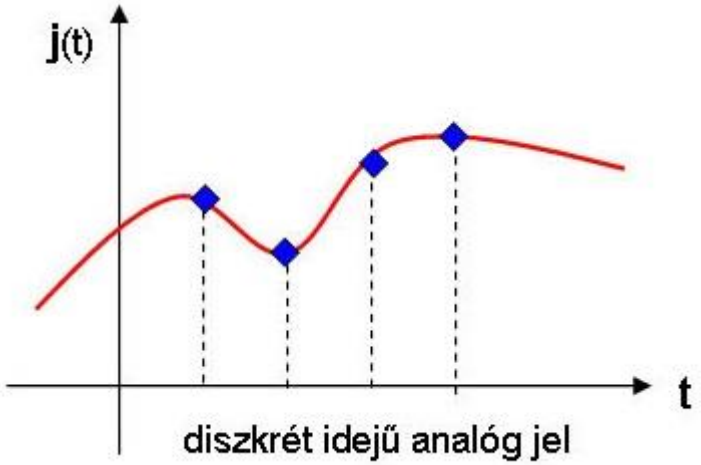
Egy jelet diszkrét amplitúdójúnak nevezünk, ha a jel értelmezési tartománya folytonos, de az amplitúdó értékek ugrásszerűen változnak (1.2. ábra).



1.2. ábra: Diszkrét amplitúdójú jel.

Értelmezési tartományon diszkrét jelek

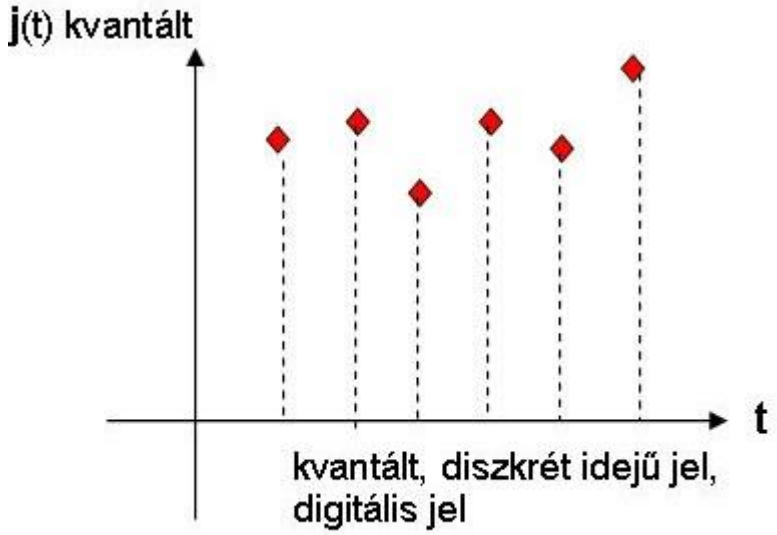
Egy jelet az értelmezési tartományon diszkrétnek nevezzük, ha a jel értékkészlete folytonos, de a jel értékek csak az értelmezési tartomány diszkrét pontjaiban állnak rendelkezésre (1.3. ábra).



1.3. ábra: Diszkrét amplitúdójú jel.

Digitális jelek

Egy jelet digitálisnak nevezünk, ha mind értékkészletében, mind értelmezési tartományában diszkrét számokból áll (1.4. ábra).

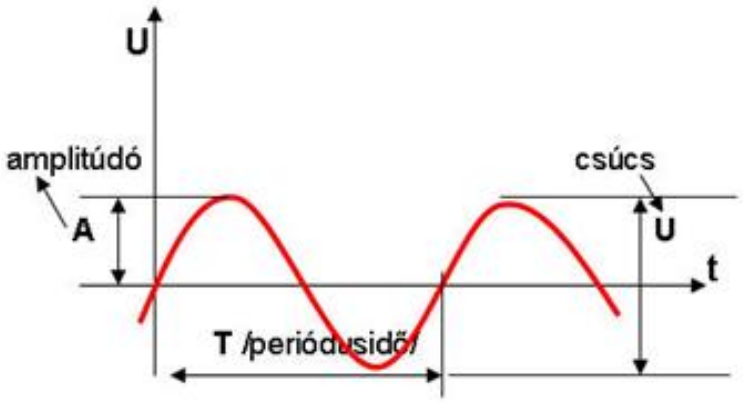


1.4. ábra: Digitális jel

Periódikus jelek

Az *f(t)* jelet periodikusnak nevezünk, ha adott időközönként a jel felveszi ugyanazt az értéket, azaz minden egyes *t* időpontra fennáll a következő összefüggés:

$f(t) = f(t \pm T)$, ahol T>0. A T-t a jel periódusidejének nevezzük (1.5. ábra).



1.5. ábra: Periodikus jel.

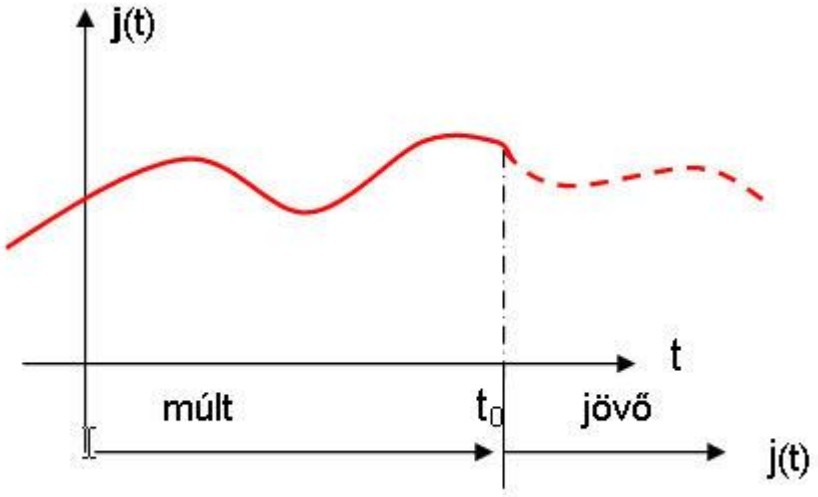
Tranziens jelek

Egy jelet tranziens jelnek nevezünk, ha a jel nem periodikus, egyszeri, és a jel által szállított energia véges. A tranziens jeleket a felfutási és lecsengési idejükkel jellemezhetjük

legjobban. Felfutási időn itt azt az időtartamot értjük, amíg a jel amplitúdója a kezdeti értékről a maximális érték eléréséig tart. Lecsengési időn pedig azt az időtartamot, amíg az amplitúdó a maximális érték közeléből a végérték közelébe csökken.

Determinisztikus jelek

Azokat a jeleket, ahol a jel múltbeli viselkedésből és a jelenkori értékéből matematikai módszerek segítségével meg tudjuk határozni, a jel jövőbeli viselkedését, determinisztikus jeleknek nevezzük (1.6. ábra).

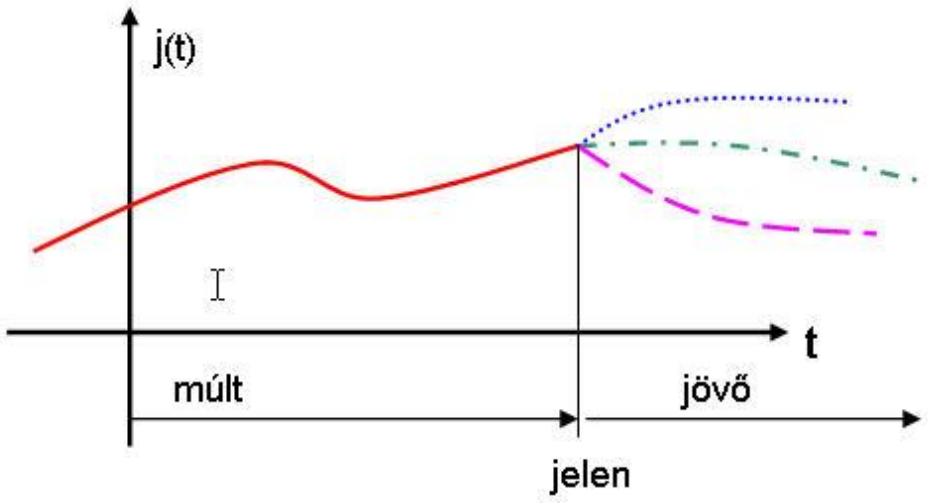


1.6. ábra: Determinisztikus jel.

Determinisztikus jelek közé tartoznak a periodikus, kvázi periodikus, valamint a tranziens jelek.

Sztochasztikus jelek

Azokat a jeleket, ahol a jel múltbeli viselkedésből és a jelenkori értékéből nem lehet következtetni a jel jövőbeli viselkedésre sztochasztikus jeleknek nevezzük (1.7. ábra).

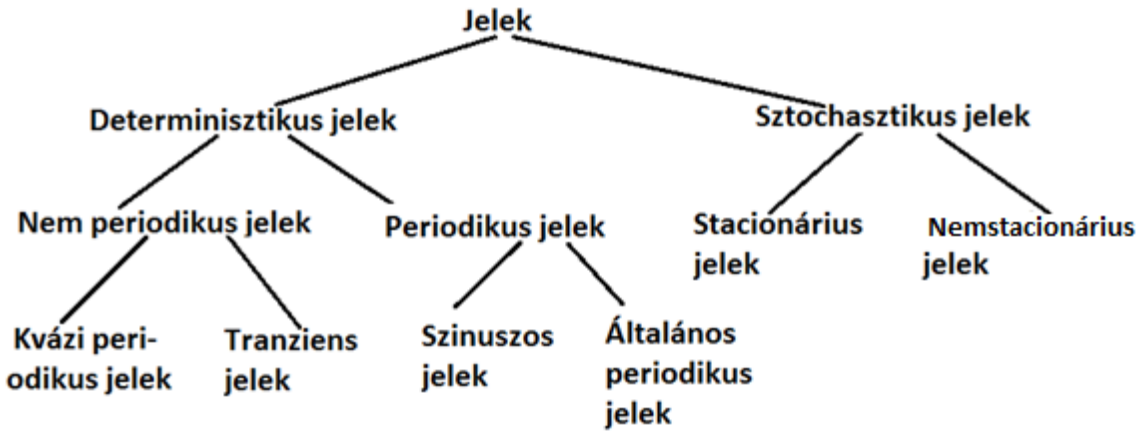


1.7. ábra: Sztochasztikus jel.

Ez azt is jelenti, hogy a sztochasztikus jelek csak részben kezelhetőek matematikai módszerekkel és statisztikai jellemzőket (várható érték-idő függvény, négyzetes középérték-idő függvény, variancia stb.) használunk a jel kvantitatív elemzésekor.

Jelek osztályozása

A fentiek alapján a jeleket az 1.8. ábra szerint osztályozhatjuk:



1.8. ábra: Jelek osztályozása.

1.2 Jelek jellemzői

Energia

Egy *f(t)* jel energiáján analóg esetben az

$$E = \int_0^\infty f^2(t)dt\,,$$

kifejezést, míg digitális esetben az

$$E = \sum_{n=-\infty}^{\infty} f_n^2$$

kifejezést értjük. Ennek alapján egy *f* jelet véges energiájúnak nevezünk, ha analóg esetben

$$E = \int_0^\infty f^2(t)dt < \infty\,,$$

illetve digitális esetben

$$E = \sum_{n=-\infty}^{\infty} f_n^2 < \infty$$

teljesül.

Véges idejű jelek

Az $f(t)$, $t \in (-\infty, \infty)$ jelet véges idejűnek nevezzük, ha van olyan t_1 és t_2 $(t_1 < t_2)$ véges időpontpár, amelyre érvényes, hogy $f(t) = 0$, $\forall t < t_1$ és $\forall t > t_2$ esetén.

Korlátos jelek

Egy jelet korlátosnak nevezünk, ha a jel értelmezési tartományán a jel értékkészlete véges.

Jelek frekvenciája

A frekvencia egy esemény ismétlődésének a gyakoriságát jelenti. Azt mutatja meg, hogy egy adott időegység alatt az esemény hányszor következett be. Mértékegysége a Hz (Hertz).

Inflexiós pont

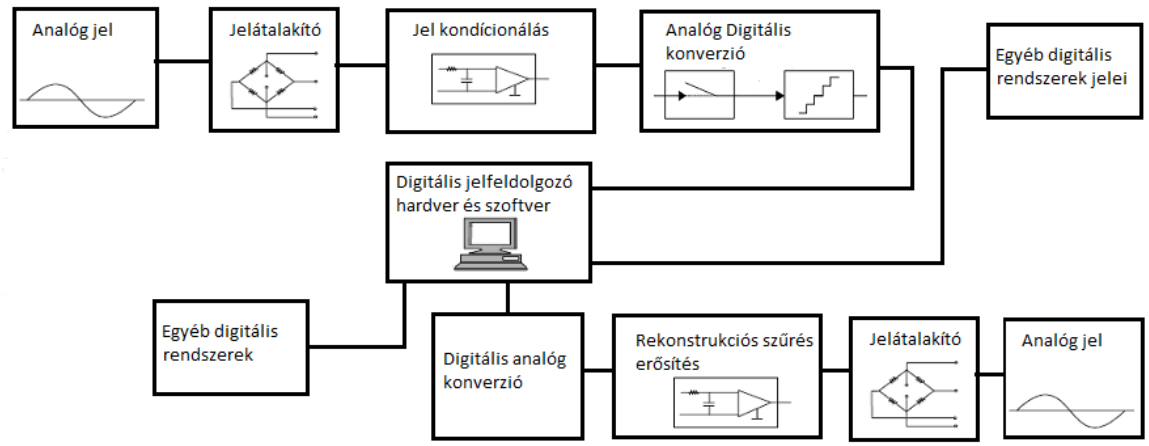
Az a pont a zajoktól megtisztított jelben, ahol a jel görbületet vált. Az inflexiós pontban a jel első deriváltjának lokális szélsőértéke van: minimum vagy maximum. Amennyiben a jel második deriváltját nézzük, annak érétke az inflexiós pontban zérus.

1.3 Kérdések, feladatok

1. Mit nevezünk analóg jelnek? Adjon rá példát!
2. Mit nevezünk digitális jelnek?
3. Mikor nevezünk egy jelet determinisztikusnak?
4. Mikor periodikus egy jel? Adjon rá példát!
5. Milyen jelek jellemzéséhez használunk statisztikai jellemzőket?
6. Mit nevezünk egy jel energiájának?
7. Mikor véges idejű egy jel?
8. Mikor tranziens egy jel?
9. Mi a frekvencia?
10. Mi az inflexiós pont?

2. A digitális jelfeldolgozás folyamata

Minden digitális jelfeldolgozó rendszer (2.1 ábra) központi eleme a hardver és a rajta futó szoftver, mely a bemeneti jelből (jelekből) meghatározza a jel információ tartalmát. A jelfeldolgozó rendszerekben ezt digitális jelfeldolgozó egységnek (angolul: *Digital Signal Processing* Unit) röviden DSP-egységnek nevezzük. A DSP-egység kivitelezése, gyakorlati megjelenése feladatfüggő. Lehet pl. személyi számítógép, melyen a DSP feladatokat ellátó szoftver fut, de lehet egy berendezésbe beépített mikrokontrolleres egység is, ilyen utóbbi pl. az otthon használatos vérnyomásmérő készülék. Mivel a DSP-egység a bemenő jel információtartalmát digitális úton (leggyakrabban szoftveres módszerekkel) határozza meg, ezért, ha valamilyen analóg jelet szeretnénk vele feldolgozni a következő lépéseket kell elvégezni:



2.1. ábra: A digitális jelfeldolgozás folyamata

A valós fizikai folyamat által szolgáltatott analóg jelet először egy jelátalakítóval analóg elektromos jellé kell átalakítani.

A következő lépés az így kapott elektromos jel kondicionálása. A kondicionálás során az elektromos jelet erősítjük, mert a jelátalakítók által szolgáltatott elektromos jel általában túl gyenge ahhoz, kábelen elvezethető legyen. Az erősítés mellett ugyanakkor zajszűrést is alkalmazni kell, mert a jelátalakítók jele általában zajjal erősen terhelt.

A kondicionált analóg elektromos jelből az analóg-digitális átalakító segítségével digitális jelet állítunk elő. Az analóg-digitális átalakítás során a mintavételezésnek nevezett eljárással a jel értelmezési tartományát (idő tengely) diszkrét pontokra bontjuk. A kvantálásnak nevezett eljárással pedig előállítjuk a mintavételi pontokban a jel számszerű értékét.

A fenti lépéssorozat eredményeképpen kapott digitális jelet aztán bemenetként átadhatjuk a DSP-egységnek. Természetesen a DSP-egységbe több csatornán, és különböző eszközökről is érkezhetnek a jelek, melyeken végrehajtja a szoftverében lekódolt algoritmust. A szoftver bonyolultsága, összetettsége erősen függ a feladat jellegétől. Mivel az információ kinyerése a bemenő jelből gyakran nem egyszerű feladat, így a jelfeldolgozó szoftver igen komoly matematikai eljárásokat, valamint a jelek valamilyen speciális tulajdonságát kihasználó *heurisztikákat* is alkalmaz. A feldolgozás eredményét a DSP-egység tárolás, megjelenítés stb. céljából továbbítja más digitális rendszereknek, Gyakori feladat, hogy a digitálisan feldolgozott jelet vissza kell alakítani analóg jellé. Ilyen feladat pl. az MP3-lejátszókban tárolt hangállományok lejátszása. Digitális jel analóg jellé alakításához a következő lépéseket kell végrehajtani:

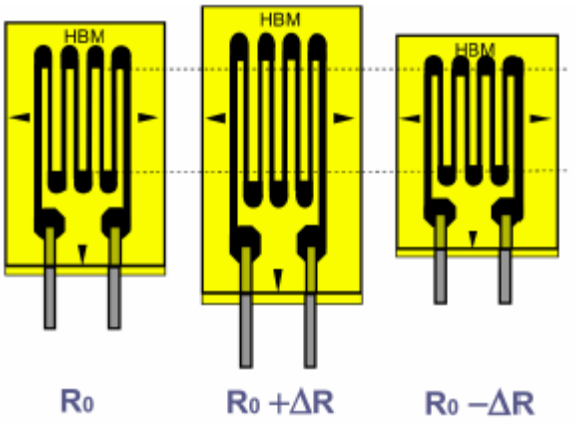
- Egy digitális- analóg átalakítón keresztül a digitális jelből analóg elektromos jelet készítünk.
- Ezt az analóg elektromos jelet egy rekonstrukciós szűrővel és egy erősítővel kondicionáljuk.
- A kondicionált analóg jelet egy jelátalakítóval a kívánt fizikai jellé alakítjuk.

Napjainkban már igen elterjedtek az egyetlen integrált áramkörként megvalósított *DSP processzorok*, melyek könnyen beépíthetőek más rendszerekbe. A DSP processzorok mindig egy adott feladatra készülnek (pl. a zajszűrés, gyors Fourier transzformáció stb.), beágyazott szoftverük ennek megfelelően kiválóan optimalizált és nagyon gyors. Analóg jelek mérésére és digitális feldolgozásukra széleskörűen használhatók különböző mikrokontrollerek.

2.1 Jelátalakítók

Amennyiben nem elektromos természetű jelet akarunk digitális módon feldolgozni, akkor a jelfeldolgozás első lépésében ezt a jelet analóg elektromos jellé (feszültséggé) alakítjuk, amelyre jelátalakítókat használunk. A kapott feszültség egy folytonos függvény, aminek

az a következménye, hogy az analóg jel a két szélső értéke között minden közbülső értéket is felvehet. A jelátalakítók különböző fizikai törvényszerűségeket használnak ki a jel elektromos feszültséggé alakítására. Például a mechanikai erők mérésére alkalmas nyúlásmérő bélyegben (2.2. ábra) a terhelés függvényében változik a bélyegbe épített huzal ellenállása.



2.2. ábra: Nyúlásmérő bélyeg elvi felépítése

JOHN TYNDALL[1]kísérletei (1865) óta tudjuk, hogy a legalább két különböző atomot tartalmazó gázok a hősugárzást jól elnyelik. Ez lehetővé teszi optikai elven működő gázérzékelők (pl. szén-monoxid, szén-dioxid) építését. Természetesen a különböző többatomos gázok különböző hullámhosszú infravörös sugarakra érzékenyek, ezeket nyelik el a legjobban. Az optikai elven működő szenzorok alapvetően három fő részből állnak: fényforrásból, hullámhossz-szűrőből, és detektorból. A technikai megvalósítást tekintve az NDIR (Nondispersive infrared) módszer a legegyszerűbb (2.3 ábra). Ennél a módszernél a fényforrásból induló infravörös fény áthalad a mintavételi csatornában levő levegőn és egy az adott hullámhosszra állított szűrön keresztül az érzékelőre jut

2.3. ábra: NDIR elven működő gázérzékelő elvi felépítése

[1] Ír fizikus (1820-1893)

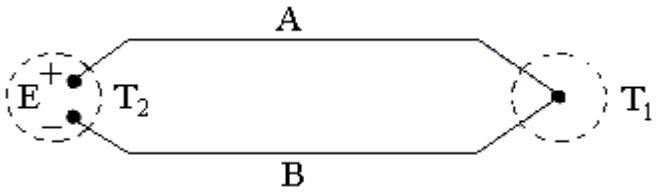
Mivel a detektált gázmolekulák számával egyenesen arányos az elnyelt infravörös fény mennyisége, ezért az érzékelőre jutó fény intenzitása fordítottan arányos a csatornában levő gáz koncentrációjával. Az intenzitást a *Lampert-Beer* törvény alapján határozhatjuk meg:

$$I = I_0 e^{kC}$$

ahol, I a mért intenzitás, I_0 a detektálni kívánt gázt nem tartalmazó csatornán mért intenzitás, k a csatornára jellemző arányossági tényező, C a detektálni kívánt gáz koncentrációja. A képletből látható, hogy olyan detektorra van szükség, mely két érzékelőt tartalmaz: egy szűrővel ellátottat és egy szűrő nélkülit. A szűrővel ellátott méri I-t, míg a szűrő nélküli pedig I_0 –t. A k értékét kalibrációval határozhatjuk meg. Ennek alapján a C koncentráció az alábbi képlettel számolható:

$$C = \frac{1}{k} \ln \frac{I}{I_0}$$

Az érzékelő működése a termo-elektromos effektuson alapul. Ennek lényege, hogy ha két különböző fémet egyik végükön egymáshoz kapcsolunk és a másik végüket szabadon hagyjuk, akkor e két vég között a két fém hőmérséklet különbségével arányos feszültség keletkezik (2.4. ábra).



1.4. ábra: A hőelem elvi felépítése

Természetesen a fényforrás kiválasztásánál ügyelni kell arra, hogy a hőingadozása kicsi legyen.

A jelátalakítókra harmadik példa a széles körben használt mikrofonok, melyek a beszédhangot alakítják át elektromos jellé. A mikrofonok különböző fizikai elveket használnak a hanghullámok elektromos jellé alakításra (piezoelektromosság, kapacitásváltozás). A kondenzátormikrofon működése azon alapul, hogy a hanghullámok a kondenzátor lemezeit rezgésbe hozzák. Ezen rezgés hatására a rezgés amplitúdójának függvényében változik a kondenzátor lemezeinek egymás közötti távolsága, ezáltal a kondenzátor kapacitása. Mivel a kondenzátoron a mérhető feszültség arányos a kapacitással, ezért a mikrofont érő

hanghullámok intenzitásának és frekvenciájának függvényében változik a kimenő feszültség.

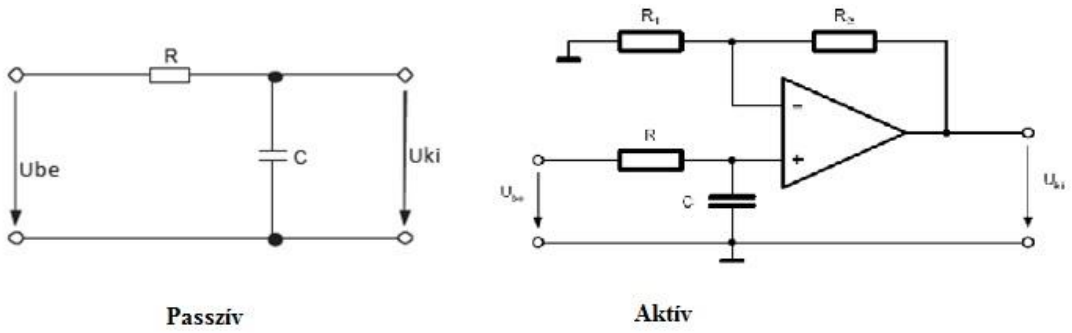
2.2 Jelkondicionálás

A jelkondicionálás során – még a jel digitalizálása előtt – az elektromos jelet erősítjük és szűrjük. Az erősítésre azért van szükség, mert a jelátalakítók által szolgáltatott elektromos jel általában túl gyenge ahhoz, kábelen elvezethető legyen. A szűrésre pedig két okból van szükség:

1. A jelátalakítók jele általában zajjal erősen terhelt, így kívánatos még az analóg oldalon annyi zajtól megszabadulni, amennyitől lehet. Maga a"zaj"a feldolgozni kívánt információt hordozó jeltől eltérő frekvenciájú és intenzitású jelek zavaró összessége. Más megfogalmazásban: minden olyan jel, amely nem része az átviendő információnak, de benne van a jelben. A jelfeldolgozás egyik központi feladata a jelben levő zajoktól történő megszabadulás. Ezt a folyamatot a jelek szűrésének nevezik. A technikában általában nem a zaj abszolút nagysága az érdekes, hanem a hasznos jel nagyságához való viszonya. Mértékegysége a dB (decibel).

2. Az A/D konverzió során az esetleges alul-mintavételezésből adódó jelmeghamisítás kizárása.

A fenti két követelmény már egyértelműen meghatározza azt, hogy az analóg oldalon szűrőként passzív, vagy aktív **aluláteresztő** szűrőt szokás használni (2.5. ábra). Az aluláteresztő szűrő olyan szűrő megvalósítás, mely a szűrő bementére adott jelet úgy módosítja, hogy a jelből, csak a kis frekvenciás komponenseket engedi át. Az átengedés itt azt jelenti, hogy a szűrőre jellemző f_0 vágási frekvenciánál kisebb frekvenciájú komponenseket engedi át változatlan amplitúdó értékkel a vágási frekvencia felettieknél viszont az amplitúdó értéket jelentősen csökkenti. Azt a frekvencia tartományt, ahol szűrő áteresztő áteresztési tartománynak, ahol pedig nem azt vágási tartománynak nevezzük.



2.5. ábra: Passzív és aktív alul-áteresztő szűrő kapcsolási rajza

A legegyszerűbb passzív aluláteresztő szűrő egy ellenállásból ás egy kondenzátorból áll. A szűrő frekvenciafüggő működésének számításához fel kell írni a kapcsolás feszültségerősítését:

$$H(j\omega) = \frac{U_{ki}}{U_{be}} = \frac{\frac{1}{j\omega C}}{R + \frac{1}{j\omega C}} = \frac{1}{1 + j\omega RC}$$

ebből az amplitúdó és fáziskarakterisztika:

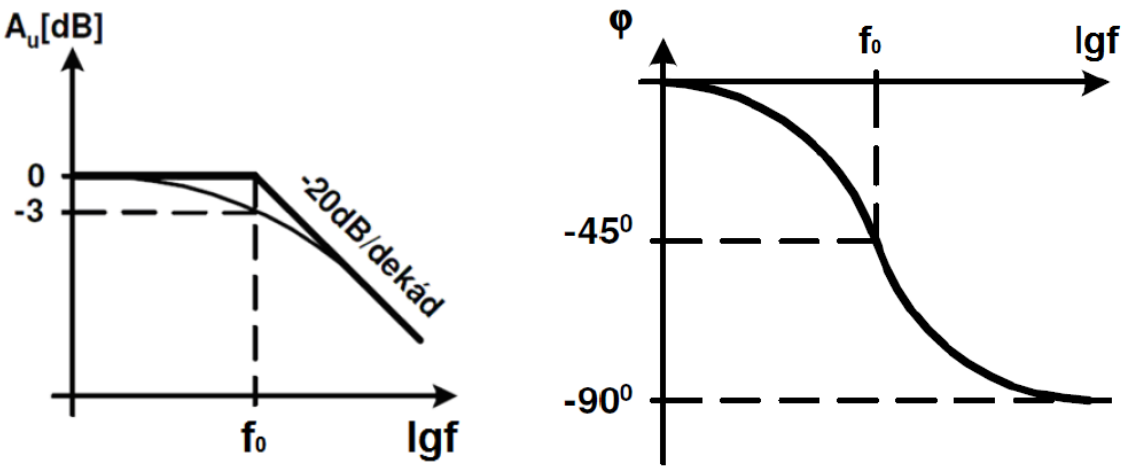
$$A(\omega) = |H(j\omega)| = \frac{1}{\sqrt{1 + \omega^2 R^2 C^2}}$$

$$\varphi(\omega) = -\arctan(\omega RC).$$

Ha ábrázoljuk az erősítés és a fáziseltolás frekvenciafüggését logaritmusos frekvenciaskálán, akkor ezt a szűrő Bode diagramjának nevezzük (2.6. ábra). Ha az erősítés diagramját vizsgáljuk, akkor azt látjuk, hogy az erősítés frekvenciafüggése aszimptotikusan kis frekvenciákon 1-el (0 dB) közelíthető, nagy frekvenciákon pedig $\frac{1}{\omega RC}$-vel. A két egyenes metszéspontja a törési (vágási) frekvencia:

$$f_0 = \frac{1}{2\pi RC}$$

Az első fokú aluláteresztő szűrő -20 dB/dekád meredekséggel megy át az áteresztési tartományból a vágási tartományba, ha ennél meredekebb átmenetre van szükség, akkor magasabb fokszámú szűrőket kell használni.

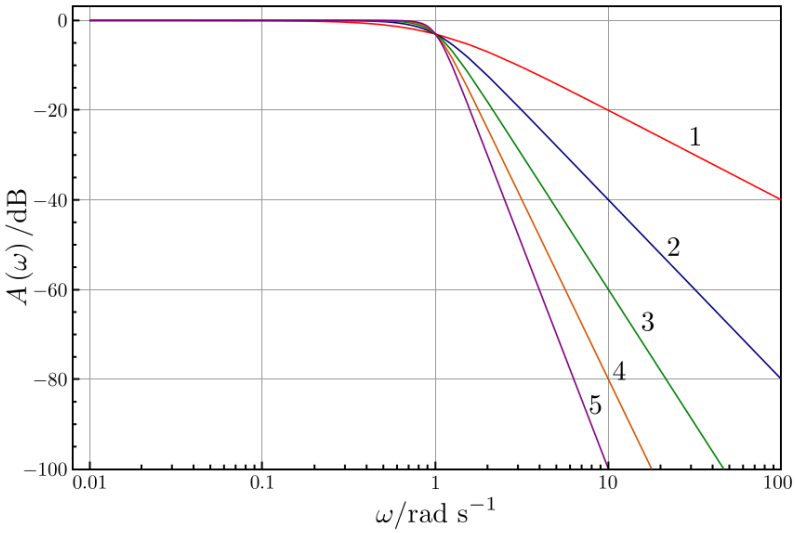


2.6. ábra: Aluláteresztő szűrő Bode diagramja

A szűrő típus kiválasztásánál fontos, hogy az áteresztési tartományban mennyire sima a frekvencia menet, a vágási frekvenciánál mennyire meredeken vág, valamint mennyire bonyolult az áramköri megvalósítása. A gyakorlatban Butterworh típusú szűrőket használnak erre a feladatra. A szűrő átviteli függvénye:

$$H(j\omega) = \frac{1}{\sqrt{1+\varepsilon^2\left(\frac{\omega}{\omega_c}\right)^{2n}}}$$

ahol $\omega=2\pi f$, ω_c a vágási frekvencia, ε az áteresztési tartományban a frekvencia maximális amplitúdója, n a szűrő fokszáma. A 2.7. ábrán a különböző fokszámú Butterworth szűrők átviteli karakterisztikája látható.



2.7. ábra: Különböző fokú Butterworth szűrők átviteli karakterisztikái

Ezt a szűrőt a szakirodalomban *antialiasing* szűrőnek is nevezik, ugyanis ez a szűrő végzi el az analóg jel sávkorlátozását a jel digitalizálása előtt. Ez azért fontos, mert a helyesen megválasztott vágási frekvencia kizárja az aliasing (alul-mintavételezés) effektust. Magát az effektust a **2.4 MINTAVÉTELEZÉS** című alfejezetben részletesen tárgyaljuk.

2.3 Analóg-digitális átalakítás

Az analóg-digitális átalakítók (*A/D konverter*) a bementi feszültséget a feszültség nagyságával arányos egész számmá alakítják. Azonban az A/D átalakítóknak több olyan paramétere is van, amely befolyásolja a kapott számot:

- **Referencia feszültség** (U_{ref}) az A/D konverter által elfogadott maximális elfogadott analóg feszültségérték.
- **Teljes tartomány** (Full Scale = FS) a bemeneti analóg jelnek azt az intervallumát jelenti, amelyben az átalakító telítésmentesen működik. Szokás vezérlési tartománynak is nevezni.
- **Felbontás** (Resolution) adja meg azt, hogy hány egymástól eltérő binárisan kódolt kimenő érték jeleníthető meg a digitális oldalon. Mértékegysége a bit. Egy 4 bites A/D átalakító $2^4 = 16$ különböző értéket jelenít meg a kimenő oldalon. A személyi számítógép hangkártyájában alkalmazott A/D konverter felbontása 16 bit. Az audio CD-k gyártásánál alkalmazott pedig 20 bit. Az U bemeneti feszültségnek megfelelő Z konvertált értéket a következő egyenlet adja meg:

$$Z = \left\lfloor \frac{U 2^b}{U_{ref}} + 0.5 \right\rfloor,$$

- **Legkisebb szignifikáns bit** (Least significant bit = LSB) adja meg a bementi jel egy bitnyi információ megváltozásához szükséges átlagos megváltozását. Ha *N* jelöli az A/D konverter felbontását bitekben, akkor $N = 2^b$.

$$LSB = \left\lfloor \frac{FS}{2^b - 1} \right\rfloor$$

- **Sávszélesség** a bemeneti (analóg) függvény frekvenciájának egy olyan tartománya, amelyen belül az A/D konverter fogadja a bemeneti jelet. Más összefüggésben ez azt jelenti, hogy az A/D konverter úgy kezeli a bemeneti jelet, mint egy olyan függvényt, amelyre igaz az, hogy a sávszélesség tartományán kívüli frekvenciákra a függvény értéke nulla.
- **Mintavételi frekvencia** adja meg azt a másodpercenkénti maximális gyakoriságot, amellyel az eszköz működni tud. Mértékegységét általában SPS (Sample per Second) jelöli.

Konverziós idő rögzíti az egy minta átalakításához szükséges időintervallumot.

