Mikrokontroller alapú rendszerek

Elektronikus jegyzet

1. fejezet

Készítette: Dr. Tevesz Gábor c. egyetemi tanár
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
Fax: 463-2871 (adm.)
Mail: tevesz@aut.bme.hu

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki alapszak (BSc)
III. évfolyam, 5. félév
Beágyazott és irányító rendszerek specializáció hallgatói

COPYRIGHT

Köszönetemet fejezem ki Kiss Domokos tanársegédnek az 1.3 fejezet kidolgozásában és ábraanyagának összeállításában nyújtott segítségéért.

TARTALOMJEGYZÉK

BEVEZETÉS

Ez a tantárgy a Budapesti Műszaki Egyetem Villamosmérnöki és Informatikai Kar nappali tagozatán kerül előadásra azon villamosmérnöki alapképzésben résztvevő hallgatók számára, akik képzési szakterületként a **Beágyazott és irányító rendszerek** specializációt választották.

A differenciált szakmai ismeretek elsajátításához az 5. félévtől a hallgatóknak **specializációt** és azon belül **ágazatot** kell választaniuk. Minden specializációhoz 4-4 elméleti tantárgy tartozik, melyeket valamennyi, az adott specializációhoz tartozó hallgatónak el kell sajátítania. Az ágazatok közötti különbségek a Témalaboratórium, a Specializáció laboratórium és az Önálló laboratórium tantárgyakban, majd a 7. félévben a Szakdolgozatban jelennek meg. A villamosmérnöki szakos hallgatók számára meghirdetett Szabadon választható tantárgyak kínálatából – a specializációtól és az ágazattól függetlenül - tetszése szerint választhat mindenki.

A **Beágyazott és irányító rendszerek specializáció** felépítése tehát:

5. félév:
Specializáció elméleti tantárgyak (ezek mindegyikét meg kell hallgatniuk):
- Beágyazott és ambiens rendszerek (MIT)
- Ipari irányítástechnika (IIT)
- Mikrokontroller alapú rendszerek (AUT)
- Témalaboratórium (minden hallgató a saját anyatanszékén veszi fel)

6. félév:
A specializáció ágazatai szerint különböző tantárgyak:
- A **Beágyazott információs rendszerek ágazat**ot választó hallgatók számára:
  - Párhuzamos és eseményvezérelt programozás beágyazott rendszerekben (MIT)
  - Beágyazott és ambiens rendszerek laboratórium (MIT)
  - Önálló laboratórium (AUT, IIT, MIT)
- Az **Irányítórendszerek ágazat**ot választó hallgatók számára:
  - Ipari képfeldolgozás és képmegjelenítés (IIT)
  - Irányítórendszerek laboratórium (IIT)
  - Önálló laboratórium (AUT, IIT, MIT)
- A **Számítógép-alapú rendszerek ágazat**ot választó hallgatók számára:
  - Beágyazott operációs rendszerek és kliens alkalmazások (AUT)
  - Mikrokontroller laboratórium (AUT)
  - Önálló laboratórium (AUT, IIT, MIT)

7. félév:
- Szakdolgozat (AUT, IIT, MIT)

A **Mikrokontroller alapú rendszerek** c. tantárgy célkitűzése, hogy a hallgatókat megismertesse az iparban legelterjedtebben használt mikrokontroller architektúrákkal, azok kiválasztási szempontjaival. A megszerzett ismeretek segítségével a hallgatók képessé válnak mikrokontroller alapú rendszerek hardver tervezésére és alacsonyszintű szoftver rendszerének megvalósítására. A kettő közötti elválaszthatatlan kapcsolatot rövid esettanulmányok mutatják be. A létrehozott egység monitorozási és diagnosztikai információs rendszerét gyors alkalmazásfejlesztő módszerek alkalmazásával alakítjuk ki a legelterjedtebb ipari platformokon.

A tantárgy 5 fő fejezetet tartalmaz, ezek a következők:

1. Architektúrális alapok
Ebben a fejezetben megismerkedünk a digitális rendszerek központi egységeivel, a mikroprocesszorok és mikrokontrollerek fontosabb architektúráival, kiválasztási szempontjaival (8/16/32 bites rendszerek, CISC/RISC architektúra). Bemutatásra kerül az elterjedtebb mikrokontroller-architektúrák egy-egy neves képviselője (8051 és ARM Cortex M4 mikrokontrollerek).

2. Hardverközeli programok fejlesztése
Itt a korábbi programozási ismeretekre építve megismerjük az assembly és a C szintű, hardver közeli programfejlesztés sajátosságait. Bemutatunk egy tipikus ASM/C fejlesztő környezetet (SiLabs, Keil, STM), a beágyazott programrendszerek szerkezetét és specialitásait. Egész és törtszámok ábrázolása, szabványok, áttérések különböző számábrázolási méretek között.

3. Mikrokontrollerek integrált perifériái
Ebben a fejezetben a mikrokontrollerek legfontosabb belső egységeivel és perifériáival ismerkedünk meg (órajel-generátorok, reset-, watch-dog áramkörök, memória elemek, időzítő és számláló egységek, aszinkron és szinkron kommunikációs egységek és protokollok, digitális és analóg be- és kimenetek).

4. Mikrokontrollerek környezete, illesztések
Külső egységek illesztése a mikrokontrollerekhez, analóg és digitális be- és kimenetek tipikus illesztése, speciális perifériák. Jelkonvertálás fizikai rétegre (RS232, RS485, CAN). EMC szempontok, leválasztások.

5. Beágyazott rendszerek tervezésének alapelvei és lépései
Megtanuljuk a CAD rendszerek használatát a hardvertervezésben: megismerjük a kapcsolási rajz, a szimuláció, és a nyomtatott áramkör tervező rendszerek legfontosabb tulajdonságait. Foglalkozunk a formai és a tartalmi követelményekkel, az alkatrészek és az áramkörök technológiai kérdéseivel (hagyományos/felületszerelt, rétegszám megválasztása, forrasztási technológia választása stb.). Bemutatásra kerülnek az élesztés, programozás, tesztelés interfészei.

A tantárgy anyaga előadásokon és gyakorlatokon kerül ismertetésre. Az előadások és a gyakorlatok az anyag ütemében váltogatják egymást, a gyakorlatokon példák, és esettanulmányok formájában kerül elmélyítésre az előadásokon elhangzott elméleti tananyag. A tantárgyat felvevő hallgatóknak egy **otthoni feladatot** és egy **zárthelyi feladatot** kell teljesíteniük a félév során, majd a félév végén (írásbeli) **vizsgát** kell tenniük.

1. ARCHITEKTÚRÁLIS ALAPOK

1.1 Beágyazott rendszerek

Először is definiáljuk: mit nevezünk (manapság rendkívül népszerű szóösszetétellel) beágyazott rendszernek?

Beágyazott rendszernek nevezzük az olyan mikroprocesszoron, mikrokontrolleren alapuló, programvezérlésű elektronikus eszközt (illetve az ilyen eszközökből alkotott rendszert), amely a befogadó fizikai / kémiai / biológiai környezetét autonóm módon képes érzékelők segítségével megfigyelni, és beavatkozók segítségével befolyásolni. (***Autonóm:*** *a kitűzött célt önállóan, külső segítség nélkül képes elérni, működése a változó feltételek között automatikus.*)
Felvetődik azonnal a specializáció másik szaktárgyának nevében található fogalom, az ambiens szó értelmezése [*ambient = környező, körülvevő*]: ***Ambiens rendszereknek*** nevezzük azokat a beágyazott rendszereket, amelyek az emberi (pl. otthoni vagy munkahelyi) környezet részévé válva elsősorban az életvitel és az életminőség szolgálatában állnak. Ennek megfelelően az ambiens rendszerek legfőbb tulajdonsága az emberközpontúság, mégpedig úgy, hogy a működésük a lehető legkevesebb terhet jelentse azoknak, akiknek az érdekében létrehozták őket.

Minden beágyazott rendszert program és processzor működtet, de nem minden programozott és processzorral működtetett rendszer beágyazott rendszer. A megkülönböztetés nem egyszerű, mivel működési elvük azonos, és ezért a fejlesztésükhöz használt módszerek és eszközök is hasonlóak. Tovább nehezíti az elhatárolást, hogy a beágyazott rendszer gyakran nem önálló, fizikailag jól elkülöníthető része az általa működtetett rendszernek. A beágyazott rendszer jellegzetességeként szokták említeni azt is, hogy számítógépszerűen nem használható („nem PC”). Az esetek nagy részében a beágyazott rendszer kívülről nem látható, azaz az általa működtetett tárgyról nem látszik, hogy mi vezérli.

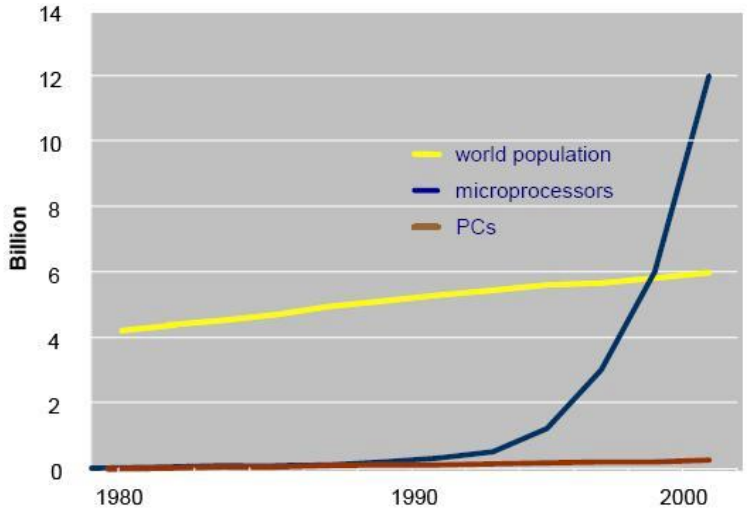
Fentiek megértésére gyorsan néhány példa beágyazott rendszerekre:
- legelőször az elektronikai ipar termékeiben jelentek meg a feladatok elhatárolása, szeparálása, szétosztása végett. A feladat olyan bonyolultságú és sebességű, hogy annak megoldásához célszerű programozható logikát alkalmazni, ennek változtathatósága, a felhasználó által való programozhatósága azonban tudatosan nem cél (számítógép billentyűzetek, nyomtatók, winchesterek, CD-meghajtók, router – CD/DVD lejátszók, televíziók, mobiltelefonok, videó berendezések stb.)
- ezt követően (elsősorban alacsony áruk miatt) megjelentek a kommersz háztartási berendezésekben is (mosógép, mikrohullámú sütő, óra, hőmérő, riasztó, távirányító stb.)
- a jármű- és közlekedésiparban – autóink, közlekedési eszközeink egyre jobban „elektronizálódnak” (ABS, sebességváltó, fedélzeti számítógép, motorvezérlő elektronikák, követőradarok, kerékkipörgés-gátlók, adaptív tempomat, sávtartás stb.)

- orvostechnikai berendezésekben – kommersz és professzionális szinten is (vérnyomásmérők, vércukorszint-mérők, de a CT-, MR-, EKG-, röntgen berendezésekben, életfunkciókat felügyelő műszerekben stb.)
- az ipar számos egyéb területén

Nem tekintjük beágyazott rendszernek az általános célú számítógépeket (pl. az asztali és hordozható személyi számítógépeket, a táblagépeket és a szervergépeket), az ipari vezérlésre használt számítógépeket, és a kézi számítógépeket sem, viszont egyes komponenseiket (monitoraikat, lemezmeghajtóikat, nyomtatóikat stb.) ugyancsak beágyazott rendszerek működtetik.

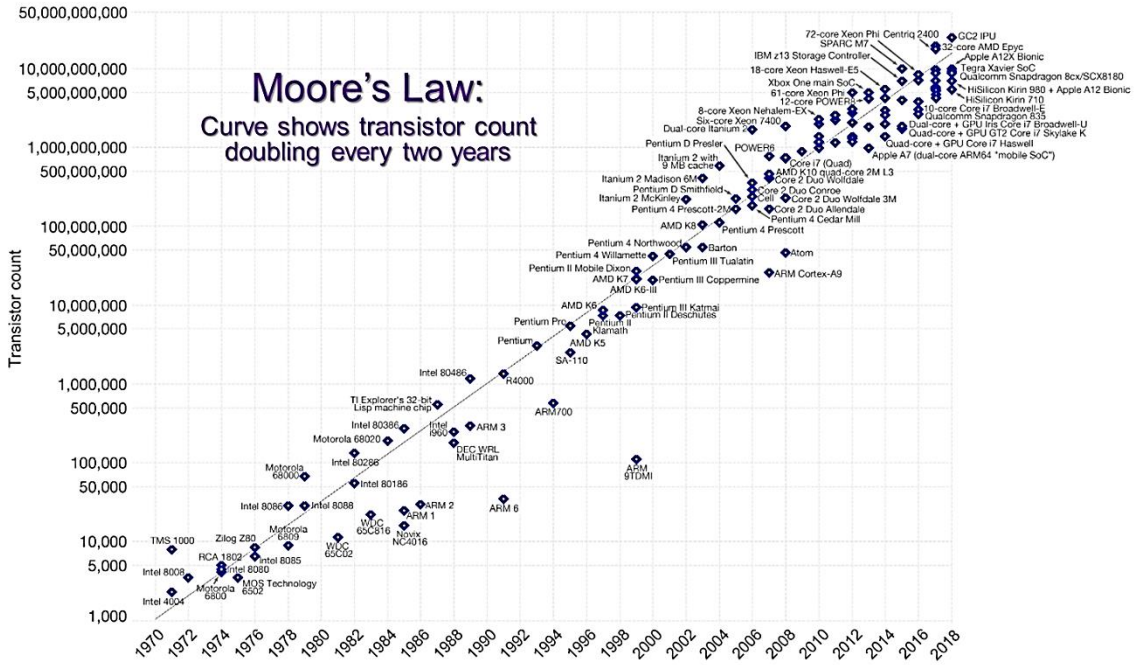
Egy német cég (FAST GmbH) a müncheni Műszaki Egyetemmel együttműködve 2005-ben 280 oldalas tanulmányt készített a brüsszeli Európai Bizottság (az Európai Unió „kormánya") számára a beágyazott rendszerek szerepéről és várható fejlődéséről a közeljövőben. Ebben a következő érdekesebb adatok találhatók:

- 2000-ben 10 milliárd mikroprocesszor volt használatban, **98 százalékuk beágyazott rendszerekben.**
- Becslések szerint 2014-re a mikroprocesszorok száma 2000-hez képest megduplázódott (20 milliárd), azaz **minden emberre 3 mikroprocesszor jut a Földön** (2005-ben az ENSZ Népességi Hivatal jelentése szerint a Föld lakossága 6.5 milliárd volt, 2023-ban átlépte a 8 milliárdot). A beágyazott rendszerek világpiaca 2009-re 71 milliárd euróra nőtt, növekedési üteme 2004 és 2009 között évi 14%. A PC-piac növekedési üteme ennél jóval kisebb, kb. évi 8%, ami azt jelenti, hogy a beágyazott rendszerek piaca az elektronikai ipar fő hajtóereje ezekben és az eljövendő években. A beágyazott szoftver piaci értékének növekedését még a hardverénél is nagyobbra, évi 16%-ra taksálják 2004-től 2009-ig.

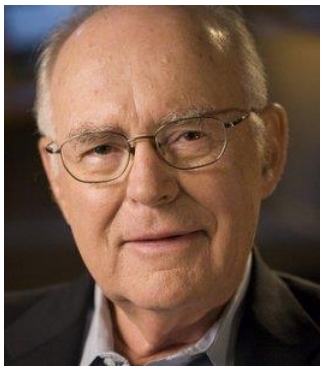


A mikroprocesszorok számának növekedése a világ lakosságához képest

- Mindez az évi 2%-os GDP-növekedés tükrében a beágyazott rendszerek fokozódó jelentőségét mutatja ebben az időszakban.