Példák A/D konverter paramétereinek számítására

1. Legyen a teljes mérési tartomány [-5V,15V], és tekintsünk egy 8 bites felbontású A/D konvertert. Ekkor a legkisebb bit megváltozásához szükséges átlagos bemeneti feszültségváltozás:

$$LSB = \left\lfloor \frac{FS}{2^b - 1} \right\rfloor = \left\lfloor \frac{20}{2^8 - 1} \right\rfloor = \left\lfloor \frac{20}{1024} \right\rfloor = 0{,}01955V = 19{,}55mV.$$

2. Tegyük fel, hogy az A/D konverter 3 bites felbontást fog előállítani, azaz $N = 3$, és a maximálisan figyelembe vehető feszültség 24 V ($U_{ref} = 24$). Ekkor az A/D konverzió során $\Delta U = U_{ref} / 2^N = 24 / 8 = 3$. Ez azt jelenti, hogy 3 egységenként „ugrik” az A/D konverter bináris értéke. Tekintsünk két példát:

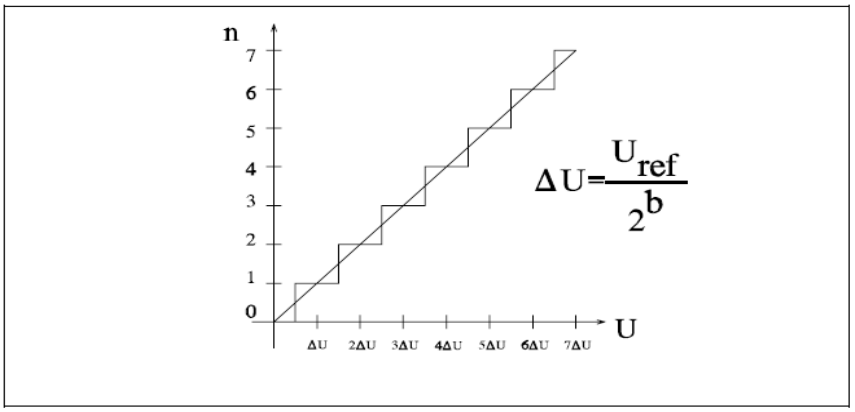
   A. Legyen a bemenő feszültség 1,7 V, akkor a kimenő bináris érték

$$Z = \left\lfloor \frac{U 2^b}{U_{ref}} + 0.5 \right\rfloor = \left\lfloor \frac{1{,}7 \cdot 2^3}{24} + 0{,}5 \right\rfloor = \left\lfloor \frac{13{,}6}{24} + 0{,}5 \right\rfloor = 1.$$

B. Legyen a bemenő feszültség 13,1 V. Ekkor

$$Z = \left\lfloor \frac{U 2^b}{U_{ref}} + 0.5 \right\rfloor = \left\lfloor \frac{13{,}1 \cdot 2^3}{24} + 0{,}5 \right\rfloor = \left\lfloor \frac{104{,}8}{24} + 0{,}5 \right\rfloor = 4.$$

Az analóg-digitális konverzióként előálló „lépcsős” függvényt mutatja be a 2.8. ábra.



2.8. ábra: Az A/D konverzió függvénye.

Az ábrából jól látszik, hogy a generált lépcsős függvény ugrási pontjai az $i\Delta U+\Delta U/2$ pontokban vannak, ahol $i = 0,1,\ldots, 2^N-1$.

Felépítéstől függetlenül az A/D átalakító a következő műveleteket hajtja végre a konverzió során:

- mintavételezés – adott időközönként mintát vesz az analóg jelből,
- kvantálás – a mintavételi időpontokhoz diszkrét feszültségértéket rendel,
- kódolás – a kvantált amplitúdó értéket bináris számmá alakítja.

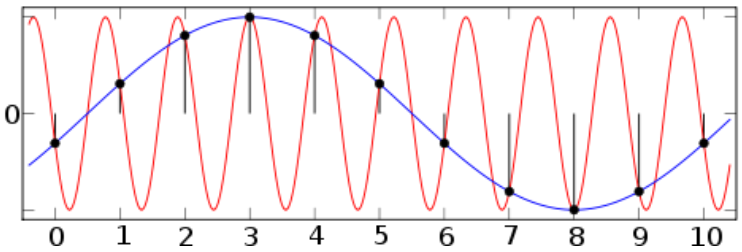
2.4 Mintavételezés

Mint korábban már említettük, a mintavételezés azt jelenti, hogy a folytonosan érkező analóg jelből meghatározott időnként (diszkrét időpontokban) kiolvassuk az analóg jel értékét. A digitális jelfeldolgozásban a mintavétel a legtöbb esetben egyenletes. Ez azt jelenti, hogy az időtengelyt *ekvidisztáns* módon osztjuk fel, és az osztópontokban történik az analóg jelből a mintavétel. Ekkora az *f(t)* diszkrét mintaértékeinek f_n sorozata a következőképpen adható meg:

$$f_n = f(t_0 + kT),$$

ahol *k* egész szám és *T* a mintavétel periódus ideje.

Felvetődik a kérdés, hogy milyen sűrűn kell mintát venni az analóg jelből, azaz mekkora legyen a periódus idő, más néven a *mintavételi frekvencia*? Az 2.9. ábra az *aliasing* jelenséget demonstrálja. Az ábrán a fekete színű pontokban történt a mintavételezés. Nyilvánvaló, hogy az ezekben a pontokban vett minták elemei lehetnek mind piros, mind a kék görbének. Tehát nem dönthető el egyértelműen, hogy melyik jelből vettük a mintát. Ezt a jelenséget nevezzük alul mintavételezésnek vagy más néven *aliasing* jelenségnek.



2.9. ábra: Az aliasing jelenség

Az aliasing effektus furcsa jelenségeket is tud produkálni. Tipikus példa a régi westernfilmeken a hátrafelé forgó kerék. Adódik a kérdés, hogy milyen sűrűn kell akkor a mintavételezést csinálni. A választ a **SHANNON[2]-NYQUIST** féle mintavételezési tétel adja meg: Egy analóg jel akkor állítható maradéktalanul helyre véges számú mintából, ha a mintavételi frekvencia legalább kétszerese a jelben előforduló legnagyobb frekvenciának:

$$f_{\min\ tav;tel} \geq 2 f_{\max}.$$

[2] Claude Shannon (1916-2001) amerikai híradástechnikus

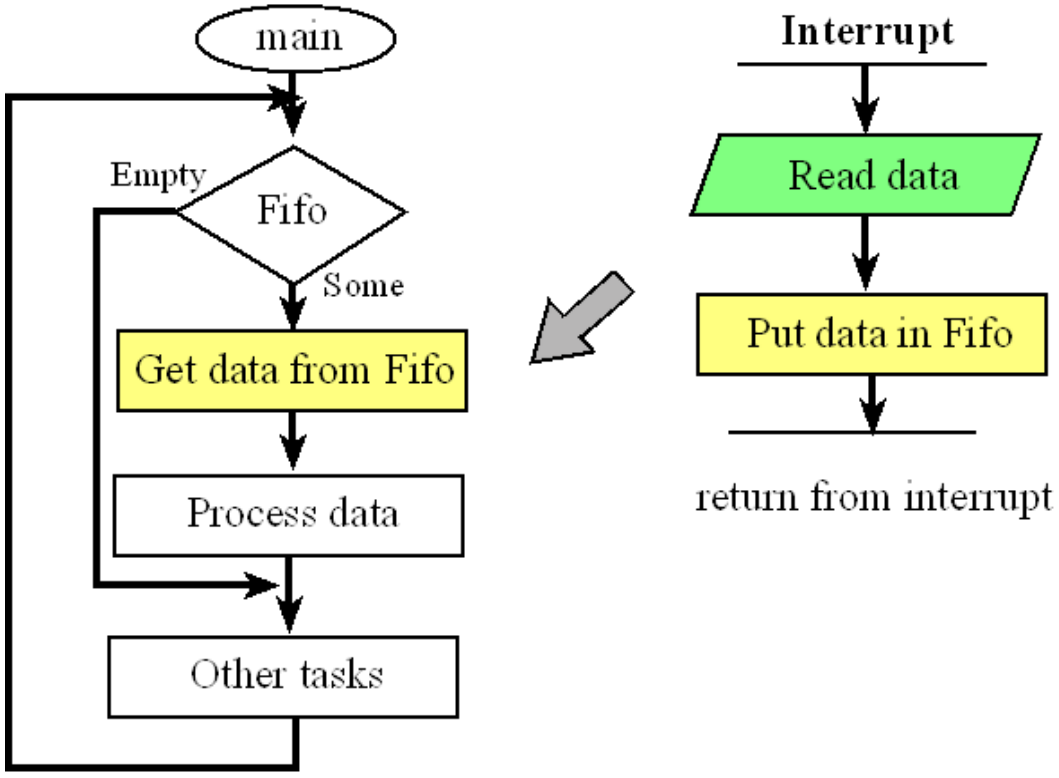
A gyakorlatban nem szoktak megelégedni meg a kétszeres szorzóval, hanem nagyobb biztonságra törekednek. Általában úgy állítják be az értékeket, hogy $f_{\min tavétel} \geq 10 f_{\max}$ legyen. A gyakorlatban a mintavételi frekvencia meghatározásánál figyelembe kell venni az A/D-átalakító konverziós idejét. Ügyelni kell arra is, hogy ha a mintavételezési frekvenciát túl nagyra választjuk, akkor nő a keletkezett adatok mennyisége, amely értelemszerűen lassítja a feldolgozási sebességet, és a tárolásnál nagyobb tárterületet igényel, ugyanakkor a hordozott információ szempontjából nem ad jelentős többletet.

2.5 Kvantálás, kódolás

Az analóg-digitális átalakítás következő lépése a mintavételezett jel folytonos értékkészletű amplitúdójának diszkrét értékkészletűvé alakítása. Ennek során az AD-átalakító az eredeti folytonos értékkészletet kvantálási lépcsőkkel intervallumokra bontja. Minden intervallumban referencia értékként kijelöl egy kvantálási szintet, és a pillanatnyi amplitúdóhoz ezt a szintet rendeli. Ezáltal a kvantálási folyamat egy lépcsős függvénnyel írható le Nyilvánvaló, hogy az A/D-átalakító felbontása függ a kvantálási szintek számától.

A digitális jelfeldolgozásban fontos szempont, hogy az A/D-konverter a kvantálás eredményét olyan alakban adja, hogy azokat a különböző CPU-kon gyorsan és egyszerűen lehessen feldolgozni. A kvantálás eredményét az A/D-konverterek egész szám formájában szolgáltatják.

Szoftver szempontból nézve az adatvesztést elkerülendő az AD konverternek megszakítás vezérelten kell működnie. Ez azt jelenti, hogy a rendszer egyik időzítőjét a mintavételi frekvenciára programozzuk. Az időzítő minden egyes aktiválásakor az AD konverter regiszterében levő értéket elhelyezzük egy körkörös FIFO pufferba. A fő program pedig folyamatosan futva ellenőrzi, hogy a van –e a FIFO-ban érvényes adat. Ha van, akkor kiolvassa és feldolgozza (2.10. ábra). Olyan beágyazott rendszernél, ahol a rendszernek van operációs rendszere, ott a gyártó driver szinten elrejti a megszakításkezelést és egy API felületet biztosít a FIFO-ban levő adatok kezelésére.



2.10. ábra: ADC adatok megszakítás vezérelt feldolgozása

Forrás: http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm

2.6 A DSP egység

A DSP-egység tartalmazza azt, a leggyakrabban szoftveres módon megvalósított logikát, melynek feladata az információ kinyerése és feldolgozása a digitalizált jelből. A szoftveres megvalósítás előnyei nyilvánvalóak, ugyanis a szoftvereket könnyen lehet cserélni, az adott feladathoz igazítani. Természetesen a hibamentesség mellett törekedni kell a minél gyorsabb, lehetőleg valósidejű futásra.

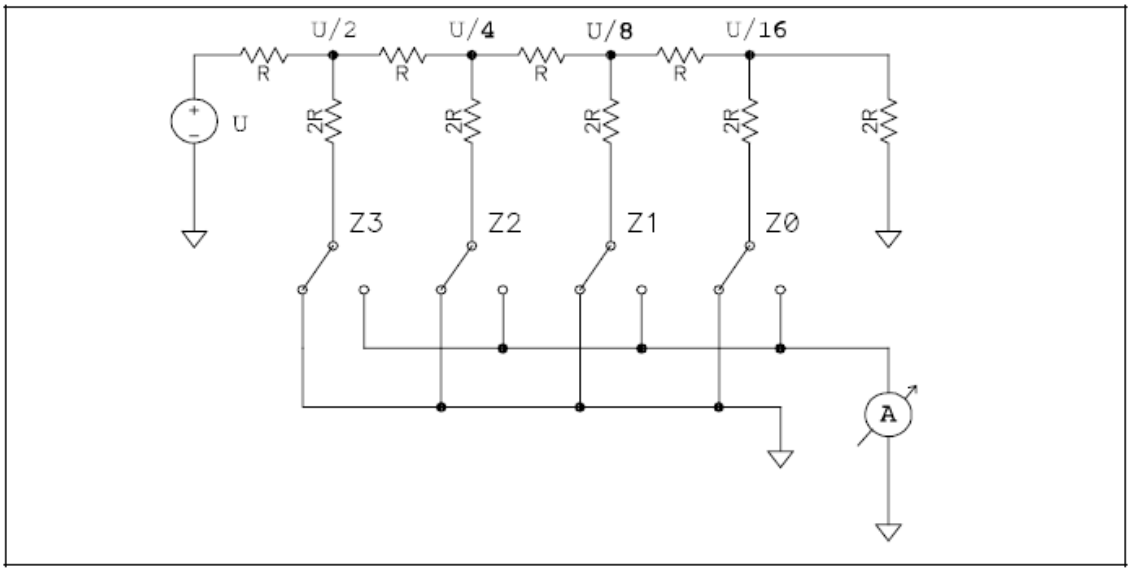
A fentekből látszik, hogy a DSP-egységeknek gyakran igen bonyolult algoritmusokat kell végrehajtaniuk a kívánt cél érdekében. Mint említettük, ezért ezeket általában célszerű szoftverként megvalósítani. Ennek előnyei nyilvánvalóak. A szoftvereket könnyű az adott feladathoz igazítani. Természetesen a DSP-egységekben megvalósított szoftvereknél törekedni kell a minél gyorsabb – lehetőleg valós idejű – futásra.

2.7 Digitális-Analóg átalakítás

Gyakran előfordul az a feladat, hogy a DSP egység kimenő jelét analóg jellé kell konvertálni. Gondoljunk arra, hogy a digitálisan megtisztított hangfelvételt meg szeretnénk hallgatni. Ekkor a digitális jelet analóg jellé kell alakítani. A D/A konverter feladata, hogy a digitális jelben megjelenő egész számmal arányos feszültséget generáljon. Matematikailag ez a következő formulával írható le:

$$U = Z \frac{U_{ref}}{2^b}$$

ahol Z egy –maximum b biten ábrázolható – bemenetként kapott egész szám és U_{ref} a referencia feszültség. Vegyük észre, hogy matematikai szempontból az A/D konverter felbontásánál bemutatott képlet és a fenti formula – a kerekítéstől eltekintve – egymás inverz függvényeinek tekinthetők. A digitális jelek analóg (feszültség) jellé alakításához a leggyakrabban az R-2R ellenállás-hálózatos sémát alkalmazó készülékeket használják (2.11 ábra).



2.11. ábra: R-2R DAC konverter

2.8 Kérdések, feladatok

1. Mi a jelkondicionálás feladata? Milyen egységekből áll?

2. Mi az antialiasing szűrő? Milyen szabály alapján kell megválasztani a vágási frekvenciáját?

3. Az A/D konverternek milyen feladatokat kell elvégeznie?

4. Mit jelent a mintavételezés?

5. Mondja ki a Shannon féle mintavételezési tételt!

6. Mi az aliasing effektus? Mondjon rá példát!

7. Milyen módszereket használhat egy A/D konverter az amplitúdó kvantálásra?

8. Mi a BCD kódolás?

9. Hogyan ábrázoljuk az előjeles egész számokat kettes komplemens alakban?

10. Hány byte adatmennyiség keletkezik az A/D konverter kimenetén 10 s alatt, ha a mintavételi frekvencia 144,4 kHz, és a kvantálás hossza 16 bit?

11. Egy jelkondícionáló egység aluláteresztő szűrője 250 Hz vágási frekvenciával dolgozik. A digitalizáló egységben legalább mekkora mintavételezési frekvenciát kell alkalmazni?

3. Jelfeldolgozási módszerek

A jel legfontosabb jellemvonása a jel által hordozott információ. A jelfeldolgozás feladata ezen információtartalom meghatározása. Ehhez a jeleken különböző műveleteket kell elvégezni Ilyen művelet pl. a jelek zajoktól való megszabadítása, a jelek rekonstruálása, a jelparaméterek meghatározása, a jelek tárolása, megjelenítése stb. E műveletekkel szemben elvárás, hogy mind algoritmikus, mind elektronikai szemszögből könnyen kezelhetőek legyenek. A jelfeldolgozás kétféle stratégia szerint történhet:

- abban az esetben, ha a jel időbeli viselkedésén akarunk változtatni, akkor **időalapú** jelfeldolgozásról beszélünk.
- abban az esetben, ha a feldolgozás alapja a jelben levő frekvenciákon végzett művelet, akkor **frekvenciaalapú** jelfeldolgozásról beszélünk.

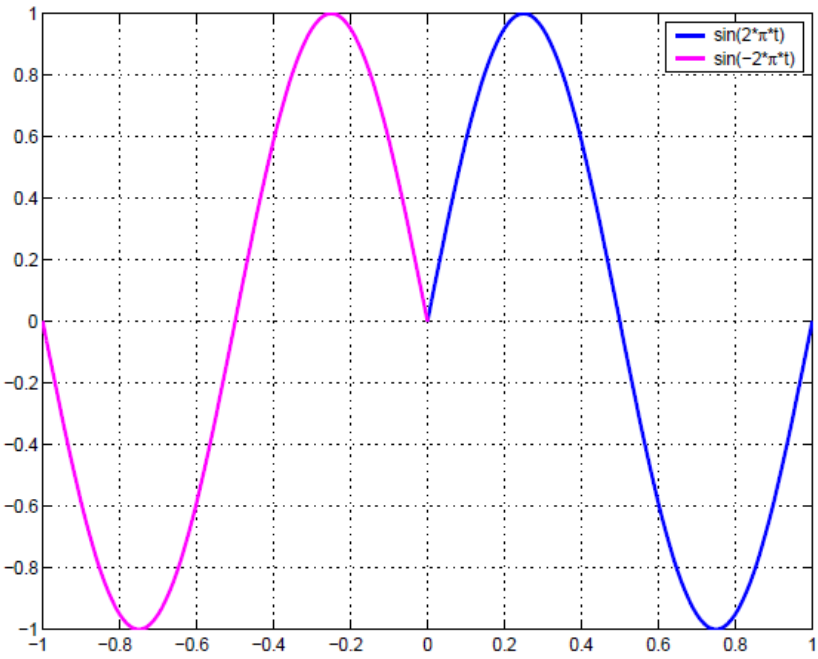
Az időalapú jelfeldolgozásnál a mért jel (idősor) mutathat trendet, szezonális ingadozást, valamint ciklikusságot.

3.1 Időalapú jelfeldolgozás

Az időlapú jelfeldolgozásnál két fő csoportra oszthatjuk a jelfeldolgozási műveleteket: időtengely mentén módosító és amplitúdó tengely mentén módosító.

Reflexió

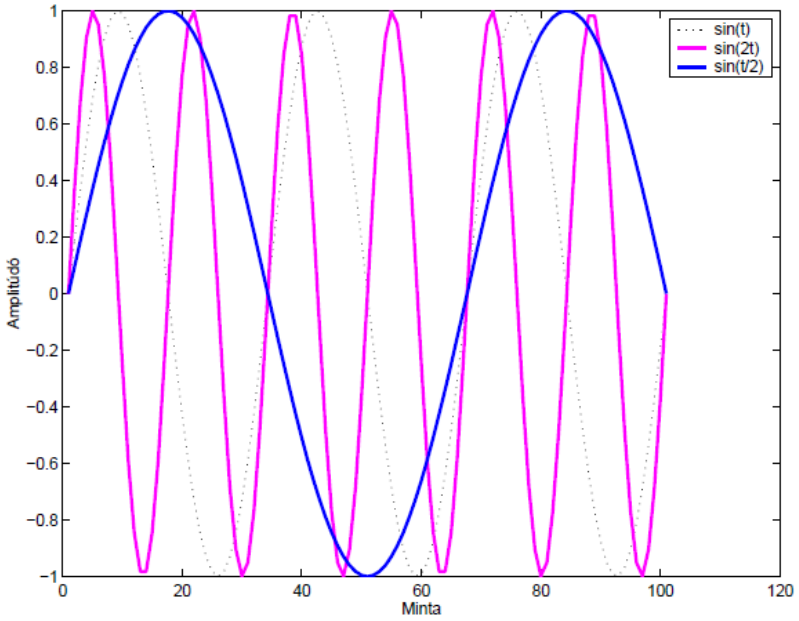
A jel reflexióján az amplitúdó tengelyre való tükrözését értjük. Matematikai alakban analóg jel esetén $y(t) = f(-t) \quad \forall t$-re, digitális esetben $y_n = f_{-n} \quad \forall n$-re.



3.1. ábra Reflexió. A kék színű az eredeti görbe, a lila az amplitúdó tengelyre tükrözött [2].

Időintervallum skálázás

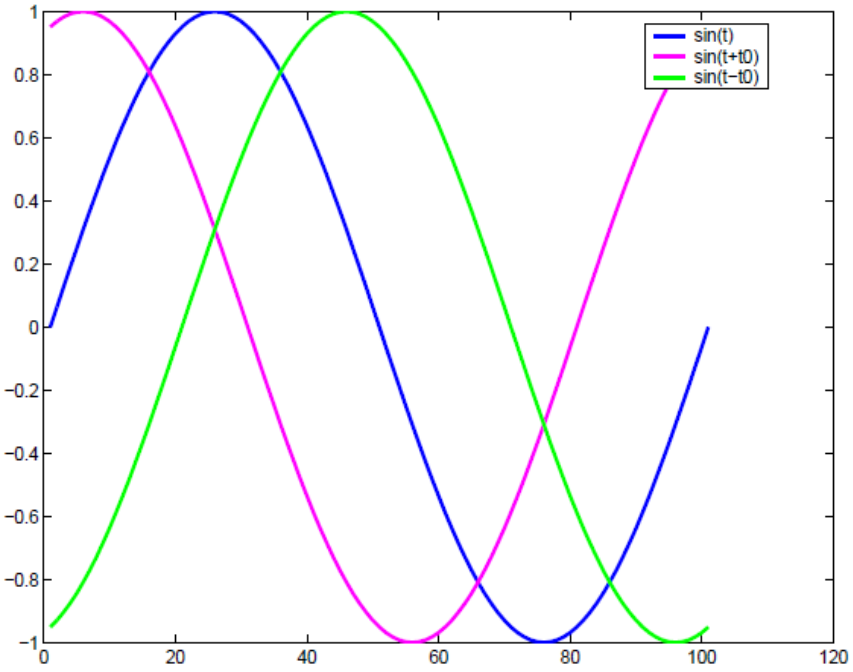
Matematikai formában megadva analóg jel esetén $y(t) = f(at) \quad \forall t$-re, digitális esetben $y_n = f_{an} \quad \forall n$-re.



3.2. ábra: Időintervallum skálázás [2].

Időeltolás

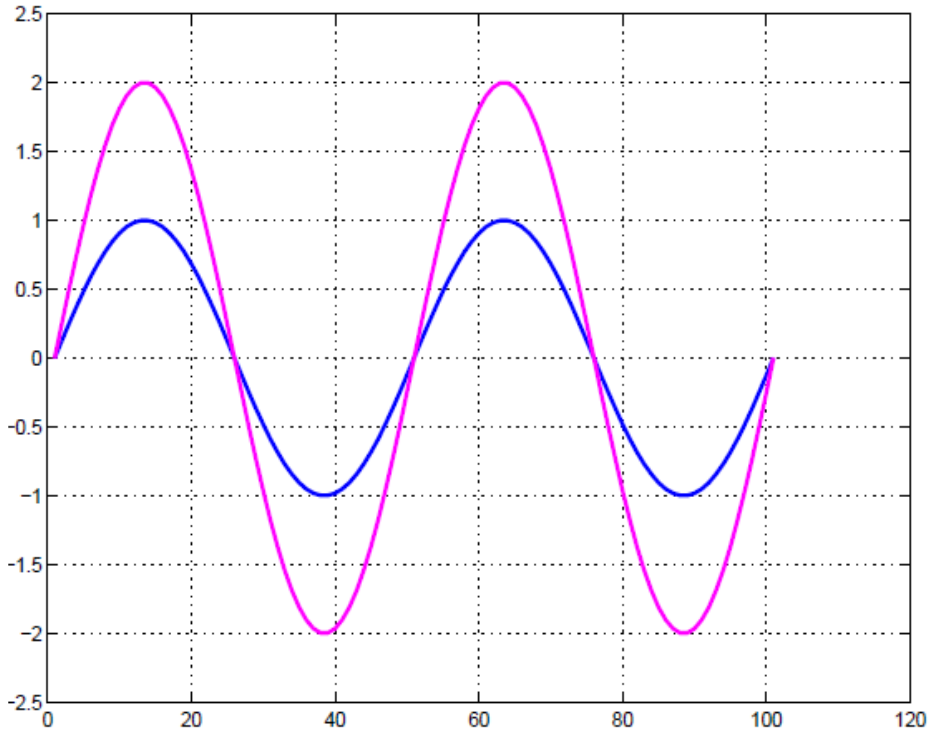
Matematikai formában analóg jel esetén $y(t) = f(t - t_0)$ $\forall t$-re, digitális esetben $y_n = f_{n-n_0}$ $\forall n$-re.



3.3. ábra: Időeltolás [2].

Valós számmal való szorzás (amplitúdó skálázás)

Az f jelnek az a valós számmal való szorzásán a jel amplitúdó értékeinek és az a valós számnak a szorzatát értjük. Matematikai formában analóg jel esetén $y(t) = af(t)$ $\forall t$-re, digitális esetben $y_n = af_n$ $\forall n$-re. Technikai értelemben a valós számmal való szorzás a jel erősítésének felel meg.



3.4. ábra: Valós számmal való szorzás (amplitúdó skálázás) [2].

Invertálás

A jel invertálásán a jelnek a -1-el történő szorzását értjük. Matematikai formában analóg jel esetén $y(t) = -f(t) \quad \forall t$-re, digitális esetben $y_n = -f_n \quad \forall n$-re.

Összeadás, kivonás

Az *f* és *g* jel összegén minden időpontban az adott időponthoz tartozó amplitúdók összegét értjük. Matematikai alakban analóg jel esetén $y(t) = f(t) \pm g(t) \quad \forall t$-re, digitális esetben $y_n = f_n \pm g_n \quad \forall n$-re.

Numerikus-differenciálás

Egy jel lokális szélsőértékeinek, inflexiós pontjainak meghatározásához szükség van pontonként a jel első, második és esetleg harmadik deriváltjának az ismeretére. A jel differenciálhányados függvényének zérushelyei a jel lokális szélsőértékeit jelöli ki. A második derivált zérushelyei pedig a jel inflexiós pontjait határozza meg, amennyiben a harmadik derivált az adott helyen nem zérus. Digitális jelek estén a jel analitikusan általában nem ismert, így numerikus-deriválást kell végezni. A derivált értéket az adott pontban differencia hányadossal közelítjük.

Első derivált:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Amennyiben *h*-t elég kicsinek választjuk ($h \to 0$), akkor differencia hányados jól közelíti a derivált értékét.

Második derivált közelítő képlete:

$$f''(x) \approx \frac{f(x) - 2f(x+h) + f(x+2h)}{h^2}$$

Jelek konvolúciója

A konvolúció az átlag fogalmának az általánosítása. Sztochasztikus jeleknél gyakran felmerül az a kérdés, hogy tudunk –e becslést mondani a jel következő értékére. Természetesen adódik az az ötlet, hogy ezt az értéket az őt megelőző jelértékek átlagából becsüljük. A kérdés csak az, hogy hány értékből számoljuk az átlagot. Általában a jel egy sokkal korábbi értéke csak kis súllyal szól bele a becsülni kívánt értékbe, így az átlagolást csak az adott pont környezetére érdemes elvégezni. Definíció szerint *mozgó-átlagnak* nevezzük, ha egy adatsorból visszamenőleg egy részhalmazt átlagolunk.

$$y_n = \frac{1}{N} \sum_{i=0}^{N-1} y_{n-i}$$

A fenti képletben N jelöli a mozgó-átlag sugarát, azaz azt, hogy hány pontból kell átlagolni. A mozgó-átlag képlete felfogható két vektor skaláris szorzataként. Az egyik vektor egy N elemű csupa $\frac{1}{N}$ értékből áll. A másik pedig a jel adott intervallumba eső értékei. Az első vektort az átlagolás *kernelének* nevezzük. Tovább általánosíthatjuk az átlagolást, ha a kernel vektor értékei is tetszőleges értékek lehetnek. Ekkor

$$y_n = \sum_{i=0}^{N-1} k_i y_{n-i}$$

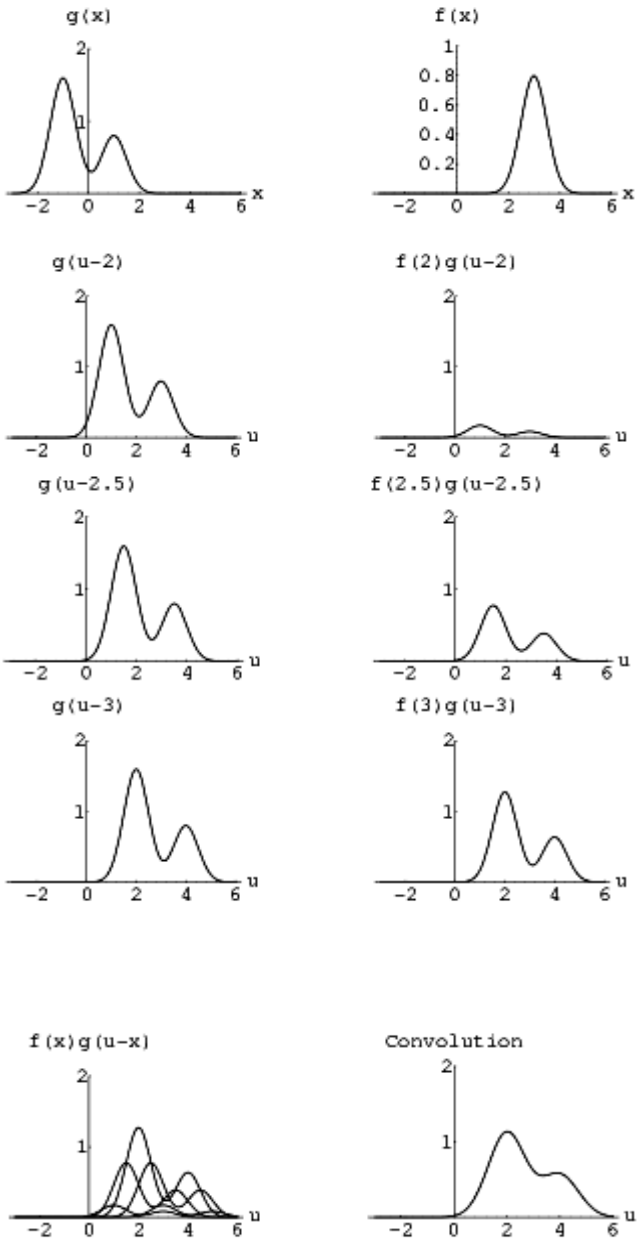
Ahol az egyenletben k_i a kernel vektor i-edik eleme.

Analóg jelek estén a folytonos függvényekre kell a konvolúciót kiterjeszteni. Ekkor a szummából integrál lesz és definíció szerint a $(-\infty, \infty)$ intervallumon értelmezett *f, g* integrálható függvények konvolúcióján az

$$f * g = \int_{-\infty}^{\infty} f(x)g(u-x)dx$$

integrált értjük. A konvolúciós integrálban két változónk van – *x* és *u* – és az integrálást az *x* változó szerint végezzük, ezért az *u* az integrálás szempontjából konstansnak számít. Ez azt eredményezi, hogy a szorzatfüggvény értékét kell kiszámítanunk véges sok helyen, és ezeket kell összegezni. A 3.5 ábrán látható ábrasorozat jól mutatja ezt a folyamatot. Az első sorban szerepel a két függvény. Esetünkben *g(x)* a bemenő jel függvénye és *f(x)* egy programozott – a szűrőben „zsinórozott" – simító függvény, ami úgy lett tervezve, hogy a bemenő függvény „kilengéseit" a kimenő jel oldalán simítsa el. A következő 3 sor szolgáltatja az *x=2, x=2.5 és x=3* értékekhez tartozó függvényértékeket a bal oldali ábrán, és a hozzá tartozó szorzatfüggvény-értékeket az adott sor jobb oldalán. A jobb oldalon a függvények alatti területek felelnek meg az integrál-közelítés egy-egy kis téglalapjának. Ezeket kell összegezni ahhoz, hogy a konvolúció értékét egy jó közelítéssel megkapjuk. Az utolsó sor bal oldali ábráján az egyes „részterületek" vannak felrajzolva, és a jobb oldalon kapjuk a konvolúciónak – mint integrálfüggvénynek – egy olyan közelítését, amelyet a jobb oldali három függvények és az f(x) függvény abszcissza-pontjaihoz tartozó függvényértékek összegzéséből kaphatunk meg. Az utolsó sor a konvolúció eredményeként létrejött simítás eredményét mutatja abban az esetben, amikor három pontos felbontást alkalmaztunk. A simítás eredményét jól láthatjuk, ha összehasonlítjuk az ábra első sorának bal oldalán látható bemenő jelet a konvolúció eredményeként kapott kimenő jellel.

Ha úgy ítéljük meg, hogy a közelítés nem elég jó, akkor a három pont helyett a [2, 3] intervallumot több részre kell bontanunk. Ennek megfelelően finomabb simítást is kapnánk.



3.5. ábra: Példa a konvolúció kiszámítására
(forrás: The convolution theorem and its application).

A mozgó-átlag és diszkrét konvolúció a jelfeldolgozásban kiválóan használható a jelek simítására. Ne felejtsük el azonban azt, hogy az átlagolás mindig csökkenti a jel fel és lefutásának meredekségét, Minél nagyobb az N értéke, annál laposabb lesz a jelünk.

A következőkben megadjuk a fent bemutatott jelfeldolgozási módszerek egy lehetséges implementációját:

```
class SignalProcessing
 {

   public double[] Reflex(double[] y)
   {
     double[] ret = new double[y.Length];

     for (int i = 0; i < ret.Length; i++)
     {
          ret[i] = y[y.Length - 1 - i];
     }

     return ret;
   }

        public double[] TimeScaling(double[] y, double scalefactor)
        {
            double[] ret = new double[y.Length];


            return ret;
        }

        public double[] TimeShift(double[] y, int t0)
        {
            double[] ret = new double[y.Length-t0];

            for (int i = 0; i < ret.Length-t0; i++)
            {
                ret[i] = y[i+t0];
            }

            return ret;
        }

        public double[] MultiplyByScalar (double[] y, double a)
        {
            double[] ret = new double[y.Length];

            for (int i = 0; i < ret.Length; i++)
            {
                ret[i] = a * y[i];
            }

            return ret;
        }

        public double[] Add(double[] y, double[] z)
        {
            double[] ret = new double[y.Length];

            for (int i = 0; i < ret.Length; i++)
            {
                ret[i] =  y[i]+z[i];
            }
```

```
        return ret;
    }

    public double[] Sub(double[] y, double[] z)
    {
        double[] ret = new double[y.Length];

        for (int i = 0; i < ret.Length; i++)
        {
            ret[i] = y[i] - z[i];
        }

        return ret;
    }

    public double[] Derive1(double[] y, int h)
    {
        double[] ret = new double[y.Length-h];

        for (int i = 0; i < ret.Length; i++)
        {
            ret[i] = (y[i+h] - y[i])/h;
        }

        return ret;
    }

    public double[] Derive2(double[] y, int h)
    {
        double[] ret = new double[y.Length - 2*h];

        for (int i = 0; i < ret.Length; i++)
        {
            ret[i] = (y[i + 2*h] - 2*y[i+h]+y[i]) / (h*h);
        }

        return ret;
    }

    public void movingAverage(double[] array, int window)
    {
        double sum = 0;

        for (int i = 0; i < (array.Length - window); i++)
        {

            sum = 0;
            // Loop through window numbers from current i position
            for (int j = i; j < i + window; j++)
            {
                sum += array[j];            // Increment sum.
            }

            // Calculate moving average.
            double movingAverage = sum / window;
            array[i] = movingAverage;
        }

        for (int i = array.Length - window; i < array.Length; i++)
        {
            sum = 0;
```

```
                for (int j = 0; j < window; j++)
                {
                    sum += array[i - j];
                }
                array[i] = sum /window;
            }
        }

    public void Convolution(double[] array, double [] kernel)
    {
       double sum = 0;
       for (int i = 0; i < (array.Length -kernel.Length); i++)
       {

          sum = 0.0;
          // Loop through window numbers from current i position
          int k = 0;
          for (int j = i; j < i + kernel.Length; j++)
          {
              sum += kernel[k] * array[j];
              k++;
          }
          array[i] = sum/kernel.Length;
      }

      for (int i = array.Length - kernel.Length; i < array.Length; i++)
      {
        sum = 0;
        for (int j = 0; j < kernel.Length; j++)
        {
          sum +=kernel[j]* array[i - j];
        }
        array[i] = sum / kernel.Length;
      }
}
```

3.2 Frekvencialapú jelfeldolgozás

Jelfeldolgozási szempontból az analóg jelek az időben keletkeznek, és az időben változnak. Azonban sok esetben kényelmesebb, ha az időbeli változás helyett a jel frekvencia szerinti változását vizsgáljuk. Ebben a részben bemutatjuk azokat a módszereket, amelyek lehetővé teszik az időtérből a frekvenciatérbe és onnan visszafelé történő transzformációt.

3.2.1 Periodikus jelek Fourier sora

A Fourier transzformáció feladata, hogy egy összetett jelet felbontsunk komponenseire, azaz egyszerűbb jól ismert függvényértékek összegeivel közelítsünk. A módszer Joseph Fourier-től származik (3.6. ábra). Ő vette észre, hogy minden periodikus $f(t) = f(t \pm T)$ függvény egyértelműen felírható megfelelő amplitúdókkal és fázisállandókkal bíró harmonikus rezgések összegeként.

3.6. ábra: Jean Baptiste Joseph Fourier (1768-1830)

A harmonikus rezgés az nem más, mint a tiszta szinuszos rezgés, melynek matematikai alakja:

$$f(t) = A\sin(\omega t + \phi),$$

ahol A a rezgés amplitúdója, ω a rezgés körfrekvenciája és ϕ a fázisállandó.

Ennek alapján a T periódus idővel rendelkező periodikus függvényekre vonatkozó Fourier tétel az alábbi matematikai formulával írható fel:

$$f(t) = A_0 + A_1 \sin(\omega t + \phi_1) + A_2 \sin(2\omega t + \phi_2) + ... + A_n \sin(n\omega t + \phi_n) + ...$$

A szinusz függvényre vonatkozó addíciós szabály szerint:

$$A_n \sin(n\omega t + \phi_n) = A_n \sin\phi_n \cos n\omega t + A_n \cos\phi_n \sin n\omega t$$

Ezért $f(t)$ felírható az alábbi alakban is:

$$f(t) = a_0 + (a_1 \cos\omega t + b_1 \sin\omega t) + (a_2 \cos 2\omega t + b_2 \sin 2\omega t) + ... + (a_n \cos n\omega t + b_n \sin n\omega t) + ...$$

Itt $a_0 = A_0$, $a_n = A_n \sin\phi_n$, és $b_n = A_n \cos\phi_n$, ahol n=1,2,…

Másképpen:

$$f(t) = a_0 + \sum_{n=1}^{\infty} (a_n \cos n\omega t + b_n \sin n\omega t)$$

Az a_n és b_n együtthatókat a függvény Fourier együtthatóinak nevezzük. Ezek az alábbi formulával számolhatok ki:

$$a_n = \frac{2}{T} \int_0^T f(t) \cos(n\omega t) dt$$

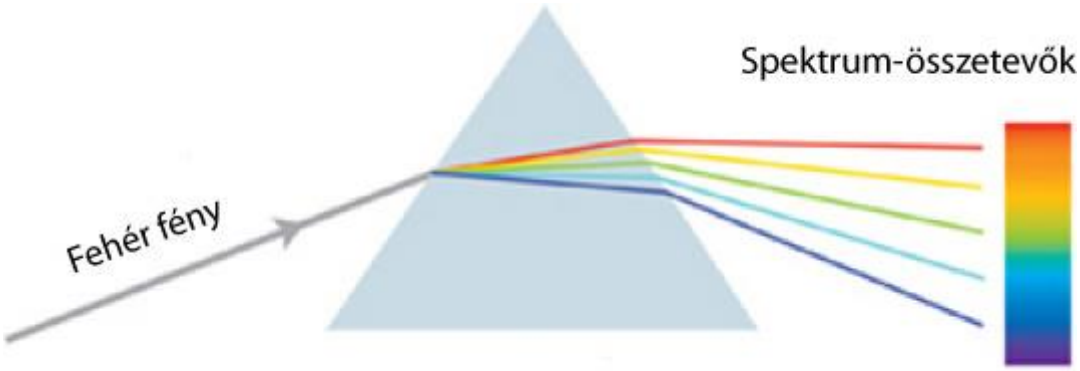
$$b_n = \frac{2}{T} \int_0^T f(t) \sin(n\omega t) dt$$

Az a_0 a függvény stacionárius egyenáramú komponense, amely nem más, mint a függvény egy periódusra átlagolt középértéke. Matematikai alakban:

$$a_0 = \frac{1}{T} \int_0^T f(t) dt$$

A Fourier sorban a $\cos(\omega t)$ valamint $\sin(\omega t)$ kifejezéseket alapharmonikusoknak, a $\cos(n\omega t)$, valamint a $\sin(n\omega t)$ tagokat pedig felharmonikusoknak nevezzük.

Ha az $f(t)$ periodikus függvény Fourier sorában a jobboldalon levő szinusz függvények mindig pozitívnak választható amplitúdóit ábrázoljuk a szinuszok frekvenciáinak függvényében, akkor megkapjuk az illető periodikus függvény színképét vagy idegen szóval spektrumát. A színkép (spektrum) elnevezés a fizikai optikából származik. Isaac Newton 1664-ben vékony résen át vékony fehér fénynyalábot bocsátott üvegprizmára. Az tapasztalta, hogy a fehér fény már az első törésnél színes nyalábokra bomlott (3.7. ábra).



3.7. ábra: fehér fény színes nyalábokra bontása prizmával
A kép forrása: http://www.videopraktika.hu/articles/466

Csökkenő hullámhossz szerint a színek: vörös, narancs, sárga zöld, kék, ibolya.

Történeti érdekességként megjegyezzük, hogy Fourier 1807-ben publikált dolgozatában a hőmérséklet eloszlást próbálta szinusz függvényekkel közelíteni. A dolgozatot J. L. Lagrange véleménye alapján visszautasították. Lagrange szerint a „sarkot” nem lehet szinuszokkal előállítani. Végül a dolgozat 1822-ben Lagrange halála után jelent meg. Azóta az elméletet kiterjesztették nem periodikus függvényekre, valamint diszkrét idejű és diszkrét értékkészletű függvényekre is.

Fourier-nek ez a felfedezése hatalmas jelentőséggel bír a jelfeldolgozásban. Egy részt ugyanis lehetővé teszi, hogy bármilyen jelet a matematikailag és számítástechnikailag nagyon jól kezelhető szinusz illetve koszinusz függvényekkel közelítsünk. A jelfeldolgozás szempontjából a legnagyobb előny, hogy a szinuszos illetve koszinuszos jelet egy lineáris rendszeren átengedve a kapott jel is szinuszos, illetve koszinuszos lesz. Legfeljebb csak a fázisában, illetve amplitúdójában különbözhet. Frekvenciájában nem.

3.2.2 A Fourier sora komplex alakja

A Fourier sor eddig tárgyalt valós alakja sok esetben jól használható a gyakorlatban, azonban a jelfeldolgozási problémák általános tárgyalásához nehézkes. Az írásmód a komplex számok használatával jelentősen egyszerűsödik. Írjuk fel a z komplex számot valamint a konjugáltját is trigonometrikus és exponenciális alakban is.

$$R(\cos\alpha + j\sin\alpha) = \mathrm{Re}^{j\alpha}$$

$$R(\cos\alpha - j\sin\alpha) = \mathrm{Re}^{-j\alpha}$$

R-el egyszerűsítve és összeadva a két egyenletet kapjuk, hogy

$$2\cos\alpha = e^{j\alpha} + e^{-j\alpha}$$

amiből

$$\cos\alpha = \frac{1}{2}(e^{j\alpha} + e^{-j\alpha})$$

Hasonlóan, ha az R-el történő egyszerűsítés után a két egyenletet kivonjuk egymásból, akkor

$$2j\sin\alpha = e^{j\alpha} + e^{-j\alpha}$$

Ezt átrendezve

$$\sin\alpha = \frac{1}{2j}(e^{j\alpha} + e^{-j\alpha})$$

Azaz a szinusz és a koszinusz (és nyilván az összes trigonometrikus) függvény felírható komplex exponenciális függvényekkel.

Ekkor a Fourier sor felírható a következő módon:

$$f(t) = a_0 + \sum_{n=1}^{\infty}\frac{1}{2}\left(a_n e^{jn\omega t} + a_n e^{-jn\omega t} - jb_n e^{jn\omega t} + jb_n e^{-jn\omega t}\right)$$

Átcsoportosítva a szummában levő tagokat

$$f(t) = a_0 + \sum_{n=1}^{\infty}\left(\frac{a_n - jb_n}{2}e^{jn\omega t} + \frac{a_n + jb_n}{2}e^{-jn\omega t}\right)$$

formulát kapjuk. Bevezetve a

$$C_n = \frac{a_n - jb_n}{2}$$

komplex amplitúdót, ahol $C_{-n} = \overline{C_n}$ és $C_0 = a_0$ felírható a Fourier sor komplex számokkal kifejezett alakja

$$f(t) = \sum_{n=-\infty}^{\infty} C_n e^{(jn\omega t)}$$

3.2.3 Nem periodikus jelek Fourier analízise

A valós műszaki problémáknál a jeleknek csak viszonylag kis része periodikus. A Fourier sorfejtés viszont csak periodikus jelekre teszi lehetővé az időtartományból a frekvenciatartományba és viszont történő átalakítást. Felvetődik a kérdés, hogy hogyan lehetne a módszert kiterjeszteni nem periodikus jelekre is. Nos, a nem periódusos jel felfogható úgy is, mint olyan periodikus jel melynek periódus ideje $T = \infty$. Az előző fejezetben láttuk, hogy periodikus jelek spektruma vonalas. A nem periodikus jeleknél a $T = \infty$ határátmenet alkalmazása miatt a spektrum vonalak távolsága $\Delta f \to 0$, ezért a spektrum folytonos. Bizonyítás nélkül kimondjuk, hogy ekkor a Fourier sor komplex alakjában az összegzés integrállá alakul, hiszen ez nem más, mint a végtelen kicsi Δf tartományokra vett függvényértékek összege.

$$f(t) = \int_{-\infty}^{\infty} C(\omega)e^{j\omega t}df$$

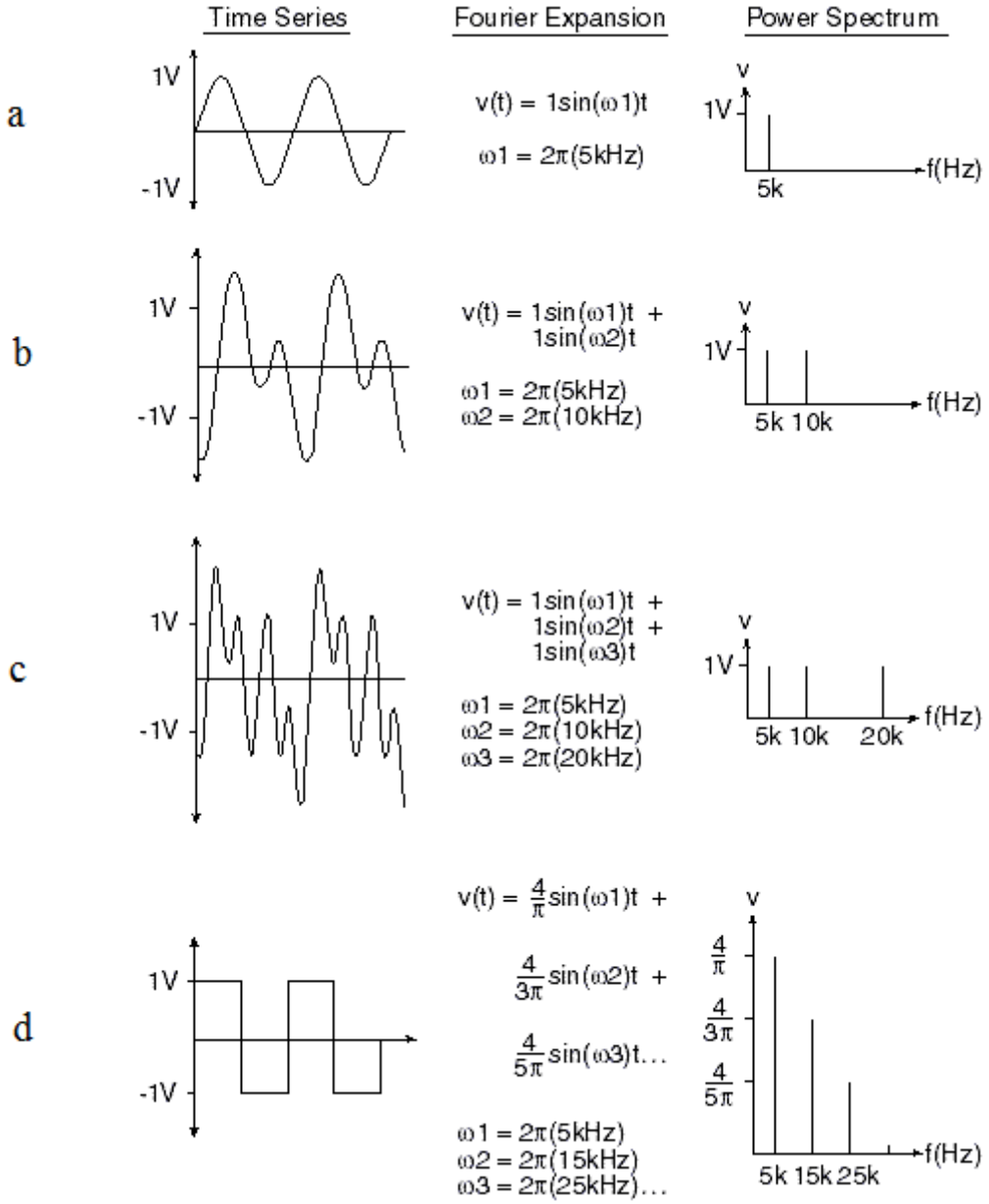
ahol $C(\omega)$ egy komplex értékű folytonos amplitúdó függvény, Természetesen a visszatranszformálás is létezik:

$$C(\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}dt$$

A $C(\omega)$ függvényt az $f(t)$ függvény **Fourier transzformáltjának** nevezzük. Az irodalomban az inverz transzformációs képletekben gyakran feltűnő $\frac{1}{2\pi}$ skálaparaméter akkor jelentkezik, ha az integrálást a körfrekvenciára végezzük el.

Komplex írásmód esetén a fentiek alapján jel frekvenciatartománybeli képét (spektrumát) a $C(\omega)$ komplex amplitúdó függvény értékei írják le az ω függvényében. Az ábrázolhatóság érdekében egyrészt lehetőség van a komplex amplitúdók valós és képzetes részeinek elkülönítésére. Másrészt ábrázolhatjuk az amplitúdó spektrumot a $C(\omega)$ együtthatók abszolút értékeiből. Ezt teljesítmény spektrumnak nevezik, s a gyakorlati életben ezt szokták használni.

A 3.8. ábrán néhány analóg jel és annak Fourier spektruma látható. A 3.8/a ábrán egy 5 kHz frekvenciájú tiszta szinusz jel és annak spektruma, a 3.8/b ábrán egy 5 kHz-es szinusz és egy 10 kHz-es szinuszból álló jel és spektruma, a 3.8/c ábrán egy 5 kHz-es szinusz, egy 10 kHz-es szinuszból és egy 20 kHz-es szinuszból álló jel és spektruma látszik. A 3.8/d ábrán pedig egy négyszög jel és annak spektruma látható.



3.8. ábra: Jelek Fourier spektruma [4]

3.2.4 A Fourier transzformáció tulajdonságai

Bizonyítás nélkül röviden összefoglaljuk a Fourier transzformáció fontosabb tulajdonságait (3.1 táblázat):

	Tulajdonság	Idő tér	Frekvencia tér
1	A transzformáció lineáris	$a_1 f_1(t) + a_2 f_2(t)$	$a_1 C_1(\omega) + a_2 C_2(\omega)$
2	Rövidebb jelnek szélesebb a spektruma (léptékváltás)	$f(t/a)$	$aC(a\omega)$
3	Az időeltolás nem változtatja meg a spektrum alakját	$f(t-t_0)$	$C(\omega)e^{-j\omega t_0}$
4	Szinuszos moduláló jel eltolja a spektrumot, de nem változtatja meg az alakját	$f(t)e^{j\omega t_0}$	$C(\omega-\omega_0)$
5	Az időtérben végzett konvolúció a frekvencia térben szorzásnak felel meg	$\int_{-\infty}^{\infty} f_1(t) f_2(t-\tau) d\tau$	$C_1(\omega)C_2(\omega)$

3.1 táblázat: A Fourier transzformáció tulajdonságai

3.3 Diszkrét Fourier transzformáció

Az előző fejezetekben tárgyalt Fourier transzformáció nagyon hasznos eszköz azokon a területeken, ahol képlettel megadott függvényeken kell az időtartományban bonyolult műveleteket végezni. Azonban a műszaki gyakorlatban nem tudunk mindig képlettel megadott függvényekkel dolgozni. Sokszor előfordul, hogy a kérdéses jel csak diszkrét időpontokban táblázatos formában áll rendelkezésre. Így felmerült az igény egy olyan módszer megalkotására, amellyel a Fourier transzformáció kiterjeszthető véges hosszúságú diszkrét adatsorokra.

Legyen N db minták egy jelből, melyek egyenlő távolságra vannak egymástól az időtengelyen. Tároljuk ezeket a mintákat egy x[N] vektorban. Ekkor definíció szerint az x[N] vektor diszkrét Fourier transzformáltján (továbbiakban DFT) az alábbi y[N] vektort értjük.

$$y_k = \sum_{n=0}^{N} x_n e^{\frac{-j2\pi kn}{N}} \qquad \text{k=0,1,2,…,N-1}$$

Természetesen a diszkrét esetben is létezik a transzformáció inverze:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k e^{\frac{j2\pi kn}{N}} \qquad \text{n=0,1,2,…,N-1}$$

Vegyük észre, hogy sem a T periódus idő, sem az ω körfrekvencia nem szerepel a DFT képleteiben. Tehát a DFT egy olyan függvény transzformáció (operáció), mely egy N db mintából álló sorozatból egy másik N db mintából álló sorozatot képez, a dimenzió nélküli *k* index segítségével. Fizikai jelentése akkor van a transzformációnak, ha az x_n vektor valamilyen fizikai jellemzővel rendelkezik. Jelfeldolgozási szempontból a fizikai jellemző a $k\omega$, tehát a DFT ebben az esetben azt mondja meg, hogy az N db mintával megadott jelben az ω körfrekvencia N db egész számú többszöröse mekkora súllyal van jelen.

A DFT elvégezhetőségének feltétele, hogy az x_n jel energiája véges legyen, azaz

$$0 \le \sum_{n=0}^{N} x_n^2 < \infty$$

Ez a gyakorlatban mindig teljesül az x_n tagjainak korlátossága, valamint a véges mintaszám miatt.

A DFT szempontjából mindegy, hogy a bemenő jel ténylegesen periodikus vagy sem. Úgy tekinti az N db mintát tartalmazó x_n vektort, mint egy végtelen periodikus jel egy periódusát.

A transzformáció eredményeképpen előálló y_k vektor N db komplex számot tartalmaz, ahol az egyes tagok abszolút értékeiből képzett sorozat adja az amplitúdó spektrumot, a fázis szögek sorozata pedig a fázis spektrumot.

Az alábbi C# nyelvű rutin a definíció szerint számolja ki a DFT-t.

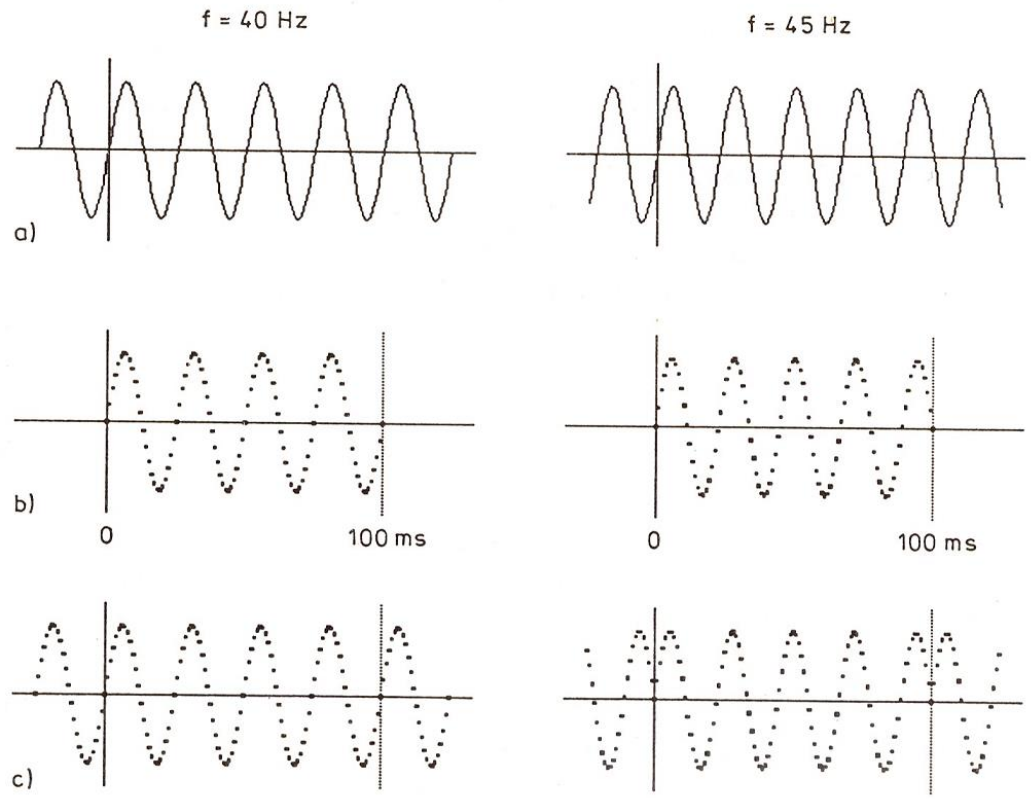
```
void dft(Complex[] in, Complex[] out, int N)
{
 double re, im;
 int k, n;
```

```
 int N=in.Length;
 for(k=0; k<N; k++)
 {
  re=0; im=0;
  for(n=0;n<N;n++)
  {
      re+=in[n].re*Math.Cos(2*Math.PI*k*n/N)
      re+=in[n].img*Math.Sin(2*Math.PI*k*n/N);

      im+=in[n].img*Math.Cos(2*Math.PI*k*n/N)
      im-=in[n].re*Math.Sin(2*Math.PI*k*n/N);
  }
  out[k].re=re;
  out[k].img=im;
 }
}
```

3.4 Ablakozás

Mint már említettük a DFT algoritmusa a transzformálandó jelet periodikusnak tekinti, melynek periódus ideje N, azaz $x_n = x_{n+N}$.

A 3.9/a ábrán egy 40 Hz, és egy 45 Hz frekvenciájú szinusz jelet ábrázol. Nyilvánvalóan mind a két jelnek egyetlen spektrum vonala van (40 illetve 45 Hz). Legyen a mintavételi frekvenciánk 1000 Hz, a mintavétel időtartama pedig 100 ms. (Ez azt jelenti, hogy mind a két jelből N= 100 db mintát vettünk.) Ennek eredményét a 3.9/b ábra mutatja. Periódikusan ismételve ezt az eljárást a 3.9/c látható eredményt kapjuk. Látható, hogy amikor a mintavételi frekvencia és az eredeti jelben levő frekvencia hányadosa nem egész szám, akkor a mintavételezett jelnél az időablak szélein torzulások vannak. A torzulás miatt a mintavételezett jel már nem periodikus, ezért nem csak egy szinuszos összetevője van. Ennek megfelelően a DFT az első esetben a helyes spektrumot fogja szolgáltatni, míg a második esetben a torzult mintavételezett jelnek megfelelően a DFT által számolt spektrum is torzult lesz.



3.9. ábra: Az N minta utáni periodikus folytatás torzító hatása [4].

A fent említett torzítások jelentősen csökkenthetőek az ablakozási technika segítségével. Az alapötlet a következő: az x_n vektorra alkalmazzunk egy súlyfüggvényt és a DFT-t ezen súlyfüggvénnyel szorzott mintákon végezzük el:

$$y_k = \sum_{n=0}^{N} W_n x_n e^{\frac{-j2\pi kn}{N}}$$

Itt W_n értékek a súlyfüggvény diszkrét értékei. A súlyfüggvényt úgy kell megválasztani, hogy a megfigyelési ablak szélein az értékei kicsik legyenek. Elvárt követelmény még a súlyfüggvénnyel szemben, hogy szimmetrikus legyen. A jelfeldolgozásban a súlyfüggvényeket ablakozó függvényeknek is nevezik. A gyakorlatban az alábbi ablakozó függvényeket szokták használni:

Négyszögletes ablak

Valójában ez nem más, mint az ablakozás nélküli eset. Definíciója:

$$W_n = \begin{Bmatrix} 1, & 0 \le n \le N \\ 0, & egyébként \end{Bmatrix}$$

Háromszög ablak

A függvény egy egyenlőszárú háromszög. Matematikai alakja:

$$W_n = \begin{Bmatrix} 1 - \dfrac{N/2 - n}{N/2}, & 0 \le n \le N \\ 0, & egyébként \end{Bmatrix}$$

Hanning (Hann) ablak

A függvény egy cosinus görbe, melynek két minimuma az ablak széleinél van. Matematikai alakja:

$$W_n = \begin{Bmatrix} 0.5 - 0.5\cos(n\dfrac{2\pi}{N}), & 0 \le n \le N \\ 0, & egyébként \end{Bmatrix}$$

Hamming ablak

A Hann ablakhoz hasonló, de az értéke az ablak szélein nem csökken zérusig. Matematikai alakja:

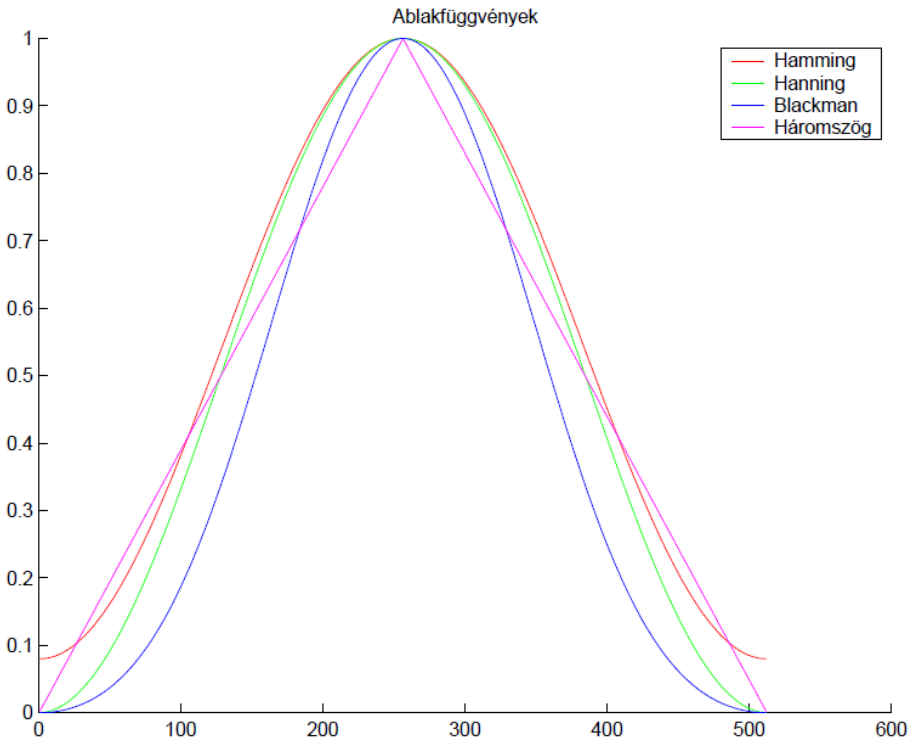
$$W_n = \begin{Bmatrix} 0.54 - 0.46\cos(n\dfrac{2\pi}{N}), & 0 \le n \le N \\ 0, & egyébként \end{Bmatrix}$$

Blackman ablak

Matematikai alakja:

$$W_n = \begin{Bmatrix} 0.42 - 0.5\cos(n\dfrac{2\pi}{N}) + 0.08\cos(2n\dfrac{2\pi}{N}), & 0 \le n \le N \\ 0, & egyébként \end{Bmatrix}$$

A 3.10. ábra grafikusan is szemlélteti a fent említett ablakozó függvényeket.



3.10. ábra: Ablakozó függvények.

3.5 A gyors Fourier transzformáció

A diszkrét Fourier transzformációt illetve annak inverzét definíció szerint megvalósító algoritmusokkal bármilyen jel spektruma meghatározható. Azonban az eljárásnak van egy hátránya: az, hogy rendkívül számításigényes. Ennek oka, hogy mind a k mind az n indexek értékeit az eljárás során 0-ról N-re kell növelni az eredmény kiszámításához. Ez N^2 számítási műveletet igényel. A diszkrét Fourier transzformáció számítási igényét Cooley és Tukey 1965-ben publikált algoritmusukkal jelentősen csökkentették. Mivel az eredeti DFT N^2 szorzást igényel, ezért, ha az adatokat szétválasztjuk két egyforma részre, akkor az egyes részek önmagukban $\left(\frac{N}{2}\right)^2$ szorzást igényelnek. A teljes transzformáció pedig nyilván $2\left(\frac{N}{2}\right)^2 = \frac{N^2}{2}$ műveletet. Így, ha a két transzformáció eredményei összekombinálhatóak, akkor ezzel a metódussal számítási időt lehet megtakarítani. Felvetődik a kérdés, hogy a fenti ötletet hogyan lehet a gyakorlatban megvalósítani.

Vezessük be a következő jelölést: $W_N = e^{-j\left(\frac{2\pi}{N}\right)}$. Ezzel a DFT a következő alakban írható fel:

$$y_k = \sum_{n=0}^{N} x_n e^{\frac{-j2\pi kn}{N}} = \sum_{n=0}^{N} x_n W_N^{kn} \qquad \text{k=0,1,2,…,N-1}$$

Csoportosítsuk úgy a fenti egyenletet, hogy külön összegezzük a páros, illetve páratlan indexű tagokat. Ekkor a fenti egyenlet így írható:

$$y_k = \sum_{n=0}^{N} x_n W_N^{kn} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_N^{(2n+1)k}$$

Másképpen:

$$y_k = \sum_{n=0}^{N} x_n W_N^{kn} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} (W_N^2)^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} (W_N^2)^{nk}$$

Azonban:

$$W_N^2 = e^{j\left(-\frac{2\pi}{N}2\right)} = e^{j\left(-\frac{2\pi}{\frac{N}{2}}\right)} = W_{\frac{N}{2}}$$

Így:

$$y_k = \sum_{n=0}^{N} x_n W_N^{kn} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} W_{\frac{N}{2}}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} W_{\frac{N}{2}}^{nk}$$

Vagyis az eredeti DFT felírható páros és páratlan indexű tagok diszkrét Fourier transzformáltjainak összegeként. Igaz a páratlan indexű tagúak transzformáltját még be kell szorozni W_N^k -val. Induláskor érdemes N-et 2 egészkitevőjű hatványának választani, mert ekkor a fenti eljárás tovább folytatható egészen addig, amíg az alsó határhoz a 2 pontos DFT-hez nem érünk.

Példaként vegyünk egy 8 számból álló sorozatot (N=8).

$$x[N]=\{0,1,2,3,4,5,6,7\}.$$

Ezt bontsuk fel két csoportra:

$$x[N]=\{0,2,4,6\} \text{ és } \{1,3,5,7\}.$$

Újra bontsuk két-két csoportra:

$$x[N]=\{0,4\}\ \{2,6\}\ \{1,5\} \text{ és } \{3,7\}.$$

Így összesen négy darab 2 pontos DFT-t kell kiszámolni.

A DFT kiszámolásának ezt az algoritmusát gyors Fourier transzformációnak nevezzük, angolul Fast Fourier Transformation (FFT).

Az algoritmus működésére tekintsünk egy egyszerű példát, ahol a hossz, N=4. Ekkor a transzformáció a következő lesz:

$$y_k = \sum_{n=0}^{4} x_n W_4^{kn} = \sum_{n=0}^{1} x_{2n} W_2^{nk} + W_4^k \sum_{n=0}^{1} x_{2n+1} W_2^{nk}$$
$$= \left[x_0 + x_2 W_2^k\right] + W_4^k \left[x_1 + x_3 W_2^k\right]$$

De:

$$W_2^k = e^{j\left(-\frac{2\pi}{2}k\right)} = e^{j\left(-\frac{2\pi}{4}2k\right)} = W_4^{2k}$$

Így a z FFT a következő lesz:

$$y_k = \left[x_1 + x_2 W_4^{2k}\right] + W_4^k \left[x_1 + x_3 W_4^{2k}\right]$$

Minden egyes *k*-ra végig kiszámolva az algoritmus a következő:

$$y_0 = \left[x_0 + x_2 W_4^0\right] + W_4^0\left[x_1 + x_3 W_4^0\right]$$

$$y_1 = \left[x_0 + x_2 W_4^2\right] + W_4^1\left[x_1 + x_3 W_4^2\right]$$

$$y_2 = \left[x_0 + x_2 W_4^0\right] + W_4^2\left[x_1 + x_3 W_4^0\right]$$

$$y_3 = \left[x_0 + x_2 W_4^2\right] + W_4^3\left[x_1 + x_3 W_4^2\right]$$

A számolásnál figyelembe vettük, hogy

$$W_4^4 = e^{j\left(-\frac{2\pi}{4}\right)} = 1 = W_4^0$$

és

$$W_4^6 = e^{j\left(-\frac{2\pi}{4}\right)6} = -1 = W_4^2$$

Az FFT legfőbb előnye a DFT-vel szemben, hogy a transzformáció végrehajtásához lényegesen kevesebb műveletszám kell. A DFT-nél a műveletszám N^2, míg az FFT fenti megvalósításánál ez az érték $N \log_2 N$. Egy 1024 pontos DFT esetén a műveletszám 1048576 addig az FFT-nél ez 10240, azaz a sebesség növekedés ebben az esetben durván százszoros.

Összefoglalva a gyors Fourier transzformáció egy rekurzív algoritmus, melynek lényege, hogy a transzformálni kívánt komplex számsort páros és páratlan indexű tagok tömbjeire bontunk. Majd az így kapott tömbökre ugyanígy elvégezzük a résztömbökre való bontást egészen addig, amíg a tömbök elemszáma kettőre csökken. A kételemű tömbökön elvégezzük a diszkrét Fourier transzformációt, majd lépünk vissza a rekurzióból. A gyors Fourier transzformációhoz 2^N darab egyenközű minta kell.