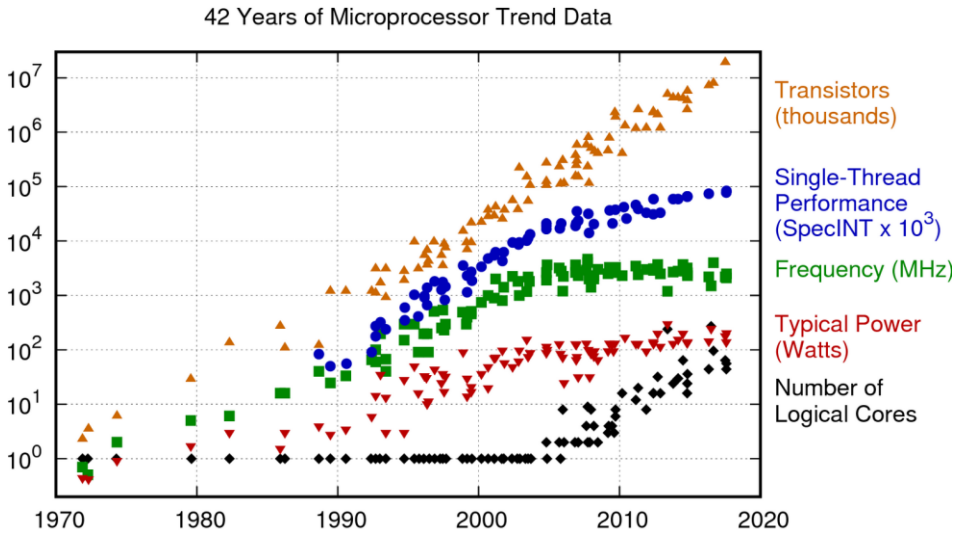
Moore törvénye a valósággal összevetve



Gordon Earle Moore, az Intel Corporation egyik alapítója az Electronics Magazine 1965. április 19-i számában megjelent „Még több komponens megvalósítása az integrált áramkörökben” című írásában a következő megállapítást fogalmazta meg: „A legalacsonyabb árú komponens összetettsége évenként durván a kétszeresére nőtt… Rövidtávon ez az ütem várhatóan nem fog jelentősen változni, esetleg valamelyest növekszik. Hosszú távon a növekedés üteme bizonytalanabb, bár jelenleg nincs okunk feltételezni, hogy az elkövetkező 10 évben ez változni fog. Ez azt jelenti, hogy 1975-ben a legalacsonyabb árú integrált áramkör 65 000 komponenst fog tartalmazni. Úgy hiszem, hogy egy ilyen összetett áramkör megépíthető egy lapkán.”

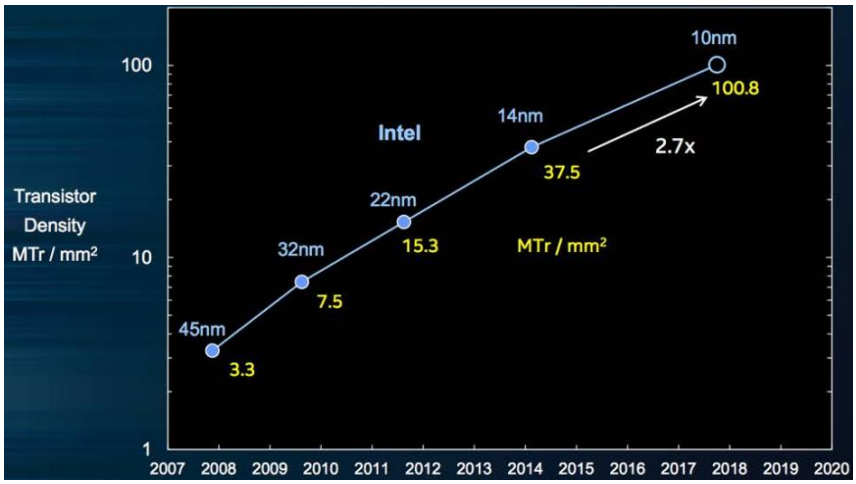
1975-ben Moore némileg „visszafogta” korábbi becslését, a tapasztalatok alapján az alkatrészsűrűség kb. kétévente történő megduplázódását jósolta. Megfigyelését nem nevezte törvénynek, a jelenséget először Carver Mead, a Caltech professzora, a VLSI technológia egyik úttörője hívta Moore törvényének.

Karl Rupp a TU-Wien számítógép kutatója évente tesz közzé felméréseket a mikroprocesszorok fejlődési trendjéről. 2018. február 15-én publikált, a mikroprocesszorok 42 éves fejlődésével foglalkozó publikációjában mind a tranzisztorok számának alakulásával, mind a teljesítmény és a működési frekvencia elemzésével is foglalkozik.



Karl Rupp publikációja a mikroprocesszorok fejlődéséről (2018. február)

Jól látható, hogy a tranzisztorok számának exponenciális növekedése töretlenül tartja magát, a 2011-es 2,6 milliárdos szám 2018-ra az AMD Epyc processzoraiban 19 milliárdra növekedett (az NVIDIA GP100 Pascal grafikus processzorok 15 milliárd tranzisztorból épülnek, azaz az előző szám még csak kiugrónak, vagy egyedülállónak sem tekinthető). A 10 nm-es technológia bevezetésével a trend a következő években várhatóan tartani fogja magát.



A 10 nm-es technológia előnye: 10^8 tranzisztor/mm^2 (Intel, 2017. szeptember)

1.2 Egy kis fejszámolás

A fenti adatok tükrében nem meglepő, hogy a mikrokontrollerek választéka nagyon nagy, a tapasztalatlan fejlesztő tanácstalanul néz: „melyiket válasszam”? Természetesen ezt a kérdést nem intézhetjük el egy sommás válasszal, hogy „ez vagy ez a legjobb” – nagyon sok szempont figyelembevétele után hozható meg az a döntés, hogy egy konkrét feladat megoldásához melyik típus választása a legcélszerűbb.

A legmeghatározóbb szempontok a választás során a következők lehetnek:

1) architektúrális tulajdonságok
2) teljesítőképesség,
3) ár
4) fogyasztás
5) szállítási idő
6) gyártó megbízhatósága, rendelkezésre állása
7) szoftver komponensek rendelkezésre állása
8) tapasztalat az eszközzel, eszközcsaláddal
9) fejlesztőeszközök ára
10) gyártási szempontok

Az elsőnek említett pont a legkomplexebb, nem is függetleníthető a többi pontoktól sem. A megfelelő **architektúra** kiválasztása befolyásolja a rendszer legfontosabb tulajdonságait (sebesség, teljesítőképesség), ezeken keresztül az árat, fogyasztást stb.

A **teljesítőképesség** (2) helyes becslése is több pontra van befolyással: érinti az árat, a fogyasztást, a rendszer fejlesztési költségeit. Nem lehet mindenre „nagyágyúval” lőni, de egy alulbecsült rendszer „szoftverrel történő korrigálása” többnyire indokolatlan lassúságot és működésbeli kompromisszumokat eredményez.

Az **ár** (3) részben az eszköz, részben a hozzá kapcsolódó környezet kölcsönhatásából alakul ki (egy jól megválasztott kontroller, amihez „szinte semmi sem kell” akár kicsit drágább is lehet). Tévhit „a hardver ára ma már nem számít” elképzelés: egy tömeggyártásba kerülő termék fejlesztési költségei sokszor több millió részre oszlanak el, de a rosszul választott mikrokontroller magasabb ára, vagy hardver környezetének többletköltségei minden egyes legyártott darabban megjelennek („minden cent számít” – szoktuk mondani az ipari fejlesztések többségében).

A **fogyasztás** (4) sok alkalmazásnál kritikus szempont, ma már azonban szinte minden fejlesztésben szóba kerül. Eladhatatlan lenne az az akkumulátoros üzemű berendezés (mobiltelefon, PDA stb.), amelyik csak néhány óráig lenne üzemeltethető két töltési művelet között. De a hatásfok részben a dráguló energiaárak, a globális energiafelhasználás csökkentése és a miniatürizálásból fakadó disszipációs problémák miatt is egyre inkább szempont a tervezés során (még akár kisebb felár árán is).

Kritikus lehet az **eszköz rendelkezésre állása** (5 és 6), a rendkívül gyorsan változó elektronikai piacon nagyon nehéz teljesíteni a vevőnek azt az elvárását, hogy pl. az autójában található elektronikus egység még egy 10 éves autóban is cserélhető legyen, mikor az elektronikában található mikroprocesszorok 2-3 éves élettartammal rendelkeznek. A nagy cégek sokszor másod-, harmadgyártót keresnek az elektronikus eszközökhöz termékeik biztosítására – ellenkező esetben marad a „bevásárlás" pl. 5 évre előre, ami egy Bosch nagyságú cég esetében mind finanszírozási, mind raktározási problémákat is felvet.

Nagyon komoly fejlesztési költségek takaríthatók meg a **szoftverkomponensek** rendelkezésre állásával (7) (kész beágyazott operációs rendszer az adott típusra, aritmetikai és egyéb funkcionális könyvtárak létezése, stb.) és a teljes- vagy részfeladatok korábbi megoldásainak hordozhatóságával.

A választásnál nem elhanyagolható szempont az eszközcsaláddal, **gyártóval kapcsolatos korábbi tapasztalat- és kapcsolatrendszer** (8). Bár említettük már, hogy a fejlesztési költségek a gyártmány árában esetleg csak töredékárban jelennek meg, ösztönös és nem is biztos, hogy elítélendő reakció minden fejlesztőtől az „ismert" irányába való elhúzódás. Jó fejlesztők azonban nem ragaszkodnak kizárólagosan egyetlen ismerethalmazhoz – nyitott szemmel figyelve a szakirodalmat és a körülöttünk zajló folyamatokat tudni kell kellő időben váltani is.

Eléggé a sor végére került a **fejlesztőeszközök** ára (9). Egy típus kiválasztásakor azonnal jusson eszünkbe, hogy ahhoz új fejlesztői környezet kellhet (fordítóprogramok, könyvtárak, emulátor vagy debuggolást támogató eszköz) – ezek ára sokszor több millió forintos beruházást jelent, ami a kisebb darabszámú terméket végeredményben nagyon megdrágíthatja.

Gyártási szempontokként (10) említhetők pl. az eszköz hőmérséklettűrése, tokozása (pl. BGA tokozású eszközöket nem minden cég tud sorozatban és jó minőségben beültetni), stb.

Talán a legmegfoghatatlanabb mindenki számára az elmondottakból az első pontok vizsgálata és teljesítése. Nézzünk három rövid példát a probléma jobb megértéséhez!

Vegyünk három egyszerű feladattípust, amit elképzelhető, hogy egy-egy beágyazott rendszerrel kívánunk elvégezni!

a) Egy egyszerű kis intelligens folyamatmodul (intelligens szenzor) digitális bemeneteket fogad az irányított folyamat (pl. autó gyártósor) egy helyszínén, azokat pergésmentesíti, megformálja az impulzustulajdonságokat (pl. minimális hossz, minimális távolság közöttük, stb.), majd az állapotokat ciklikusan továbbítja egy terepi buszon (CAN, Profibusz, stb.) az irányító PLC felé.
b) Egy mobil robot hajtómotorjainak szabályozási feladatát ellátó intelligens hajtásprocesszor megvalósítása

c) Az előbbi robotunk mozgásának pályatervezése valós időben. Ezzel majd a mesterképzés keretei között foglalkozunk részletesen, bonyolult mátrixegyenletek ciklikus megoldását jelenti a valós időben.

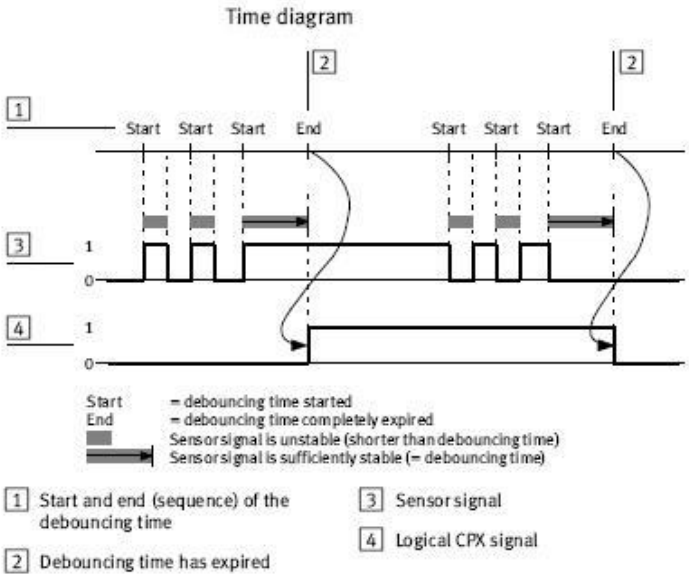
A feladathoz vessünk össze 3 mikrokontroller típust, melyek mindegyike rendelkezésünkre áll a feladat megoldásához:

1) A 70-es években az egész világot uraló Z80-as mikroprocesszor felgyorsított, „mikrokontrolleresített" változatát mind a mai napig gyártják (Zilog, Hitachi, stb.) Konzervatív fejlesztőnk a múltbeli ismeretekhez való ragaszkodása miatt mérlegeli ennek a típusnak az alkalmazását.
2) A 8 bites mikrokontrollerek világában – hasonlóan a PC-k 80x86-os világához – az Intel cég 8051-es kontrollereinek mai változatai még mindig nagy számban találhatók a piacon (lassan 40 éve). Gyártók sora kínálja ennek a típusnak a különböző klónjait (Atmel, Philips, Infineon, Silicon Laboratories, stb.)
3) Manapság rendkívül népszerűek az ARM család Cortex M4-es mikrokontrollerei, melyek már a nagyobb teljesítőképességű processzáló elemek közé tartoznak és rendkívül elterjedtek. A későbbiekben részletesen foglalkozunk majd velük.

Összeállítottunk egy táblázatot a fontosabb szóba jövő műveletek végrehajtási időszükségletéről. A logikai műveletek mindhárom mikrokontroller esetében 1-1 utasítással elvégezhetők, ezek időszükséglete megegyezik az adott kontroller típus utasítás végrehajtási idejével. A 16 és 32 bites műveletek a 8 bites mikrokontrollerek számára már mindenképpen több utasítás végrehajtását jelentik, hasonlóan a lebegőpontos műveletekhez, melyek csak a Cortex processzor számára végezhetők el egyetlen utasítással (a legkisebb teljesítményű mikrokontroller esetében itt már nem is adunk meg számokat, a végrehajtási idők ugyanis irreálisan magasak lennének a többi kontroller adataihoz képest).

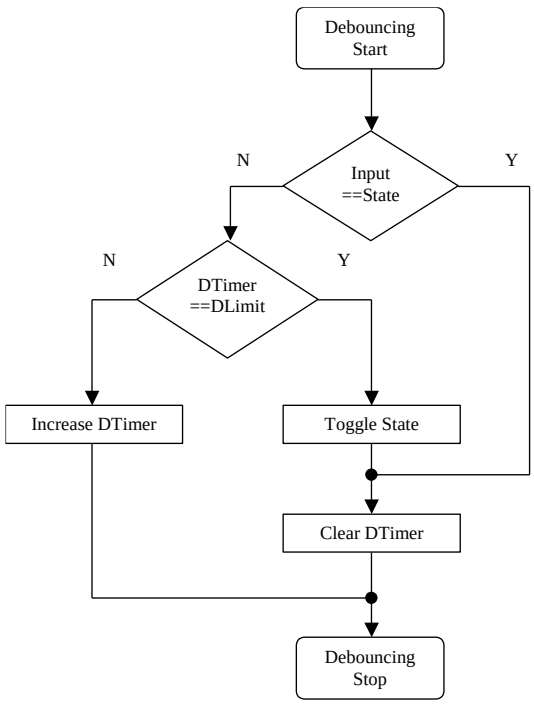
Művelet	**20 MHz Z180 8 bit átl. 6-8 T/utasítás HW szorzó**	**25 MHz EFM8BB3 8 bit átl. 1-2 T/utasítás HW szorzó, osztó**	**180 MHz STM32F446RE 32 bit 1 T/utasítás HW szorzó, osztó, FPU**
Logikai ÉS Logikai VAGY 8 + 8 = 8 bit	0.3 μsec	0.04 μsec	0.0056 μsec
16 + 16 = 16 bit 32 + 32 = 32 bit	2 μsec 8 μsec	0.5 μsec 1.0 μsec	- 0.0056 μsec
16 * 16 = 16 bit 32 * 32 = 32 bit	12 μsec 36 μsec	2.4 μsec 5.0 μsec	- 0.0056 μsec
16 / 16 = 16 bit 32 / 32 = 32 bit	80 μsec 160 μsec	10.0 μsec 22.0 μsec	- 0.0667 μsec
FADD, FSUB	-	10 μsec	0.0056 μsec
FMUL	-	12 μsec	0.0056 μsec
FDIV	-	49 μsec	0.0778 μsec

a) Digitális jel feldolgozása alapműveletekkel



A digitális jel pergésmentesítésének értelmezése

A jelfeldolgozás nem igényel komolyabb aritmetikai műveletsort. Egy későbbi esettanulmányunkban részletesen foglalkozunk majd a megoldással, egyelőre indoklás nélkül a megoldás:



A digitális jel pergésmentesítésének algoritmusa

Az algoritmus elvégzése a lefordított program elemzése alapján kb. 15-20 utasítás végrehajtását jelenti a mikrokontrollerek számára (aritmetikai és logikai műveletek, időzítők tárolása). Számoljunk 15 utasítással, viszont 32 db ilyen bemenet kezelését kell elvégeznünk, az elvárt ciklusidő max. 100 μsec. A feladat ciklikus végrehajtása mellett a terepi busszal való kommunikáció (céláramkör!) időigénye elhanyagolható.