Az FFT algoritmus pszeudo kódja:

```
X0,...,N-1 ← fft(x, N, s):
    if N = 1 then
        X0 ← x0
    else
        X0,...,N/2-1 ← fft(x, N/2, 2s)
        XN/2,...,N-1 ← fft(x+s, N/2, 2s)
        for k = 0 to N/2-1 do
            t ← Xk
            Xk ← t + exp(-2πi k/N) Xk+N/2
            Xk+N/2 ← t - exp(-2πi k/N) Xk+N/2
        end for
    end if
```

Az FFT hatékony implemetálásához szükségünk van komplex számok aritmetikáját megvalósító osztályra is. C#-ban használhatjuk a System.Numerics névtérben implementált Complex osztályt, de az alábbi példa alapján készíthetünk saját implementációt is.

```
class Complex
{
        public double re;   // the real part
        public double im;   // the imaginary part


        public Complex(double re, double im)
        {
            this.re = re;
            this.im = im;
        }


        public String toString()
        {
            if (im == 0) return re + "";
            if (re == 0) return im + "i";
            if (im < 0) return re + " - " + (-im) + "i";
            return re + " + " + im + "i";
        }


        public double abs()
        {
            return Math.Sqrt(re*re+im*im);
        }



        public Complex add(Complex b)
        {
            Complex a = this;
            double real = a.re + b.re;
            double imag = a.im + b.im;
            return new Complex(real, imag);
        }


        public Complex sub(Complex b)
        {
```

```
            Complex a = this;
            double real = a.re - b.re;
            double imag = a.im - b.im;
            return new Complex(real, imag);
        }


        public Complex mul(Complex b)
        {
            Complex a = this;
            double real = a.re * b.re - a.im * b.im;
            double imag = a.re * b.im + a.im * b.re;
            return new Complex(real, imag);
        }

        public Complex reciprocal()
        {
            double scale = re * re + im * im;
            return new Complex(re / scale, -im / scale);
        }


        // return a / b
        public Complex divides(Complex b)
        {
            Complex a = this;
            return a.mul(b.reciprocal());
        }

        public Complex conjugate()
        {
            return new Complex(re, -im);
        }

        public Complex scale(double alpha)
        {
            return new Complex(alpha * re, alpha * im);
        }

}
```

A következőkben megadjuk az FFT egy lehetséges implementációját. Az osztályban az FFT algoritmus mellett implementáltuk a `getPowerSpectrum` metódust, mely egy két dimenziós tömbben a spektrum táblázatot adja meg. A `bandPassFilter` metódus pedig egy brick wall rendszerű sáváteresztő szűrő. A digitális szűrőkről bővebben a 4. fejezetben olvashat.

```
class FFT
    {
       // radix 2 Cooley-Tukey FFT
       // length of the array must be a power of 2"
        public Complex[] fft(Complex[] x)
        {
            int n = x.Length;

            // base case
            if (n == 1)
                return new Complex[] { x[0] };
```

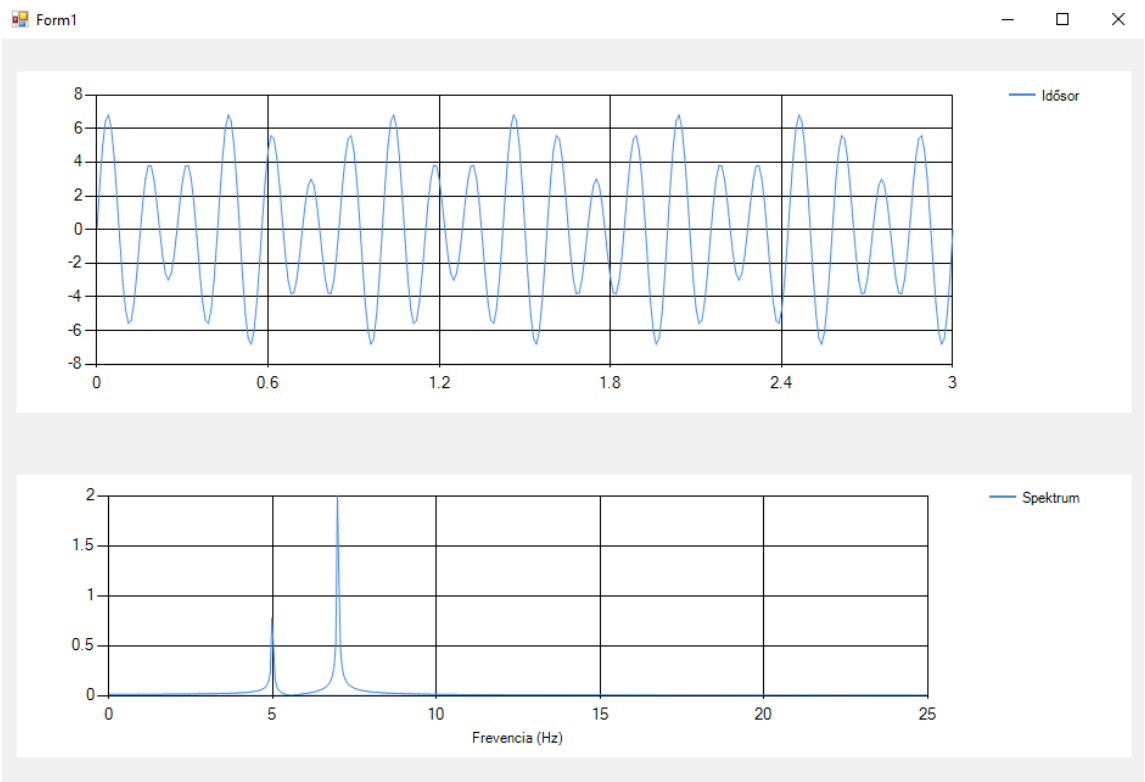
```
        // fft of even terms
        Complex[] even = new Complex[n / 2];
        for (int k = 0; k < n / 2; k++)
        {
            even[k] = x[2 * k];
        }
        Complex[] q = fft(even);

        // fft of odd terms
        Complex[] odd = even;  // reuse the array
        for (int k = 0; k < n / 2; k++)
        {
            odd[k] = x[2 * k + 1];
        }
        Complex[] r = fft(odd);

        // combine
        Complex[] y = new Complex[n];
        for (int k = 0; k < n / 2; k++)
        {
            double kth = -2 * k * Math.PI / n;
            Complex wk = new Complex(Math.Cos(kth), Math.Sin(kth));
            y[k] = q[k].add(wk.mul(r[k]));
            y[k + n / 2] = q[k].sub(wk.mul(r[k]));
        }
        return y;
    }


    // inverse Cooley-Tukey FFT
    // length of the array must be a power of 2"
    public Complex[] ifft(Complex[] x)
    {
        int n = x.Length;
        Complex[] y = new Complex[n];

        // take conjugate
        for (int i = 0; i < n; i++)
        {
            y[i] = x[i].conjugate();
        }

        // compute forward FFT
        y = fft(y);

        // take conjugate again
        for (int i = 0; i < n; i++)
        {
            y[i] = y[i].conjugate();
        }

        // divide by n
        for (int i = 0; i < n; i++)
        {
          y[i] = y[i].scale(1.0 / n);
        }

        return y;

    }
```

```
        //reorder array elements so they are in order from lowest to
        //highest frequency
        public void fftShift(double[] x)
        {
            double[] temp = new double[x.Length];
            for (int i = 0; i < x.Length; i++)
            {
                temp[i] = x[i];
            }

            for (int i = 0; i < x.Length / 2; i++)
            {
                x[i] = temp[x.Length / 2 + i];
                x[x.Length / 2 + i] = temp[i];
            }
        }

        public int pow2roundup(int x)
        {
            if (x < 0)
                return 0;
            --x;
            x |= x >> 1;
            x |= x >> 2;
            x |= x >> 4;
            x |= x >> 8;
            x |= x >> 16;
            return x + 1;
        }

        public double[] correctDataLength(double[] input)
        { //changes the length of in to a power of 2
            int n = input.Length;

            int newLength=pow2roundup(n);

            double[] correctArray = new double[newLength];
            for (int i = 0; i < newLength - 1; i++)
            {
                if (i < input.Length)
                {
                    correctArray[i] = input[i];
                }
                else
                { //pad with 0s
                    correctArray[i] = 0;
                }

            }
        return correctArray;
    }

    public double[] bandPassFilter(double[] data, uint Fs,
                                   double minFreq, double maxFreq)
    {
            data = correctDataLength(data);

            //fill x with the data in complex form
            Complex[] x = new Complex[data.Length];
            for (int i = 0; i < data.Length; i++)
            {
                x[i] = new Complex(data[i], 0);
```

```
            }

            //get the fft of the data
            Complex[] y = fft(x);
            int n = y.Length;
            double T = n / (double)Fs;

            int j = 0;
            for (int i = -n / 2; i < (n / 2) - 1; i++)
            {
                double freq= Math.Abs(i/T);

              if (freq < minFreq || freq > maxFreq)
                  y[j] = new Complex(0, 0);
                j++;
            }

            //get the fft of the data
            Complex[] z = ifft(y);
            double[] zr = new double[z.Length];
            for (int i = 0; i < zr.Length; i++)
            {
                zr[i] = z[i].re;
            }
            return zr;
        }

    public double[,] getPowerSpectrum(double[] data, int Fs)
    {
        data=correctDataLength(data);
        Complex[] x = new Complex[data.Length];
        for (int i = 0; i < data.Length; i++)
        { //fill x with the data in complex form
            x[i] = new Complex(data[i], 0);
        }

        Complex[] y = fft(x); //get the fft of the data
        int n = y.Length;

        double[] k = new double[n]; //for calculating the frequency
        int j = 0;

      //populate k with integers from -n/2 to n/2-1
        for (int i = -n / 2; i < (n / 2) - 1; i++)
        {
            k[j] = i;
            j++;
        }

        double T = n / (double)Fs;

        double [] amp = new double[n];

        //normalize data and get real magnitude
        for (int i = 0; i < n; i++)
        {
            amp[i] = y[i].divides(new Complex(n, 0)).abs();

        }

        fftShift(amp);
```

```
        //make multidimensional array of Freq & Amp
        //the 1st column of the frequency array is the frequency
        //the 2nd column is the magnitude

        double [,] freqMag = new double[n,2];
        for (int i = 0; i<n; i++)
        {

            freqMag[i,0]=k[i]/T;
            freqMag[i,1]=amp[i];
        }
        return freqMag;
    }

}
```

Demonstrációs céllal tekintsük a 3.11. ábrát. Az ábrán egy Windows form alkalmazás képernyő képe látható.



3.11. ábra: Demonstrációs alkalmazás a gyors Fourier transzformációhoz

A felső csatornán az $y(t) = 2\sin(5\omega t) + 5\sin(7\omega t)$ függvény egy részlete látható, az alsón pedig a függvény spektruma. A függvény adatsorának előállításakor 256 Hz-es mintavételi frekvenciát használtunk. Ezzel a mintavételi frekvenciával futtattuk le az FFT osztály `getPowerSpectrum` metódusát. Az elvárásoknak megfelelően a spektrumban csak az 5 és a 7 Hz-es spektrumvonal látható.

Az alkalmazásban a görbék megjelenítésére az MS .NET chart vezérlőjét használtuk. Az alkalmazás lényegi részének forráskódja:

```
using System;
using System.Windows.Forms;


namespace FFTTest
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            double[] payload = new double[2048];
            int fs = 256;    // sampling rate
            double t = 0;
            double dt = 1.0 / fs;
            for (int i=0; i< 2048; i++, t += dt)
            {
              payload[i] = 2 * Math.Sin(5 *2*Math.PI*t) +
                           5 * Math.Cos(7*2*Math.PI*t);

              chart1.Series[0].Points.AddXY(t, payload[i]);
            }
            chart1.ChartAreas[0].AxisX.Minimum = double.NaN;
            chart1.ChartAreas[0].AxisX.Maximum = double.NaN;
            chart1.ChartAreas[0].RecalculateAxesScale();

            FFT fft = new FFT();
            double [,] fMag= fft.getPowerSpectrum(payload, fs);


            for (int i = fMag.Length / 4; i < fMag.Length/2; i++)
            {
              chart2.Series[0].Points.AddXY(fMag[i,0], fMag[i,1]);

            }

            chart2.ChartAreas[0].AxisX.Minimum = double.NaN;
            chart2.ChartAreas[0].AxisX.Maximum = double.NaN;
            chart2.ChartAreas[0].RecalculateAxesScale();
        }
    }
}
```

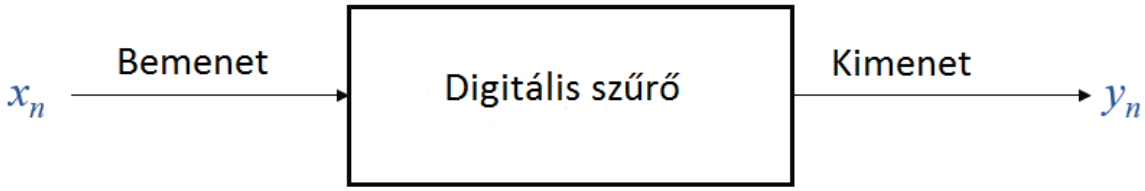
3.6 Kérdések, feladatok

1. Nevezze meg azokat a műveleteket, amelyek az időtartományban módosítják a jeleket!

2. Nevezze meg azokat a műveleteket, amelyek az amplitúdó tartományban módosítják a jeleket!
3. Ábrázolja az $y(t) = 3\sin(-2t)$ függvényt!
4. Mit értünk egy jel inverzén?
5. Mi a harmonikus rezgés matematikai alakja? Magyarázza meg, a képletben szereplő tagok jelentését!
6. A Fourier sorfejtésben mi az a_0 tag jelentése?
7. Mit nevezünk egy periodikus jel spektrumának?
8. Mi az Euler szám?
9. Mi az *f(t)* periodikus függvény Fourier sorának valós, illetve komplex alakja?
10. Adja meg a Fourier sor valós alakjában szereplő a_0, a_n és b_n együtthatók matematikai alakját!
11. Írja fel az $f(t) = \begin{cases} 1, & 0 < t < \pi \\ 2, & \pi < t \le 2\pi \end{cases}$ jel Fourier sorát!
12. Mikor vonalas egy jelnek a spektruma és mikor folytonos?
13. Írja fel az inverz Fourier transzformáció formuláját!
14. Mikor használjuk a diszkrét Fourier transzformációt.
15. Írja fel a diszkrét Fourier transzformáció formuláját!
16. Mi az ablakozási technika lényege a DFT-nél?
17. Milyen tulajdonságokkal kell rendelkeznie egy súlyfüggvénynek.
18. Mik a leggyakrabban használatos súlyfüggvények?
19. Mennyi a műveletszám igénye egy N=2048 pontos DFT-nek, illetve FFT-nek?

4. Digitális szűrők

A digitális szűrők olyan szoftveres és hardveres megoldások, amelyeket jelek helyreállítására, illetve szétválasztására használunk. Szétválasztásra akkor használjuk, ha pl. a jel zajjal terhelt. Ekkor a szűrőkkel a zajt próbáljuk leszedni a jelről. Helyreállításra pedig akkor lehet használni, ha a jel sérült, vagy torzult. Ennek alapján a digitális szűrő az x_n bemenő bináris számsorozatot y_n kimenő bináris számsorozattá alakítja át (4.1. ábra).

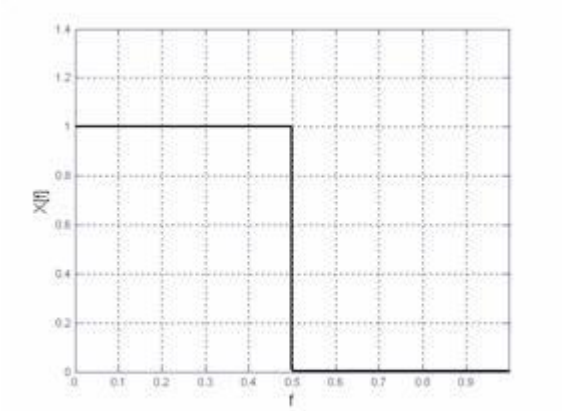


4.1. ábra: Digitális szűrés sémája

A szűrő működését tanulmányozhatjuk mind az időtartományban, mind a frekvenciatartományban. Mindig az adott feladat jellege határozza meg, hogy a tervezés és működés során mely tartománybeli működésre kell optimalizálni a szűrőt.

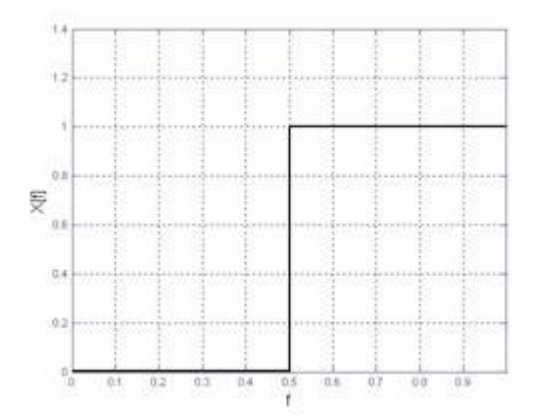
Általános esetben az alábbi szűrő típusokat különböztetjük meg:

Az **aluláteresztő** szűrő olyan szűrő megvalósítás, mely a szűrő bementére adott jelet úgy módosítja, hogy a jelből, csak a kis frekvenciás komponenseket engedi át. Az átengedés itt azt jelenti, hogy a szűrőre jellemző f_v vágási frekvenciánál kisebb frekvenciájú komponenseket engedi át változatlan amplitúdó értékkel a vágási frekvencia felettieknél viszont az amplitúdó értéket nullázza (4.2 ábra). Azt a frekvencia tartományt, ahol szűrő áteresztáteresztési tartománynak, ahol pedig nem azt vágási tartománynak nevezzük.



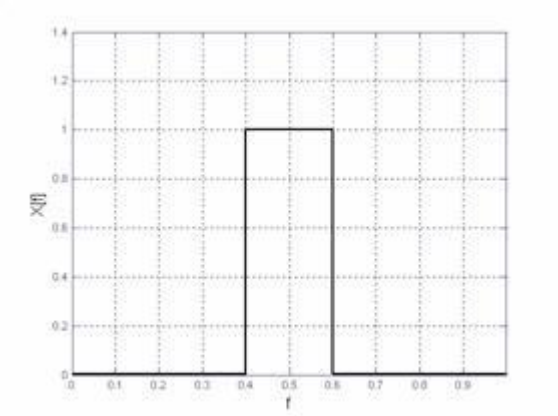
4.2. ábra: Ideális aluláteresztő szűrő frekvencia függvénye

A **felüláteresztő** szűrő az aluláteresztővel pontosan ellentétesen viselkedik. Vagyis a vágási frekvencia felett átereszt, míg alatta szűr (4.3. ábra).



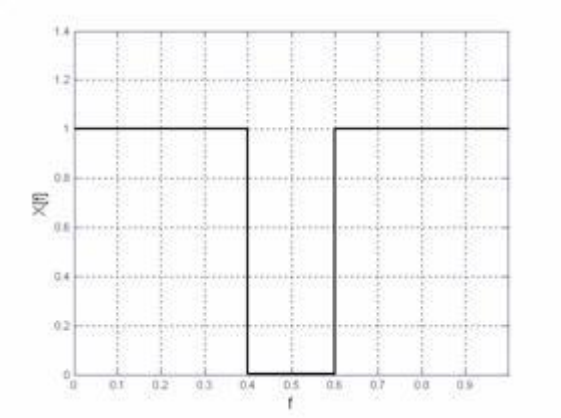
4.3. ábra: Ideális felüláteresztő szűrő frekvencia függvénye

A **sáváteresztő** szűrő egy adott frekvencia tartományon belül ereszti át a bementére adott jelet. A tartományon kívül pedig vág (4.4 ábra).



4.4. ábra: Ideális sáváteresztő szűrő frekvencia függvénye

A **sávkizáró** szűrő pedig a sáváteresztő szűrő ellentéte, azaz a frekvencia tartományon belül vág, míg a tartományon kívül átereszt (4.5 ábra).



4.5. ábra: Ideális sávátereszt ő szűrő frekvencia függvénye

A fentiek alapján azt hihetjük, hogy a digitális szűrők szűrő hatását elég csak a frekvencia tartományban vizsgálni. Ez azonban csak abban az esetben igaz, ha a szűrő feladata olyan, amit a frekvencia tartományban lehet jól elvégezni. Ilyen feladat pl. a jelek szeparálása. Amennyiben a szűrő feladata a jel véletlen zajoktól történő megszabadítása, (simítás), akkor a szűrőnek az időtartománybeli viselkedése a fontos, hiszen a szűrőtől azt várjuk, hogy a zaj kiszűrése után a jelalak lehetőleg változatlan maradjon.

Működési elvük szerint kétféle szűrőt különböztetünk meg:

- Véges válaszidejű (Finite Impulse Response – FIR) szűrők.
  Ezek olyan szűrők, melyeknek kimeneti jele csak a bemeneti jeltől függ. Ezeket a szűrőket konvolúciós szűrőknek nevezzük.
- Végtelen válaszidejű (Infinite Impulse Response – IIR) szűrők.
  Ezek olyan szűrők, melyeknek a kimeneti jele függ a bemeneti jeltől, valamint a szűrő korábbi kimenti jelétől. Azaz ezek a szűrők rekurzív szűrők.

4.1 FIR szűrők

Ezen szűrők kimeneti jele csak a bemeneti jeltől függ. A FIR szűrőket konvolúciós szűrőknek is szokták nevezni, ugyanis működési elvük matematikai alapja két függvény (a bemenő jel és a szűrő függvény) konvolúciója. Definíció szerint, ha f és g a $(-\infty, \infty)$ intervallumon értelmezett integrálható függvények, akkor ezen két függvény konvolúcióján az

$$f * g = \int_{-\infty}^{\infty} f(x)g(u-x)dx$$

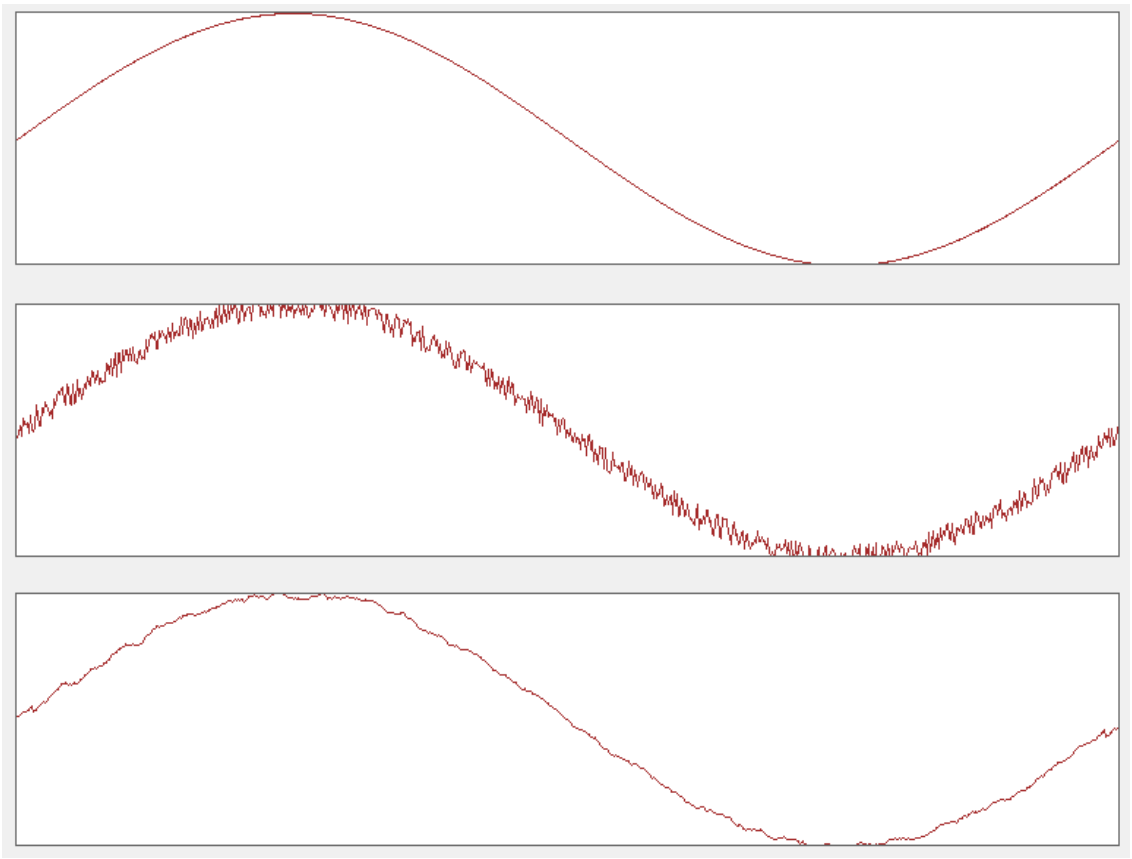
kifejezést értjük. Diszkrét esetben pedig legyen *N* számú mintánk egy jelből és tároljuk őket az *x[N]* vektorban, valamint legyen adott *k* darab, a szűrési karakterisztikát megadó valós számunk. Ezeket tároljuk az *a[k]* vektorban. Ekkor definíció szerint az *a[k]* és *x[N]* vektorok konvolúcióján az alábbi kifejezést értjük:

$$y_n = \sum_{i=0}^{k} a_i x_{n-i} \quad (0 \le n < N, k < N),$$

ahol y_n az n-edik szűrt értéket jelöli. Az *a[k]* vektort a konvolúció magfüggvényének (*kernel*), a *k* számot pedig a konvolúció fokszámának nevezzük. A fenti formula egyben a digitális FIR szűrők matematikai alakja is. A FIR szűrőkre legegyszerűbb példa az átlagoló szűrő Ez a szűrő megfelelő tervezés és használat melltett a jelben levő véletlen zaj szűrésére kiválóan alkalmas. Maga a véletlen zaj olyan zaj, melynek spektruma adott frekvenciasávban nagyjából azonos teljesítmény sűrűségű. A véletlen zaj átlaga hosszútávon a nullához tart. Az átlagoló szűrő gyakorlatilag a jelben minden egyes pontot egy adott sugarú környezetének átlagával helyettesít. Matematikai alakban

$$y_n = \frac{1}{N} \sum_{i=0}^{N-1} x_{n-i},$$

ahol *N* a környezet sugara. Látható, hogy az átlagoló szűrő képlete nem más, mint az x_n jel és 1/*N* konstans konvolúciója Az átlagoló szűrő használatával azonban vigyázni kell, ugyanis az átlagolás ugyanis a jelben lévő zaj mellett a jel fel és lefutásának meredekségét is csökkenti. Ezért az átlagoló szűrő tervezésénél a fő szempont az, hogy mekkora legyen az átlagolás sugara, hogy az a jel alakját még ne torzítsa el, de a jelben levő zajt kiszűrje. Illusztrációként az 4.5. ábrán mutatjuk be az átlagoló szűrő hatásosságát. Az ábra felső részén egy szinusz függvény egy periódusa látható. Erre egy zavarjelet helyeztünk (középső görbe), majd ezt átlagoló szűrővel simítottuk (alsó görbe).



4.5. ábra: Szinusz függvény simítása átlagoló szűrővel.

A mozgóátlagolást megvalósító programkód:

```
void movingAverage( float *array, int n, int window)
{
    float sum=0;
    for (int i = 0; i < (n - window); i++)
    {

          sum = 0.0;
          // Loop through window numbers from current i position
          for (int j = i; j < i + window; j++)
          {
                sum += array[j];            // Increment sum.
          }

          // Calculate moving average.
          float movingAverage = sum / window;
          array[i]=movingAverage;
    }
}
```

A FIR szűrőt megvalósító programkód:

```
void filterFIR( float *array, int n, float *a, int window)
{
    float sum=0;
    for (int i = 0; i < (n - window); i++)
    {

          sum = 0.0;
          // Loop through window numbers from current i position
          int k=0;
          for (int j = i; j < i + window; j++)
          {
                sum += a[k]*array[j];
                k++;
          }

          array[i]=result;
    }
}
```

Mint az eddigiekből látható, a FIR szűrőkkel a bemenő jel időtartománybeli viselkedését könnyen tudjuk befolyásolni. Azonban mit kell csinálnunk, ha a jel frekvenciatartománybeli viselkedését szeretnénk befolyásolni? Ebben az esetben a következő lépéseket kell végrehajtani:

- először a bemenő jelet Fourier transzformációval áttranszformáljuk a frekvencia térbe,
- az általunk megadott magfüggvénnyel beszorozzuk a jel spektrumát (elvégezzük a konvolúciót),
- a kapott spektrumot az inverz Fourier-transzformációval visszatranszformáljuk az időtérbe.

Bizonyítás nélkül itt jegyezzük meg, hogy a konvolúció műveletének a frekvenciatérben a szorzás felel meg.

A FIR szűrők nagy pontosságúak és jó elnyomási karakterisztikával rendelkeznek.

4.2 IIR (rekurzív) szűrők

A rekurzív szűrők kimeneti jele nemcsak a bemeneti jeltől, hanem a szűrő korábbi kimeneti jelétől is függ. A rekurzív szűrők működése az alábbi matematikai formulával adható meg:

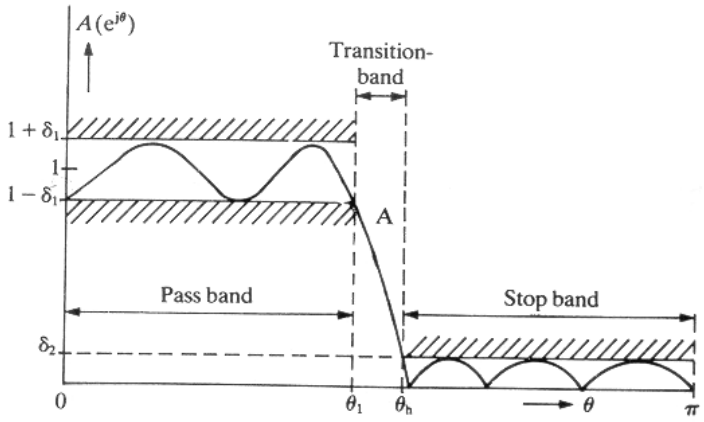
$$y_n = \sum_{i=0}^{k} a_i x_{n-i} - \sum_{i=0}^{k} b_i y_{n-i} ,$$

ahol x_n a bemenő jel, a_i és b_i a szűrő viselkedését megadó függvény i-edik értéke, míg y_n a kimenő jel n-edik értéke. Amennyiben szűrési folyamatba nem csatoljuk vissza a korábbi szűrés eredményét, azaz az egyenlet jobb oldalán nem szerepel a kimenő jel, akkor visszakapjuk a FIR szűrő működését leíró egyenletet. Az IIR szűrő működését definiáló egyenlet matematikai szempontból rekurzív egyenlet, ezért nevezik ezeket a szűrőket rekurzív szűrőknek. A matematikában rekurzívnak nevezünk egy függvényt, ha a függvényt definiáló utasítás részben önmaga egy változatát hívja.

Az IIR szűrők előnye a FIR szűrőkkel szemben a gyors végrehajtás, viszont pontosságuk nem olyan jó, mint FIR szűrőké.

4.3 Digitális szűrők méretezése

A szűrők méretezése a frekvencia tartománybeli amplitúdó karakterisztika ($H(\omega)$) megadásával kezdődik. Ez a gyakorlatban a tolerancia diagrammal történik (4.6. ábra)



4.6. ábra: Aluláteresztő szűrő paramétereinek a specifikációja [9].

A diagramon a δ_l, Θ_l, és Θ_h értékek megadásával határozzuk meg, hogy az amplitúdó karakterisztikát meddig tekintjük elfogadhatónak. A szűrők tervezésénél a lineáris fázisátmenet (a jel terjedési késleltetése minden frekvencián azonos) alapkövetelmény. A tervezés iteratív módon a következő lépésekből áll:

1. Előállítjuk az x[nT] diszkrét jel Fourier transzformáltját

$$X\left(e^{j\omega T}\right)=\sum_{n=-\infty}^{n=+\infty}x[nT]e^{-jn\omega T}$$

2. Meghatározzuk, hogy FIR vagy IIR struktúrával akarjuk-e közelíteni az előírt paramétereket
3. Megadjuk a szűrő fokszámát és meghatározzuk az együtthatókat
4. Megválasztjuk a szűrő struktúrát a kvantálás hatásának a figyelembe vételével
5. Ellenőrizzük, hogy a szűrő eleget tesz –e az előírt specifikációknak. Ha nem, akkor módosítjuk a paramétereket és a módosítás után megismételjük az eljárást.

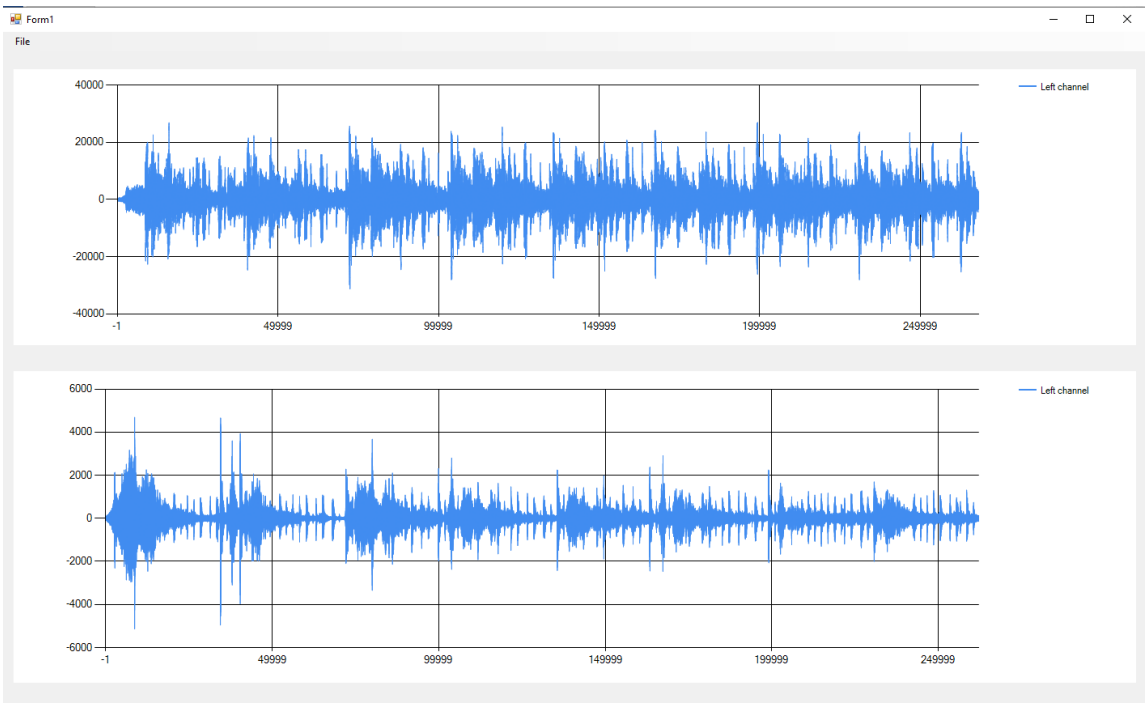
A szűrő együtthatók meghatározása

Az amplitúdó-frekvencia diagramban megadott karakterisztika megközelítése a megadott tűréshatárokon belül elsősorban az együtthatók meghatározási módjától függ. A FIR szűrők esetén a tervezés az ideális amplitúdó-frekvencia karakterisztikából ($H_d\left(e^{j\theta}\right)$) indul ki. E diagram lehető legjobb közelítése a cél. Közvetlenül az ideális átviteli karakterisztikából inverz Fourier transzformációval meghatározzuk az impulzus választ ($h_d[n]$). Ez alapján azonban a szűrőt nem lehet megvalósítani, mivel az impulzus válasz nagy vagy végtelen hosszúságú és nem kauzális, mert $n<0$ esetén $h_d[n] \neq 0$

Emiatt az impulzus válasz függvény hosszát limitálni kell egy elfogadható L hosszra, valamint a kapott válasz függvényt el kell tolni (késleltetni kell), hogy az impulzus válasz kauzális legyen. Az így kapott h[n] impulzus válasz értékei megegyeznek a szűrő együtthatókkal.

4.4 Gyakorlati példa

Gyakorlati példaként egy 8 kHz mintavételezésű wav állományon mutatjuk be a digitális szűrés hatását. A 4.7-es ábrán, a felső csatornán látható az eredeti szűrés nélküli wav állomány. Mivel a mintavételi frekvencia 8 kHz volt, ezért a jelben előforduló legnagyobb hangfrekvencia maximum 4 kHz. Ezt a hangállományt átengedtük egy ideális (brick-wall) rendszerű sávszűrőn. Az áteresztési sáv: 500-1500 Hz. A szűrés eredménye az alsó csatornán látható.



4.7. ábra: Ideális (Brick- wall) sávszűrő hatása.

A példában a hangállomány kezeléséért a Wave class felel:

```
public class Wave
{
 public struct WavHeader
 {
   public byte[] riffID; // "riff"
   public uint size;
   public byte[] wavID;  // "WAVE"
   public byte[] fmtID;  // "fmt "
   public uint fmtSize;
   public ushort format;
   public ushort channels;
   public uint sampleRate;
   public uint bitPerSec;
   public ushort blockSize;
   public ushort bit;
   public byte[] dataID; // "data"
   public uint dataSize;
}
```

```
public WavHeader Header = new WavHeader();
public List<short> lDataList = new List<short>();
public List<short> rDataList = new List<short>();

public double getDuration()
{
 double duration = (Header.size - 4) /
               (Header.sampleRate *  Header.channels * Header.bit / 8);
 return duration;
}



public void Load(string fname)
{
 using FileStream fs = new FileStream(fname, FileMode.Open,
                                      FileAccess.Read))
 using (BinaryReader br = new BinaryReader(fs))
 {
  try
  {
    Header.riffID = br.ReadBytes(4);
    Header.size = br.ReadUInt32();
    Header.wavID = br.ReadBytes(4);
    Header.fmtID = br.ReadBytes(4);
    Header.fmtSize = br.ReadUInt32();
    Header.format = br.ReadUInt16();
    Header.channels = br.ReadUInt16();
    Header.sampleRate = br.ReadUInt32();
    Header.bitPerSec = br.ReadUInt32();
    Header.blockSize = br.ReadUInt16();
    Header.bit = br.ReadUInt16();
    Header.dataID = br.ReadBytes(4);
    Header.dataSize = br.ReadUInt32();

    for (int i = 0; i < Header.dataSize / Header.blockSize; i++)
    {
      lDataList.Add((short)br.ReadUInt16());
      rDataList.Add((short)br.ReadUInt16());
    }
  }
  finally
  {
    if (br != null)
    {
     br.Close();
    }
    if (fs != null)
    {
     fs.Close();
    }
   }
  }
 }
}
```

A példa programban a görbék megjelenítésére az MS .NET chart vezérlőjét használtuk. Az alkalmazás lényegi részének forráskódja:

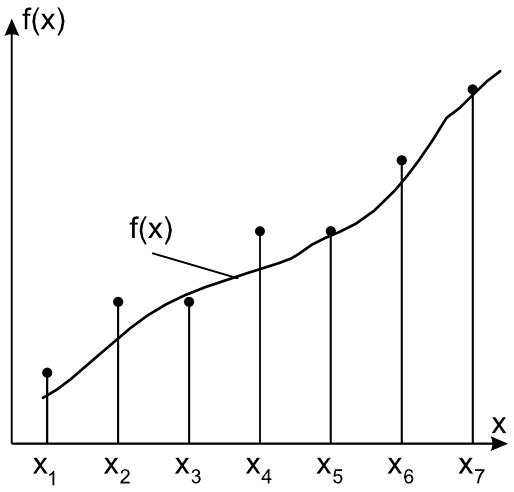
```
public partial class Form1 : Form
{
  protected Wave wave;

  public Form1()
  {
     InitializeComponent();
     wave = new Wave();
  }


 private void openToolStripMenuItem_Click(object sender, EventArgs e)
 {
  OpenFileDialog ofd = new OpenFileDialog();
  if (ofd.ShowDialog() == DialogResult.OK)
  {
   try
   {
    wave.Load(ofd.FileName);

    double duration = wave.getDuration();

    double[] left = new double[wave.lDataList.Count];
    double[] right = new double[wave.rDataList.Count];

    for (int i=0; i< wave.lDataList.Count; i++)
    {
      chart1.Series[0].Points.AddXY(i, wave.lDataList[i]);

      left[i] = wave.lDataList[i];

    }

    FFT fft = new FFT();
    double[] zr;
    zr= fft.bandPassFilter(left, wave.Header.sampleRate,500, 1500);
    for (int i = 0; i < zr.Length/2; i++)
    {
       chart2.Series[0].Points.AddXY(i, zr[i]);
    }

  }
  catch (Exception ex)
  {
     MessageBox.Show(ex.Message);
  }
 }
}
```

4.5 Kérdések, feladatok

1. Frekvenciatartománybeli viselkedésük alapján milyen fajta szűrőket különböztetünk meg?
2. Mit nevezünk a szűrő átmeneti függvénynek?
3. Írja fel két folytonos függvény konvolúciójának matematikai alakját!
4. Írja fel két diszkrét függvény konvolúciójának matematikai alakját!
5. Mit nevezünk FIIR szűrőnek?
6. Írja fel a FIIR szűrő matematikai alakját!
7. Írja fel az átlagoló szűrőt definiáló kifejezést.
8. Mire használható az átlagoló szűrő?
9. Írja fel a IIR szűrő matematikai alakját!
10. Matematikai szempontból miben különbözik az IIR szűrő a FIR szűrőtől?
11. Milyen módszereket ismer FIIR szűrő tervezésére?

5. Modellillesztés, paraméterbecslés

A jelfeldolgozás során gyakori feladat az, hogy a mért jelekből *kvantitatív* következtetéseket vonjunk le a lezajlott folyamatokról. Ennek során az első lépés a jel (mérési eredmények) grafikus ábrázolása. Ez gyakran lehetővé teszi a mért adatok tulajdonságainak vizuálisan (szemmel) történő felismerését, s az első kvalitatív modell megadását. A mérési eredményeket leggyakrabban derékszögű koordinátarendszerben ábrázoljuk. A koordinátarendszerben a vízszintes tengelyen az időt (független változó), míg a függőleges tengelyen a mért mennyiséget (függő változó) ábrázoljuk. A grafikus ábrázolás során a kapott ábra akkor nem lesz torz, ha a két tengely beosztása megközelítően „egyforma”. Ezt követően olyan görbe vonalat kell húzni a mért adatokat ábrázoló pontok közé, hogy pontról pontra az adott pont és a görbe közötti távolság kicsi legyen (5.1 ábra). Hangsúlyozni kell, hogy nem a pontok közvetlen összekötése a cél, hanem a mért folyamat törvényszerűségeit legjobban megközelítő kvantitatív összefüggés megtalálása. Azt a függvényt, mely kielégíti a fenti követelményt *illesztési függvénynek* nevezzük.



5.1. ábra: A mért pontokhoz illesztett görbe

Az illesztési függvény kiválasztásakor figyelembe kell venni:

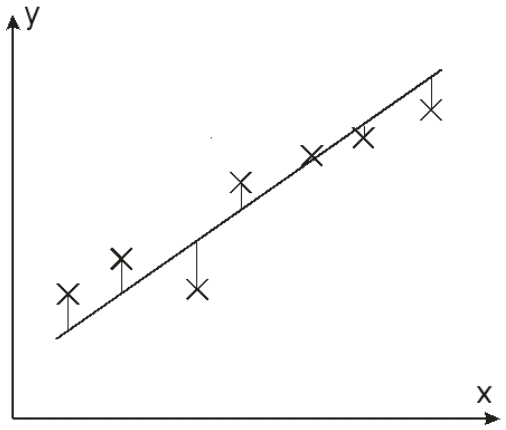
- az adott mérési pontokhoz tartozó elméleti megfontolásokat,
- az ábrázolás során kapott grafikont,
- a lehetséges illesztési függvények matematikai formuláját, együtthatóik jelentését.

Összefoglalva olyan illesztési függvényt kell választanunk, mely összhangban van a tapasztalatainkkal, a konkrét jelenség elméleti hátterével, és ugyanakkor a mért adatokhoz

a lehető legközelebb van. Mivel a matematikában bizonyított tény, hogy bármely folytonos függvény tetszőleges pontossággal közelíthető polinomokkal, ezért mondhatnánk azt is, hogy válasszuk az illesztési függvényt elég magas fokszámú polinomnak. Ekkor azonban semmi sem garantálja, hogy a polinom együtthatói, valamint az illesztett függvény alakja tényleges fizikai jelentéssel bír.

5.1 Legkisebb négyzetek módszere

Miután kiválasztottuk a megfelelőnek ítélt illesztési függvényt, meg kell határoznunk annak együtthatóit. A szakirodalomban a számítások elvégzésére a **GAUSS** által javasolt legkisebb négyzetek módszerén alapuló algoritmust használják. Ez gyakorlatilag a következőt jelenti: az illesztési függvény együtthatóit úgy határozzuk meg, hogy a függvény egyenletéből szerkesztett görbe és a mérési adatok közötti („függőleges") távolságok négyzeteinek összege minimális legyen (5.2. ábra).



5.2. ábra: Legkisebb négyzetek módszere

Matematikai alakban:

$$\min S = \sum_{j=1}^{N} \left(y_j - f(t_j) \right)^2,$$

ahol y_j a mért érték, $f(t_j)$ pedig az illesztési függvény értéke a t_j időpontban (a *min* szócska pedig egy bevett matematikai, optimalizáláselméleti jelölésmód, megegyezés szerint azt jelenti, hogy a feladatunk minimalizálási, azaz egyfajta – korlátozó feltételek melletti – szélsőérték meghatározása).

Az illesztés jóságát az *r* korrelációs együttható adja meg:

$$r = \frac{N\sum_{i=1}^{N} x_i y_i - \sum_{i=1}^{N} x_i \sum_{i=1}^{N} y_i}{\sqrt{\left(N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2\right)\left(N\sum_{i=1}^{N} y_i^2 - \left(\sum_{i=1}^{N} y_i\right)^2\right)}},$$

ahol $x_i = f(t_i)$. A korrelációs együttható értéke mindig -1 és 1 között van. A gyakorlati életben ezért ennek a négyzetét az r^2-et szokták használni az illesztés minősítésére. Minél közelebb van az r^2 az 1-hez annál jobb az illesztés. Élettani rendszereknél az 1-hez túl közeli r^2 érték már gyanús, ugyanis ezeknél a rendszereknél összetettségük miatt a mérések során nagy a bizonytalansági tényező. Orvos – biológiai területen a gyakorlati tapasztalatok szerint a 0,85-nél nagyobb r^2 már kiváló értéknek számít.

5.2 Egyenes illesztése

A számítások menetét a legegyszerűbb példán, az egyenes illesztésén keresztül mutatjuk be. Legyen a keresett egyenes egyenlete a következő:

$$y = mt + b.$$

Az illesztés során meghatározandó paraméterek az egyenes meredeksége (m) és a tengelymetszete (b). Ekkor a minimum feltétel a következőképpen írható:

$$\min \; S = \sum_{j=1}^{N} \left(y_j - f(t_j)\right)^2 = \sum_{j=1}^{N} \left(y_j - mt_j - b\right)^2,$$

ahol az illesztési egyenes két paramétere ismeretlen. A matematikából ismert, hogy egy többváltozós függvény szélsőértéke (esetünkben minimuma) ott lehet, ahol az egyes változók szerinti parciális differenciálhányadosok értéke 0. Matematikai alakban megfogalmazva, ahol

$$\frac{\partial S}{\partial m} = 0; \quad \frac{\partial S}{\partial b} = 0.$$

Beírva a konkrét függvényalakot és elvégezve az egyes változók szerinti parciális deriválást az alábbi egyenletrendszerhez jutunk:

$$\frac{\partial S}{\partial m} = \sum_{j=1}^{N} -2\left(y_j - mt_j - b\right)t_j = 0,$$

$$\frac{\partial S}{\partial b} = \sum_{j=1}^{N} -2\left(y_j - mt_j - b\right) = 0 .$$

Átrendezve az egyenleteket kapjuk, hogy:

$$\sum_{j=1}^{N} t_j y_j = m\sum_{j=1}^{N} t_j^2 + b\sum_{j=1}^{N} t_j ,$$

$$\sum_{j=1}^{N} y_j = m\sum_{j=1}^{N} t_j + bN ,$$

Ennek alapján keresett két paramétert meghatározó összefüggés:

$$m = \frac{N\sum_{j=1}^{N} t_j y_j - \sum_{j=1}^{N} t_j \sum_{j=1}^{N} y_j}{\sum_{j=1}^{N} t_j^2 - \left(\sum_{j=1}^{N} t_j\right)^2},$$

$$b = \frac{N\sum_{j=1}^{N} y_j - m\sum_{j=1}^{N} t_j}{N} .$$

Az, hogy ez az illesztés mennyire jó, azt az r^2 kiszámolásával tudjuk megmondani. Az r^2 értéke mellett az alábbi képlettel kiszámolható az illeszkedési szórás négyzet, mely azt mutatja meg, hogy a mért pontok mennyire térnek el az illesztési függvénytől:

$$S_r^{\,2} = \frac{\sum_{j=1}^{N}\left(y_j - mt_j - b\right)^2}{N-2}$$

Az egyenes illesztést megvalósító programkód:

```
class LinearFit
{
  public double A;
  public double B;
  public double R2;
  public double S2;

  public void Fit(double []x, double[]y)
  {
   double sumx=0;
   double sumx2=0;
   double sumy=0;
   double sumxy=0;

   protected int N=x.Length;

   // calculate A,B
   for( int =0; i<N; i++)
   {
     sumx=sumx +x[i];
     sumx2=sumx2 +x[i]*x[i];
     sumy=sumy +y[i];
     sumxy=sumxy +x[i]*y[i];

   }

   A = ((sumx2*sumy -sumx*sumxy)/(N*sumx2-sumx*sumx));
   B = ((N*sumxy-sumx*sumy)/(N*sumx2-sumx*sumx));

   //calculate R2

   double r= N*sumxy-sumx*sumy;
   double q=Math.Sqrt((N*sumx2-sumx*sumx)*(N*sumy2-sumy*sumy));

   R2=r/q;

   //calculate S2

   double v=0;
   for( int i=0; i<N; i++)
   {
      v+=(y[i]-A*x[i]-B)* (y[i]-A*x[i]-B);
   }
   S2=v/(N-2);
 }
}
```

5.3 Nemlineáris függvény illesztése

Sok esetben a különböző fizikai mennyiségek között *nem lineáris* jellegű kapcsolat van. Ekkor a mért értékekre *nemlineáris függvényt* kell illeszteni. Erre két lehetőségünk van:

Amennyiben az adott függvény csak két paraméterrel rendelkezik, akkor az adott függvényt valamilyen transzformációval lineáris függvénnyé transzformáljuk, majd erre egyenest illesztünk. A kapott paramétereket, a transzformáció inverzével visszatranszformáljuk. Az 5.1. táblázatban néhány linearizálható függvény látható.

Típus	Függvény	T	Y	A	B
Lineáris	$y = At + B$	t_i	y_i	α	β
Exponenciális	$y = Ae^{Bt}$	t_i	$\ln(y_i)$	e^{α}	β
Hatványfüggvény	$y = At^B$	$\ln(t_i)$	$\ln(y_i)$	e^{α}	β
Arrhenius	$y = Ae^{-\frac{B}{t}}$	$-\frac{1}{t_i}$	$\ln(y_i)$	e^{α}	β
Reciprok	$y = \frac{1}{A + Bt}$	t_i	$\frac{1}{y_i}$	α	β
Racionális tört	$y = \frac{At}{1 + Bt}$	t_i	$\frac{t_i}{y_i}$	$\frac{1}{\alpha}$	$\frac{\beta}{\alpha}$
Kvadratikus	$y = t(A + Bt)$	t_i	$\frac{y_i}{t_i}$	α	β
Vektoriális	$y = \sqrt{A + Bt^2}$	t_i^2	y_i^2	α	β
Hiperbola	$y = A + \frac{B}{t}$	$\frac{1}{t_i}$	y_i	α	β
Logaritmikus	$y = A\ln(Bt)$	$\ln(t_i)$	y_i	β	$e^{\frac{\alpha}{\beta}}$

5.1. táblázat: Linearizálható függvények.

Általános esetben szintén a legkisebb négyzetek módszerét lehet alkalmazni. Ekkor a minimalizálandó kifejezés

$$\min\ S = \sum_{j=1}^{N} \left(y_j - f(t_j, a, b, c, ...) \right)^2,$$

ahol $y = f(t, a, b, c, ...)$ az illesztendő függvény és *a*, *b*, *c*, ... a keresett paraméterek.

Például parabola esetén a kifejezés a következő lesz:

$$\min \quad S = \sum_{j=1}^{N} \left(y_j - at_j^2 - bt_j - c \right)^2.$$

Ekkor a minimum kiszámításához a következő negyedfokú egyenletrendszert kell megoldani:

$$\begin{cases} cN + b\Sigma t + a\Sigma t^2 = \Sigma y \\ c\Sigma t + b\Sigma t^2 + a\Sigma t^3 = \Sigma ty \\ c\Sigma t^2 + b\Sigma t^3 + a\Sigma t^4 = \Sigma t^2 y \end{cases}$$

ahol *a*, *b* és *c* a keresett paraméterek értékei.

Általános esetben a legjobb illeszkedést biztosító paraméterek kiszámítása a **LEVENBERG-MARQUARDT** algoritmussal is történhet.

5.4 Kérdések, feladatok

1. Mi az illesztési függvény, és sorolja fel a kiválasztásakor figyelembe veendő szempontokat!
2. Ismertesse a legkisebb négyzetek módszert!
3. Mondjon példákat linearizálható függvényekre!
4. Mit mutat meg az r^2 és az illeszkedési szórás négyzet?

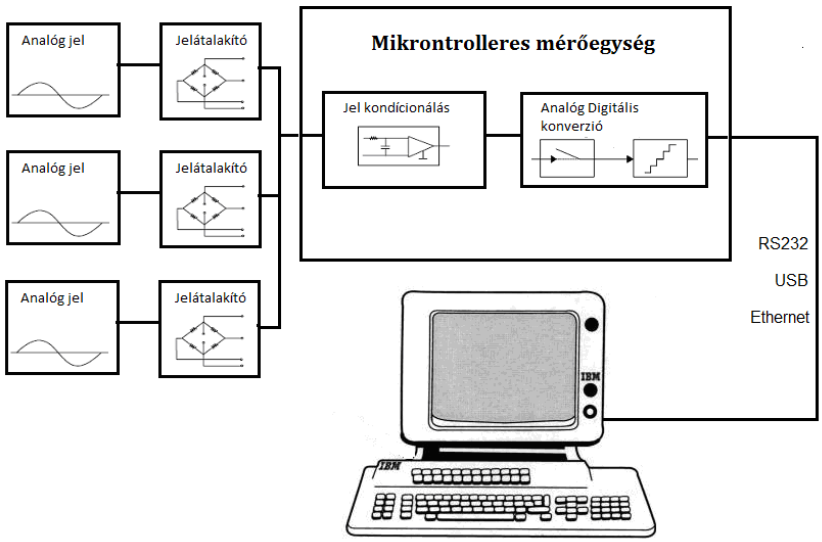
6. Számítógépes méréstechnika

6.1 Számítógépes mérőrendszer feladata, struktúrája

A számítógépes méréstechnikában a számítógép feladata az információ összegyűjtése, feldolgozása, tárolása. Azonban napjainkban széleskörűen használt okos telefonok, tabletek, személyi számítógépek (laptop, desktop) az estleges mikrofon bemeneten kívül nem rendelkeznek olyan adatbeviteli kapukkal, amely lehetővé tenné az analóg jelek fogadását. Így az ilyen számítógépekkel megvalósított mérő rendszerekben a számítógépek feladata az az alábbiakban foglalható össze:

- az analóg jelek tényleges mérését végző műszerek és egyéb perifériák vezérlése
- a mérőműszerről érkező digitális jelfolyamok gyűjtése, kiértékelése, tárolása
- a jelekből kinyert információ megjelenítése.
- mérési folyamat dokumentálása

Egy ilyen rendszerben a tényleges mérést végző mérőműszernek az analóg jelet mindenképpen digitalizálnia kell. Erre a feladatra kiválóan alkalmasak a különböző *mikrokontroller*ek. A 2.1. ábra egy mikrokontrollerrel és számítógéppel megvalósított mérő rendszer blokksémája látható.

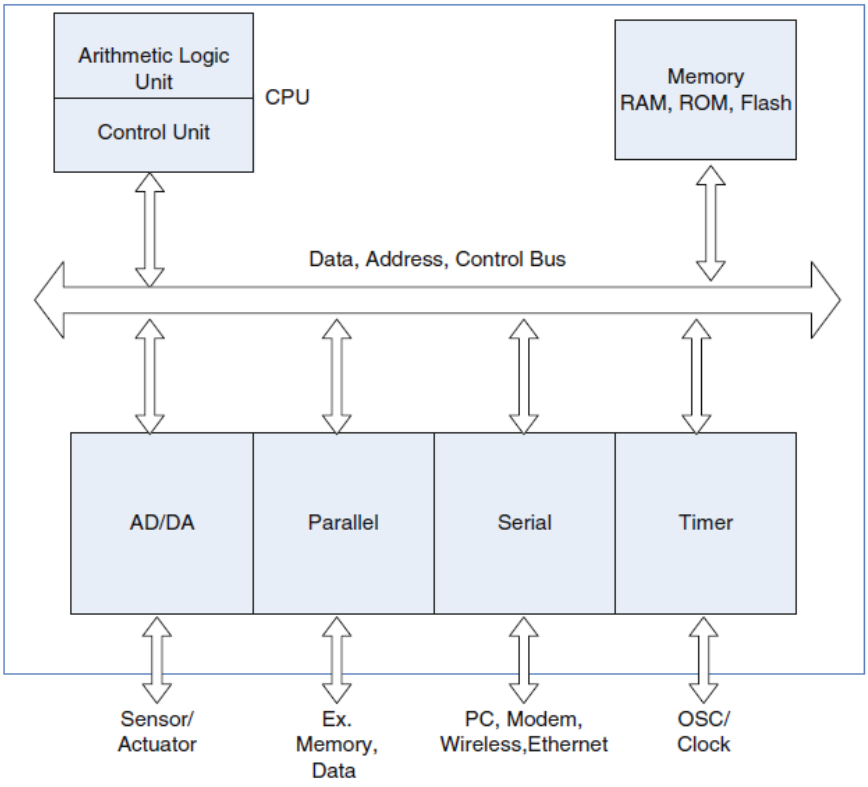


2.1. ábra: Számítógépes mérőrendszer blokksémája .

Az akár több csatornás analóg jel a mérőegységben csatornánként kondicionálásra kerül, majd az AD átalakító digitális jellé alakítja. Az A/D átalakítóból kijövő minden egyes digitális jel érték egy a mikrokontroller memóriájában lefoglalt pufferbe kerül. Innen a megfelelő digitális interfészen (USB, RS232, Ethernet) keresztül az adattároló, jelfeldolgozó számítógépbe kerül. Egy ilyen rendszerben két egymástól teljesen elkülönülő szoftver dolgozik: a mérőegység beágyazott szoftvere, és a számítógépen futó magas szintű feladatokat ellátó jelfeldolgozó szoftver. A mikrokontroller szűkös memória kapacitása és a személyi számítógépeknél sokkal kisebb sebessége miatt ebbe az egységbe komolyabb számításokat igénylő algoritmusokat nem építenek bele. Legfontosabb feladata az analóg beviteli csatornákon érkező jelek digitalizálása és ezen a digitalizált jeleknek szabványos kommunikációs portokon történő továbbítása a feldolgozó számítógép felé. A számítógépen futó jelfeldolgozó szoftver feladata az alapfeladatokon (jelek fogadása a mikrokontrolleres egységtől, jelek tárolása, grafikus megjelenítése) kívül azon algoritmusok végrehajtása, melyek a felhasználó számára értelmezhetővé teszik a jel által hordozott információt. Azaz a valós világ jeleinek mérése a mikrokontrolleres oldalon, míg tényleges jelfeldolgozás a számítógépes oldalon valósul meg.

6.2 Mikrokontrollerek felépítése

A mikrokontrollerek, hasonlóan a mikroprocesszorokhoz egyetlen áramköri lapkán megvalósított számítógépek. A 6.2. ábrán látható, hogy a mikrokontrollerek a központi számolóegységen (CPU) kívül tartalmaznak még: beépített memóriát (RAM, ROM, flash-memória), az egységeket összekapcsoló buszrendszert, időzítő egységeket, és a külvilággal kapcsolatot tartó beviteli kiviteli (I/O) kapukat (port).



6.2. ábra: Mikrokontroller sematikus blokkvázlata.

Az I/O portokon keresztül a mikrokontroller adatot küldhet a külvilágnak, másrészt fogadhat adatot. Ezek az adatok mind *analóg*, mind *digitális* természetűek lehetnek.

Kommunikációs kapuk (portok) szolgálnak más mikrokontrollerekkel, illetve számítógépekkel történő kommunikációra. Felépítésüket tekintve ezek a kapuk *soros* (RS232, Ethernet, USB, I^2C, SPI), illetve *párhuzamos* adatátvitelt valósítanak meg. Soros adatátvitelnél egyszerre mindig csak egy bitet továbbítunk a vevőnek. Párhuzamos adatátvitel esetén egyszerre egy időben mindig bitcsoportot továbbítunk. Ehhez viszont annyi adatátviteli szálra van szükségünk, ahány bitet egyszerre továbbítani szeretnénk.

Innen már látható, hogy a mikroprocesszorokkal szemben a mikrokontrollerek beépítve tartalmazzák a külvilággal való kommunikációhoz szükséges csatoló felületeket, AD/DA átalakítókat. Azaz minden olyan eszközt, amely lehetővé teszi, hogy a valós világ analóg jeleit digitalizálni lehessen. Cserébe viszont a mikrokontroller CPU-része kisebb utasításkészlettel rendelkezik, mint a mikroprocesszor, hasonlóan a beépített memóriája is jóval kevesebb, és általában a működést biztosító órajel is kisebb. Ennek alapján a mikrokontroller önmagában nem alkalmas bonyolult, számításigényes DSP feladatok ellátására.

Ilyen jellegű feladatoknál az analóg világ jeleinek kezelésére mikrokontrollert tartalmazó mérődobozt és a már digitalizált jelek tárolását, megjelenítését, és feldolgozását végző számítógépet együttesen kell használni.

Röviden összefoglaljuk az egyes komponensek jellemzőit:

- **CPU**

  A CPU valamennyi mikrokontrollernek a lelke. Felépítését tekintve a CPU egy aritmetikai-logikai egységből (ALU) és egy vezérlő egységből (Control Unit) áll. Az adatokon történő műveletek elvégzéséhez a CPU belső adattároló egységeket un. *regisztereket* használ az adatok elérésére, tárolására illetve a kijelölt műveletek elvégzésére. Ezen kívül szintén regiszterben van tárolva a soron következő utasítás címe is (Program Counter register). A regiszterek között kitüntetett szerepet játszik az akkumulátor regiszter. Ez tárolja az aritmetikai műveletek eredményét.

  Az ALU felel a matematikai (összeadás, kivonás, szorzás, osztás) és a logikai műveletek (ÉS, VAGY, KIZÁRÓ VAGY,stb.) elvégzésért. Azt, hogy az ALU egy műveletet hány biten képes elvégezni határozza meg, hogy a mikrokontroller hány bites (8, 16, 32).

  Fontos jellemző még a CPU órajele. Valamennyi művelet elvégzéséhez a CPU-nak időre van szüksége. A CPU teljesítményét a MIPS-el (Million Instruction Per Second) adják meg.

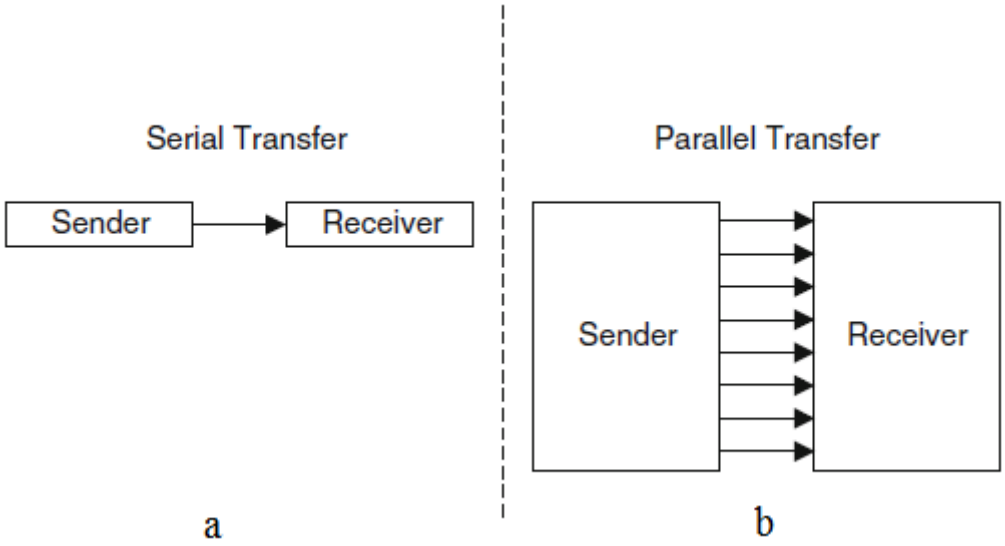
  A vezérlő egység felel a műveletek időzítéséért, sorrendiségéért.

- **Memória**

  A mikrokontroller alapvetően kétféle memóriát használhat: külső, illetve belső. Külső memóriát akkor használunk, ha a belső memória mérete nem elég nagy. Meg kell jegyezni azonban, hogy a külső memória elérésének a sebessége sokkal kisebb, mint a belsőé. Ugyanis minden egyes külső memória művelethez először meg kell címezni azt az I/O portot, melyre a memóriát kötöttük, majd ezen keresztül tudjuk a memóriát címezni. A memória fajtája szerint lehet RAM, ROM, EEPROM,. A kontrollereknél a szoftver kódját a csak olvasható memóriában (ROM, EEPROM) tárolják, míg az adatokat a RAM-ban. Ha a működés során keletkező adatokat hosszabb távon is tárolni kell (pl. naplózás), akkor azt külső memóriaként csatlakoztatott EEPROM-mal, SIM kártyával lehet megvalósítani.

- **I/O portok**

  A mikrokontrollereknél az I/O portok szolgálnak a kontroller külvilággal történő összekapcsolására. A kontroller egyrészt fogadhat adatot a külvilágtól, illetve küldhet adatot a külvilágnak. Ezek az adatok *digitális* illetve *analóg* jelek lehetnek. A digitális jeleket *soros* (6.3/a ábra) illetve *párhuzamos* (6.3/b ábra) interfészen keresztül olvashatja be illetve küldheti el a kontroller. Soros interfészre példa az RS232, az Ethernet, az USB, a CAN, az I^2C, az SPI. Párhuzamos interfészre példa a számítógépek adat-, vezérlő- és címbuszai, vagy perifériák esetén a nyomtató.



6.3. ábra: Soros (a) illetve párhuzamos (b) adattovábbítási technika. Soros adatátvitel esetén az információt bitenként sorban egymásután továbbítjuk. Párhuzamos adatátvitelnél egyszerre mindig bitcsoportot továbbítunk. Ahány bitet akarunk egyszerre továbbítani, annyi adatvonallal kell összekötni az adót és vevőt.

Analóg jelek bevitelénél a jelet először digitalizálni kell, melyre a kontrollerbe épített analóg-digitális (ADC) átalakító szolgál. Természetesen a digitalizálásnál be kell tartanunk a mintavételi törvényt. Erre a mikrokontrollerbe épített időzítő egységek (Timers) használatával van lehetőségünk. Hasonlóan, ha digitális jelet analóg formában kell külvilágba kiadni, akkor azt egy szintén a mikrokontrollerbe épített digitális-analóg (DAC) átalakító szolgál.

- **Busz rendszer**

  A busz rendszer feladata, hogy a mikrokontrollerben levő egységeket (CPU, memória, I/O portok) összekapcsolja, és biztosítsa ezen egységek között az adat áram-

lást. A busz rendszer az adatbuszból (Data bus), címbuszból (Address bus), valamint a vezérlő buszból (Control bus) áll. Az adatbuszon történik az adatok áramlása. A címbuszal történik a memória illetve I/O portok címzése. A vezérlő buszt a CPU használja a műveleti sorrend vezérlésére.

6.3 Soros kommunikáció

A jelfeldolgozó szoftver első feladata a mérőegységből érkező digitális jelek fogadása memória pufferben történő tárolása. Napjainkban s mérőműszer és a jelfeldolgozó egység közötti adatcserére soros vonali kommunikációt használnak.

A soros vonali kommunikáció során két eszköz kommunikál egymással: az *adó (Transmitter)* és a *vevő (Receiver)*. A kommunikációs csatornán (vonalon) az adatok bitekre lebontva adott időütem szerint ütemezve bitenként továbbítódnak az adótól a vevőig. Természetesen a kommunikáció megkezdése előtt ezt az időütemet definiálni kell. A kommunikáció tartama alatt ez az idő ütem nem változtatható. Az átvitel sebességét az időütemezés határozza meg. Minél rövidebb az egyes bitek átviteléhez szükséges idő, annál gyorsabb a kommunikáció. Természetesen az átviteli közeg sávszélessége fizikailag is behatárolja az ütemezést, azaz az egyes bitek átvitelhez szükséges időtartamot nem lehet tetszőlegesen kicsivé tenni.

A soros kommunikáció az alábbi 5 rétegből épül fel:

- **Fizikai réteg**
  Ez lényegében a fizikai csatlakozók kialakítását, méretét, az alkalmazott vezetékek számát határozza meg.
- **Elektromos réteg**
  A kommunikáció során alkalmazott feszültség szinteket határozza meg.
- **Logikai réteg**
  Mely feszültség szint jelenti a logikai 0-át, és mely a logikai 1-et.
- **Adat réteg**
  Itt határozódik meg az adatátvitel sebessége és az adatbitekből az adatcsomag előállítása.
- **Alkalmazási réteg**
  Ez határozza meg, hogy az adatcsomagokat milyen sorrendben kell elküldeni.

Példaként megemlítünk néhány soros vonali kommunikációs technológiát:

- RS232
  Ez egy peer-to peer azaz két eszköz között megvalósított full duplex kommunikáció.
- RS485
  Az RS485 szabvány programozói szempontból megegyezik az RS232-vel, azonban ez hardver szinten több eszköz között megvalósított kommunikáció.
- USB
  Az USB-t (Universal Serial Bus) az RS232 szabvány felváltására hozták létre.
- SPI
  Az SPI (Serial Peripheral Interface) a Motorolla cég fejlesztése. Master/Slave rendszerű kommunikációt valósít meg.
- I^2C
- Az I^2C (Inter Integrated Circuit) a Philips cég fejlesztése. Multi master kommunikációt valósít meg.

A soros kommunikációnak kétféle változata van: *szinkron* és *aszinkron*. A szinkron kommunikáció esetében az adó és a vevő a továbbítandó adatblokk átadásának idején szinkronban maradnak. Az aszinkron soros kommunikációnál viszont az adó és a vevő csak egyetlen karakter továbbításának idejéig szinkronizálódnak össze. Ráadásul mind az adó mind a vevő saját órajelet használ. Aszinkron kommunikációnál alapesetben a vonal inaktív állapotban van. Adat küldésekor a kommunikációs vonal aktív állapotba kerül, míg az adat elküldése után inaktív állapotban kerülhet.