Feladat	Z180	EFM8BB3	STM32F446RE
1 bemenet kezelése	4.5 μsec	0.6 μsec	0.083 μsec
32 bemenet kezelése	144 μsec	19.2 μsec	2.67 μsec
Terhelés	**144%**	**19.2%**	**2.67%**

b) Hajtásszabályozás

Lépjünk egy szinttel magasabbra, és tételezzük fel, hogy már tudjuk, az adott motort milyen sebességre kívánjuk szabályozni (vagy hová kívánjuk pozícionálni). A feladat csupán ezen feladat elvégzése. Vegyünk egy viszonylag egyszerű PID irányítási algoritmust:

$$Y = K_P \cdot Q_r + K_i \cdot Q_r + Y_{ie} + K_d \left(Q_r - Q_{re} \right)$$

A képletben K_P az arányos, K_i az integráló, K_d a differenciáló tag együtthatója, Q_r a rendelkező (vagy hiba-) jel, Y_{ie} és Q_{re} a szabályozó kimenő jelének és a rendelkező jelnek az előző értékei.

Ezt a feladatot egy korszerű hajtás esetében hozzávetőlegesen 100-500 μsec-ként kell elvégeznünk. Egyelőre indoklás nélkül fogadjuk el, hogy a feladat 32 bites egész (fixpontos) aritmetikával elvégezhető. Mikrokontrollereink számára ez hozzávetőlegesen a következő időigénnyel jár:

Feladat	Z180	EFM8BB3	STM32F446RE
$Q_r = Q_a - Q_e$	8 μsec	1.0 μsec	0.0056 μsec
$K_p * Q_r$	36 μsec	5.0 μsec	0.0056 μsec
$K_i * Q_r$	36 μsec	5.0 μsec	0.0056 μsec
$Q_r - Q_{re}$	8 μsec	1.0 μsec	0.0056 μsec
$K_d*(Q_r - Q_{re})$	36 μsec	5.0 μsec	0.0056 μsec
Összeadás (3 x)	24 μsec	3.0 μsec	0.0167 μsec
Összesen	148 μsec	20.0 μsec	0.0444 μsec
Terhelés	**148%**	**20.0%**	**0.044%**

c) A robotirányítás pályatervezése

Általánosságban a robot pályatervezése mátrixegyenletek megoldását jelenti, mely mátrixok tipikusan 4x4-esek és trigonometrikus függvényeket (is) tartalmaznak (ezen kijelentések indoklása csak mélyebb robotirányítási elmélet ismeretekkel lehetséges).

Próbaképpen szorozzunk össze 2 db 4x4-es mátrixot, amiben nincs kihasznált szabályosság és egyelőre hanyagoljuk el a trigonometrikus függvényeket. Z180-as mikrokontrollerünk esetében még 32 bites egész (fixpontos) aritmetikával kísérletezünk (bár az alkalmazhatóság erősen kilátástalan), másik két processzorunk lebegőpontos számításokat végezzen. A műveletből legalább 10-et kellene elvégeznünk kb. 10 msec alatt.

$$\begin{bmatrix} a_{11} & \cdots & a_{14} \\ \vdots & \ddots & \vdots \\ a_{41} & \cdots & a_{44} \end{bmatrix} * \begin{bmatrix} b_{11} & \cdots & b_{14} \\ \vdots & \ddots & \vdots \\ b_{41} & \cdots & b_{44} \end{bmatrix} = \begin{bmatrix} c_{11} & \cdots & c_{14} \\ \vdots & \ddots & \vdots \\ c_{41} & \cdots & c_{44} \end{bmatrix}$$

*Időszükséglet = 10 x [(4 db * ; 3 db +) * 16]*

Feladat	Z180	EFM8BB3	STM32F446RE
4 db szorzás	144 µsec	48 µsec	0.022 µsec
3 db összeadás	24 µsec	30 µsec	0.017 µsec
1 mátrixszorzás	2688 µsec	1248 µsec	0.622 µsec
Összesen (10x)	26800 µsec	12500 µsec	6.222 µsec
Terhelés	**268%**	**125%**	**0.062%**

Milyen következtetést tudunk leszűrni a fenti (hangsúlyozzuk, erősen közelítő!) számításokból?

a) **Z180-asunk elavult**. Legnagyobb gondja a nem pipeline jellegű műveletvégzés, ettől megközelítőleg azonos órajel mellett is stabilan 4-5-ször lassabb, mint 8051-esünk. Ehhez még hozzájárul, hogy elavult architektúrája miatt folytonos mentésekre kényszerül a részeredményeknél (kevés regiszter). A feladatot a **8051-es kiválóan el tudja látni**, mintegy 5-10-szer drágább **ARM processzorunk teljesen felesleges** a feladat számára – a neki abszolút nem passzoló egyszerű műveletek miatt alig gyorsabb kis mikrokontrollerünknél (kb. 180/25 ≈ hétszeres utasítás végrehajtási sebesség miatt kb. 7x is gyorsabb).
b) Ennél a feladatnál már 8051-es mikrokontrollerünk is kezd „leterhelődni”, korábbi 8:1 terhelésaránya a Z180-hoz képest lecsökken 6:1-re (teljesen normális, ez kb. 4-5-szeresre állna be végtelen terhelés esetén az utasítás végrehajtási idők és az órajelek eltérése miatt). A feladat a 8 bites mikrokontrollerek számára is elvégezhető, Cortex M4-esünk „üresen jár” ezzel a terheléssel.
c) Itt már 8 bites mikrokontrollereink „elvéreztek”. Az eltérés extrém: **miből adódik ez a 12500/6.222 = 2009-szeres faktor?** Az órajelek különbözőségéből? Az utasítás végrehajtási sebességek kb. 180/25 ≈ 7-szeres eltérést indokolna. A beépített lebegőpontos aritmetikából? Valamennyi lehet ebben, hiszen a lebegőpontos műveleteket összehasonlítva 10/0.0056=1786-szoros szorzóhoz jutunk. Hardver szorzónk ugyan sokat segít a helyzeten, de ha mélyebben belegondolunk, rá kell jöjjünk: **az igazi ok valójában a szóhossz.** 8 bites mikrokontrollereink csak 4 lépésben

tudják elvégezni a 32 bites feladatot, ami egyszerűbb műveletek esetében még azonos órajelnél is "csak" kb. 10-12-szeres időt jelentene (4-szer a művelet elvégzése és mindegyik művelethez a megfelelő regiszterek töltögetése/tárolása). A szorzásnál azonban a részletszorzatok miatt ez már hatványozottan jelenik meg (egész számoknál: 5.0/0.0056=893, lebegőpontos számok esetében pedig 12/0.0056=2143).

Trigonometrikus egyenletek esetén ne is próbálkozzunk: a fixpontos számítás szinte lehetetlen, a függvények számításához az aritmetikai processzor megléte elengedhetetlen. Ide bizonyítás nélkül csak egy hozzávetőleges szám: egy korszerű hibrid pozíció-erő robotirányítás számítási igényeinek elvárásainál egy asztali PC kb. 2...3-szor lassabb.

1.3 A RISC architektúra

A számítógépek kialakulásának kezdeti időszakában – kb. a 60-as évek elejéig – az alkalmazott központi egységek igen egyszerű és kevés számú utasításból álló utasításkészletet és kevés címzési módot használtak. Ennek oka leginkább a technikai/technológiai korlátokban keresendő. Ez idő tájt leginkább műveletvégzési sebességük káprázтatja el a szakembereket és a tudósokat: az 1946-ban megszületett ENIAC másodpercenként 5000 összeadás elvégzésére volt képes, ami fantasztikus teljesítménynek tűnt az ember által elvégzett műveletvégzési sebességhez képest.

A számítógépek fejlődése napjainkig öt generációra osztható, ahol az új generációk mindig valamilyen jelentős technológiai fejlesztést jeleznek.

Az **1. generációba (~1945–1956)** tartozó első elektronikus számítógépek (pl. ENIAC, IBM 701, Princeton IAS) áramkörei elektroncsöveken alapultak, a memóriát relékkel valósították meg, az alkatrészeket szigetelt vezetékekkel kötötték össze, és a programozás közvetlen gépi nyelven történt. A programot lyukkártyákon vagy lyukszalagon lehetett beadni, a kimenetet is ilyenekre nyomtatva szolgáltatták. Méretük és fogyasztásuk hatalmas volt, és a nagy disszipáció miatt rendszeresen meghibásodtak. Fixpontos műveletvégzésre voltak képesek, és utasításkészletükben ismert volt már az ugró utasítások és az akkumulátor (regiszter) fogalma is.

A **2. generáció (~1956–1964)** kezdetét a tranzisztor alapú számítógépek megjelenése jelentette. A tranzisztor megalkotása (1947) után közel egy évtizeddel a diszkrét félvezető elemek megjelentek a számítógépekben is. Az összeköttetéseket nyomtatott áramkörök biztosították, így sokkal kisebb méret és fogyasztás volt elérhető. A relék helyett megjelentek a ferritgyűrűs memóriák. A második generációs gépek már alkalmasak voltak lebegőpontos műveletvégzésre, a gépi kód helyett már assembly nyelvet használtak a programok írására, és ekkortájt kezdődött el a magas szintű nyelvek (Fortran, Algol, Cobol) és fordítóprogramok alkalmazása is. A be- és kimeneteket még itt is lyukkártyák szolgáltatták. Az első tranzisztor alapú számítógép az MIT-n fejlesztett TX-0 volt 1956-ban. További reprezentatív második generációs rendszerek az IBM 7090, az Univac LARC és a CDC 1604.

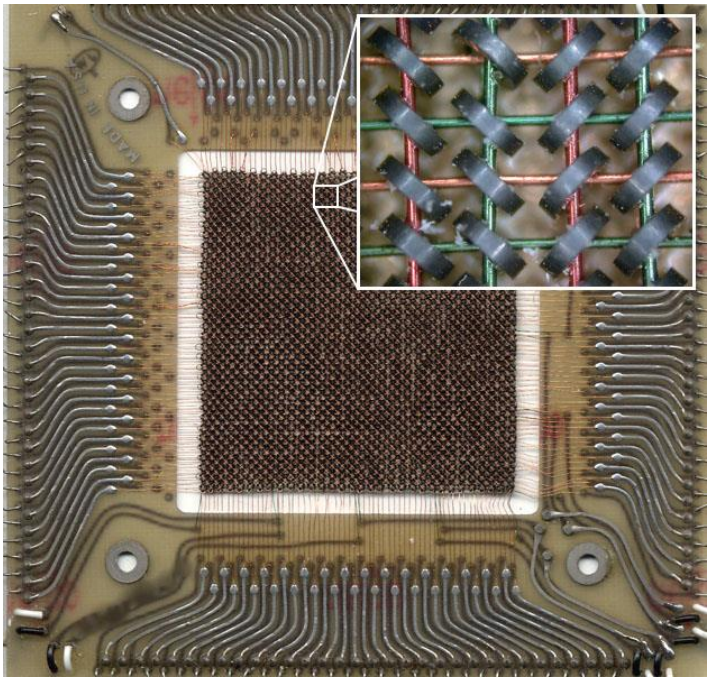
A **3. generáció (~1964–1971)** már integrált áramköröket használt logikai és memória funkciók megvalósítására, kis és közepes integráltsági fokú gyártástechnológiával (SSI – small-scale integration, MSI – medium-scale integration, több tíz vagy száz tranzisztor egy lapkán). Az alkatrészeket többrétegű nyomtatott áramkörökön helyezték el. Ekkor vált elterjedtté a mikroprogramozás (ld. később) a vezérlőművek tervezésében, és ekkor kezdték bevezetni a csővezetékes végrehajtás (pipelining) és a gyorsítótárak alkalmazását. Ennél a generációnál jelent meg a billentyűzet és monitor, mint felhasználói felület, valamint az operációs rendszerek és a virtuális memória használata. A generáció híres képviselői az 1964-ben megjelent IBM System/360-370 sorozatú, valamint a CDC 6600/7600 sorozatú és a DEC PDP-8 sorozatú mainframe számítógépek.

A **4. generációt (~1971–2000)** a mikroprocesszorok megjelenésétől számítjuk. Az Intel 1971-ben mutatta be a 4004-es mikroprocesszort, amely egy integrált áramköri lapkán tartalmazta az összes lényegi processzorfunkciót (erről később bővebben is olvashatunk). A félvezető (MOS) alapú memóriák az 1970-es évek végére kiszorították a ferritgyűrűs memóriákat, ami az integrált áramköri gyártástechnológia fejlődésével együtt elindította a számítógépek azóta is megfigyelhető hihetetlen mértékű miniatürizálását és teljesítménynövekedését. A nagyfokú miniatürizálásnak köszönhetően ebben a korszakban jelentek meg az otthoni felhasználásra szánt asztali és hordozható számítógépek és a grafikus felhasználói felület.

Az **5. generációba (~2000-től napjainkig)** a mesterséges intelligencia szélesebb körben való használatát lehetővé tévő architektúrák megjelenése óta beszélhetünk. A mély tanuló algoritmusok (deep learning) és más számításigényes MI alkalmazások a 2010-es évektől váltak népszerűvé, elég, ha csak a televíziós vetélkedőben is sikerrel szereplő IBM Watson-ra, az emberi beszéddel vezérelhető virtuális asszisztensekre (Apple Siri, Microsoft Cortana, Amazon Alexa stb.) és az önvezető járművek fejlesztésére gondolunk.

Az egymást követő generációk során a számítógépek működésének alapelve nem változott, de a belső architektúrájuk, illetve a funkcionális elemek technológiai megvalósítása hihetetlen fejlődésen ment keresztül néhány évtized alatt. Érdemes kiemelni a számítógépek fejlődésének két, a további tárgyalásunk szempontjából fontos részfolyamatát: a főmemória-egységek és a processzorok vezérlőművének különböző megvalósításait.

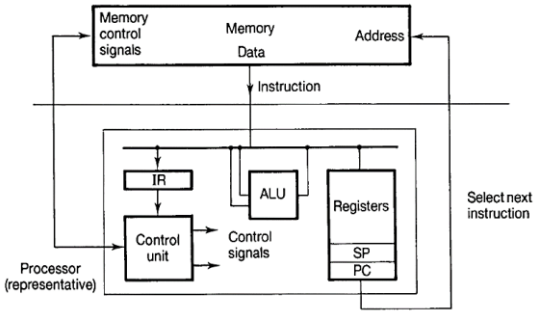
Ahogy a generációk áttekintésénél is láttuk, a számítógépek főmemóriáit az 1960-as évek végéig (2. és 3. generáció) elsősorban ferritgyűrűs tárakkal lehetett megvalósítani. Ezek a tárak az információ tárolását mágnesezhető kerámiagyűrűk mágneses polaritásának felhasználásával végezték. A gyűrűket rácsszerűen elhelyezett áramjárta vezetőkre fűzték fel úgy, hogy a vízszintes és függőleges vezetőkre az átmágnesezéshez szükséges áramerősség felét lehetett kapcsolni. Így egy sor és egy oszlop kiválasztása esetén csak a metszéspontban lévő gyűrű mágneses polaritása változott. Ezeknek a memóriaegységeknek a gyártása bonyolult és emiatt drága volt (nem lehetett jól automatizálni, aprólékos kézi munkával ruhaipari dolgozók végezték a gyűrűk fűzését), és a gyűrűk fizikai mérete korlátozta az elérhető adatsűrűséget (nem igazán lehetett 1 bit/mm^2 értéknél sűrűbbet elérni).