6.3.1 RS232

Ez egy aszinkron soros kommunikációs protokoll, melynél az adatbiteket kiegészítő bitek (Start bit, Paritás bit, Stop bit) veszik körül. Azaz a kommunikáció *keretekben* (frame) történik. Az adás kezdetét a start bit lefutó éle jelzi, amely mindig alacsony szintű átmenet. Ezt követik az adatbitek. Az adatbiteket a paritás bit követi. A szabvány szerint ennek megadása nem kötelező. Az adatátvitel végét a stop bit jelzi. Ha a vevő a stop bitet érzékelte, akkor az adó újabb start bit küldésével újabb adatbájt vételére szólíthatja fel a vevőt.

A fentiek alapján az RS232 szabványú soros adatátvitelt négy paraméterrel írhatjuk le:

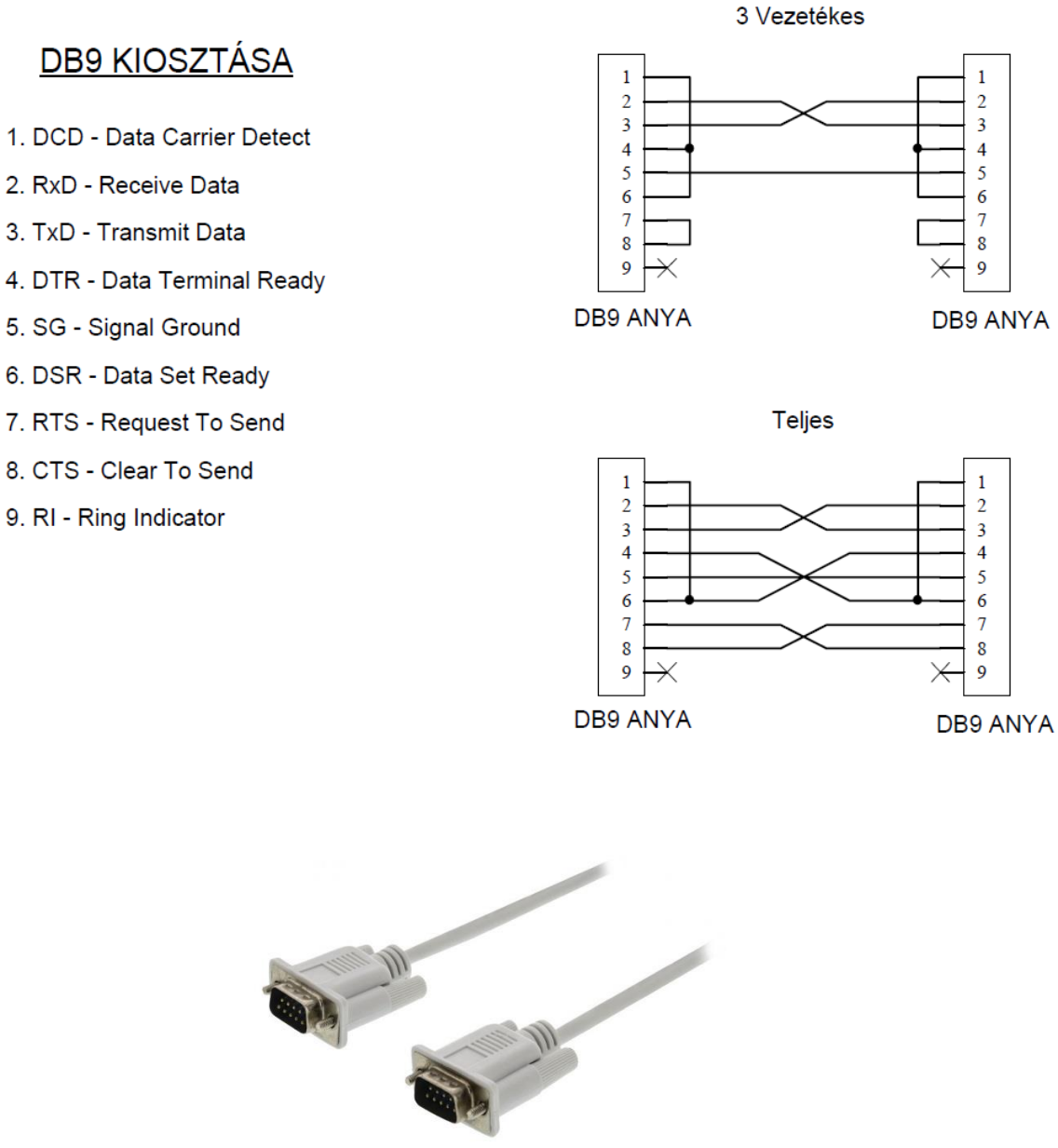
- Az *adatbitek* száma a szabvány szerint 5,6,7 és 8 lehet.
- Szintén a szabvány szerint a *stop bitek* száma 1, 1.5 és 2 lehet. A méret itt természetesen a stop bit idejére vonatkozik, azaz 1.5 stop bit időtartama másfél bitütemet jelent.
- A kommunikáció során az adatátviteli hiba detektálására *paritás biteket* használhatunk. A szabvány szerint ez 0, 1 vagy 2 lehet. A 0 esetén nem használunk paritás ellenőrzést (NO_PARITY). Az 1 esetén páratlan paritás ellenőrzés van (ODD_PARIRTY). A 2 esetén pedig páros paritás ellenőrzés van. (EVEN_PARITY).
- Az átvitel *sebessége* (BAUD RATE) az időegység alatt átvitt bitek száma. Mértékegysége a *bps* (bit/s). Az alkalmazható Baud Rate-re nincs elvi megkötés. Korlátokat csak az alkalmazott átviteli közeg sávszélessége valamint a rendelkezésre álló órajel jelent. A megbízható kommunikációhoz azonban az RS232 szabvány rögzíti a megengedett sebességeket:

Baud Rate (bps)	Egyetlen bit átvitelének időtartama (ms)
150	6.66666
300	3.33333
600	1.66666
1200	0.83333
2400	0.41666
4800	0.20833
9600	0.10416
19200	0.05208
38400	0.02604
57600	0.01736
115200	0.00868

Azonban ez nem az adatbitek tényleges átviteli sebessége, ugyanis az átvitel sebességébe a kiegészítő bitek is benne vannak. Ennek alapján az adatbitek tényleges átviteli sebessége mindig kisebb az alkalmazott Baud Rate-nél.

Példa: 115200 baud = 115200 bit/s. Ha 8 Data bitet, 1 Start bitet, 1 Stop bitet és nem használunk paritásbitet, akkor a valódi átviteli sebesség 115200 *8/10 = 92160 bit/s.

Fizikai szinten a kommunikáció megvalósítására 9 (régebben 25) pólusú un. D-Sub csatlakozót használ (6.4 ábra).



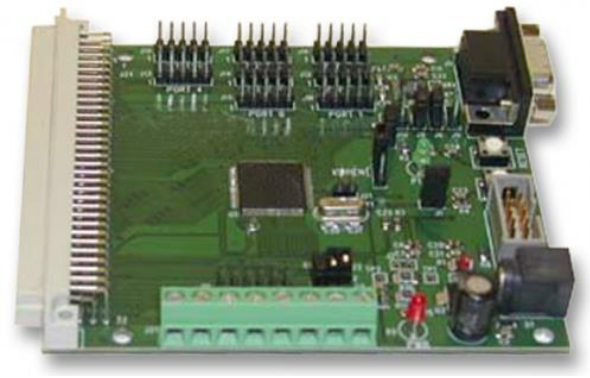
6.4. ábra: D-Sub csatlakozó és lábkiosztásai

Az aszinkron soros kommunikáció során elég az alábbi három vonalat használni:

- RxD : adatfogadás, 2-es láb
- TxD : adatküldés, 3-as láb
- Föld (GND): 5-ös láb

6.4 Fejlesztői csomagok

A fejlesztési megkönnyítése érdekében a gyártók fejlesztői csomagokat is forgalmaznak. Ezek a csomagok fejlesztő kártyából (6.5. ábra), interfészekből és a fejlesztést lehetővé tevő szoftverekből állnak.



6.5. ábra: Mikrokontrolleres fejlesztő kártya

A tényleges szoftverfejlesztés általános célú számítógépen történik és az elkészült szoftver forráskódját keresztfordító program segítségével fordítják le a mikrokontrolleren futó kódra. A futtatás közbeni nyomkövetése kereszt nyomkövetővel történik, mely az általános célú számítógépen futó fejlesztői környezetet egy speciális interfészen (JTAG) kapcsolja össze a beágyazott rendszer processzorával. Ez az interfész egyben a program kód letöltésére is szolgál. Természetesen a beágyazott rendszer szoftvere a szabványos kommunikációs csatornákon (USB, RS232, Ethernet) futás közben nyomkövetési céllal küldhet üzeneteket a fejlesztői gépnek.

6.5 A C8051F120 fejlesztői csomag

Jelen jegyzetben modellként a Silicon Laboratories C8051F120-as mikrokontrollerét használjuk. Ez a kontroller fejlesztői csomagban (C8051F120-DK) is kapható (6.6. ábra) A csomag tartalma:

- fejlesztői kártya,
- JTAG modul,
- USB kábel
- tápegység (AC/DC adapter),
- fejlesztő szoftvert, dokumentációt és példaprogramokat tartalmazó CD.

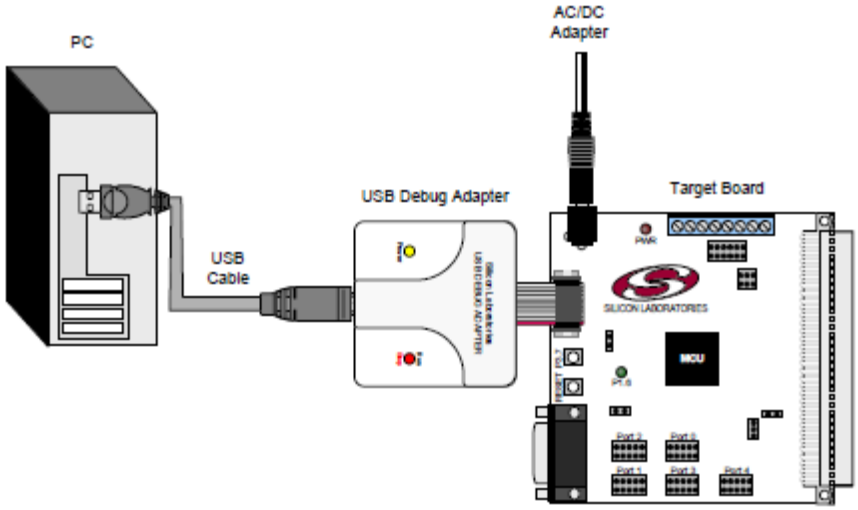
6.6. ábra: A C8051F120 fejlesztői csomag (Development Kit)

A jegyzetben lévő összes példa ezzel a fejlesztő rendszerrel készült.

6.6. Hardver előkészítés

A fejlesztői kártya üzembe helyezése a következő lépésekből áll (6.7. ábra):

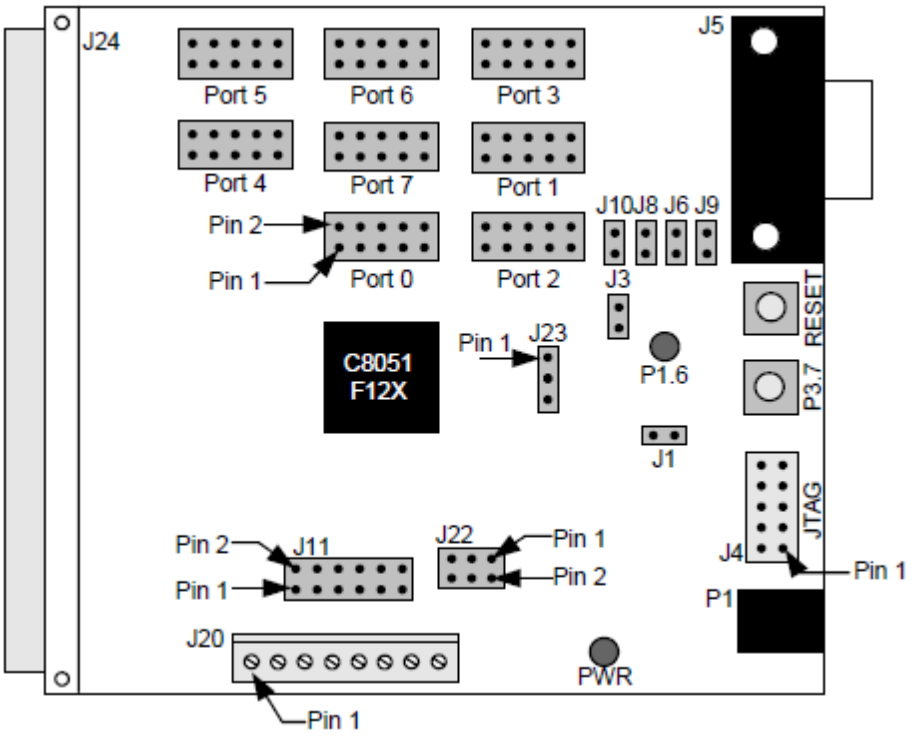
- csatlakoztassuk az USB kábel a JTAG modulhoz
- csatlakoztassuk a JTAG modult a fejlesztői kártyához
- csatlakoztassuk az USB kábel másik végét a fejlesztői géphez
- csatlakoztassuk a tápegységet a fejlesztői kártyához.



6.7. ábra: A C8051F120 fejlesztői kártya és PC összekapcsolása a JTAG modullal

A későbbiek során szükségünk lesz még egy RS232 összeköttetést megvalósító soros kábelre is. Amennyiben a fejlesztői gépen nincs RS232 csatlakozó, akkor használjunk USB - RS232 átalakítót.

A 6.8 ábrán a fejlesztői kártya blokksémája látható.



6.8. C8051F120 fejlesztői kártya

Az egyes eszközök jelentése:

Eszköz	Funkció
P1	Tápegység csatlakozó
J1	P3.7 kapcsoló
J3	P1.6 –ra kötött LED
J4	JTAG csatlakozó
J5	RS232 csatlakozó
J6	UART0 TX port
J8	UART0 RTS port
J9	UART0 RX port
J10	UART0 CTS port
J11	Analóg I/O csatlakozó
J12-19	Digital I/O port csatlakozó
J20	Analóg I/O csatlakozó
J22	Referencia feszültség csatlakozó
J23	V_{dd} monitor letiltó jumper
J24	96 tüskés I/O csatlakozó

6.8. A fejlesztői szoftver telepítése

A fejlesztői csomagban levő CD-ről, vagy a Silicon Laboratories weboldaláról az alábbi szoftver komponenseket kell telepíteni:

- **Silicon Labs IDE**

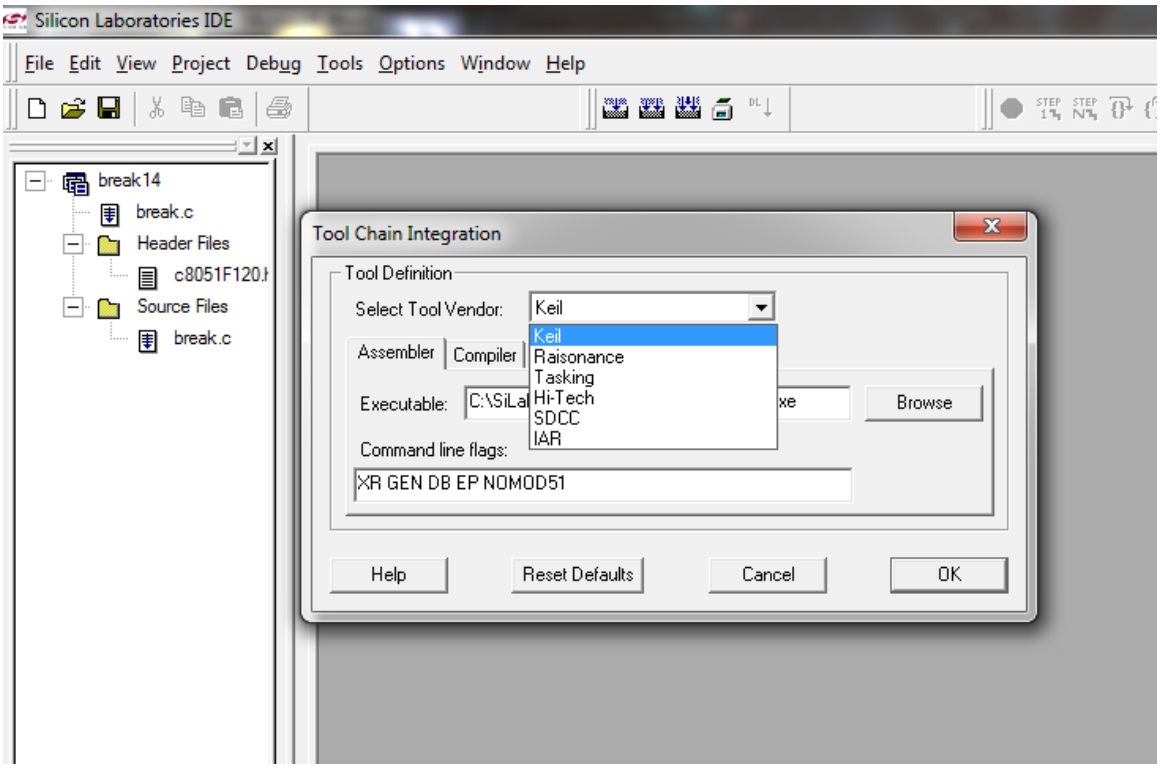
  Ez a Silicon Laboratories integrált fejlesztői környezete. A jegyzet írásának idején a 4.500-ás verziónál tartanak. Mint ahogy a neve is mutatja ez a környezet a szoftverfejlesztés összes munkafolyamatát (forráskód szerkesztés, fordítás, szerkesztés, a fejlesztői kártyára való letöltés, nyomkövetés) támogatja.

- **Configuration Wizard**

  Ez a konfigurációs varázsló a grafikus felületen segíti a beágyazott szoftver által kezelt perifériák (I/O portok, időzítők, stb), helyes beállítását.

- **Keil Compiler Tools Evaluation**

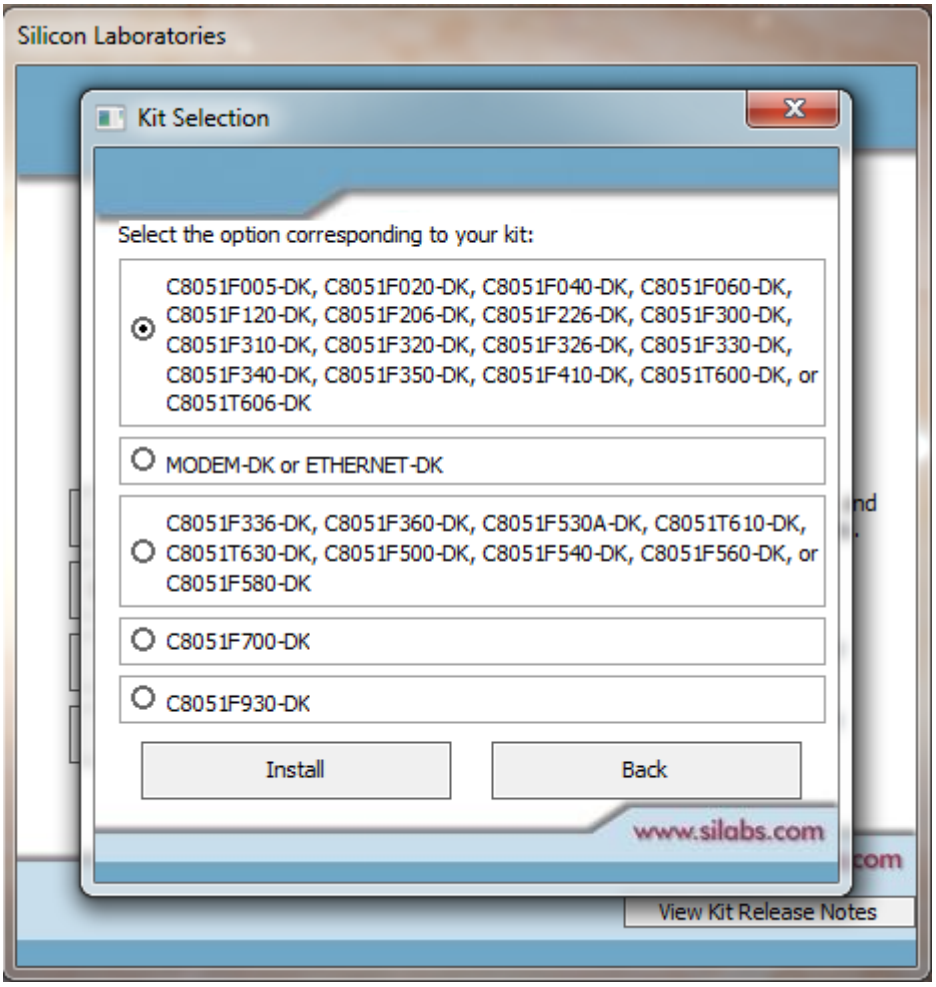
  A fejlesztő környezet (IDE) többféle fordító program használatát támogatja (6.9. ábra). A CD-n a Keil cég fordítóprogramja van.



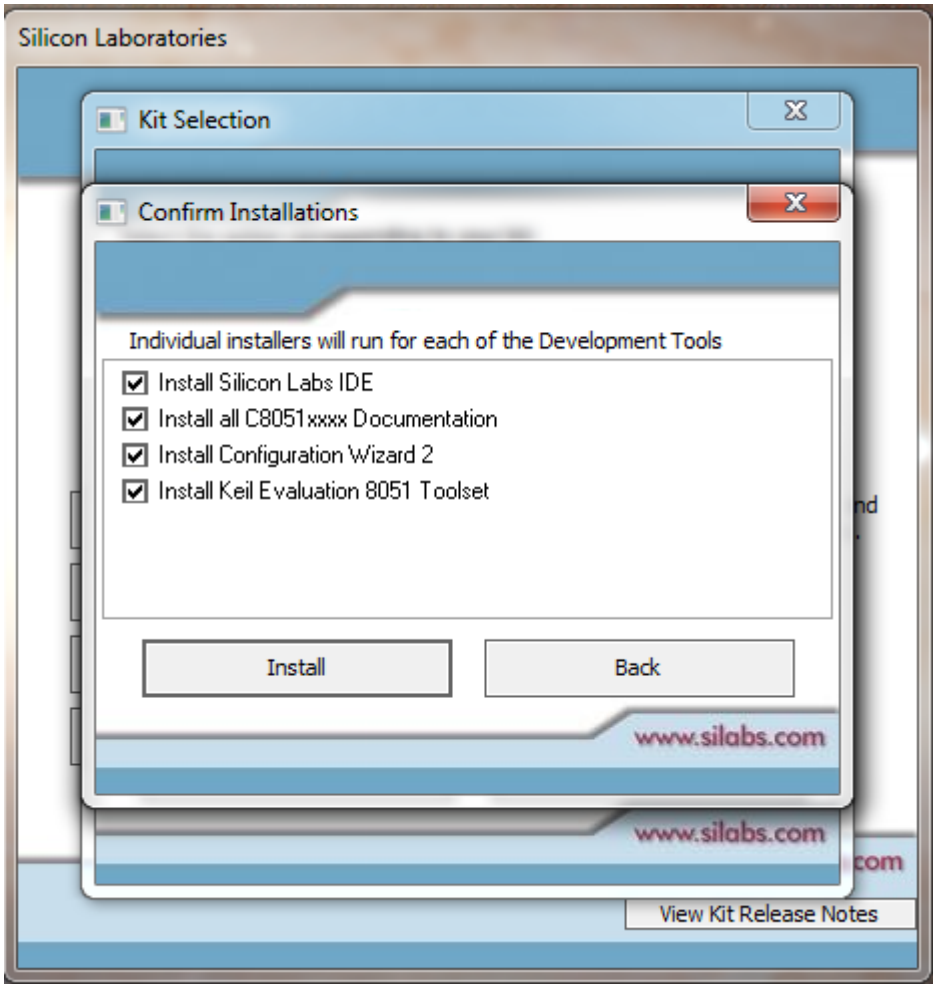
6.9. ábra: Fordítóprogram támogatás

Az induló képernyőről válasszuk ki az *Install Development Tools*-t (6.10. ábra), majd a fejlesztői csomagot (6.11. ábra), végül a csomagban telepíteni kívánt modulokat (6.12. ábra). A sikeres telepítés után fejlesztő környezetben be kell állítani, hogy a JTAG modul milyen módon kapcsolódik a fejlesztői számítógéphez. Ez USB illetve RS232 lehet. A beállítást a fejlesztői környezet *Option / Connection Options* menüpontján keresztül lehet elvégezni (6.13. ábra).

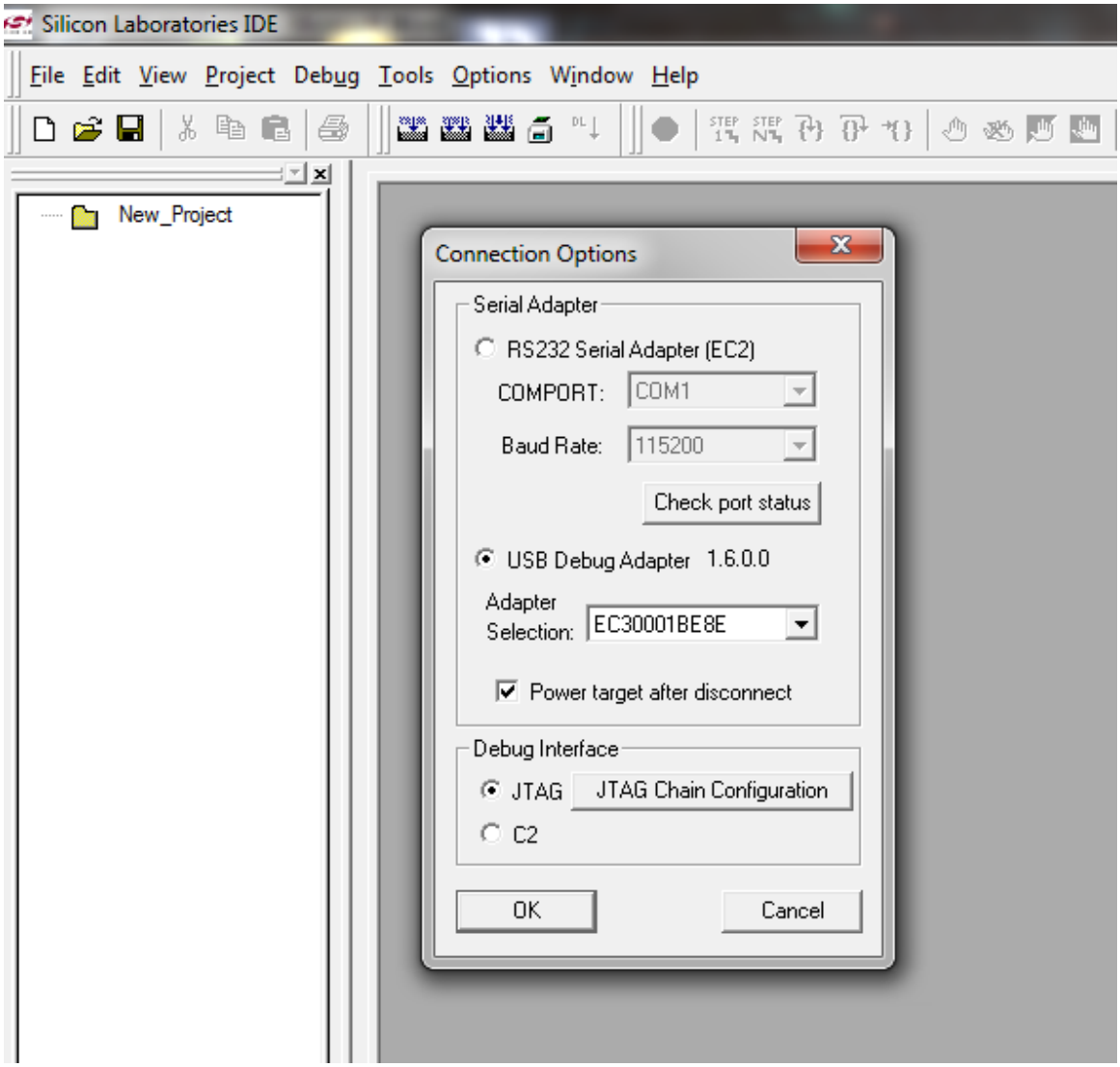
6.10. ábra: A telepítő CD induló képernyője



6.11. ábra: Fejlesztői csomag választása



6.12. ábra: A fejlesztő csomag telepítendő moduljai

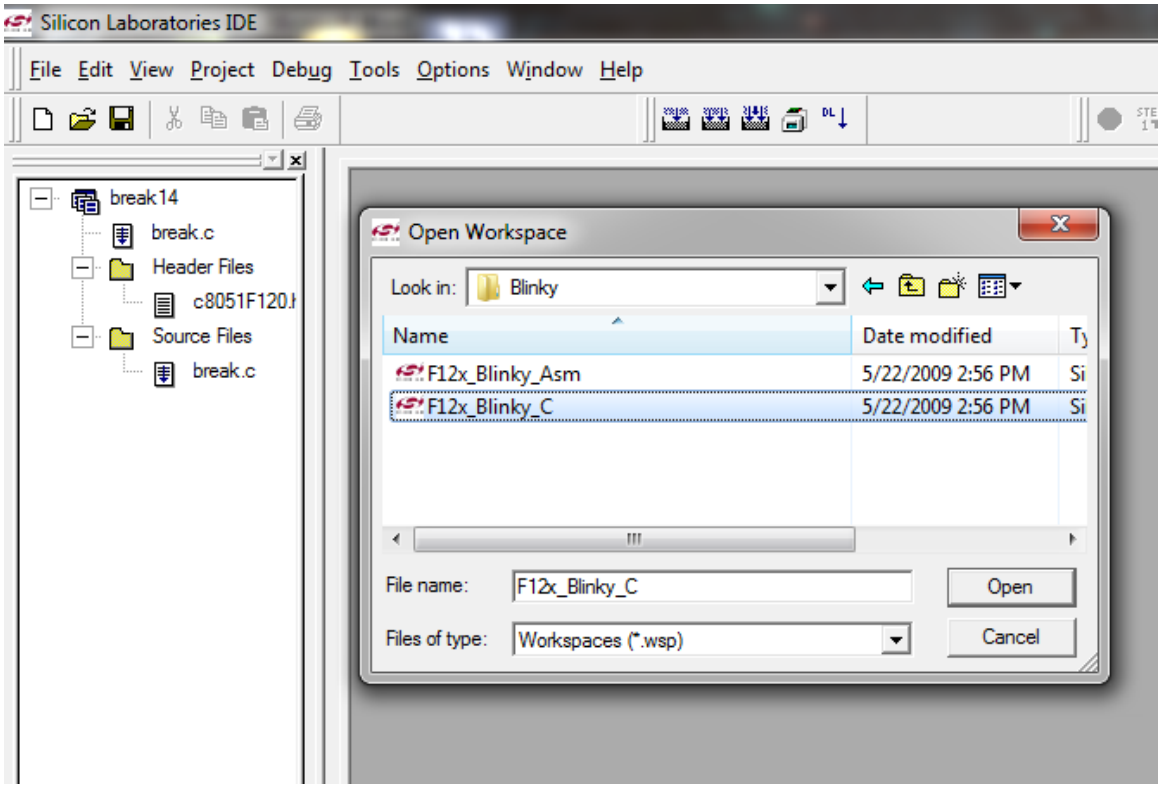


6.13. ábra: A JTAG kapcsolat beállítása

6.9. A fordítás, letöltés, futtatás

A telepítés és konfigurálás helyességének ellenőrzésére a mellékelt példaprogramok közül töltsük le a fejlesztői kártyára és futtassuk az *F12x_Blinky_C* projectet az alábbi lépéseken keresztül:

- a *Debug / Connect* menün keresztül kapcsoljuk össze a fejlesztő környezetet és a fejlesztői kártyát.
- a *Project / Open project…* menün keresztül töltsük be a *F12x_Blinky_C* projektet (6.14. ábra)
- a Project / Rebuild project menün keresztül fordítsuk le. A fordítás eredménye a projekt névvel megegyező nevű kiterjesztés nélküli állomány.



6.14. ábra: az F12x_Blinky_C projekt betöltése

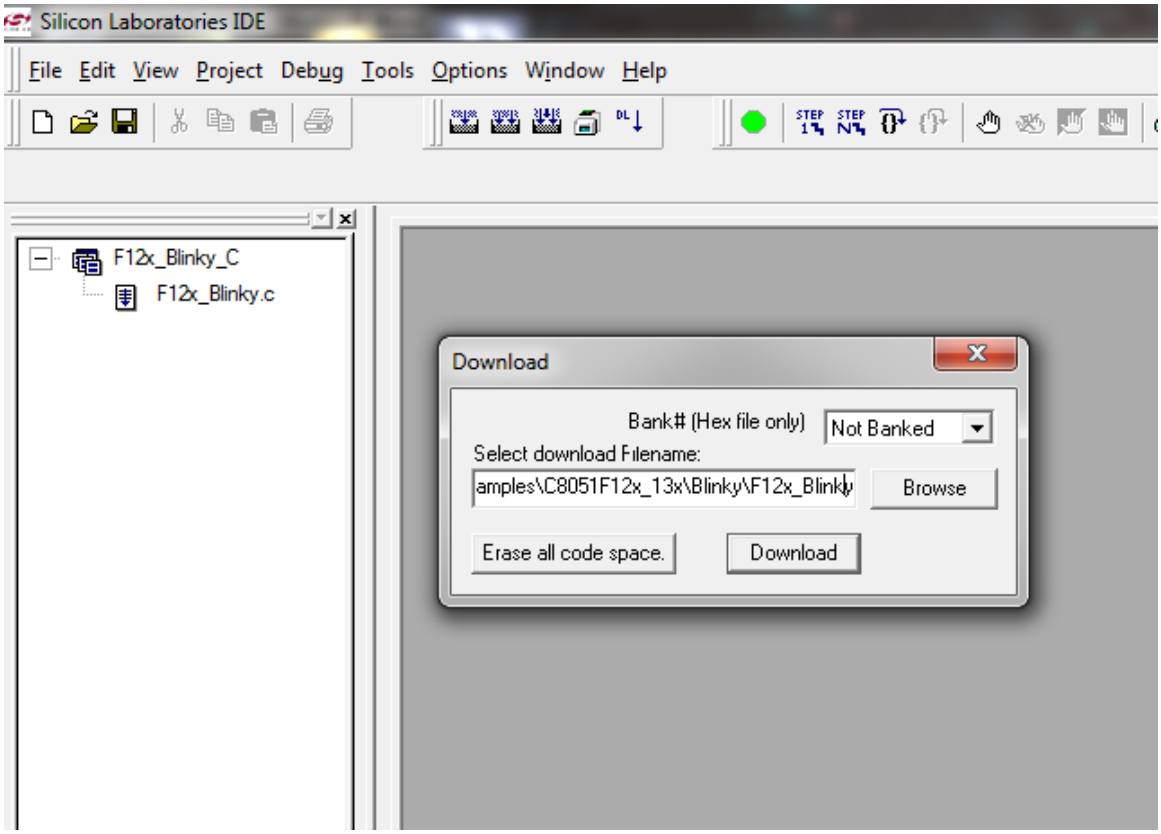
- a *Debug / Download Object File…* menün keresztül töltsük le a lefordított projektet a fejlesztői kártyára (6.15. ábra)
- a *Debug / Go* menüponton keresztül futtassuk a programot.

Amennyiben mindent jól csináltunk, akkor fejlesztői kártyán levő zöld színű LED 10 Hz frekvenciával villog.

A futtatást a a *Debug / Stop* menüponton keresztül tudjuk leállítani. Természetesen a fejlesztő környezet lehetővé teszi a lépésenkénti futtatást is a megfelelő menü pontok használatával.

Amennyiben a fejlesztő környezet nélkül szeretnénk futtatni a programunkat, akkor:

- a *Debug / Disconnect* menüpont segítségével szüntessük meg a kapcsolatot a fejlesztőgép és a fejlesztői kártya között,
- áramtalanítsuk a kártyát, és húzzuk ki a JTAG-et a kártyából, majd kapcsoljuk be a kártyát.

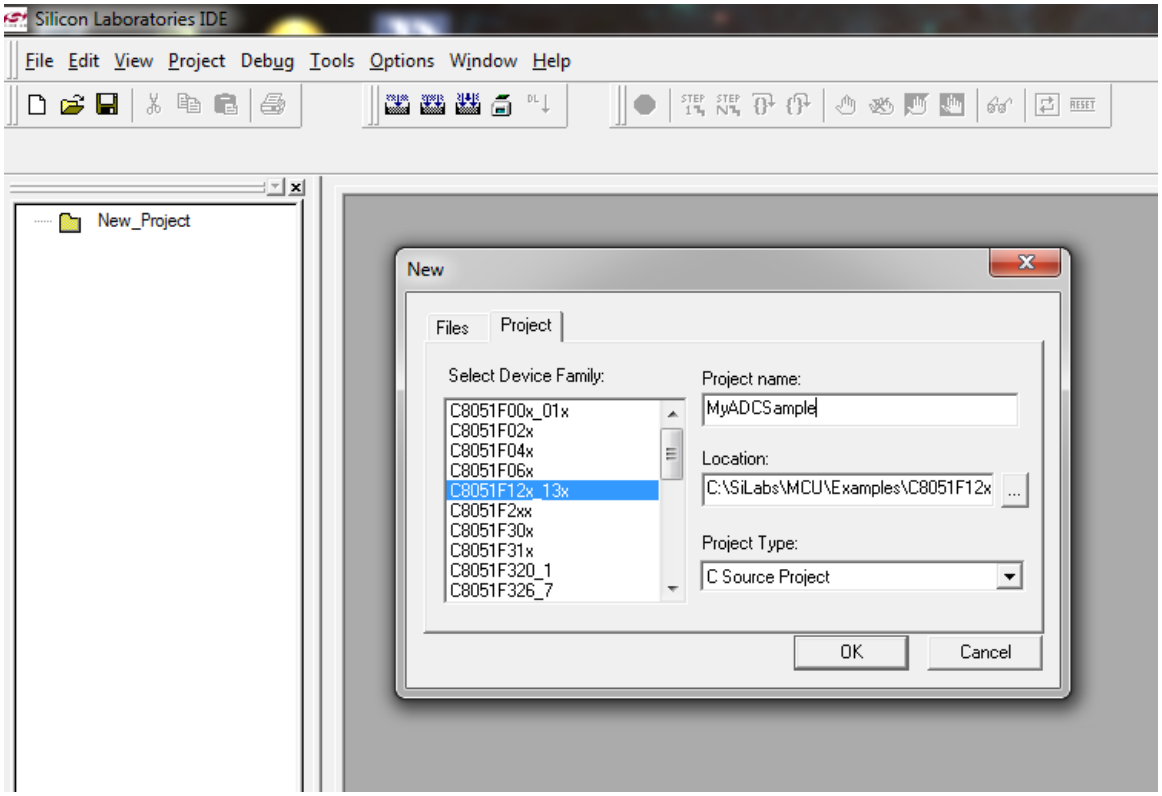


6.15. ábra: A lefordított F12x_Blinky_C letöltése a fejlesztői kártyára

6.10. Új projekt készítése

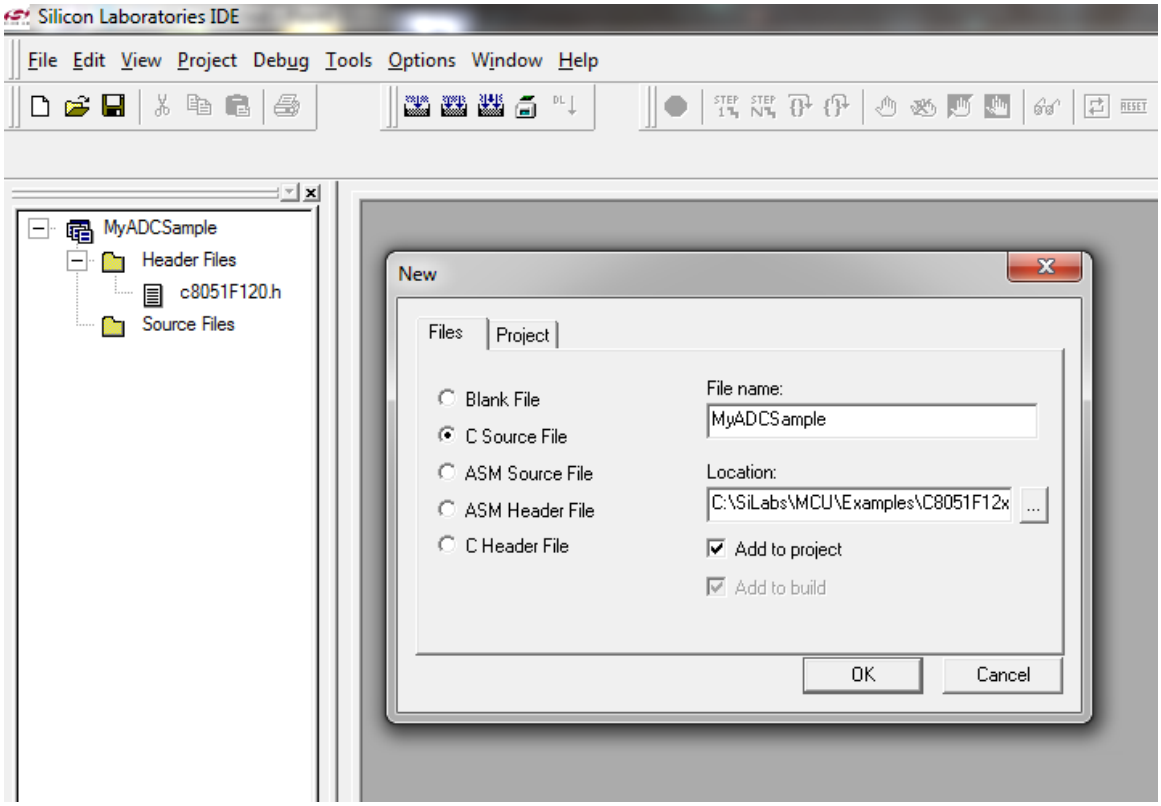
Új projekt létrehozásához hajtsuk végre a következő lépéseket:

- a *Project / New project…* menüponton keresztül hozzunk létre egy új projektet. A projekt létrehozásakor meg kell adnunk a mikrokontroller típusát (C8051F12x_3x), a projekt nevét, helyét a fejlesztői gépen valamint a projekt típusát (6.16. ábra). Ekkor a fejlesztő rendszer a projekt könyvtárában két fájlt hoz létre: a projekt nevével megegyező „`.wsp`” projekt fájlt és a „`c8051f120.h`” include fájlt. Ez tartalmazza a mikrokontroller specifikus deklarációkat és praktikusan minden a projekthez tartozó „.c” fájlhoz hozzá kell adni.



6.16. ábra: MyADCSample project generálása

- a *File / New File…* menüponton keresztül adjuk hozzá a projekthez a készítendő szoftver forráskódját tartalmazó fájlt (6.17. ábra). Kezdetben ez egy üres fájl.



6.17. ábra: Új forrás file hozzáadása a projekthez

6.11 Kérdések, feladatok

1. Mi az SFR regiszter?
2. Mi az SFR lap és miért van rá szükség?
3. Mi az XRAM és mire használhatjuk?
4. Mire használhatjuk a FLASH memóriát
5. Adja meg a *main* függvény általános felépítését!
6. Mi a megszakítás fogalma?
7. Milyen lépéseket kell a mikrokontrollernek elvégeznie egy megszakítás bekövetkezésekor?
8. Mit értünk a megszakítások prioritásán?
9. Mi a megszakítási rutin feladata?
10. Milyen módszerekkel lehet az SFR regiszterekben tárolt értékeket megváltoztatni?
11. Mondjon gyakorlati példát az időzítő használatára!
12. Mi a mintavételi frekvencia?
13. Mi a különbség az RS232 és az RS485 között?

14. Hol használják a CAN buszt?
15. Mi az I^2C protokoll?
16. Milyen paraméterekkel írjuk le az RS232 szabványban az adat átvitelt?
17. Mi a különbség a szinkron és az aszinkron soros kommunikáció között?
18. Hány byte adatmennyiség keletkezik az A/D konverter kimenetén 25 s alatt, ha a mintavételi frekvencia 40 Hz, és a kvantálás hossza 12 bit?
19. Az A/D konverternek milyen feladatokat kell elvégeznie?

7. Beágyazott szoftver készítése C nyelven

Ebben a fejezetben tárgyaljuk azokat az ismereteket, amelyek alapján az olvasó képes lesz a fent bemutatott platformon futó megbízható hatékony szoftverek fejlesztésére. Mint már említettük a cél nem a „C” programozási nyelv ismertetése, a nyelv ismeretét feltételezzük.

7.1. A main függvény

Beágyazott rendszereknél a *main* függvény soha nem lép ki, mert a program alatt nincs semmilyen operációs rendszer, mely ezt kezelné. Ezért a *main* függvényben el kell helyezni egy hurkot, mely a beágyazott szoftver folyamatos futásához szükséges rutinokat állandóan futtatja. Ilyen hurkot legegyszerűbben végtelen ciklussal tudunk implementálni. Ennek alapján a *main* általános szerkezete a következő:

```
void main(void)
{
   Init();
   while(1)
   {
      func1();
      func2();
   }
}
```

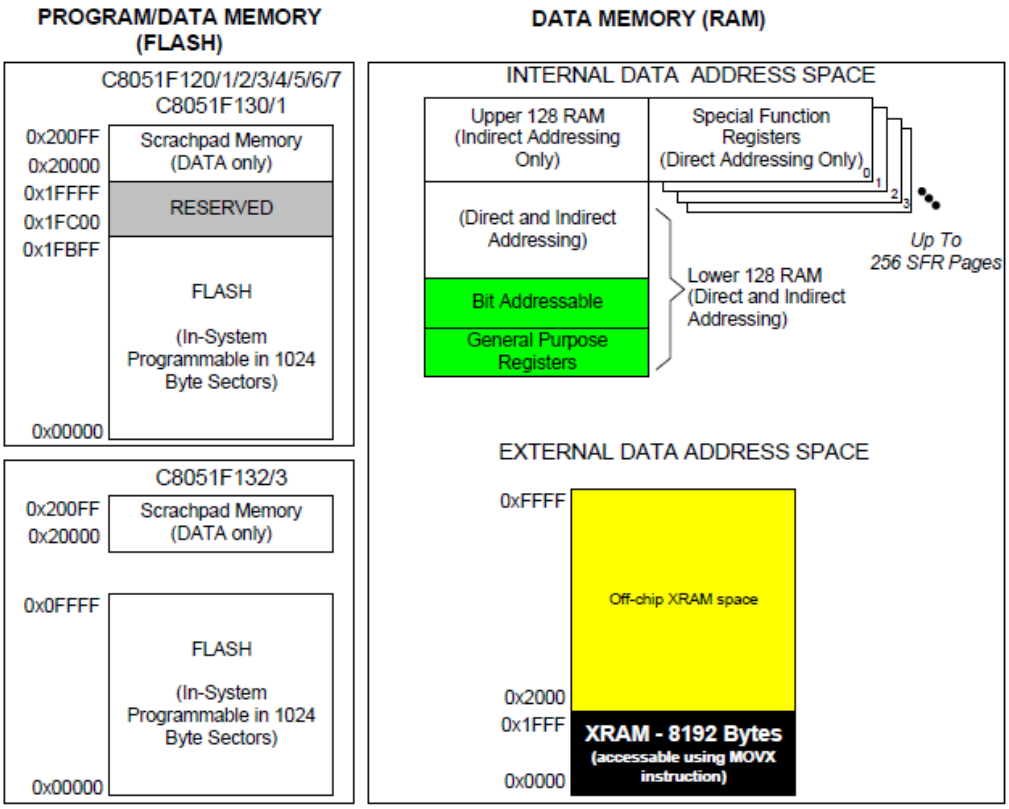
ahol az `Init()` az inicializációs függvényhívást jelöli, a `func1()`, illetve `func2()` pedig az állandóan futó kódot.

7.2. A C8051F120 memória kezelése

A mikrokontroller adattárolásra 8 kB XRAM (eXtended RAM) és 256 bájt RAM belső memóriával rendelkezik és szükség esetén még 64 kB külső memória csatlakoztatható a párhuzamos interfészen keresztül. A beágyazott szoftver kódjának tárolására 128 kB FLASH memória áll rendelkezésre. A memóriakezelés az eredeti 8051-el kompatibilis. Ez azt jelenti, hogy a beágyazott szoftver kódja a FLASH memóriában van tárolva és ott van futtatva, míg a program futása során változó adatok a belső memóriában helyezkednek el. A program futása során a FLASH memóriába semmilyen módon nem tud adatot írni. A belső memóriában a 256 bájt RAM közvetlenül címezhető. Gyakorlatilag ez tekinthető

a hagyományos értelemben vett RAM-nak. Nagyobb mennyiségű adatok tárolására az XRAM szolgál. Ha az XRAM-ban akarunk adatot tárolni ezt a programunkban az adattípus után írt `xdata` kulcsszóval jelezni kell. Példa:

```
char xdata Buffer[2000];
```



7.1. ábra: A C8051F120 memória térképe

Azonban mint ahogy a 7.1. ábrán látható a 256 bájt RAM-ból a felső 128 byte speciális célra fent van tartva. Nevezetesen mint ahogy már a 6. fejezetben utaltunk már rá a mikrokontroller számos beépített perifériával rendelkezik. Ezeket a perifériákat használatuk előtt általában inicializálni, engedélyezni kell. A perifériákat egy speciális regiszteren az SFR (Special Function Register) regiszteren keresztül lehet beállítani. Minden egyes perifériának saját SFR regisztere van. Ezek az SFR regiszterek a RAM felső 128-bájtnyi területére vannak leképezve. Ez lehetővé teszi az SFR regiszterek közvetlen memória művelettel (írás/olvasás) történő elérését. Másrészt lehetőség van az SFR regiszterek egyes bitjeinek közvetlen elérésére is. Előre hozott példaként tekintsük az UART1 soros port

beállítását. Az UART1 SFR regisztere az SCON1. A 8 bites változó baud rate-ű kommunikáció beállítása történhet:

- Az SFR közvetlen írásával

```
SCON1 = 0x10;
```

- Az SFR bitcímzésével

```
SCON1^4 = 1;
```

A C8051F120-as mikrokontrollerben levő SFR regiszterek által lefedett memória terület azonban nagyobb, mint a rendelkezésre álló 128 bájt, ezért az SFR regisztereket SFR lapokba szervezték. Összesen 20 db előre definiált SFR lap van. A kívánt SFR regiszter használata előtt az őt tartalmazó SFR lapot aktívvá kell tenni. Ez a C nyelvű programban meglepően egyszerű:

```
SFRPAGE = UART1_PAGE;
```

A fejlesztés során a projekt generálásakor keletkező `c8051f120.h` include fájl tartalmazza azokat az előre definiált makrókat és változókat, melyekkel az SFR regiszterek és SFR lapok könnyen elérhetőek.

7.3. Digitális portok kezelése

A fejlesztői kártyán 8 db 8 bites digitális port van (P0-P7). Amennyiben egy C nyelvű programból egy bájtot a P1 portra írni akkor szintaxis a következő:

```
P1 = 0x12;
```

Ha a második bitet szeretnék 1-be állítani, akkor:

```
P1 ^1= 1;
```

Amennyiben beolvasni szeretnénk, akkor:

```
unsigned char B, sbit b;


B=P1;

b= P1 ^1;
```

Az `sbit` (single bit) adattípus lehetővé teszi a portok bitenkénti elérését.

7.4. Megszakításkezelés

Mind az általános célú számítógépeknél, mind a beágyazott rendszereknél a programok futása során bekövetkezhetnek olyan események, melyek az éppen futó program futásának *megszakítását* és más magasabb prioritású műveletek végrehajtását követelik meg a rendszertől. Ennél fogva minden mikrokontrollernek és processzornak rendelkeznie kell olyan képességgel, amely lehetővé teszi fenti megszakítási folyamatok végrehajtását. Alapvető követelmény, hogy a megszakítási program lefutása után a vezérlésnek folytatni kell a félbehagyott program futatását. Összefoglalva megszakításnak (interrupt) nevezzük azt a műveletsorozatot, amikor mikrokontroller (processzor) az éppen feldolgozás alatt levő folyamat futását felfüggeszti és egy másik magasabb prioritású folyamatot indít el úgy, hogy ennek a magasabb prioritású folyamatnak a végén az eredeti felfüggesztett folyamat folyatatni tudja a futását. Ennek alapján egy megszakítási esemény bekövetkezésekor a mikrokontroller a következő lépéseket hajtja végre:

- a mikrokontroller letiltja a további (jelenleginél kisebb prioritású) megszakításokat,
- befejezi az aktuális utasítás végrehajtását,
- elhelyezi a veremben a következő utasítás címét (visszatérési cím),
- elugrik a végrehajtást kiszolgáló rutin címére és végrehajtja az ott levő utasításokat,
- kiolvassa a veremből a visszatérési címet és folytatja az eredeti programot.

A C8051F120-as mikrokontroller 22 db különböző megszakítási eseményt tud kezelni (7.2. ábra). A megszakítások között van:

- 4 db külső megszakítás, melyekkel a mikrokontrollerhez kapcsolt külső hardver eszközöket tudjuk kezelni.
- 2 db soros port (részletesen a 7.6 fejezetben tárgyaljuk)
- 5 db időzítő megszakítás (részletesen a 7.5 fejezetben tárgyaljuk)

Source	Vector	Priority	Flag	Enable Flag	Priority Control
Reset	0000	Top	None	Always Enabled	Always Highest
External Interrupt 0 (/INT0)	0003	0	IE0 (TCON.1)	EX (IE.0)	PX0 (IP.0)
Timer 0 Overflow	000B	1	TF0 (TCON.5)	ET0 (IE.1)	PT0 (IP.1)
External Interrupt 1 (/INT1)	0013	2	IE1 (TCON.3)	EX1 (IE.2)	PX1 (IP.2)
Timer 1 Overflow	001B	3	TF1 (TCON.7)	ET1 (IE.3)	PT1 (IP.3)
UART0	0023	4	RI0 (SCON0.0) TI0 (SCON0.1)	ES0 (IE.4)	PS0 (IP.4)
Timer 2 Overflow	002B	5	TF2 (T2CON.7)	ET2 (IE.5)	PT2 (IP.5)
Serial Peripheral Interface	0033	6	SPIF (SPI0CN.7)	ESPI0 (EIE1.0)	PSPI0 (EIP1.0)
SMBus Interface	003B	7	SI (SMB0CN.3)	ESMB0 (EIE1.1)	PSMB0 (EIP1.1)
ADC0 Window Comparator	0043	8	AD0WINT (ADC0CN.2)	EWADC0 (EIE1.2)	PWADC0 (EIP1.2)
Programmable Counter Array	004B	9	CF (PCA0CN.7) CCFn (PCA0CN.n)	EPCA0 (EIE1.3)	PPCA0 (EIP1.3)
Comparator 0 Falling Edge	0053	10	CP0FIF (CPT0CN.4)	ECP0F (EIE1.4)	PCP0F (EIP1.2)
Comparator 0 Rising Edge	005B	11	CP0RIF (CPT0CN.5)	ECP0R (EIE1.5)	PCP0R (EIP1.5)
Comparator 1 Falling Edge	0063	12	CP1FIF (CPT1CN.4)	ECP1F (EIE1.6)	PCP1F (EIP1.6)
Comparator 1 Rising Edge	006B	13	CP1RIF (CPT1CN.5)	ECP1R (EIE1.7)	PCP1F (EIP1.7)
Timer 3 Overflow	0073	14	TF3 (TMR3CN.7)	ET3 (EIE2.0)	PT3 (EIP2.0)
ADC0 End of Conversion	007B	15	AD0INT (ADC0CN.5)	EADC0 (EIE2.1)	PADC0 (EIP2.1)
Timer 4 Overflow	0083	16	TF4 (T4CON.7)	ET4 (EIE2.2)	PT4 (EIP2.2)
ADC1 End of Conversion	008B	17	AD1INT (ADC1CN.5)	EADC1 (EIE2.3)	PADC1 (EIP2.3)
External Interrupt 6	0093	18	IE6 (P3IF.6)	EX6 (EIE2.4)	PX6 (EIP2.4)
External Interrupt 7	009B	19	IE7 (P3IF.7)	EX7 (EIE2.5)	PX7 (EIP2.5)
UART1	00A3	20	RI1 (SCON1.0) TI1 (SCON1.1)	ES1 (EIE2.6)	PS1 (EIP2.6)
External Crystal OSC Ready	00AB	21	XTLVLD (OSCXCN.7)	EXVLD (EIE2.7)	PXVLD (EIP2.7)

7.2. ábra: A C8051F120 mikrokontroller megszakítási táblája.

Minden egyes megszakítási eseményhez tartozik egy:

- *memória cím (Vector)*

  Ez a cím a megszakítást kiszolgáló rutin – Interrupt Service Routin (ISR) – címe. Az ISR rutin tartalmazza azokat az utasításokat, melyeket a megszakítás kiszolgálásakor futtatni akarunk. Nagyon fontos követelmény a megszakítást kiszolgáló rutinnal szemben, hogy a lehető legrövidebb időn belül hajtsa végre a futását. Azaz

ebben a rutinban semmilyen más eseményre történő várakozás, valamit hosszadalmasabb számítás nem megengedett. Egy C nyelvű programban a megszakítási rutint az alábbi módon kell implementálni:

```
void pelda_ISR (void) interrupt prioritas
```

azaz a függvény fejlécében meg kell adni az `interrupt` kulcsszót és a prioritás számát. A prioritás számmal azonosítja a C fordító, hogy melyik megszakítási vektorról van szó.

- *prioritás (Priority)*
  Ez a megszakítási esemény fontosságát jelzi. Minél alacsonyabb ez a szám annál nagyobb a prioritása a megszakításnak. Egy magasabb prioritású megszakítás megszakíthatja egy alacsonyabb prioritású megszakítás futását, de fordítva nem.

- *jelző bitek (Flag)*
  Minden megszakítási esemény rendelkezik egy vagy több jelzőbittel. Ezek a bitek a megszakítás bekövetkezésekor 1-be állítódnak. Ha szoftverből, vagy hardverből nem történik meg e bitek törlése (0-ba állítása), akkor az aktuális megszakítás folyamatosan be fog következni, melynek következménye a rendszer lefagyása is lehet. Amennyiben a bitet szoftverből kell törölni azt a mikrokontroller kézikönyve jelzi.

- *engedélyező/tiltó bitek (Enable Flag)*
  Ezen bitek szolgálnak a megszakítások engedélyezésére vagy tiltására. Minden megszakítási eseményhez tartozik egy egyedi engedélyező/tiltó bit (ezeket az IE, EIE1 és az EIE2 SFR regiszterek tartalmazzák), valamint van egy globális engedélyező/tiltó bit (IE.7). Általában mind az egyedi mind a globális engedélyező/tiltó bitet engedélyezni kell a megszakítás kezeléséhez.

- *prioritást beállító bitek*
  Ezekkel a bitekkel a prioritási sorrendet lehet állítani a RESET megszakítás kivételével.

Mint ahogy fentebb említettük a megszakítási regisztereken keresztül tudjuk engedélyezni/tiltani a megszakításukat valamint beállítani a prioritási sorrendet. Ezekről a regiszterekről részletesen olvashat a mikrokontroller dokumentációjában.

7.5. Időzítők programozása

Az időzítők olyan egységei a mikrokontrollernek, melyeket felhasználhatunk megszakítások generálására, különböző események számlálására. Valamennyi időzítő áramkör működéséhez szükség van egy órajelre, mely meghatározza, hogy az időzítő milyen frekvenciával dolgozzon. Ezt a jelet a mikrokontroller oszcillátor egysége szolgáltatja. A C8051F120 –as mikrokontrollernél a beépített oszcillátor 24.5MHz frekvenciájú. A C8051F120 –as mikrokontroller 5 db számláló/ időzítő egységet tartalmaz. Ezek közül a Timer0 és Timer1 az eredeti 8051-el kompatibilis 16 bites számláló/időzítők. Ezek a következő üzemmódok szerint működhetnek:

- 13 bites számláló/időzítő.
- 16 bites számláló/időzítő.
- 8 bites számláló/időzítő automatikus újraindulással.
- Timer0: 2 db 8 bites számláló/időzítő.

A harmadik üzemmód kivételével a számlálás vagy időzítés befejezésekor, ha szükséges, akkor az időzítőt szoftverből kell újra indítani.

A Timer2, Timer3 és Timer4 16 bites időzítők, automatikus újraindulással.

A fentieken kívül a mikrokontroller tartalmaz egy a lefagyások esetén hasznos watchdog (házőrző) időzítő egységet is, melyet a szoftvernek periodikusan nullázni kell. Amennyiben ez a nullázás a watchdog lejárta előtt nem történik meg, akkor a watchdog áramkör hardveresen újraindítja a mikrokontrollert. Ettől talán elmúlik a lefagyást előidéző állapot. Meg kell jegyezni, hogy a lefagyások legnagyobb részét valamilyen szoftver vagy hardver hiba okozza. Ha sűrűn lefagy a rendszerünk, akkor ne hagyatkozzunk a watchdogra, hanem keressük meg a hibát. Természetesen a watchdog kikapcsolható az alábbi utasítással:

```
WDTCN = 0xde;

WDTCN = 0xad;
```

Beágyazott rendszereknél az időzítők tipikus használata például a megfelelő mintavételi frekvencia biztosítása az AD átalakító részére. Gyakorlati példaként nézünk egy olyan példát, ahol a Timer3 időzítőt használjuk a fejlesztői kártyán levő LED villogtatására.

A megvalósítás elve:

A Timer3 inicializációs függvénye paraméterben kapja meg azt a 16 bites számot, amelyről az időzítőnek vissza kell számolnia. A számlálás frekvenciájaként a rendelkezésre álló órajel 12-dét használjuk. Amikor a számláló elérte a 0 értéket az időzítő megszakítást generál, mely meghívja a megszakítást kiszolgáló rutint. A megszakítást kiszolgáló rutinban töröljük a megszakítást jelzőt és a LED-et ellentétes állapotra állítjuk.

A fenti elvet megvalósító kódrészlet:

```
void Timer3_Init (int counts)
{
char SFRPAGE_SAVE = SFRPAGE;  // Save Current SFR page
SFRPAGE = TMR3_PAGE;
TMR3CN = 0x00;           // Stop Timer3; Clear TF3;
                         // use SYSCLK/12 as timebase
   RCAP3   = -counts;    // Init reload values 16 bit
   TMR3    = 0xffff;     // set to reload immediately
   EIE2    |= 0x01;      // enable Timer3 interrupts
   EA = 1;               // Enable global interrupts
   TR3 = 1;              // start Timer3
   SFRPAGE = SFRPAGE_SAVE;

}

void Timer3_ISR (void) interrupt 14
{
   TF3 = 0;              // clear TF3
   LED = ~LED;           // change state of LED
}
```

Felmerül a kérdés, hogy mekkorára kell választani az inicializációs rutinnak átadandó paramétert, hogy a villogási frekvenciája 5 Hz legyen. Nos, a C8051F120-as mikrokontroller belső órajele 24.5 MHz, azonban alapértelmezett frekvenciaként ennek a 8-ada van beállítva (3.0625MHz).

Ennek alapján a villogtatás késleltetési ideje:

$$T = \frac{1}{f} = \frac{1}{5Hz} = 200ms$$

Az időzítő frekvenciája:

$$T_f = \frac{SYSCLK}{12} = \frac{3.0625}{12} = 0.2552MHz$$

Egy ciklus időtartama:

$$T_c = \frac{1}{T_f} = \frac{1}{0.2552MHz} = 0.3918\mu s$$

A számláló értéke:

$$count = \frac{T}{T_c} = \frac{200ms}{03918\mu s} = 50146$$

Ez egy 16 bites szám ezért

$$count = 65535 - 51046 = 14489$$

Természetesen az inicializációs rutinban a kivonást – előjelel valósítottuk meg.

Az időzítőkről részletesen olvashat a mikrokontroller dokumentációjában.

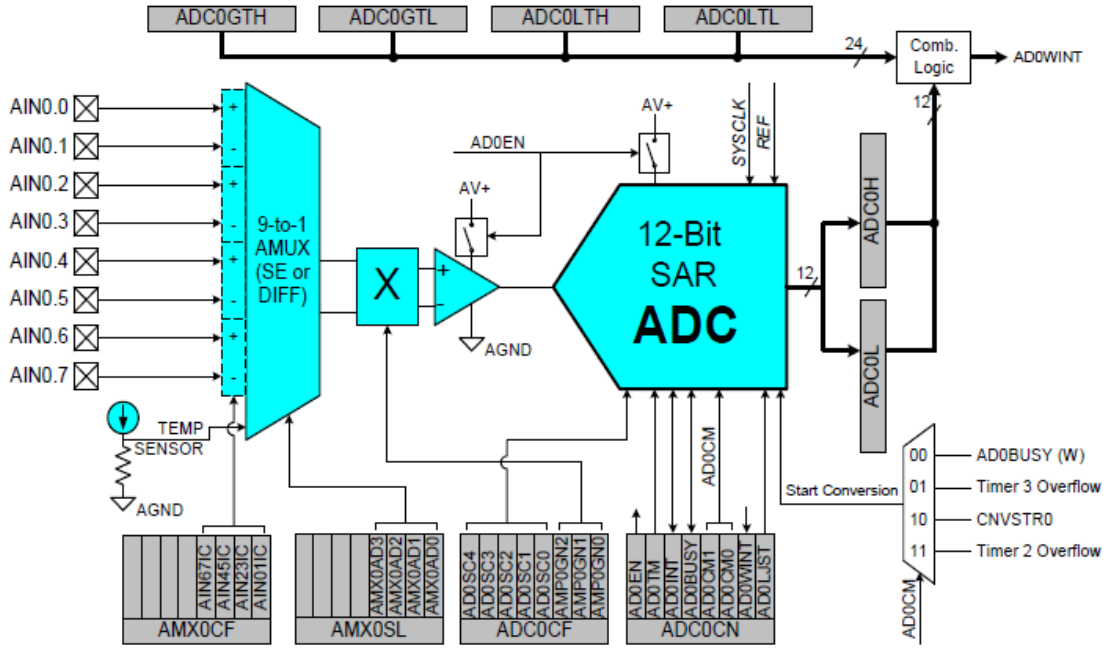
7.6. A 12-bites AD átalakító használata

A mikrokontrollerben 1db 12-bites szukcesszív approximációs elven működő AD átalakító (ADC0) van. A mikrokontroller ADC0 alrendszere (7.3 ábra) a következő elemekből épül fel:

- 9 csatornás analóg multiplexer (8 db analóg input + a mikrokontroller hőmérséklet szenzor-a).
- Programozható előerősítő (PGA0 – gain amplifier)
- 12 bites szukcesszív approximációt (SAR) alkalmazó analóg-digitális átalakító
- Referenciafeszültség választó

A multiplexer feladata, hogy a különböző csatornákon levő analóg inputok közül mindig csak a kiválasztott csatorna jele kerüljön az analóg–digitális átalakító bemenetére.

A programozható előerősítő feladata, hogy a bemenő jelet a megadott szorzó tényezővel erősítse. Ennek akkor van jelentősége, hogy ha a bemenő jel erőssége nem éri el azt a szintet, amire az analóg–digitális átalakítónak szüksége van. Értéke 0.5, 1, 2, 4, 8, 16 lehet. Alapértelmezésben értéke 1.



7.3. ábra: Az ADC0 alrendszer

Az ADC0 programozásának lépései a következők:

- Referenciafeszültség beállítása (REFC0N).
- Multiplexer konfigurálása (AMX0CF).
- Csatorna kiválasztása (AMX0SL).
- SAR órafrekvencia és előerősítés beállítása (ADC0CF, PGA0).
- A megfelelő vezérlő bitek beállítása (ADC0CN).
- Az ADC0 bekapcsolása.
- Az eredmény tárolása (ADC0)

Az ADC0 alrendszer belső referencia feszültsége 2.43 mV. A referencia feszültség beállítása az REFC0N regiszteren keresztül történik (7.4. ábra). Analóg-digitális konverzió esetén az 1-bitet (BIASE) 1-be kell állítani. A belső referenciafeszültség használata esetén a 0-s bitet 0-ba kell állítani.

A 7.5 ábrán látható, hogy az analóg bemenetek kétféle módban programozhatóak: független bemenetként, illetve differenciális bemenetpárként. Alapértelmezésben az analóg bemeneteket független bemenetként kezeli a rendszer.

Bit	Symbol	Leírás
7-5	-	Használaton kívül
4	AD0VRS	**ADC0 Voltage Reference Select** 0: ADC0 feszültség referencia:VREF 0 pin. 1: ADC0 feszültség referencia:DAC0 output.
3	AD1VRS	**ADC1 Voltage Reference Select** 0: ADC1 feszültség referencia:VREF 1 pin. 1: ADC1 feszültség referencia:AV+
2	TEMPE	**Temperature Sensor Enable Bit** 0: Off 1: On.
1	BIASE	**ADC/DAC Bias Generator Enable Bit** (ADC or DAC használatakor: 1 !) 0: Off 1: On.
0	REFBE	**Internal Reference Buffer Enable Bit** 0: Off 1: On. Internal voltage reference is driven on the VREF pin.

7.4. ábra: Az RECC0N regiszter bitkiosztása

Bit	Symbol	Leírás
7-4	-	Használaton kívül
3	AIN67IC	**AIN0.6, AIN0.7 Input Pair Configuration Bit** 0: AIN0.6 és AIN0.7 független bemenetek 1: AIN0.6, AIN0.7 +,- differenciális bemenetpár
2	AIN45IC	**AIN0.4, AIN0.5 Input Pair Configuration Bit** 0: AIN0.4 és AIN0.5 független bemenetek 1: AIN0.4, AIN0.5 +,- differenciális bemenetpár
1	AIN23IC	**AIN0.2, AIN0.3 Input Pair Configuration Bit** 0: AIN0.2 és AIN0.3 független bemenetek 1: AIN0.2, AIN0.3 +,- differenciális bemenetpár
0	AIN01IC	**AIN00., AIN0.1 Input Pair Configuration Bit** 0: AIN0.0 és AIN0.1 független bemenetek 1: AIN0.0, AIN0.1 +,- differenciális bemenetpár

7.5. ábra: Az analóg bemenetek konfigurálása

Az aktuális csatorna kiválasztása a multiplexer segítségével történik (AMX0SL)(7.6. ábra).

Bit	Symbol	Leírás
7-4	-	Használaton kívül
3-0	AMX0 AD3-0	**AMX0 Address Bits** 0000: AIN0.0 0001: AIN0.1 0010: AIN0.2 0011: AIN0.3 0100: AIN0.4 0101: AIN0.5 0110: AIN0.6 0111: AIN0.7 1xxx: Temp Sensor

7.6. ábra: Az aktuális bemeneti csatorna kiválasztása.

Az ADC0 konverziós frekvenciáját és az előerősítést az ADC0CF regiszteren állíthatjuk be (7.7. ábra). A 3-7 bit szolgál a frekvencia beállítására. Itt a képletben SYSCLK a rendszer órakristályának a frekvenciája (24.5 MHz). A CLK_{SAR} a kívánt konverziós frekvencia.