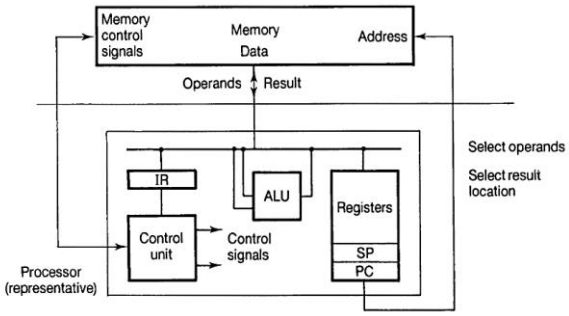
A CDC 6600-as számítógép ferritgyűrűs memóriamodulja
(méret: 10,8×10,8 cm, kapacitás: 64×64 bit = 4 kbit)

Az 1960-as évek elejétől kezdtek megjelenni a félvezető alapú memóriák. Először diszkrét bipoláris tranzisztorokból építettek fel memóriamodulokat (Texas Instruments, 1961), de néhány évvel később már integrált áramköri formában is állítottak elő memória-chip-eket (Fairchild, 1965). Bár ezeknek a memóriáknak a sebessége jelentősen meghaladta a ferritgyűrűs tárakét, a bipoláris technológia ebben az időben nem tudott még árban versenyezni a mágneses megoldásokkal. Ekkoriban jelentek meg az első MOS technológiára épülő félvezető memóriák (Fairchild, 1964). A MOS tranzisztoroknak a bipolárishoz képest jelentősen kisebb áramfelvétele, illetve olcsóbb gyártástechnológiája előrevetítette a MOS alapú integrált áramkörök megjelenését és elterjedését. A MOS IC technológia kifejlesztése Federico Faggin, a Fairchild mérnöke nevéhez fűződik, az első ilyen memóriachip-et 1968-ban jelentette meg a Fairchild (Faggin nevével később még találkozni fogunk). A MOS alapú memóriák az integrált áramköri gyártástechnológiának köszönhetően árban, sebességben és természetesen adatsűrűség tekintetében is hamar maguk mögé utasították a ferritgyűrűs memóriákat, és az 1970-es évek első felétől kezdve egyre inkább kiszorították azokat a piacról.

A számítógépek processzor egységének alapvető feladata a programutasítások végrehajtása. Ennek egyszerűsített folyamatát mutatja be a következő ábra. A processzor funkcionális elemei közül kiemelt szerepe van a vezérlőegységnek (control unit), az aritmetikai-logikai egységnek (ALU) és a regisztereknek, amelyek között megtalálhatjuk a speciális szereppel rendelkező programszámlálót (program counter, PC), veremmutatót (stack pointer, SP) és utasításregisztert (instruction register, IR). A processzor mellett található a főmemória, amely egyszerűsített modellünkben mind az utasítások, mind az adatok tárolására szolgál (Neumann-elvű memóriaszervezés).



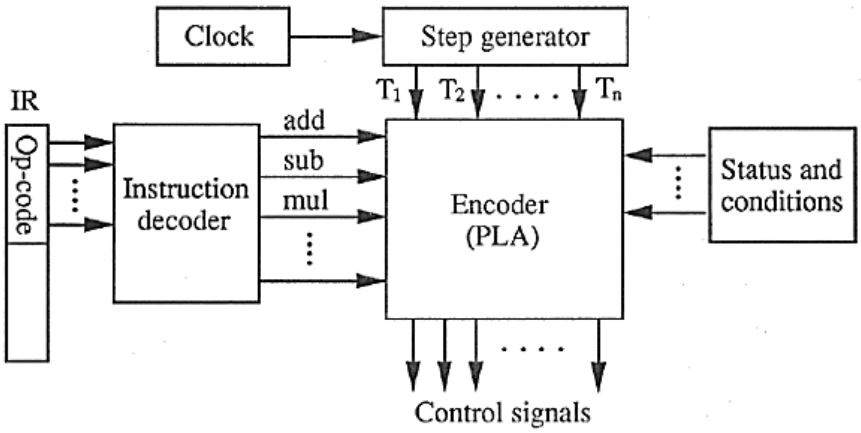
Utasítás felhozatala



Utasítás végrehajtása

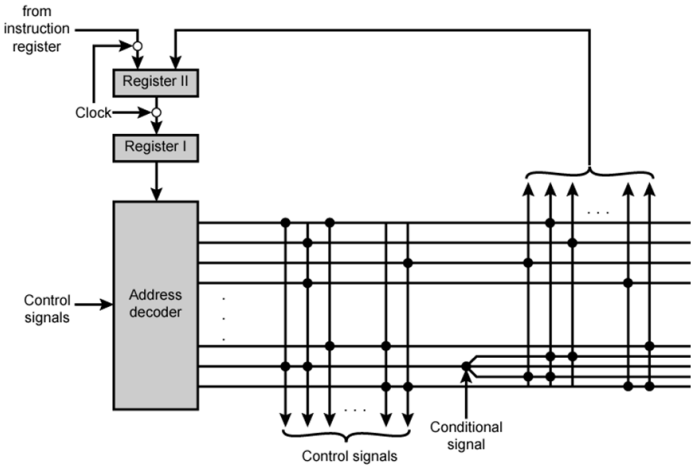
Az utasítások felhozatalát és végrehajtását a vezérlőegység bonyolítja le, ennek a feladata a megfelelő vezérlőjelek előállítása a különböző funkcionális részegységek számára. Az utasításfelhozatal során a PC tartalmával címezi a memóriát, és a felhozott utasításszó a processzor buszrendszerén keresztül az utasításregiszterbe kerül. Ezt követően az IR-ben lévő utasításkódot a vezérlőegység dekódolja, és előállítja a megfelelő vezérlőjeleket az ALU számára a megfelelő művelet elvégzéséhez, illetve a regisztertár és/vagy a memória számára az operandusok felhozatalához, majd később az eredmény tárolásához. Emellett frissíti a PC tartalmát az utasítástól függően (egyszerű inkrementálás vagy más cím beállítása ugró utasítás esetén), és a folyamat indul elölről a következő utasítás felhozatalával.

A számítógép vezérlőegysége tehát egy nagybonyolultságú, komplex logikai áramkör. A CPU nagyszámú vezérlési (kapcsoló-) ponttal rendelkezik, ezeket kell vezérelni a végrehajtandó utasítás által elvégzendő művelet, operandus, címzési mód, stb. szerint. Ennek eredményeként az ún. huzalozott (erre a célra tervezett) vezérlőegység struktúrája mindig kevéssé áttekinthető. Ezért a huzalozott vezérlőegység megtervezése sok munkát igényel, így drága, melyben a hibakeresés is rendkívül költséges. A vezérlőjel szekvenciák pedig különféle utasítások esetében is sok hasonlóságot mutathatnak egymáshoz.



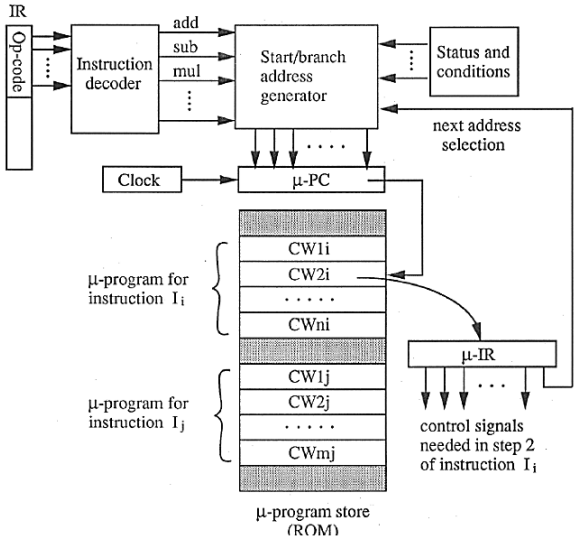
Huzalozott végrehajtómű

1950 körül Maurice V. Wilkes (Cambridge University) számítógép-fejlesztő felvetette egy rugalmas és szisztematikusan felépített vezérlőáramkör tervezési mód kialakításának szükségességét.



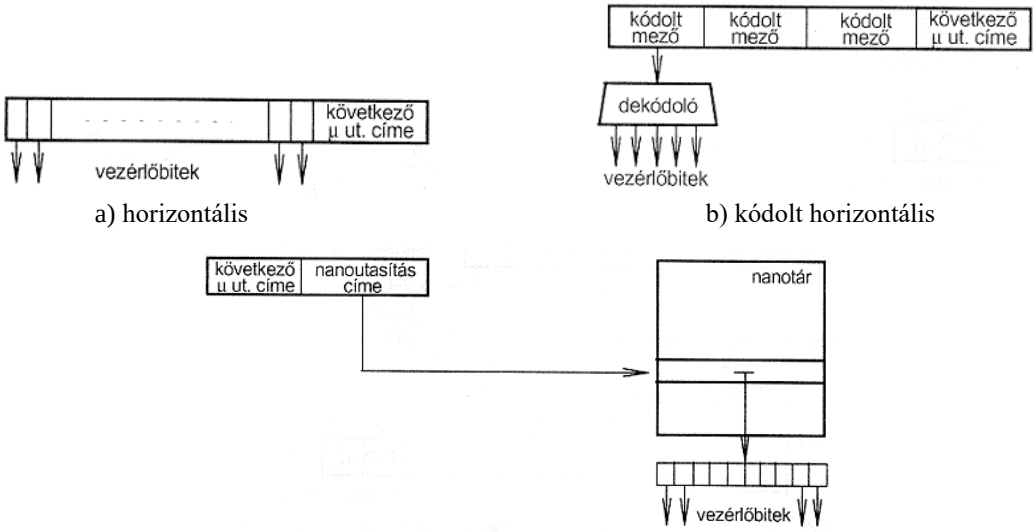
A Wilkes-féle vezérlési modell

Ennek a lényege abban állt, hogy Wilkes egy általános szekvencia-vezérlőt tervezett, amivel minden gépi utasítás végrehajtását mikroutasítások sorozatának végrehajtására vezette vissza. A mikroutasításokat (Control Word – CW) a Control Memory-nak nevezett címezhető memóriában tárolta. A Control Memory-ból lehívásra kerülő mikroutasítások tartalmát részben a gépi utasítást végrehajtó mikroutasítás-szekvencia leírására használjuk (mely mikroutasításokat kell egymás után végrehajtani a gépi utasítás elvégzéséhez), részben a vezérlővonalakat aktíváljuk segítségükkel (a vezérlőpontokat kapcsoljuk). A Control Memory melletti vezérlőegységet mikroprogram-vezérlőegységnek nevezzük.



Mikroprogramozott végrehajtómű

A mikroprogramozás a vezérlőegység tervezést egyszerűbbé, áttekinthetőbbé tette, mivel a kapcsolópontok vezérlését jól meghatározott formátummal rendelkező szavakba (mikroutasításokba) szervezte. Mivel az utasításkészletet ezen vezérlőszavak segítségével írjuk le, a fejlesztés változtatásai és könnyedén megvalósíthatók a Control Memory tartalmának cseréjével. A módszer elterjedt, 1964-ben, az IBM cég 360-as gépcsaládjának bevezetésével jelent meg a számítógépekben a mikroprogramozott műveletvezérlés az utasítások végrehajtására.



a) horizontális

b) kódolt horizontális

c) horizontális-vertikális

Mikroutasítások felépítése

Egy normál gépi utasítás tehát mikroutasítások sorozatának végrehajtását jelenti. A mikroutasítások a Wilkes által Control Memory-nak nevezett mikroprogram tárban helyezkednek el, ennek utasításait értelmezi és hajtja végre közvetlenül a hardver. Az eredeti elv szerinti ún. horizontális mikroutasítás rendkívül hosszú, mivel adott esetben több száz kapcsolópontot kell irányítania a vezérlőműben. Ezért a „többnyire hasonlóan” vezérelt pontokat összefoghatjuk egy-egy kódolt mezőbe, és a pontok néhány vezérlési kombinációját egy-egy rövidebb kódszóban tároljuk. A dolog még tovább is bonyolítható: a nagyméretű mikroprogramtár elkerülésére a sokszor ismétlődő részek a későbbiekben már egy nanoprogramtárba kerültek, és innen mintegy „rutinonként” hajtódtak végre. Kívülről nézve a processzor továbbra is úgy viselkedik, mintha az eredeti gépi utasítást közvetlenül egy lépésben a hardver hajtaná végre.

Az IBM S/360 számítógép

A mikroprogramozás nagymértékben megnöveli a gépek rugalmasságát. A mikroprogramtár kicserélésével egész egyszerűen megváltozik a gép utasításkészlete. Így az is megvalósítható, hogy egy másik gép gépi kódját közvetlenül végrehajtsa, emulálja a másik gép működését. A mikroprogramozás – mint lehetőség – következtében növekedett az utasítások bonyolultsága, azaz egyre több funkciót bíztak egyetlen utasításra, emellett a használható utasítások száma is növekedett (több, mint 200-250 utasítás).

A mikroprogramozott gépekben az utasítások bonyolultsága azért is növekedett különösen, mert a belső ROM tárolóban elhelyezett mikroprogram végrehajtása gyorsabb volt, mint az akkoriban elterjedten használt ferritgyűrűs központi tárak által biztosított sebesség. Célszerű volt tehát csökkenteni a tárhoz fordulások számát, inkább legyen egy-egy utasítás minél komplexebb. Egy másik tényező, amely a bonyolult gépi utasítások kialakulása felé vezetett, a magas szintű programozási nyelvek egyre szélesebb körű elterjedése volt. Az ezekben használt összetett utasítások, tömb- és táblakezelési módszerek megvalósításához meg kellett alkotni ezek gépi szintű megfelelőjét – nos ezeket összetett mikroprogramok, mikroeljárások segítségével tűnt legegyszerűbbnek megvalósítani. Ezek támogatására különböző, bonyolult címzési módokat is kidolgoztak. Ekkor alakultak ki az *összetett utasításkészletű* **CISC számítógépek** (Complex Instruction Set Computer). A folyamat teljes mértékben visszatükröződött az egyre jobban fejlődő mikroprocesszorok világában is: a 70-es évek közepén/végén elinduló két nagy rivális, az Intel 8086-os és a Motorola 68000-es család mikroprocesszorai is egyaránt mikroprogramozott végrehajtóművet tartalmaztak.

A 70-es évek technológiai fejlődésének eredményeként megjelenő, gyors működésű félvezetős RAM memóriák és az egyre inkább elterjedő cache-tárak már nem tették szükségessé a központi tárhoz fordulások erőteljes csökkentését, sőt, az egyre bonyolultabb mikroprogramok lelassították a feldolgozás gyorsaságát. Emellett egyre nehezebb lett a bonyolult mikroprogramok hibátlan előállítása, tesztelése. Ezek a tények vezettek el az egyszerűsítés szükségességének felismerése felé. Megszületett a **RISC számítógépek** (Reduced Instruction Set Computer) alapgondolata, amely – mint látni fogjuk – a korábbi számítógépek és mikroprocesszorok működési tapasztalatai alapján leszűrhető elemzések eredménye.

Mint említettük, a kutatási folyamat a 70-es években indul el. Az első alapvető kérdés:

Mit tekintünk a teljesítőképesség mérőszámának?

Nem szabad szem elől tévesztenünk, hogy a számítógépekkel bizonyos feladatokat kívánunk megoldani, a lehetséges mérték a feladat megoldásához szükséges idő. Ennek összetevői a következők:

$$\frac{\text{Időszükséglet}}{\text{Feladat}} = \frac{\text{Utasításszám}}{\text{Feladat}} * \frac{\text{Órajelperiódus}}{\text{Utasítás}} * \frac{\text{Periódusidő}}{\text{Órajel}}$$

A jobboldal 3 tényezőjének jelentése egyszerűen a következő (balról jobbra haladva):

1. Egy adott feladat leírásához hány CPU-utasítás szükséges (utasítások komplexitása)
2. Átlagosan hány órajel periódus alatt hajtható végre egy-egy CPU-utasítás (utasítások végrehajtási ideje)
3. A CPU órajel-frekvenciája

A felhasználó számára a lényeges szempont a teljes szorzat értékének minimalizálása, a végeredmény szempontjából közömbös, hogy az egyes tényezők milyen egyedi minimummal rendelkeznek. Nincs tehát önmagában perdöntő jelentősége az oly gyakran hangoztatott CPU órajel-frekvenciának (3. tényező), a MIPS-értékeknek (a 2. és a 3. tényező szorzatának reciproka), vagy az utasítások komplexitásának (1. tényező).

A 70-es évek végétől a számítógép-gyártók az általános célú számítógépek kialakításánál alapvetően az ún. család-architektúrát követték (IBM, DEC, Intel, Motorola). Ennek előnyeit és hátrányait pl. az Intel 8086-os, vagy a Motorola 68000-es családjánál is figyelemmel kísérhetjük. Alapvető célkitűzés volt, hogy egy-egy család olyan családtagokat kínáljon, ahol a tagok szoftver kód-kompatibilisek, tehát a család valamely kisebb teljesítőképességű tagján fejlesztett program a nagyobb családtagokra egy az egyben átvihető és újrafordítás nélkül futtatható legyen.

A RISC architektúra kialakulásának folyamatában nem pusztán arról van szó, hogy a fejlesztők „megszűrték”, „leredukálták” az eddigi számítógépek és mikroprocesszorok utasításkészletét, hanem ezt a folyamatot sokkal inkább úgy lehetne jellemezni, hogy a

- számítógép/mikroprocesszor tervezők és memóriagyártók
- programtervezők, rendszerprogramozók
- számítógép-kutatók

"egymásra találtak", tehát egyeztették a végeredmény szempontjából mindazt, ami egy nagy teljesítőképességű rendszer (és nem számítógép!) létrehozásához szükséges. A RISC irányzat alapvető célkitűzése háttérbe szorítja a „szuper CPU/hardver” és a „szuper szoftver rendszer” fogalmait, és a teljes rendszert igyekszik optimalizálni, a szükségtől függően hardver vagy szoftver kompromisszumok árán. A legnagyobb számítógép/mikroprocesszor gyártók azonnal reagáltak is erre a felhívásra, így ma a piacon már viszonylag kiforrott RISC architektúrájú számítógépek/mikroprocesszorok találhatók. (Az már egy egészen más kérdés, hogy kifejezés egy előnyt jelentő divatszó lett, és sokszor olyan kontrollerek esetében is alkalmazzák, ahol a most összefoglalásra kerülő elvek fele sem teljesül. Mindig fenntartásokkal fogadjuk ezeket az állításokat.)

A hagyományos CISC-architektúrákon végzett vizsgálatok azt mutatták, hogy a leggyakrabban használatos fordítóprogramok (compilerek) által előállított kódok a processzorok utasításkészletének és címzési módjainak egy igen kis részét használják csak nagy számban. Felvetődik a kérdés: mi értelme ekkor a mikroprocesszor-gyártók technológiai erőlködésének, akik a CPU-k teljesítőképességét alapvetően 3 módon igyekeznek növelni:

- az órajel-frekvencia növelésével (ennek technológiai korlátai vannak)
- egyre több egységnek a CPU-ba történő integrálásával (pl. cache, MMU, FPU, ennek is technológiai korlátai vannak)
- egyre komplexebb utasítások és címzési módok kialakításával (amit a szoftvertervezők egyre kevésbé tudnak kihasználni, ilyen módon „el sem jutnak" a felhasználókhoz).

A hagyományos CPU-k kb. ez idő tájt 5-600 000 tranzisztorfunkciójából egyes kutatók szerint a felhasználások alapján mintegy 50-60 000-re lenne szükség az alapján, amit a szoftverek ki is használnak a valóságban. Nem lehetne tehát az itt felszabaduló kapacitást együttműködve a szoftverrel valóban a teljesítmény növelésére felhasználni?

(Érdekességként: az Intel 2012-es i7-es 4 magos Ivy-Bridge névre keresztelt processzora 160 mm^2-es (kb. 13x13mm-es) lapkán készül 22 nm-es Tri-Gate technológiával készült, és hozzávetőlegesen 1,48 milliárd tranzisztort tartalmaz.)

A következőkben pontról-pontra haladunk végig a RISC architektúra tulajdonságain, illetve azon problémákon, melyek ezekkel kapcsolatban felvetődnek.

Intel i7 Ivy Bridge

1.3.1 Az utasításkészlet tulajdonságai

A RISC-architektúra utasításkészletének legjellemzőbb tulajdonságai az egyszerűség és a szabályosság. Alapvető cél, hogy a RISC processzorok hozzávetőlegesen 30-50 utasítással rendelkezzenek. Az utasítások lehetőleg egyszerűek, ún. elementárisak legyenek, kerülendő és felesleges bonyolult műveletsorok alkalmazása. Nézzük pontokba szedve a követelményeket:

1) Az utasítások egyenszilárdságúan használhatók legyenek
Valóban az szerepeljen utasításként, ami felhasználásra is kerül, egyébként ne bonyolítsa feleslegesen a CPU utasítás dekódoló- és végrehajtó egységét (idő, felépítési szabályosság).

2) Az utasítások általános felépítésűek legyenek
A RISC filozófia kifejezetten „tiltakozik" a mikroprogramozott CPU ellen - elsősorban a sebesség növelése érdekében. Huzalozott logikával azonban akkor lehet gyorsan dolgozni, ha az utasításkód felépítése szabályos, az egyes bitpozíciók adott funkciókhoz dedikáltak, az utasításokon belüli funkciómezők elhelyezkedése lehetőleg utasítástól függetlenül fix, azaz az utasítások felépítése szabályos és általános (amennyiben az utasítások valamennyi adattípust valamennyi címzési móddal képesek használni, ún. ortogonális utasításkészletről beszélünk).

3) Az utasítások időben gyorsan végrehajthatók legyenek
Általános követelményként jelenik meg az 1 utasítás = 1 órajel-periódus követelmény. (Ez a pipeline technika révén nem jelenti azt, hogy az utasítást egyetlen órajel-periódus alatt végre is kell hajtani! A mai mikroprocesszor gyártók ezt a követelményt már 1/2 órajel-periódusra vagy több utasítás egyidejű elkezdésére fokozzák; ez csak a belső architektúra bonyolultságának kérdése és mindaddig nem jelent minőségi változást, ameddig a processzor nem „rendezi át" a program utasításainak szekvenciáját.)

4) Az utasítások időbeli lefolyása lehetőleg egységes és előre kiszámítható legyen
Egy szoros szoftver-hardver együttműködéshez az szükséges, hogy mindaz, ami a CPU „belsejében zajlik", előre egyszerűen végigkövethető legyen kívülről is a szoftver számára. A végrehajtás elemzésénél látni fogjuk, hogy a szoftvernek szüksége van arra, hogy órajel szinten tudja azt, mi történik a CPU belsejében. A végrehajtás ilyen mértékű követése akkor megy egyszerűen, ha időbeli lefolyásuk általános szabályosságokat mutat.

5) Load-store architektúra
Külön kell választani a memóriareferens utasításokat (töltő/tároló utasítások) a többitől a memóriák lassúsága miatt. Ezen utasítások kezelése elkülönítetten történik, minden más utasítás regiszter referens kell legyen (regiszterből regiszterbe dolgozik), ami nem korlátozza a CPU sebességét. Ehhez természetesen megfelelő számú regiszter szükséges.

6) Különválasztott társprocesszor felület
Nem nélkülözhetők a bonyolultabb aritmetikai utasítások sem. Ezeket azonban ne próbáljuk meg „begyömöszölni" a RISC processzorok egyszerű és szabályos utasításai közé, bízzuk ezt egy (vagy több) konkurensen működő társprocesszorra ill. lebegőpontos műveletvégző egységre (FPU-ra). Hasonlóan a load/store utasításokhoz, a teljes végrehajtási időt tekintve ezek a társprocesszor utasítások is „lassabbak", mint az alap utasítások, ezekkel nem szabad a CPU utasításkészletének egységét megbontani. (Ez az előírás napjainkban már elavultnak tekinthető.)

7) Az utasítások 3 címűek legyenek
Kutatások azt mutatják, hogy az a kétcímű gépekre jellemző tulajdonság, mely szerint az egyik forrás- és a céloperandus címe megegyezik, rengeteg utántöltési és mentési művelet forrása. Ebből kiindulva javasolják a kutatók a 3 című utasítások kialakítását, innen származik a RISC betűszó másik értelmezése is: ***R**eusable **I**nformation **S**torage **C**omputer*.

1.3.2 Az utasítások végrehajtása

Mint azt az előző pontban már említettük, célkitűzés az, hogy az utasítások lehetőleg "egyetlen órajel-periódus alatt végrehajthatók" legyenek.

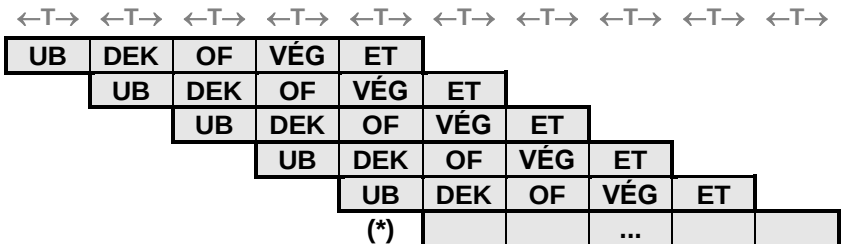
Általánosságban egy CPU-utasítás végrehajtása a következő fázisokból építhető fel:

- Utasítás beolvasása (UB)
- Dekódolás (DEK)
- Operandus felhozatala (OF)
- Végrehajtás (VÉG)
- Eredmény tárolása (ET)

A hagyományos (nem pipeline) műveletvégzésű számítógépek és mikroprocesszorok esetében az utasításfázisok végrehajtása is szekvenciálisan történik:

Utasítások hagyományos (egymás után történő) végrehajtása

Elképzelhető azonban a következő végrehajtási mód is:



A pipeline utasítás végrehajtás elve

Egy-egy utasítás végrehajtása ugyan valójában nem egy órajel-periódus alatt zajlik le, mivel azonban minden órajel-periódusban véget ér egy utasítás, a felhasználó számára úgy látszik, mintha az utasítás csupán egy órajel-periódus alatt hajtódna végre.
Igen ám, de mit feltételez ez a módszer ?

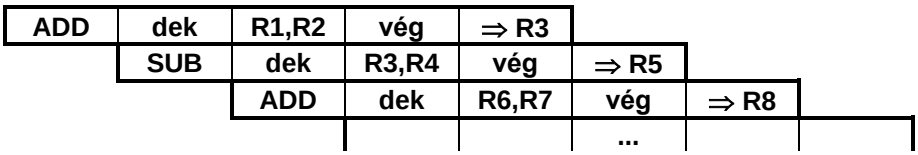
- nagybonyolultságú végrehajtó egységet, amely egyidejűleg képes több utasítás „kvázi párhuzamos” feldolgozására: pl. a (*)-gal jelölt esetben 3 tárolási művelet (UB, ET, OF), egy utasítás dekódolása és egy megint másik végrehajtása zajlik egyidejűleg. A 3 tárolási művelet megoldhatatlan? Később látni fogjuk, hogy nem (különválasztott utasítás- és adatbusz, ill. regiszterreferens utasítások).
- olyan utasításstruktúrát, amely egységes szerkezetű, mert az egyedileg változó hosszúságú utasítások, valamint a végrehajtási idők különbözőségének nagyszámú variációs lehetősége gyakorlatilag lehetetlenül bonyolulttá teszi a fenti végrehajtóművet.

(A nagyteljesítményű CISC processzorok, így pl. a 68020/30/40-es már részben alkalmazták ezt a módszert: ott prefetch ill. 3/6 fokozatú pipeline elnevezésekkel. Ezeknél azonban pontosan az utasítások egyedisége egy „tiszta” pipeline műveletvégző egységet oly mértékben elbonyolított volna, hogy annak megvalósítása már technológiai korlátok miatt is lehetetlen, nem beszélve arról, hogy mindez kívülről szinte követhetetlen lenne.)

A fenti pipeline művelet végrehajtást gyakorlatilag minden RISC-processzor alkalmazza. Ez azonban további, eddig nem létező problémákat vet fel:

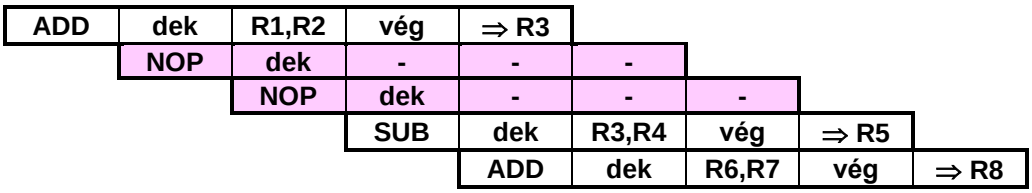
A műveletek sorrendi végrehajtásában operandus konfliktusok léphetnek fel. Vegyük a következő példát:

```
ADD R1,R2,R3    (R3:=R1+R2)
SUB R3,R4,R5    (R5:=R3-R4)
ADD R6,R7,R8    (R8:=R6+R7)
```



Operandus konfliktus pipeline utasítás végrehajtásnál

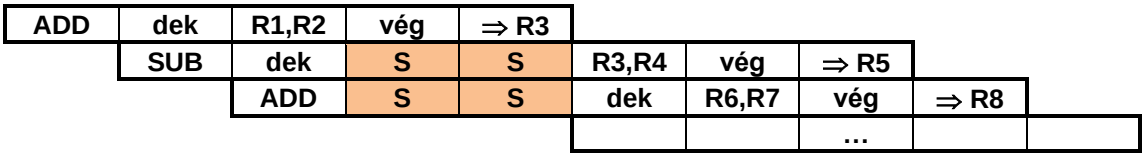
Látható, hogy a második műveletben R3 hamarabb felhasználásra kerül, mint az első utasítás eredményének tárolása megtörténne. Ennek elkerülésére NOP utasítás(oka)t kell beszúrni:



A konfliktus feloldásának szoftver megoldása: NOP-ok beszúrása

Speciális regiszterhozzáférés-szervezéssel (register-bypass) elérhető, hogy a második NOP utasítást ne kelljen beszúrni (ld. a regiszterek szervezésénél).

Nagy vita zajlik a szoftver- és a hardver-szakemberek között, hogy a NOP(ok) beszúrása kinek a feladata. A hardver-szakemberek érthetően nem akarnak olyan egységet kiadni a kezükből, amely működése hibás lehet „nem megfelelő” programozás esetén. Így a RISC processzorok legtöbbje tartalmaz olyan ún. pontozótábla egységeket (**Scoreboarding-Register**), amely a fenti hibát hardver úton kiküszöböli. Ekkor nem tényleges NOP utasítások beszúrása, hanem a konfliktusban érintett pipeline fázisok időleges felfüggesztése, ún. „stall” (felfüggesztés, elakadás) fázisok beiktatása történik addig, amíg az operandusok közötti konfliktus fennáll.



A konfliktus feloldásának hardver megoldása: „stall” fázisok beszúrása

A szoftver-szakemberek ezt felesleges bonyolításnak tartják, mivel céljuk az, hogy még az assembly szintű programozás is olyan optimalizáló fordítóprogramok segítségével történjen, amelyek a fenti hibát kiküszöbölik, sőt, pl. a jelen esetben intelligensen, az utasítások sorrendjének felcserélésével még a felesleges NOP beszúrását is megtakarítják:

ADD	**dek**	**R1,R2**	**vég**	**⇒ R3**		
	ADD	**dek**	**R6,R7**	**vég**	**⇒ R8**	
		SUB	**dek**	**R3,R4**	**vég**	**⇒ R5**
					...	

A konfliktus másik lehetséges feloldása: az utasítások sorrendjének felcserélése

Azon parancsok, melyek szerkezete nem felel meg a fentieknek, speciális kezelést igényelnek. Ezek a következők:

- LOAD, STORE utasítások (mivel a memória lassú)
- FPU-utasítások (végrehajtási idő)
- vezérlésátadó műveletek (mi jön utána ?)

A memóriareferens műveletek hasonlóan NOP-ok beszúrásával kezelhetők, illetve irodalmi hivatkozások szerint kb. 90%-ban az előzőekhez hasonlóan utasításcserével áthidalhatók (az ilyen utasítások „mögé" tesszük a felhozataltól független nem memóriareferens utasításokat, melyek adott esetben csak a külön erre a célra fenntartott utasításbuszt használják).

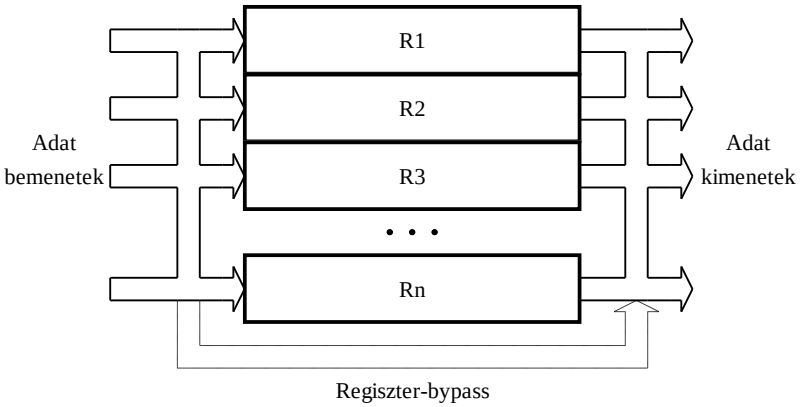
A pipeline folyamatos működtetésében a programban lévő vezérlésátadó utasítások (feltétel nélküli, feltételes ugró utasítások, ciklusutasítások) okozzák a legtöbb nehézséget. Ennek egyrészt oka az, hogy a feltétel teljesülése, az ugrási cím, csak az utasítás részbeni vagy teljes feldolgozása után válik ismertté, másrészt ha az ugrási címtől kell folytatni a végrehajtást, akkor az adatcsatornában lévő utasításokat törölni kell. A törléssel együtt vissza kell állítani az eredeti állapotot is. Ez megint csak történhet hardver úton, NOP-ok beszúrásával, vagy más megoldással (ún. késleltetett ugrás, vagy **delayed JMP**). Fejlettebb módszerek az elágazás-előrejelezés (branch prediction), az utasítássorrend átrendezése vagy a csővezeték többszörözése (részletekbe idő hiányában itt nem tudunk belemenni, mesterképzésen a Nagyteljesítményű mikrokontrollerek és interfészek tantárgyban részletesen foglalkozunk majd ezzel).