Az analóg-digitális konverzió indítása és leállítása az ADC0 regiszter 7. bitjének engedélyezésével/tiltásával történik (7.8. ábra).

Bit	Symbol	Leírás
7-3	AD0SC4-0	**ADC0 SAR0 Conversion Clock frequency Bits** A rendszer-órajelből származik: $AD0SC = \frac{SYSCLK}{CLK_{SAR0}} - 1$ A maximális frekvencia 2.5MHz
2-0	AD0VRS	**ADC0 Internal Amplifier Gain (PGA)** 000: Gain = 1 001: Gain = 2 010: Gain = 4 011: Gain = 8 10x: Gain = 16 11x: Gain = 0.5

7.7. ábra: A mintavételi frekvencia és az előerősítés konfigurálása (ADC0CF)

Bit	Symbol	Leírás
7	AD0EN	**ADC0 Enable Bit** 0: ADC0 letiltva. 1: ADC0 engedélyezve. ADC0 aktív, és készen áll a konverzióra.
6	AD0TM	**ADC0 Track Mode Bit** 0: Folyamatos mintavételezés, kivéve feldolgozás alatt (és ha AD0EN = 0). 1: Alacsony külső jel (CNVSTR = 0) alatt mintavételez, a külső jel felfutó élére (CNVSTR 0→1) kezdi a feldolgozást.
5	AD0INT	**ADC0 Conversion Complete Interrupt Flag** 0: ADC0 nem fejezte be az átalakítást az utolsó törlés óta. 1: ADC0 befejezte az átalakítást. Ez a bit csak szoftveresen törölhető!
4	AD0BUSY	**ADC0 Busy Bit** 0: ADC0 átalakítás befejezve vagy nincs folyamatban. 1: ADC0 átalakítás folyamatban. AD0BUSY lefutó éle állítja be AD0INT bitet.

7.8. ábra: Az ADC0 vezérlése

A fentiek alapján az ADC0 alrendszer belső referencia feszültséggel történő használatát bemutató kódrészlet a következő:

```
#define SYSCLK   24500000      // Internal clock frequency
#define SAR_CLK  2500000       // Desired SAR clock speed


void ADC0_Init()
{
    char SFRPAGE_SAVE = SFRPAGE;
    SFRPAGE = LEGACY_PAGE;

    ADC0CN = 0x80;      // ADC0 enabled; normal tracking mode;
                        // ADC0 conversions are initiated
                        // on writing a 1 to AD0BUSY

    REF0CN = 0x03;       // enable on-chip VREF and VREF output buffer
    AMX0SL = 0x00;       // Select AIN0.0 as ADC input


    ADC0CF = (SYSCLK/SAR_CLK) << 3;    // ADC conversion clock = 2.5MHz
    ADC0CF |= 0x00;                     // PGA gain = 1 (default)

    AD0EN = 1;           // enable ADC

    SFRPAGE = SFRPAGE_SAVE;
}


long ADC0_Measure(short channelNum,  short gain)
{
    long adcData = 0;
    int i = 0;
    char SFRPAGE_SAVE = SFRPAGE;

    SFRPAGE = ADC0_PAGE;

    switch (channelNum)
    {
        case 0: AMX0SL = 0x00;
             break;
        case 1: AMX0SL = 0x01;
             break;
        case 2: AMX0SL = 0x02;
             break;
        case 3: AMX0SL = 0x03;
             break;
        case 4: AMX0SL = 0x04;
             break;
        case 5: AMX0SL = 0x05;
             break;
        case 6: AMX0SL = 0x06;
             break;
        case 7: AMX0SL = 0x07;
             break;
        default: AMX0SL = 0x00;
             break;
    }
```

```
    ADC0CF &= 0xF8;
    switch (gain)
    {
        case 0: ADC0CF |= 0x06;  // gain is 0.5;
            break;
        case 1:ADC0CF |= 0x00;
            break;
        case 2:ADC0CF |= 0x01;
            break;
        case 4:ADC0CF |= 0x02;
               break;
        case 8:ADC0CF |= 0x03;
               break;
        case 16:ADC0CF |= 0x04;
            break;
        default:ADC0CF |= 0x00;
            break;
    }

    //accumulate 256 samples
    for (i=0; i<256; i++)
    {
        AD0INT = 0;
        AD0BUSY = 1;
        while(!AD0INT);
        adcData += ADC0;
    }

    // take the average
    adcData >>= 8;

    SFRPAGE = SFRPAGE_SAVE;

    return adcData;
}



//                            Vref (mV)
//   measurement (mV) =   --------------- * Result (bits)
//                       (2^12)-1 (bits)
long getResultmV( int value )
{
      return value * 2430 / 4095;
}
```

7.7. Soros vonali kommunikáció

A mikrokontrollerben az RS232 szabványú full duplex aszinkron soros kommunikáció az UART (Universal Asynchronous Receiver Transmitter) interfészen keresztül valósul meg. A C8051F120-as kontrollerben két soros port van, mindegyikhez egy-egy UART (UART0, UART1) tartozik. A továbbiakban az UART1-et fogjuk használni. Az UART1 két SFR regisztert használ:

- SCON1 (Serial Control Register 1 (SCON1)
  Mint ahogy a nevében benne van, ezt regisztert használjuk a soros vonali kommunikáció paramétereinek beállítására.

- SBUF1 (Serial Data Buffer 1)
  Ez a regiszter a soros vonali kommunikáció adatregisztere.

A kommunikációs paraméterek beállítása:

Az UART1 kétféle aszinkron kommunikációt ismer:

- 8 bites kommunikáció
  Ennél a módnál a 1 start bit, 8 adatbit, nincs paritás bit és 1 stop bit van. A Baud Rate a Timer1 időzítő órajeléből van származtatva.

- 9 bites kommunikáció
  Ennél a módnál a 1 start bit, 8 adatbit, 1 bit mely paritás ellenőrzésre vagy egyéb felhasználói célra használható és 1 stop bit van. A Baud Rate szintén a Timer1 időzítő órajeléből van származtatva.

Az UART1 a Baud Rate-et a következő képlet alapján határozza meg:

$$BaudRate = \frac{1}{2} * \frac{T1_{CLK}}{256 - T1H}$$

Itt $T1_{CLK}$ a Timer1 időzítő által szolgáltatott órajel, *T1H* pedig a Timer1 regiszter felső bájtja (reload value).

A kommunikációs paraméterek beállítására az SCON1 regiszter szolgál (7.9. ábra)

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
S1MODE	-	MCE1	REN1	TB81	RB81	TI1	RI1
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0

7.9 ábra: Az SCON1 regiszter felépítése

Az egyes bitek jelentése a következő:

Bit	Jelentés
Bit 7.	Működési mód. 0: 8 bites mód 1: 9 bites mód
Bit 6.	Nem használt
Bit 5.	Multiprocesszoros kommunikációnál használatos (jelen jegyzetben nem foglalkozunk vele)
Bit 4.	Vétel engedélyzése. 0: a vétel tiltva van 1: a vétel engedélyezve
Bit 3.	9 bites kommunikációnál az adás során a 9. bit értéke. Szoftverből kell állítani. (jelen jegyzetben nem foglalkozunk vele)
Bit 2.	9 bites kommunikációnál a vétel során a 9. bit értéke. Szoftverből kell állítani. (jelen jegyzetben nem foglalkozunk vele)
Bit 1.	TI1: adási interrupt flag Az adás végén az UART1 ezt a bitet 1-be állítja. Ezzel jelzi az adás végét. A bitet szoftverből kell 0-ba állítani, hogy a következő adat küldése elinduljon.
Bit 0.	RI1: vételi interrupt flag A vétel végén az UART1 ezt a bitet 1-be állítja. Ezzel jelzi a vétel végét. A bitet szoftverből kell 0-ba állítani, hogy a következő adat vétele elinduljon.

A fentiek alapján az UART1 inicializálása 8 bites kommunikációra következő:

```
void UART1_Init (void)
{
   char SFRPAGE_SAVE = SFRPAGE;
   SFRPAGE = UART1_PAGE;
   SCON1   = 0x10;          // SCON1: mode 0, 8-bit UART, enable RX

   SFRPAGE = TIMER01_PAGE;
   TMOD   &= ~0xF0;
   TMOD   |=  0x20;         // TMOD: timer 1, mode 2, 8-bit reload


   if (SYSCLK/BAUDRATE/2/256 < 1)
   {
      TH1 = -(SYSCLK/BAUDRATE/2);
      CKCON |= 0x10;        // T1M = 1; SCA1:0 = xx
```

```
   }
   else if (SYSCLK/BAUDRATE/2/256 < 4)
   {
      TH1 = -(SYSCLK/BAUDRATE/2/4);
      CKCON &= ~0x13;      // Clear all T1 related bits
      CKCON |=  0x01;      // T1M = 0; SCA1:0 = 01
   }
   else if (SYSCLK/BAUDRATE/2/256 < 12)
   {
      TH1 = -(SYSCLK/BAUDRATE/2/12);
      CKCON &= ~0x13;      // T1M = 0; SCA1:0 = 00
   }
   else
   {
      TH1 = -(SYSCLK/BAUDRATE/2/48);
      CKCON &= ~0x13;      // Clear all T1 related bits
      CKCON |=  0x02;      // T1M = 0; SCA1:0 = 10
   }

   TL1 = TH1;              // initialize Timer1
   TR1 = 1;                // start Timer1

   SFRPAGE = UART1_PAGE;
   TI1 = 1;                // Indicate TX1 ready
   REN1=1;
   EIE2|=0x40;

   SFRPAGE = SFRPAGE_SAVE;
}
```

Adatok küldése és fogadása:

Programozói szinten az SBUF1 regiszter szolgál az adatok vételére és adására. Ténylegesen az SBUF1 olvasásánál az UART1 a belső vételi pufferéből (Receive buffer) a soron következő adatot olvassuk ki. Az SBUF1-be történő írással pedig az UART1 adási puffe-rébe (Transmit regiszter) helyezünk el adatot. Amíg az UART1 az adattovábbítással van elfoglalva, addig nem foglalkozik az SBUF1 tartalmával. Tehát az oda beírt adat elvesz-het. Felmerül a kérdés, hogy akkor mikor lehet írni adatot az SBUF1 regiszterbe. Nos, minden egyes adatátvitel végén az UART1 megszakítást generál. Adattovábbítás esetén az SCON1 regiszterben a TI1 interrupt flag 1-be állítódik. Adat adásakor tehát meg kell várni, míg a TI1 flag 1-be áll. Közvetlenül az adás előtt a TI1-et 0-ba kell állítani. Ennek alapján az 1 bájt adatot a soros vonalra küldő rutin a következő kódja a következő:

```
void SendChar( char B )
{
   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page
   SFRPAGE = UART1_PAGE;
      while(!TI1);
      TI1=0;
      SBUF1=B;
      SFRPAGE=SFRPAGE_SAVE;
}
```

Adatok vételnél az RI1 interupt flag állítódik, ezért a vételnél hasonlóan lehet eljárni, mint az adásnál.

```
char ReceiveChar( void )
{
   char SFRPAGE_SAVE = SFRPAGE;
   SFRPAGE = UART1_PAGE;
   while(!RI1);
   RI1=0;
   B=SBUF1;
   SFRPAGE=SFRPAGE_SAVE;
   return B;
}
```

A fenti vételt megvalósító rutinnak azonban van egy gyenge pontja. Mivel a kommunikáció aszinkron, így simán előfordulhat, hogy az adó oldal minden előzetes nélkül leáll a kommunikációval. Ekkor a szoftverünk a vételi rutin `while` ciklusában arra vár, hogy az RI1 flag 1-be álljon, ami nem következik be. Azaz a szoftverünk egy helyben topog, s egyéb feladataival nem tud foglalkozni. Ez nyilvánvalóan nem jó. Felmerül a kérdés, hogy lehet megoldani hogy ha a soros vonalon vétel van, akkor azt észrevegyük, ugyanakkor a programunk futása ne legyen blokkolva. A megoldás a vételi megszakítás rutin használata. Az elv a következő: Mivel a megszakítási rutinunk minden egyes adat vételének a végén lefut, ezért a megszakítási rutin a vett adatot egy cirkuláris pufferba (ring buffer) teszi be. A programunk pedig ezt puffert olvassa. A pufferhez két indexváltozó is tartozik. Az írási változót a megszakítási rutin kezeli, míg az olvasási változót a fő programunk. Ha a két változó azonos értéken áll, akkor nincs adat a vételi pufferban. A fenti elvet megvalósító kódrészlet a következő:

```
#define MAXSIZE      1000

char xdata RBuffer[MAXSIZE]; // receive puffer
short WIndex;                // write index
short RIndex;                // read index

char GetBytesFromUART( void )
{
    char b;
    char ret;
    ret=0;
    while(RIndex!=WIndex)
    {
        b=RBuffer[RIndex];
        // data processing here
        RIndex++;
```

```
        if( RIndex==MAXSIZE)
            RIndex=0;
    }
    return ret;
}

void UART_ISR (void) interrupt 20
{
   char SFRPAGE_SAVE = SFRPAGE;
   SFRPAGE = UART1_PAGE;
   if( RI1==1 )
   {
    RI1=0;
    RBuffer[WIndex]=SBUF1;
    WIndex++;
    if(WIndex==MAXSIZE )
        WIndex=0;
   }
   SFRPAGE = SFRPAGE_SAVE;
}
```

7.8. Kérdések, feladatok

1. Írjon olyan programot, mely a P1 porton beolvasott értéket a P2 portra írja!
2. Írjon olyan programot, mely az A,B, C, D, E, F karaktereket a P1 portra kiírja!

   Megoldás:

```
#include <c8051f120.h>
void main(void)
{
      unsigned char array[]=”ABCDEF”;
      unsigned char z;
      for (z=0;z<=10;z++)
            P1=array[z];
}
```

3. Írjon olyan programot, mely a fejlesztői kártyán levő zöld LED-et 10 Hz frekvenciával ki és bekapcsolja!

   Megoldás:

```
#include <c8051f120.h>

sfr16 RCAP3 = 0xCA;
sfr16 TMR3  = 0xCC;

#define SYSCLK 3062500

sbit  LED = P1^6;

void main (void)
{
      WDTCN = 0xde;
```

```
        WDTCN = 0xad;

        SFRPAGE = CONFIG_PAGE;
        XBR2    = 0x40;
        P1MDOUT |= 0x40;

        SFRPAGE = TMR3_PAGE;
        TMR3CN = 0x00;
        RCAP3   = -(SYSCLK / 12 / 10);
        TMR3    = 0xffff;
        EIE2   |= 0x01;
        TR3 = 1;

        EA = 1;

        SFRPAGE = LEGACY_PAGE;

        while (1)
        {

        }
}

void Timer3_ISR (void) interrupt 14
{
        TF3 = 0;
        LED = ~LED;
}
```

4. Írjon olyan programot, mely az UART1-re folyamatosan kiírja a „Hello world!” üzenetet.

   Megoldás:

```
#include <c8051f120.h>

void SendChar( char B )
{
        char SFRPAGE_SAVE = SFRPAGE;
        SFRPAGE = UART1_PAGE;
        while(!TI1);
        TI1=0;
        SBUF1=B;
        SFRPAGE=SFRPAGE_SAVE;
}

void SendString( char *str )
{
        while( *str )
        {
                SendChar(*str);
                str++;
        }
}

void main (void)
{
        WDTCN = 0xde;
        WDTCN = 0xad;

        //SysClock init
```

```
        SFRPAGE = CONFIG_PAGE;
        OSCICN = 0x83;
        CLKSEL = 0x00;

        //port init
        SFRPAGE = CONFIG_PAGE;
        XBR0     = 0x00;
        XBR1     = 0x00;
        XBR2     = 0x44;

        P0MDOUT |= 0x01;

        //UART1 init
        SFRPAGE = UART1_PAGE;
        SCON1   = 0x10;
        SFRPAGE = TIMER01_PAGE;
        TMOD   &= ~0xF0;
        TMOD   |=  0x20;
        TH1 = -(SYSCLK/BAUDRATE/2);
        CKCON |= 0x10;
        TL1 = TH1;
        TR1 = 1;
        SFRPAGE = UART1_PAGE;
        TI1 = 1;
        REN1=1;
        EIE2|=0x40;

        EA = 1;
        SFRPAGE = LEGACY_PAGE;

        while (1)
        {
              SendString("Hello World!");
        }
}
```

8. Adatfogadás személyi számítógéppel

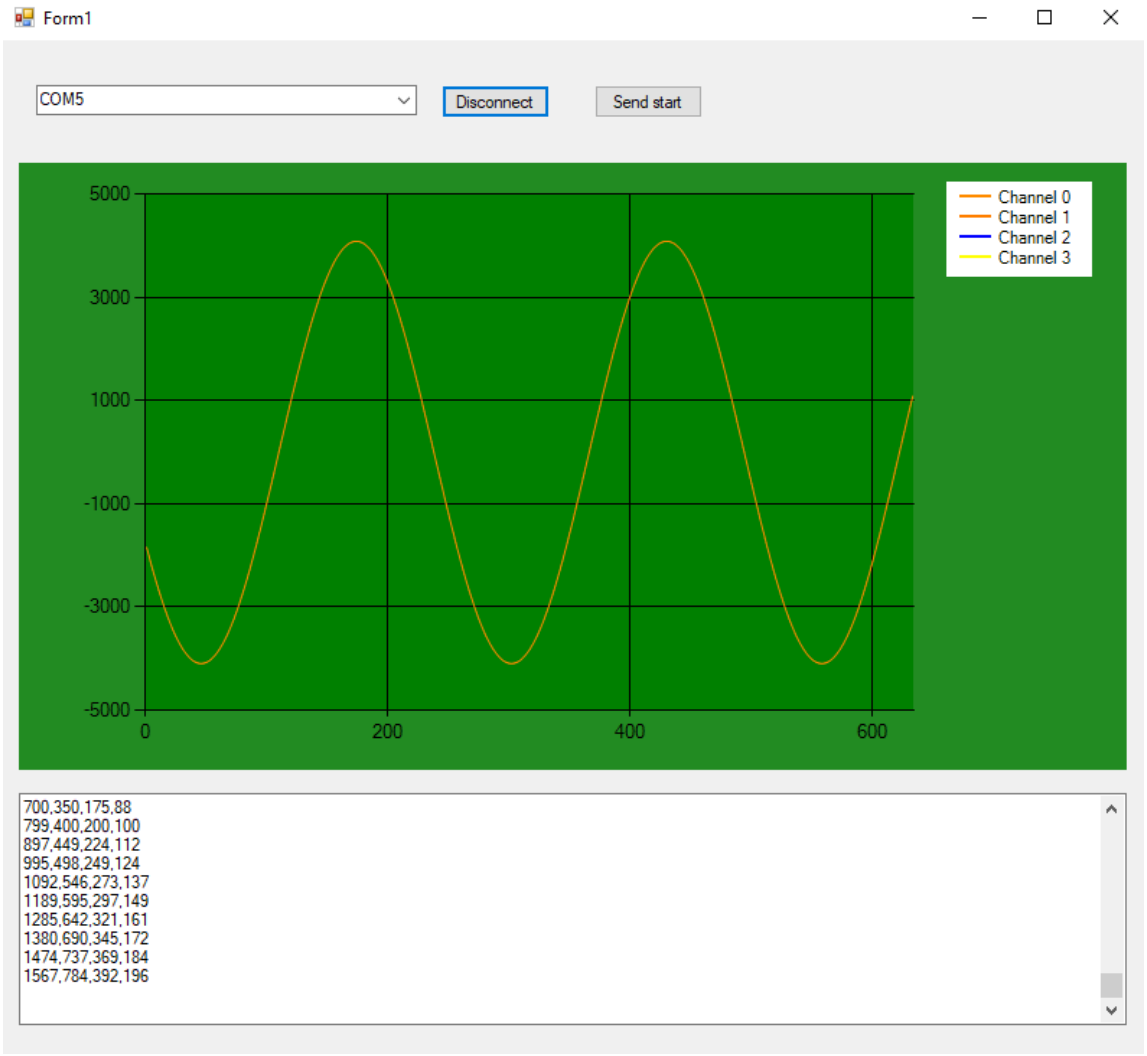
Mint már a 6. fejezetben említettük a személyi számítógép feladata a jelek fogadása a mikrokontrolleres egységtől, jelek tárolása, grafikus megjelenítése, valamint azon algoritmusok végrehajtása, melyek a felhasználó számára értelmezhetővé teszik a jel által hordozott információt. A mai számítógépek soros vonali kommunikációra az USB porton keresztül képesek, azonban a legtöbb mikrokontroller RS232 szabványú soros kommunikációt támogat. Ezt RS232-USB átalakító használatával hidalhatjuk át (8.1 ábra). Ezek az átalakítók virtuális soros portot hoznak létre a számítógép operációs rendszerében. Azoknál a mikrokontrollereknél, amelyek már USB porttal vannak ellátva, ott a mikrokontroller meghajtó szoftvere hozza létre a virtuális portot.

8.1 ábra: RS232-USB átalakító

Demonstrációs céllal tekintsük a 8.2 ábrát, melyen egy soros vonalról adatot fogadó Windows form alkalmazás képernyő képe látható. Az alkalmazás a soros vonalról jövő adatokat grafikusan és szöveges formátumban is megjeleníti. A grafikus megjelenítéshez az MS .NET Chart vezérlőjét, a szöveges megjelenítéshez pedig a TextBox vezérlőt használja.

A soros vonal kezelésére a .NET `SerialPort` osztályát érdemes használni a következő módon:

- Példányosítjuk az osztályt a kívánt kommunikációs paraméterekkel (port, baud, stb).
- Megnyitjuk a portot.
- Létrehozzuk a soros kommunikáció callback függvényét, melyet az operációs rendszer kezel. Ebben a callback függvényben helyezzük el azt a kódot, amely a soros vonali pufferből áthelyezi a beérkezett adatokat a programunk saját tárhelyére.
- A saját térhelyen levő adatokat szövegesen és grafikusan is megjelenítjük.



8.2 ábra: Soros vonali adatmegjelenítő szoftver

A szoftver forráskódja:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO.Ports;
using System.IO;


namespace WinScope
{

 public partial class Form1 : Form
```

```
 {
  Timer timer;

  private SerialPort _serialPort;
  Queue<string> dataBuffer;

  public Form1()
  {
   InitializeComponent();
   ListComPorts();
   dataBuffer = new Queue<string>();

   chart1.ChartAreas[0].AxisX.Minimum = double.NaN;
   chart1.ChartAreas[0].AxisX.Maximum = double.NaN;
   chart1.ChartAreas[0].AxisY.Minimum = -5000;
   chart1.ChartAreas[0].AxisY.Maximum = 5000;
 }


 public void SerialInit(string portName)
 {
  _serialPort = new SerialPort(portName,
                               115200,
                               Parity.None,
                               8,
                               StopBits.One);
 _serialPort.Open();
 _serialPort.DataReceived += serialPort_DataReceived;

}

void serialPort_DataReceived(object s, SerialDataReceivedEventArgs e)
{
  string InputData = _serialPort.ReadLine();
  dataBuffer.Enqueue(InputData);
  textBox1.AppendText(InputData+"\n");

}

public void Dispose()
{
  if (_serialPort != null && _serialPort.IsOpen)
  {
    _serialPort.Dispose();
  }
}

private void ListComPorts()
{
   // Get a list of serial port names.
   string[] ports = SerialPort.GetPortNames();

   // Add each port name to the list box.
   foreach (string port in ports)
   {

     comboBox1.Items.Add(port);
   }
   if(ports.Length>0)
       comboBox1.SelectedIndex = 0;
 }
```

```
private void Form1_Load(object sender, EventArgs e)
{
  timer = new Timer();
  timer.Tick += Timer_Tick;
  timer.Interval = 10;
}


private void Timer_Tick(object sender, EventArgs e)
        {

            while(chart1.Series[0].Points.Count > 1256)
            {
                chart1.Series[0].Points.RemoveAt(0);
            }

            for (int i=0; i<dataBuffer.Count; i++)
            {
                string str = dataBuffer.Dequeue();
                string[] sor = str.Split(' ');
                if( sor.Length>1)
                    chart1.Series[0].Points.AddXY(sor[0],sor[1]);
            }


        }

        private void button1_Click(object sender, EventArgs e)
        {
             if (timer.Enabled)
                  timer.Stop();
              else
                  timer.Start();


            if (button1.Text == "Connect")
            {
                textBox1.Text = "";
                string port = comboBox1.SelectedItem.ToString();
                SerialInit(port);
                button1.Text = "Disconnect";
            }
            else
            {

                Dispose();
                button1.Text = "Connect";
            }

        }

        private void button2_Click(object sender, EventArgs e)
        {
            _serialPort.Write("F");
        }
    }

}
```

9. Gyakorlati példa

Működő gyakorlati példaként egy egyszerű feszültségmérő alkalmazás elkészítését mutatjuk be lépésről lépésre. A feszültséget külső analóg jelként kötjük rá az ADC0-ra, melyet 1 kHz frekvenciával mintavételezünk. A mért feszültség értékeket soros vonalon (baud:115200, data bits: 8, stop bit:1, parity: none) elküldjük a külvilágnak. Valamint ha a feszültség értéke nagyobb, mint 2 V akkor a fejlesztői kártyán levő LED-et bekapcsoljuk. Ha feszültség értéke kisebb, mint 2 V, akkor kikapcsoljuk.

9.1. Megoldás

A megoldás főbb lépései:

- Definiáljuk a globális konstansokat a feladatban megadott paraméter értékekkel.
- Deklaráljuk a globális változókat és függvény prototípusokat.
- Implementáljuk a függvényeket

```
//-----------------------------------------------------------------------
// Includes
//-----------------------------------------------------------------------

#include <c8051f120.h>                  // SFR declarations
#include <stdio.h>
#include <stdlib.h>

//-----------------------------------------------------------------------
// 16-bit SFR Definitions for 'F12x
//-----------------------------------------------------------------------

sfr16 DP       = 0x82;                  // data pointer
sfr16 ADC0     = 0xbe;                  // ADC0 data
sfr16 ADC0GT   = 0xc4;                  // ADC0 greater than window
sfr16 ADC0LT   = 0xc6;                  // ADC0 less than window
sfr16 RCAP2    = 0xca;                  // Timer2 capture/reload
sfr16 RCAP3    = 0xca;                  // Timer3 capture/reload
sfr16 RCAP4    = 0xca;                  // Timer4 capture/reload
sfr16 TMR2     = 0xcc;                  // Timer2
sfr16 TMR3     = 0xcc;                  // Timer3
sfr16 TMR4     = 0xcc;                  // Timer4
sfr16 DAC0     = 0xd2;                  // DAC0 data
sfr16 DAC1     = 0xd2;                  // DAC1 data
sfr16 PCA0CP5  = 0xe1;                  // PCA0 Module 5 capture
sfr16 PCA0CP2  = 0xe9;                  // PCA0 Module 2 capture
sfr16 PCA0CP3  = 0xeb;                  // PCA0 Module 3 capture
sfr16 PCA0CP4  = 0xed;                  // PCA0 Module 4 capture
sfr16 PCA0     = 0xf9;                  // PCA0 counter
sfr16 PCA0CP0  = 0xfb;                  // PCA0 Module 0 capture
sfr16 PCA0CP1  = 0xfd;                  // PCA0 Module 1 capture
```

```
//-----------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------

#define BAUDRATE     115200      // Baud rate of UART in bps
#define SYSCLK       24500000    // Internal oscillator frequency in Hz

#define SAMPLE_RATE  1000       // Sample frequency in Hz

#define TIMER_SYSCLK     24500000/12

sbit SHOLD = P1^6;                 // LED pin



//-----------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------

void Hardware_Init();

void ADC0_ISR (void);
void UART_ISR (void);
void SendChar( char B );
void SendString( char *str );
void SendShort( short num);
void SendData();


//-----------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------

short result1, result2, result3;   // ADC0 decimated value
short ADBuffer[3];
short Timer_tick;
short Timer_reload;

char ADReady;

short VOLT;


//-----------------------------------------------------------------------
// MAIN Routine
//-----------------------------------------------------------------------

void main (void)
{

   WDTCN = 0xde;                        // disable watchdog timer
   WDTCN = 0xad;

   WIndex=0;
   RIndex=0;
   CIndex=0;

   ADReady=0;

   Timer_reload= SAMPLE_RATE;
   Timer_tick=Timer_reload;
```

```
   VOLT=3413;                          // (2V/2.4V)*4096

   ClearCommandBuffer();

   Hardware_Init();

   SFRPAGE = LEGACY_PAGE;

   while (1)
   {
      if( ADReady==1)
      {
            SendString("AD: ");
            SendShort(ADBuffer[0]);
            SendString("\n");

            SHOLD= ADBuffer[0] >=VOLT ? 1 : 0;

            ADReady=0;
      }

   }
}


//-----------------------------------------------------------------------
// Initialization Subroutines
//-----------------------------------------------------------------------


//-----------------------------------------------------------------------
// Harware_Init
//-----------------------------------------------------------------------
//
void Hardware_Init ()
{
   char SFRPAGE_SAVE = SFRPAGE;     // Save Current SFR page

  //SysClock init
   SFRPAGE = CONFIG_PAGE;           // set SFR page

   OSCICN = 0x83;                   // set internal oscillator to run
                                    // at its maximum frequency

   CLKSEL = 0x00;                   // Select the internal osc. as
                                    // the SYSCLK source
   //port init
   SFRPAGE = CONFIG_PAGE;           // set SFR page

   XBR0     = 0x00;
   XBR1     = 0x00;
   XBR2     = 0x44;                 // Enable crossbar and weak pull-up
                                    // Enable UART1

   P0MDOUT |= 0x01;                 // Set TX1 pin to push-pull
   P1MDOUT |= 0x40;                 // Set P1.6(SHOLD) to push-pull

   //UART1 init
   SFRPAGE = UART1_PAGE;
   SCON1   = 0x10;              // SCON1: mode 0, 8-bit UART, enable RX

   SFRPAGE = TIMER01_PAGE;
```

```
   TMOD   &= ~0xF0;
   TMOD   |=  0x20;               // TMOD: timer 1, mode 2, 8-bit reload

   TH1 = -(SYSCLK/BAUDRATE/2);
   CKCON |= 0x10;


   TL1 = TH1;                     // initialize Timer1
   TR1 = 1;                       // start Timer1

   SFRPAGE = UART1_PAGE;
   TI1 = 1;                         // Indicate TX1 ready
   REN1=1;
   EIE2|=0x40;


   //Timer3 init
   SFRPAGE = TMR3_PAGE;

   TMR3CN = 0x00;                       // Stop Timer3; Clear TF3;
   TMR3CF = 0x08;                       // use SYSCLK as timebase

   RCAP3   = -(SYSCLK/SAMPLE_RATE);     // Init reload values
   TMR3    = RCAP3;                     // set to reload immediately
   EIE2   &= ~0x01;                     // disable Timer3 interrupts
   TR3 = 1;                             // start Timer3

   // ADC0 init
   SFRPAGE = ADC0_PAGE;

   ADC0CN =0x04;                 // ADC0 disabled; normal tracking
                                 // mode; ADC0 conversions are initiated
                                 // on overflow of Timer3; ADC0 data is
                                 // right-justified
   REF0CN = 0x03;//0x07;

   AMX0SL =0; //0x0f;
   ADC0CF = (SYSCLK/2500000) << 3; // ADC conversion clock = 2.5MHz

   EIE2 |= 0x02;                   // enable ADC interrupts


   SHOLD = 0;
   AD0EN = 1;                      // enable ADC

   EA = 1;                         // Enable global interrupts

   SFRPAGE = SFRPAGE_SAVE;        // Restore SFR page
}


void SendChar( char B )
{
   char SFRPAGE_SAVE = SFRPAGE;         // Save Current SFR page
   SFRPAGE = UART1_PAGE;
      while(!TI1);
      TI1=0;
      SBUF1=B;
      SFRPAGE=SFRPAGE_SAVE;
}
```

```
void SendString( char *str )
{
	while( *str )
	{
		SendChar(*str);
		str++;
	}
}


void SendShort( short num)
{
	char str[5];
	short maradek,hanyados;
	int i;
	str[0]='0';
	str[1]='0';
	str[2]='0';
	str[3]='0';
	str[4]='\0';
	i=3;
	hanyados=num;
	while(hanyados)
	{
		maradek=hanyados%10;
		hanyados/=10;
		str[i]=maradek+'0';
		i--;
	}
	SendString(str);
}


//-----------------------------------------------------------------------
// Interrupt Service Routines
//-----------------------------------------------------------------------

//-----------------------------------------------------------------------
// ADC0_ISR
//-----------------------------------------------------------------------
//
// ADC0 end-of-conversion ISR
// Here we take the ADC0 sample, add it to a running total
// <accumulator>, and decrement our local decimation counter <int_dec>.
// When <int_dec> reaches zero, we post the decimated result in the //
// global variable <result>.
//
void ADC0_ISR (void) interrupt 15
{
   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page

   SFRPAGE = ADC0_PAGE;

   AD0INT=0;
   result1 = ADC0;
   Timer_tick--;
   if( Timer_tick==0)
   {
        Timer_tick=Timer_reload;
        ADBuffer[0]=result1;
        ADReady=1;
   }
```

```
   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}

//-----------------------------------------------------------------------
// UART1_ISR
//-----------------------------------------------------------------------
void UART_ISR (void) interrupt 20
{
   char SFRPAGE_SAVE = SFRPAGE;        // Save Current SFR page
   SFRPAGE = UART1_PAGE;
   if( RI1==1 )
   {
      RI1=0;
      RBuffer[WIndex]=SBUF1;
      WIndex++;
      if(WIndex==MAXSIZE )
            WIndex=0;
   }
   SFRPAGE = SFRPAGE_SAVE;             // Restore SFR page
}
```

10. Hivatkozások, irodalom

1. https://www.fammed.wisc.edu/medstudent/pcc/ecg/axis.html
2. http://upload.wikimedia.org/wikipedia/commons/thumb/2/28/AliasingSines.svg/500px-AliasingSines.svg.png
3. http://www.noise.physx.u-szeged.hu/Education/Elektronika/AKonvDerterek.pdf
4. http://hu.wikipedia.org/wiki/F%C3%A1jl:Flash_ADC.png
5. http://www.ms.sapientia.ro/~manyi/teaching/dsp/eloadas2.pdf
6. Kelemen András: Digitális jelfeldolgozás, Budapest: Digitális Tankönyvtár, 2013.
7. Székely Vladimir: Képkorrekció, Hanganalízis, Térszámítás PC-n. CompoterBooks Budapest 1994. ISBN: 9636180172
8. Valkó Péter, Vajda Sándor: Műszaki-tudományos feladatok megoldása személyi számítógéppel, Műszaki könyvkiadó Budapest 1987, ISBN 963-10-6928-1
9. Norbert Hesselmann: Digitális jelfeldolgozás, Műszaki könyvkiadó Budapest 1985, ISBN 963 10 6422 0
10. http://e-oktat.pmmf.hu/kepeshang_2_fejezet
11. http://e-oktat.pmmf.hu/kepeshang_3_fejezet
12. http://e-oktat.pmmf.hu/kepeshang_4_fejezet
13. http://e-oktat.pmmf.hu/kepeshang_5_fejezet
14. https://randomnerdtutorials.com/getting-started-with-esp32/
15. U. Tietze, Ch. Schenk: Analóg és digitális áramkörök, Műszaki könyvkiadó Budapest, 1993, ISBN 963 16 0010 6
16. Max-Peter Gottlob: Elektrotechnikai és elektronikai számítások komplex számokkal, Műszaki könyvkiadó Budapest,1983, ISBN 963 10 4408 4
17. http://www.silabs.com/Support%20Documents/TechnicalDocs/C8051F12x-13x.pdf
18. https://www.mccdaq.com/pdfs/anpdf/Data-Acquisition-Handbook.pdf
19. Kai Qian, David den Haring, Li Cao: Embedded Software Development with C, Springer Science+Business Media, LLC 2009, ISBN 978-1-4419-0605-2