1.3.3 A regiszterek szervezése

A RISC architektúra igen fontos eleme a CPU regiszterblokkja, hiszen a korábban elmondottakból több tulajdonság is feltételezi ezek egyrészről megfelelő számát és a következő tulajdonságait:

1. a szükséges szimmetria teljesen egyenértékű regisztereket feltételez (ne legyen akkumulátor ill. cím-/adat-/bázis- stb. regiszter).

2. a regiszterreferens utasítások kialakítása megfelelően nagy számú regisztert feltételez
3. a 3 című utasítások kialakítása is a szükséges regiszterek számát növeli
4. a pipeline végrehajtás olyan regiszterblokk-kialakítást feltételez, amelynél lehetőség van arra, hogy a végrehajtómű különböző egységei több különböző regiszterhez férhessenek hozzá egyidejűleg. A fent példaként megjelent második NOP kiszűréséhez az szükséges, hogy az éppen tárolásra kerülő végeredmény már ugyanezen órajel-periódusban elérhető legyen a töltés számára (ún. **regiszter bypass**)



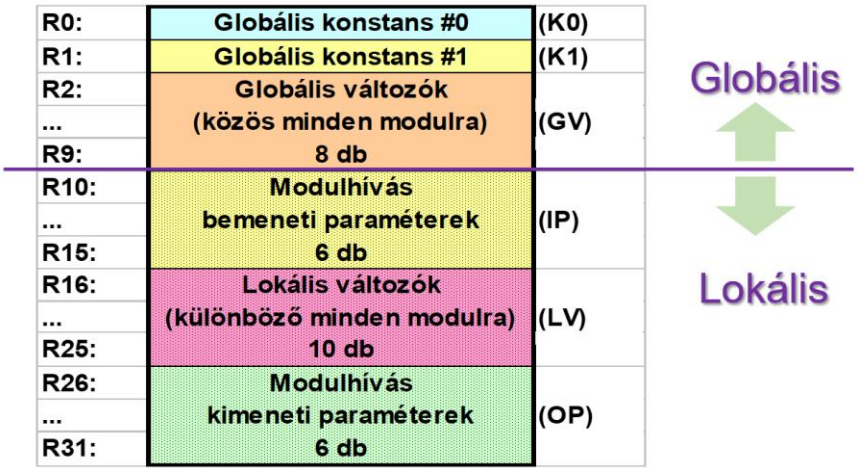
Regiszter bypass

A „sok regiszter” az elméleti elképzelések szerint akár több száz tárolóegységet is jelent. Ehhez képest a valóságos RISC processzorok nagyságrendileg 32-64 regiszterrel jelennek meg. A fenti elképzeléseknek azonban teljes mértékben ellentmond az a követelmény, hogy minél több a regiszter, modulhíváskor (magas szintű nyelvek!) annál nagyobb overhead-del történhet meg a regiszterblokk mentése. Ezen hivatott segíteni a bank- illetve az ablaktechnika.

- **Regiszterbankok** (register banking) alkalmazása esetében a regisztertömb nem átlapolódó, azonos méretű részekre, ún. bankokra van felosztva. Egy-egy ilyen bank mérete 2 valamely hatványa. A processzor számára az aktuálisan használt bank kezdetét a bank-pointer (CBP = current bank pointer) jelöli ki.
- Az **ablaktechnika** (register windowing) alkalmazásakor a regisztertömb egy-egy azonos méretű, de átlapolható része látható és használható a processzor számára. Az “ablak” mérete mindig azonos és 2-nek valamely hatványa. Az aktuális ablak kezdetét az ablak-mutató (CWP = current window pointer) jelöli ki. Nézzünk erre egy konkrét példát:

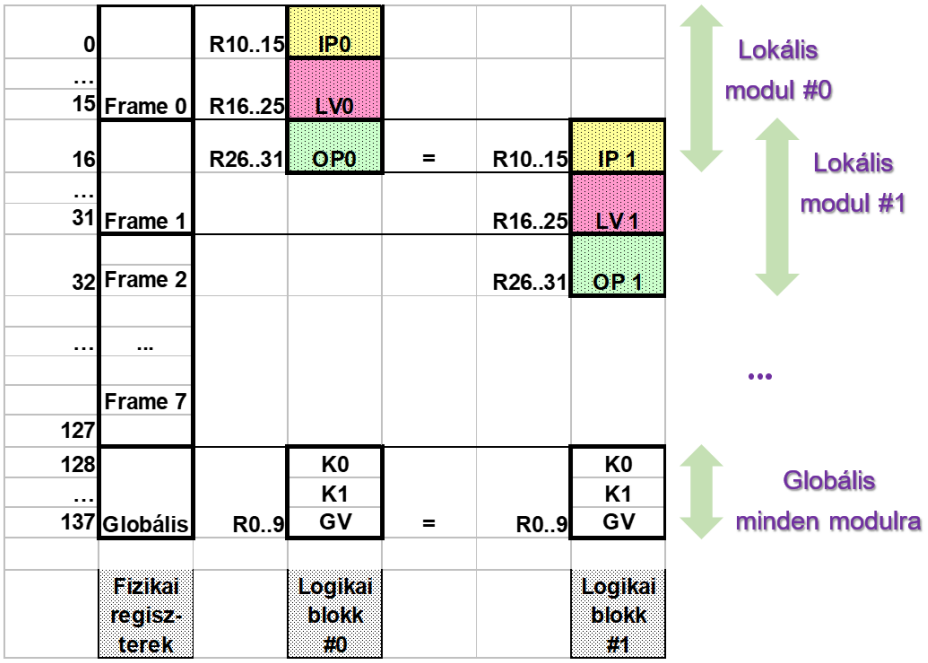
A vizsgálatok azt mutatják, hogy egy-egy modul szintjén a valóban kihasznált munkaregiszterek száma tipizálható (8-10), a modulhívási mélység pedig ritkán haladja meg a 8-at. Ily módon megvalósítható lenne egy olyan elképzelés, amely egy CPU-ban pl. 8 db regiszterblokkot feltételez, blokkonként az említett számú regiszterrel, a blokkok közötti átkapcsolás pedig modulhíváskor egyszerűen megvalósítható.
Probléma viszont a modulok közötti paraméterátadás megoldása, hiszen így minden modul csak a "saját" regiszterblokkját látja, a szükséges be/kimeneti paraméterek átadása mégis csak a lassú memóriaműveletek segítségével történhetne. (A probléma hasonló a

bankosított memóriakezeléshez.) Ennek áthidalására javasolják a fix funkcionális kiosztású átlapolt regiszterblokkok kialakítását a következők szerint:
Tartalmazzon minden, egy-egy a modul számára elérhető regiszterblokk 32 regisztert (R0...R31) a következő kialakítás szerint:



Programmodulok regiszterkészlete

Rendelkezzen a központi egységünk összesen 138 db regiszterrel a következő kialakítási filozófia szerint:



Egymást hívó programmodulok regiszterkészletei

Az OP/IP mezők átlapolódása a hívási/visszatérési paraméterek átadását szolgálja, a konstans és a globális változók területei mindegyik modul számára azonos módon érhető

el. A módszer automatizmusát magyarázni felesleges, hatékony működéséhez a hardvertámogatás elengedhetetlen.

1.3.4 A memória szervezése

Mint láttuk, a RISC architektúra szűk keresztmetszetként kezeli a memóriát, elsősorban sebességi okokra hivatkozva. Az idevonatkozó javaslatok a következők:

A pipeline feldolgozás miatt (párhuzamos töltő műveletek!) célszerűnek látszik az utasítás- és az adatforgalom szétválasztása. Ez a **Harvard-architektúrához** való visszatérést jelenti a Neumann-elvű számítógépektől. A ma létező RISC processzorok többsége elkülönített utasítás/operandus busszal rendelkezik, és a felhasználóra bízza, hogy e kettős buszrendszert fizikailag azonos memóriához vezeti-e (amivel még nem feltétlenül tettük tönkre a filozófiát, ld. cache!)

Előtérbe kerül a **cache memória** alkalmazása. A struktúrából adódóan elkülönített utasítás és adat cache-ről beszélhetünk, célszerű értéket igen nehéz javasolni, mivel az optimum a költségek függvénye, költségektől függetlenül nyilván az lenne az optimum, ha a teljes memória cache-sebességű lenne. A mai gyakorlatban használatos nagyságok 4-32 kbájt (belső!) ill. 128-256 kbájt (külső) közé esnek.

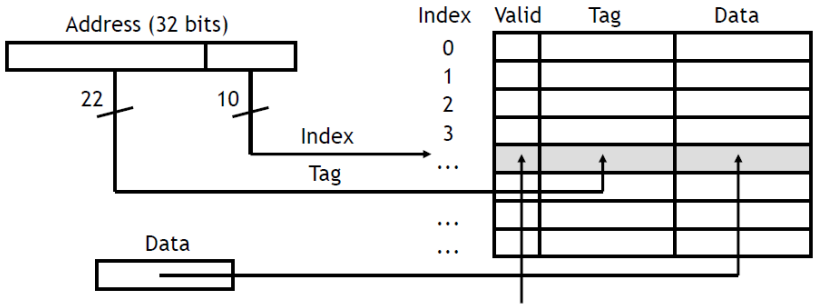
Rövid emlékeztető a cache memóriák működéséhez:

A gyorsítótár működése azon alapul, hogy az egy sokkal gyorsabb működésű tárolóelem, mint a környezetében található hasonló tárolóelemek. Bármely elem felhozatala a nagy tömegű adat tárolására szolgáló főmemóriából történik, azonban ez a processzorban történő felhasználáson túl a cache memóriában is tárolásra kerül, így ha az elemre többször is szükségünk lesz, másodszor, harmadszor, stb. már a sokkal gyorsabb cache memóriából tudjuk azt beolvasni. Amennyiben a keresett elem megtalálható a cache memóriában, találatról (hit) beszélünk, ellenkező esetben hibáról/hiányról (miss). Megkülönböztetünk utasítás és adat cache memóriákat (a processzor architektúrájától függően a kettő elkülönülhet), külön problémát jelent, ha adatok esetében a cache tartalma a főmemóriából (olvasás) vagy a mikroprocesszorból (írás) is származhat.

A cache mérete sokkal kisebb, mint a főmemóriáé, ezért nincs minden elemnek rögzített helye abban – a különböző szervezési módok pontosan a kiszorítások hatékonyságát próbálják javítani az egyes elemek között. Minden tárolandó elemmel együtt tárolni kell tehát annak egy azonosítóját (címét, címrészletét, ezt tag-nek nevezzük), amivel az elem pontos származása a későbbiekben is azonosítható. Így tehát pl. egy 32 bites utasítás mellett tipikusan 16-20-24 bitnyi tag információ tárolása is szükséges, hogy az elem pontosan milyen címről került beolvasásra – a cím többi része már a cache-ben való elhelyezkedéséből azonosítható.

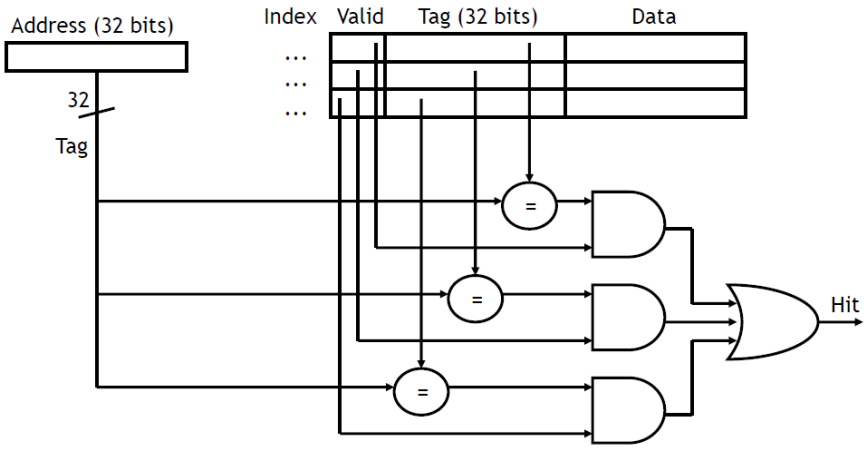
Az alapvető cache szervezések a következők:

a) **Direct mapping cache (DMC)**: A cím alsó bitjei indexelik a cache egyes sorait, a hozzárendelés fix. A cím felső része (pl. 20 bit) a tag, alsó része (12 bit) az index. Találathoz az index által meghatározott sor tagjének kell egyeznie a felső címbitekkel (továbbá a cella státusza érvényességet kell mutasson). *Előnye*: igen egyszerű és gyors, *hátránya*: a fix hozzárendelés miatt indokolatlanul magas lehet a kiszorítási arány.



Direkt hozzárendelésű cache

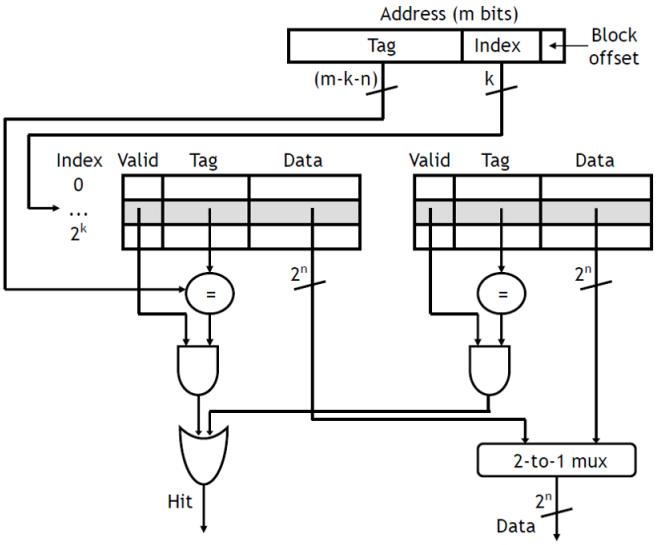
b) **Fully associative cache (FUC)**: A cache tag-része teljesen asszociatív memóriában tárolódik (CAM = Content Adressable Memory), semelyik bejegyzésnek "nincs fix helye". *Előnye*: optimális a kiszorítási arány, *hátránya*: bonyolultsága miatt nagyon drága és lassú.



Asszociatív cache

c) **Set-associative cache (SAC)**: Ebben a megoldásban több direkt mappelt cache-t építünk egymás "mellé", ezek mindegyike egy-egy cache-blokkot (**set**) alkot, számuk jelenti azt, hogy hány utas (way) a cache. A direkt mappelésből adódóan a cím index része meghatározza a cache 1 sorát, a blokkok pedig egymás között asszociatív cache-t alkotnak (set-associative), azaz itt már foglaltságfüggő, hogy melyik set-ben kerül elhelyezésre az információ (taggel együtt !), kereséskor pedig minden setnek az index által meghatározott sora asszociatív módon kell keressen. *Előnye*: összesíti az előző két megoldás előnyeit, *hátránya*: bonyolultabb, mint a direkt mappelés, nem olyan jó

a találati arány, mint a teljesen asszociatív cache esetében. Manapság az egyik legelterjedtebben használt cache-filozófia.



2 utas set-asszociatív cache

d) **Sector mapping cache (SMC)**: Ma már szinte eltűnt mechanizmus, mikor a memóriát szektorokra osztották fel és a cache-ben minden szektornak saját cache területe volt. Ez nem jelenti azt, hogy a szektor tartalma "egyszerre került mozgatásra", inkább úgy értendő, hogy minden memóriaszektor saját direkt mappelt cache-el rendelkezett, információ kiszorítás csak azonos szektoron belüli információk között léphetett fel. *Előnye*: jobb a kiszorítási arány, mint a direkt mappelt cache esetében, *hátránya*: viszonylag nagy cache-méret egyenetlen kihasználtsággal.)

A speciális hozzáférések elkerülése érdekében kerülni kell az utasítások (és esetleg az adatok) "tömörítését". A CISC gépek a memóriával való takarékoskodás érdekében igen erős szélsőségekig hajlandók voltak elmenni. A 32 bites 68030-as pl. megengedi az utasításnak 16 bites szóhatáron történő elhelyezkedését is, bizonyos gépek (iAPX 432) ezt a „takarékoskodást" egészen a bithatárig fokozták. Az adatoknak a memóriában történő tetszőleges elhelyezhetősége érdekében a 68020-as hajlandó egy operandus felhozásáért két hozzáférést is kezdeményezni (non-aligned operand). Az említett tört- („folytató-") ciklusok kiküszöbölése érdekében a RISC gépekre vonatkozó javaslat a következő: az utasítások/adatok elhelyezkedése igazodjon a gép adatbuszainak méretéhez (tipikusan 32 bit), a különböző méretű adattárolók között inkább maradjon üres hely a memóriában, de akkor se kelljen komplikált címszámítási műveleteket végezni, főként pedig ne legyen szükség egy adat felhozatalához több memória hozzáférésre.

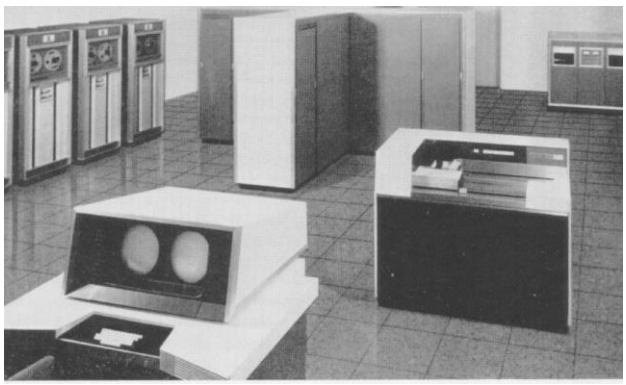
1.3.5 Összefoglalás

Foglaljuk össze a tanultakat a CISC és a RISC processzorok legfontosabb tulajdonságait összevető táblázat segítségével ! Ezt követően pedig bemutatunk néhány példát a RISC architektúra szerint megépített számítógépek és mikroprocesszorok közül.

	CISC processzorok	RISC processzorok
1	**Összetett** utasítások, melyek végrehajtása több gépi ciklust igényel	**Egyszerű** utasítások, melyek végrehajtása 1 gépi ciklust igényel
2	**Bármely**, erre alkalmas utasítás igénybe veheti a tárolót	Csak a **LOAD/STORE** utasítások fordulhatnak a memóriához
3	Az adatcsatornás (pipelining) feldolgozás **kismértékű**	**Erőteljes** adatcsatornás (pipelining) feldolgozás
4	Utasítás végrehajtás **mikroprogram** által vezérelt	**Huzalozott** utasítás végrehajtás
5	**Változó** hosszúságú utasítások	**Rögzített** utasításhossz
6	**Sokféle** utasítás és címzési mód	**Kevés** utasítás és címzési mód
7	Bonyolult **mikroprogram**	Bonyolult **fordítóprogram**
8	**Kis számú** regiszter	**Nagy méretű** regisztertár

CISC és RISC processzorok tulajdonságainak összehasonlítása

A számítógépek történetére visszatekintve a legkorábban megjelent, már bizonyos RISC jellemzőkkel leírható számítógép, a **CDC 6600**-as kifejlesztése, a 60-as évek elején Seymour Cray nevéhez fűződik. Ugyanerre az időszakra tehető az első, ilyen típusú, gyártásba is került berendezés, az **NCR 8500**-as számítógép megjelenése is.

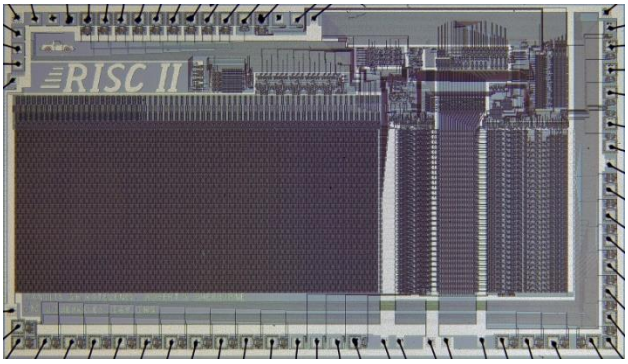


A CDC-6600 számítógép

A 70-es évek közepén indult az IBM-nél egy fejlesztés, melynek eredményeként 1978-ban elkészült az **IBM 801**-es típusú, már igazi RISC elvek szerint működő számítógép, amely igazából nem lett piaci siker. A processzor kb. 15 MHz-es órajel-frekvenciával működött, négyfokozatú utasítás pipeline-t és LOAD/STORE tárolási struktúrát használt. Minden utasítás és adat is egyaránt 32 bit hosszúságban került tárolásra. A processzor utasításkészlete 120 egyszerű utasításból állt és mindössze két címzési lehetőséggel rendelkezett. A processzorban 32 db általános célú regiszter volt.

1980 körül indult el a *University of California, Berkeley*-n a **RISC I**, majd a **RISC II** processzorok fejlesztése. A processzor VLSI technológiával készült, 3-fokozatú pipeline feldolgozással és LOAD/STORE struktúrával. Utasításkészlete 39 (!) utasításból állt,

utasításai és adatai is 32 bitesek voltak. 138 regiszterből álló regisztertárral rendelkezett, amelynél először alkalmazták az ún. regiszter-ablaktechnikát (register-windowing), amely azóta is széles körben elterjedt. A Berkeley-n kifejlesztett RISC I és II processzorok voltak strukturálisan a SPARC processzorok elődei.



Berkeley RISC II

Ezzel egy időben kezdődött a *Stanford University*-n a **MIPS** (Microprocessor without Interlocked Pipeline Stages) processzorok fejlesztése. A fejlesztésnél cél volt a hardver egyszerű volta és az utasításkészlet olyan kialakítása, amely optimalizáló fordítóprogram használatával tette lehetővé a gépi szintű program előállítását. A processzorba egy 5-fokozatú utasítás pipeline-t építettek, amely egy hasonló elvű adat pipeline-nal egészült ki. Az utasításkészlet minimális volt (31 alaputasítás) és LOAD/STORE struktúra épült rá. 32 bites utasítással és adatszóval működött.

Toshiba MIPS R4400

A MIPS és a többi nagyteljesítményű RISC architektúrán alapuló rendszerek komoly konkurenciát jelentettek az asztali számítógépek piacán, várható volt, hogy lekőrözik az Intel IA32-es architektúrát. Az Intel azonban gyorsan lépett: kihozta a lényegesen gyorsabb Pentium családot, a Microsoft pedig megszüntette a ”nem Intel” processzorok Windows támogatását. Ezzel a MIPS processzorok gyakorlatilag teljesen kiszorultak az asztali számítógépek piacáról.

A RISC architektúra elveinek tényleges hardver megvalósulását képezik az ARM processzorok, melyekkel a későbbiekben részletesen foglalkozni fogunk.

1.4 A mikrokontroller architektúra kialakulása

1969-ben a Nippon Calculating Machine Corporation, egy elektronikus számológépeket gyártó japán cég az egy évvel korábban alapított amerikai Intel céget egy, az általa gyártandó asztali számológép sorozathoz (Busicom 141-PF) használható speciális célú integrált áramkör család megtervezésével és legyártásával bízta meg. A 12 lapkából álló logikai elemcsalád kifejlesztésének feladata elég nagy volt, a viszonylag fiatal cég fejlesztői kapacitása pedig kevés. A mentő ötlettel Ted Hoff, a Stanford Egyetemről átcsábított mérnök állt elő: nem több, a Nippon által előírt specifikus feladatokra kialakított áramkört kell tervezni, hanem egy általános célú számítógépet, amelyet aztán a megadott számológép-műveletek elvégzésére kell beprogramozni. A projekt felelős tervezői Federico Faggin (ejtsd: Fadzsín) olasz fizikus és villamosmérnök, valamint Marcian Edward „Ted” Hoff, szintén villamosmérnök lettek. Az általuk létrehozott megoldást tekinthetjük a világ első mikroprocesszorának. A szabadalmi jog először a Nipponé lett, ám a dolog jelentőségét felismerve az Intel hamarosan visszavásárolta azt. A 4 chip segítségével kialakított rendszert MCS-4-nek nevezték el, melyek egyike – a tároló és a be-/kimeneti elemek mellett – a 4004-es mikroprocesszor volt. Fejlesztésük eredményét 1971. november 15-én mutatták be.

A Busicom 141-PF

A 4004-es mikroprocesszor

Marcian E. „Ted” Hoff és Federico Faggin, a 4004-es processzor kifejlesztői

A világ első 4 bites mikroprocesszora 3 x 4 mm-es lapkán 2300 tranzisztorból épült (ma az AMD EPYC 9754 Genoa 128 magos processzorokban található tranzisztorok száma 90 milliárd körül van), 740 kHz maximális órajel-frekvenciával működött és 46 utasítást ismert. Másodpercenként 92 000 művelet elvégzésére volt képes. Fél évvel később, 1972 áprilisában mutatta be az Intel i8008 névre keresztelt processzorát, amely már 8 bites, 16

kbájt memóriát kezelni képes processzor volt, 45 paranccsal rendelkezett, és 80 000 utasítást hajtott végre másodpercenként (kicsit kevesebbet, mint elődje, de 4 helyett 8 biten). Az Intelnél tovább folytatták a kutatást, aminek eredményeként 1974 áprilisában bemutatták az i8080-as mikroprocesszort.

Az Intel fejlesztéseivel párhuzamosan más cégek is elkezdték fejleszteni saját mikroprocesszoraikat. A nagy riválisa mindvégig a Motorola volt. Az általuk gyártott eszköz, a 6800 típusjelű 8-bites processzor volt (1975). A fejlesztés vezetője, Chuck Peddle átkerült a MOS Technology Inc. céghez, itt jelentek meg a 6800-hoz „erősen hasonlító” 6501, és 6502 (8 bites, 56 utasításos) mikroprocesszorok. Utóbbiból minimális bővítéssel készült a 6510-es, amely a Commodore 64 személyi számítógépek népszerű processzora volt.

Federico Faggin kivált az Intelből és Zilog néven alapított céget. 1976-ban mutatták be saját, Z80 névre keresztelt mikroprocesszorukat. A Z80 kód-kompatibilis volt az i8080-al, de kiegészítették új utasításokkal is. A Z80 korának meghatározó, legelterjedtebb processzora volt. Szintén 1976-ban került a piacra az Intel i8085-ös (8 bites) mikroprocesszora, de a Z80 ennél is erősebbnek bizonyult.

1976-ban vált ketté a mikroprocesszor és a mikrokontrollerek üzletága. Ez idő tájt már körvonalazódni látszik, hogy az alkalmazások egy meghatározó részének a logikai „magja” és felépítése teljesen hasonló: a mikroprocesszorok működtetéséhez órajelre van szükség, program- és adatmemóriák tárolják az utasításokat és az adatokat, időzítő (timer) áramkörök látják el az időzítési feladatokat, soros kommunikációval, valamint digitális be- és kimenetekkel kapcsolódnak környezetükhöz. A technológia rohamos fejlődése egyre inkább lehetővé teszi az egységek „egybeintegrálását”, ami a tervezési folyamat lényeges egyszerűsödését, a fejlesztési idő lerövidülését és a költségek csökkentését teszi lehetővé. A folyamat pedig elindul…

1.5 A 8051-es család

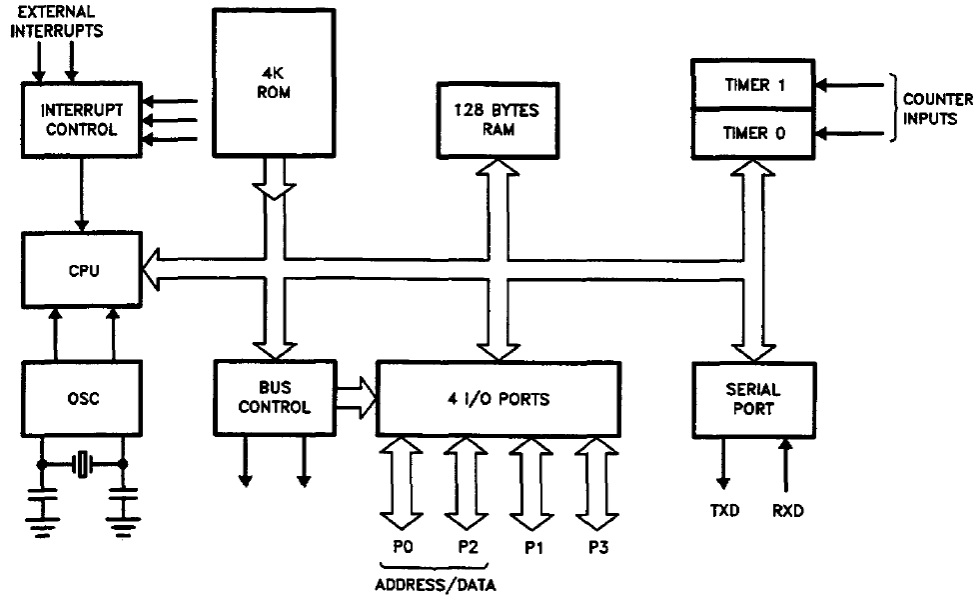
Az előző fejezetben vázolt folyamat első lépéseként az Intel megtervezte az i8048-as mikrokontrollert (ez már egyetlen lapkán 17 ezer tranzisztort, 1 kbájt ROM-ot, 64 bájt RAM-ot, egy 8 bites időzítőt, és 27 digitális be-/kimeneti vonalat tartalmazott). Kisebb alkalmazásokhoz elegendő volt, de az átütő sikert mégis az 1980-ban elkészített i8051-es mikrokontrollerrel aratta. A 8051-es vezérlő ipari szabvány lett az idők folyamán. Ez a mikrokontroller-sorozat a mai napig igen népszerű, nagynevű gyártók gyártják mind a mai napig az azóta lényegesen továbbfejlesztett „klónokat”. Ezek közös jellemzője, hogy a kód-kompatibilis 8051-es mag és az alapperiférák (cím- és funkció-kompatibilis) megtartása mellett egyre több, a mai igényekhez igazodó új tulajdonsággal ruházzák fel az újabb családtagokat. A legfontosabb gyártók, ahol érdemes ezeket a típusokat keresni:

1) Intel Corp.
2) Atmel Corp.
3) NXP (Philips) Semiconductors
4) Infineon Technologies (Siemens)

5) Texas Instruments
6) Maxim Integrated Products (Dallas Semiconductors)
7) Silicon Laboratories
8) Analog Devices
9) ST Microelectronics

1.5.1 Belső architektúra

Az anyag további tárgyalása során először minden esetben a 8051-es eredeti magjából indulunk ki, ezt követően említjük meg az idők során bekövetkezett fontosabb változásokat.



Az „ős” 8051-es architektúra

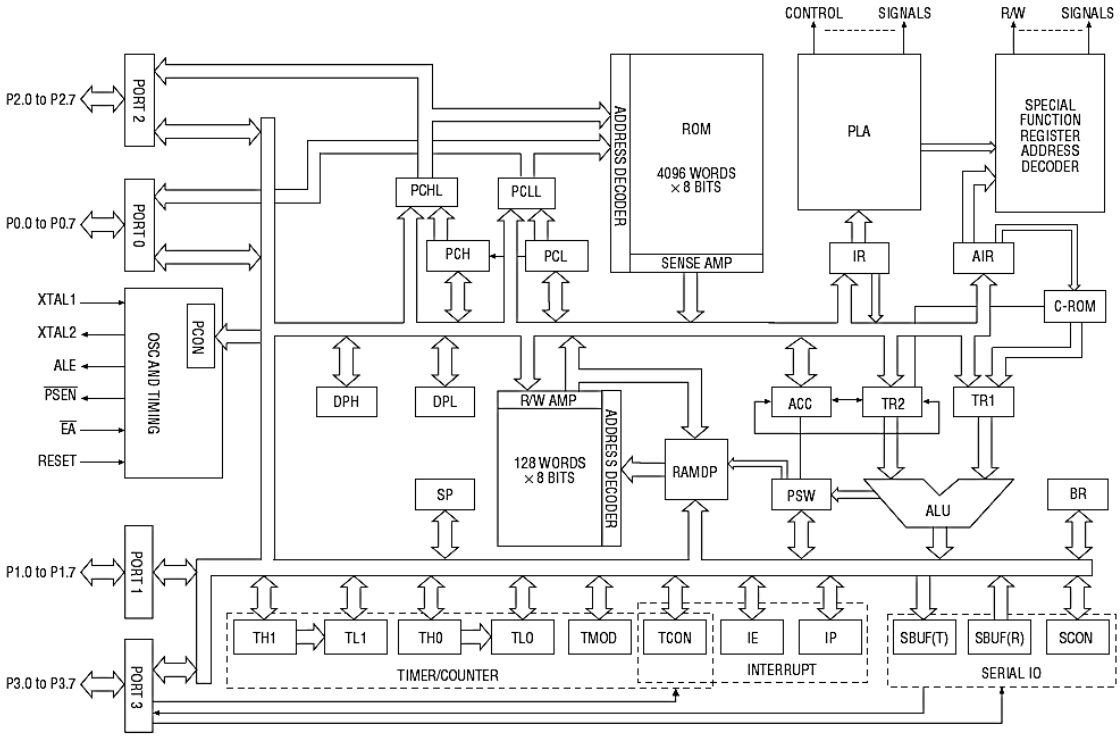
Az eredeti 8051-es architektúra azon az egyszerű felismerésen alapul, hogy – amint azt korábban már említettük – valamennyi alkalmazás számára nagy valószínűséggel a következő memória- és periféria elemek lesznek szükségesek:

1) nem felejtő (program-) memória (ROM, PROM vagy EPROM)
2) adatmemória (RAM)
3) órajel generátor (maga a kvarc nem integrálható a chipbe, de a többi elem igen)
4) időzítő/számláló egység (jobb, ha több, de legalább 1 vagy 2 mindenképpen)
5) soros kommunikációs port a környezet felé
6) párhuzamos I/O (digitális be- és kimenetek)
7) megszakítás kezelő egység a belső elemek - Timer, UART, portok - és a külvilág felé is (legalább 3-5 lineáris megszakítás kérési lehetőség)

Az elmondottak korábban legalább 8 db LSI integrált áramkör alkalmazását követelték meg (érdemes a „nagy elődök” adatlapjai után kutatni az Interneten):

- i80xx mikroprocesszor,
- EPROM,
- RAM,
- i8224 órajel-generátor,
- i8253 időzítő/számláló egység,
- i8251 USART (Universal Synchron/Asynchron Receiver/Transmitter),
- i8255 PPI (Parallel Peripheral Interface),
- i8259 megszakítás kezelő.

Az „ős” architektúra részletesebb vázlata az alábbi ábrán látható.

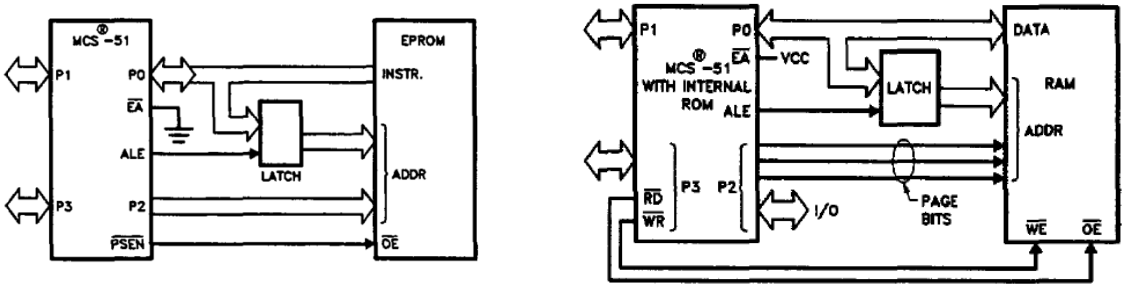


Az „ős” 8051-es architektúra részletesebb blokkvázlata

Mai, PC-s memóriaméretekhez szokott fülünk számára komolytalannak tűnhet a 4 kbájt program- és 128 bájt adatmemória mérete. A félév során találkozni fogunk több olyan alkalmazással is, amelyek majd bebizonyítják: a kisebb alkalmazásokhoz ezek a memóriakapacitások akár még ma is elegendőek. De rögtön az elején tisztáznunk kell, ezek a méretek az ez idő tájt technológiailag egyetlen lapkán megvalósítható lehetőségekből adódnak, maga a CPU ennél nagyobb méretű memóriákat is képes kezelni: a módosított Harvard architektúrájú processzor (közös adatbuszon elkülönített program- és adatmemória, az alapértelmezett kiépítésben az adatmemóriából nem hozható fel programutasítás – ld. később) összesen 64 kBájt kód- és 64 kByte adatmemória kezelésére képes külső memóriaelemek megfelelő csatlakoztatása esetén. Azt is megjegyezzük, hogy a i8751 változat ROM helyett EPROM-ot tartalmazott, a

későbbi változatokban (i80C52, i80C54, i80C58 az integrált memória mérete 8 ill. 16kbájtra nő, és az EPROM mellett megjelenik a lényegesen olcsóbb OTP ROM is).

Külső memóriák illesztése igen egyszerűen történhet a mikrokontrollerhez: a P0 port adatbuszként és a címvezetékek alsó 8 bitjének továbbítójaként is funkcionál, a többi címvezetéket a P2 porton jeleníti meg a CPU. Mint látni fogjuk, a kontroller rendelkezik olyan indirekt címzési móddal is, amely csak az alsó 8 címbit (automatikus) működtetését váltja ki egy belső regiszter segítségével. Ilyenkor a külső memória felé irányuló felső 8 címvezeték (P2) legkisebb helyiértékű bitjeivel 256 bájtos „laponként" tudjuk címezni a memóriát. Mivel a címképző egység ennél a címzési módnál a portot nem állítja automatikusan, annak a memória méretéből kiadódó szabad felső bitjeit tetszőleges I/O műveletek céljaira használhatjuk fel.



Külső memóriák illesztése a 8051-es mikrokontrollerhez

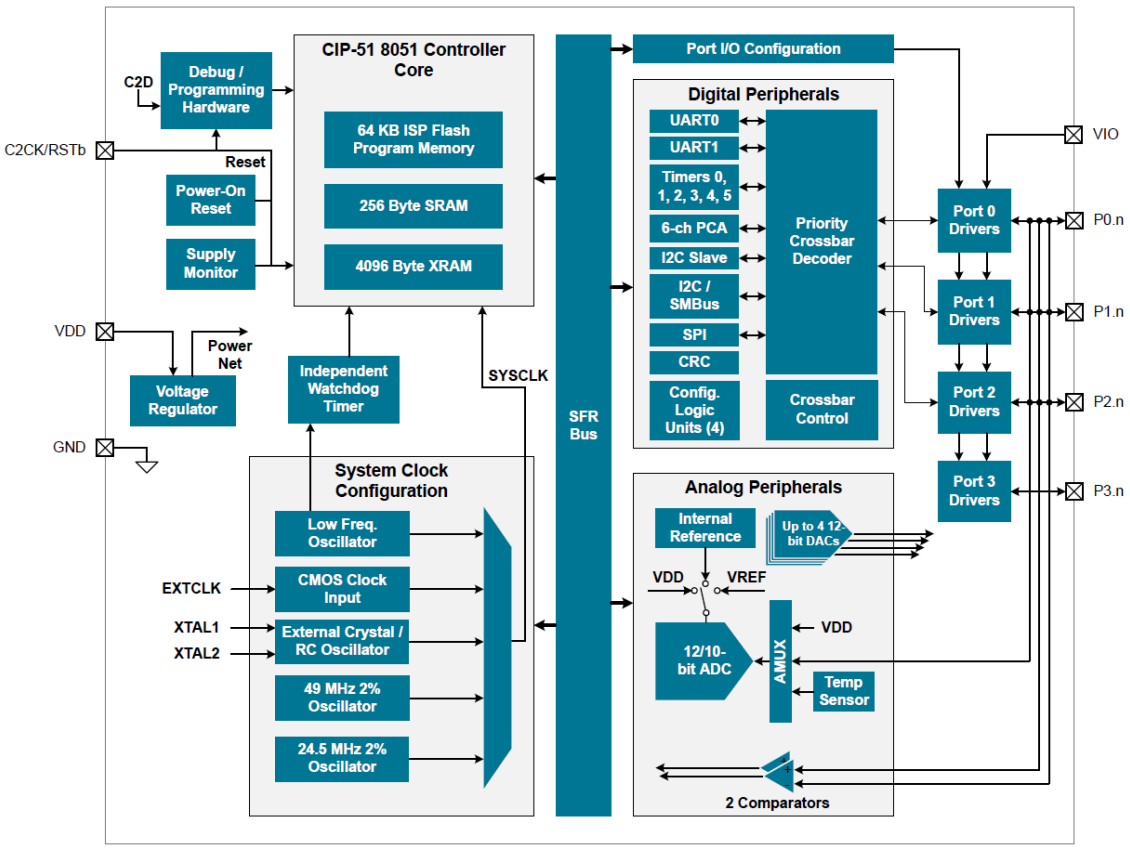
Az **/EA** (External Access) bemenettel definiálhatjuk, hogy indítás után belső (=1) vagy külső (=0) programmemóriát kívánunk használni alkalmazásunkban. Ez azért lényeges, mert ilyenkor a mikrokontrollernek már az első utasítás felhozatalakor is tudnia kell, hogy a memória kívül helyezkedik el. A szükséges vezérlőjelek mindegyikét előállítja a 8051-es: az **ALE** (Address Latch Enable) kimenet szolgáltat tároló impulzust az alsó címbájt eltárolásához, a **/PSEN** (Program Store Enable) engedélyezi a program memóriát, és a P3 porton található **/RD** és **/WR** szolgáltatják az adatmemória beíró (/WE) és kimenet-engedélyező (/OE) bemeneteinek a vezérlését.

Alapértelmezésben 32 db tetszőlegesen felhasználható digitális be- vagy kimenet áll a rendelkezésünkre a környezetből érkező információk beolvasására, ill. a szükséges vezérlésekhez. Ezen be- vagy kimenetek (a továbbiakban szakmai zsargonnal csak „portlábak") kezdenek fogyatkozni a külső elemek illesztése esetén: a 8 bites adatbusz teljes egészében lefoglalja a P0 portot, a memória méretétől függően használjuk a felső címvezetékek továbbítására a P2 portot. Külső RAM esetén a /RD és a /WR jelek a P3 porton találhatók, de ezen a porton találhatók további vezérlő vonalak is:

Pin	Name	Alternate Function
P3.0	RXD	Serial Input Line
P3.1	TXD	Serial Output Line
P3.2	$\overline{\text{INT0}}$	External Interrupt 0
P3.3	$\overline{\text{INT1}}$	External Interrupt 1
P3.4	T0	Timer 0 External Input
P3.5	T1	Timer 1 External Input
P3.6	$\overline{\text{WR}}$	External Data Memory Write Strobe
P3.7	$\overline{\text{RD}}$	External Data Memory Read Strobe

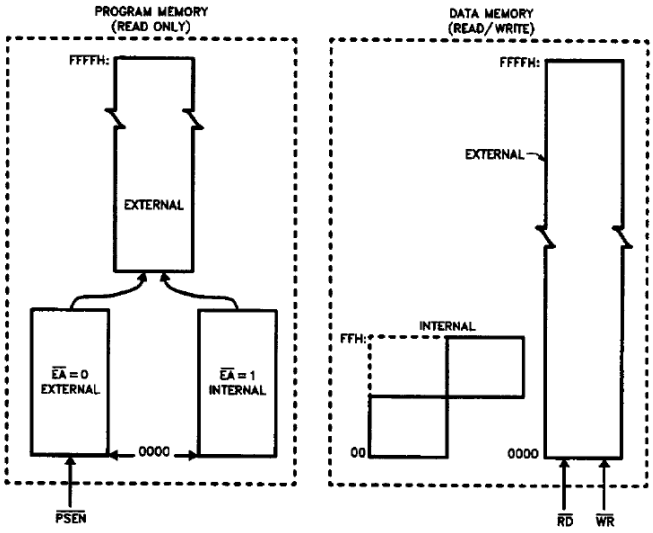
A P3 port alternatív funkciói

Végezetül nézzük meg, hogy az egyszerű „ősi” kialakítás mivé alakult át a napjainkban használatos 8051-es mikrokontrollerek esetében (ezzel a mikrokontrollerrel foglalkozunk majd legtöbbet pl. a házi feladat és az ágazati labormérések során):

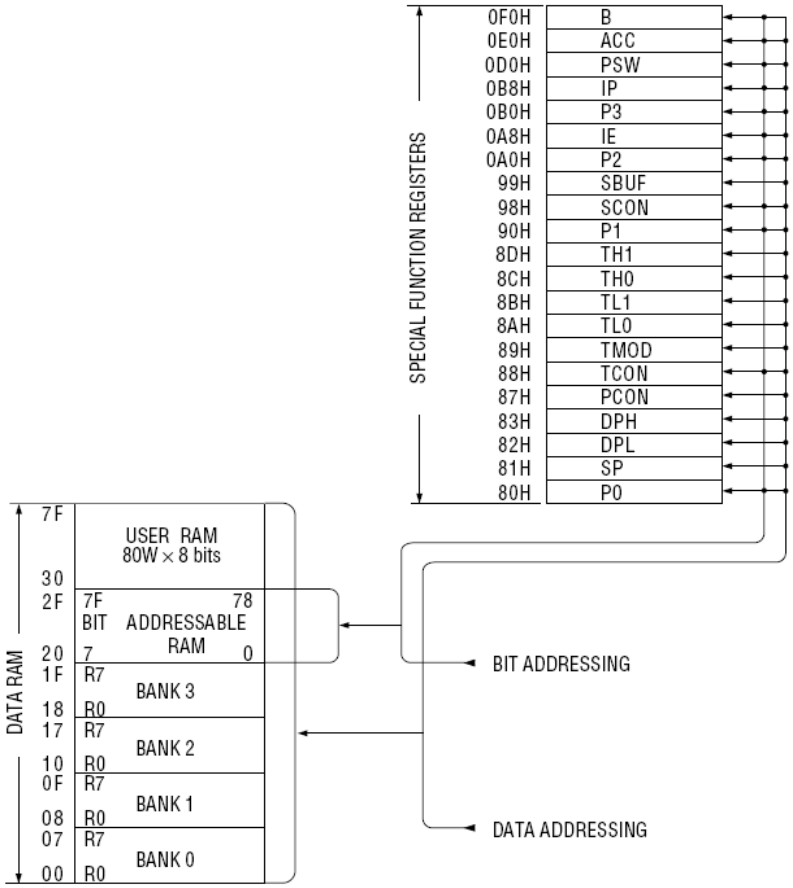


Az EFM8BB3 belső architektúrája

A további működés megértéséhez nézzük a 8051-es mikrokontroller memóriatérképét ! A programmemóriában az integrált belső memória felső határáig ezt használja a CPU, amennyiben /EA=0 vezérléssel nem tiltjuk ezt le. A felső határt követően automatikusan aktiválódik a külső memória interfész,



A 8051-es memóriatérképe



A 0..FFh belső adatmemória (IRAM és SFR) tartomány a 8051-ben

Az adatmemória esetében látni fogjuk, hogy a 8051-es utasításaiban „kódolva van”, melyik adatterülethez kívánunk hozzáférni. A belső RAM mindenképpen a 0..0x7F (későbbi verziókban már 0..0xFF) címtartományban helyezkedik el, a portok vezérlését lehetővé tevő SFR-ek (Special Function Register) minden esetben a 0x80..0xFF címtartományban találhatók. Ezzel az eltérő címzési mód használata miatt teljesen átlapolódhat a külső RAM, aminek legtermészetesebb képviselője pl. a mai 8051-es vezérlőkben az integrált ún. XRAM (tipikusan 4 kbájt). Ennek megléte az alkalmazások 99%-ában teljesen feleslegessé teszi további külső RAM használatát.

A 8051-es architektúra sikere meghatározó mértékben a fejlesztők igen fontos felismerésein alapul:

a) A kisteljesítményű vezérlők esetében meghatározó fontosságú a programmodulok (szubrutinok, megszakítási rutinok) nagysebességű váltásának lehetősége. A rendelkezésre álló regiszterek száma, regiszterbankok alkalmazása nagymértékben tudja segíteni a mikrokontrollerek ezen képességeit (a váltást lassító, sok memóriát igénylő veremműveletek elkerülésével).
b) A mikrokontrollerek világában felesleges a memória és az I/O címtartomány elkülönítése és külön utasításokkal (MOV ill. IN/OUT) történő megszólítása. Az integrált perifériák elég gyorsak ahhoz, hogy időkésés nélkül reagáljanak a hozzáférésekre, a fejlesztők által korábban is gyakran használt „memóriába ágyazott I/O” technika fenntartás nélkül elfogadható egységeink kezelésére. Cserében valamennyi címzési mód ezekre is használható. (A Motorola processzorok megjelenésük óta ezt a technikát alkalmazták).
c) Mind az alkalmazói programjainkban, mind a perifériák kezelésénél szükségünk van egy bites „bitváltozókra” (boolean). Programjainkban emiatt vagy bájtokat használunk egy-egy bit kezelésére, vagy fáradtságos és időigényes munkával „ÉS-elgetjük-VAGY-olgatjuk” bitjeinket egy-egy bájt kezelésével. Teljesen hasonló a perifériák kezelése is: a vezérlőszavakhoz és a portokhoz legtöbbször egy-egy bit lekérdezése vagy beállítása miatt fordulunk, ezeket is csak maszkolással (olvasás) vagy read-modify-write műveletekkel (írás) tudjuk megvalósítani.

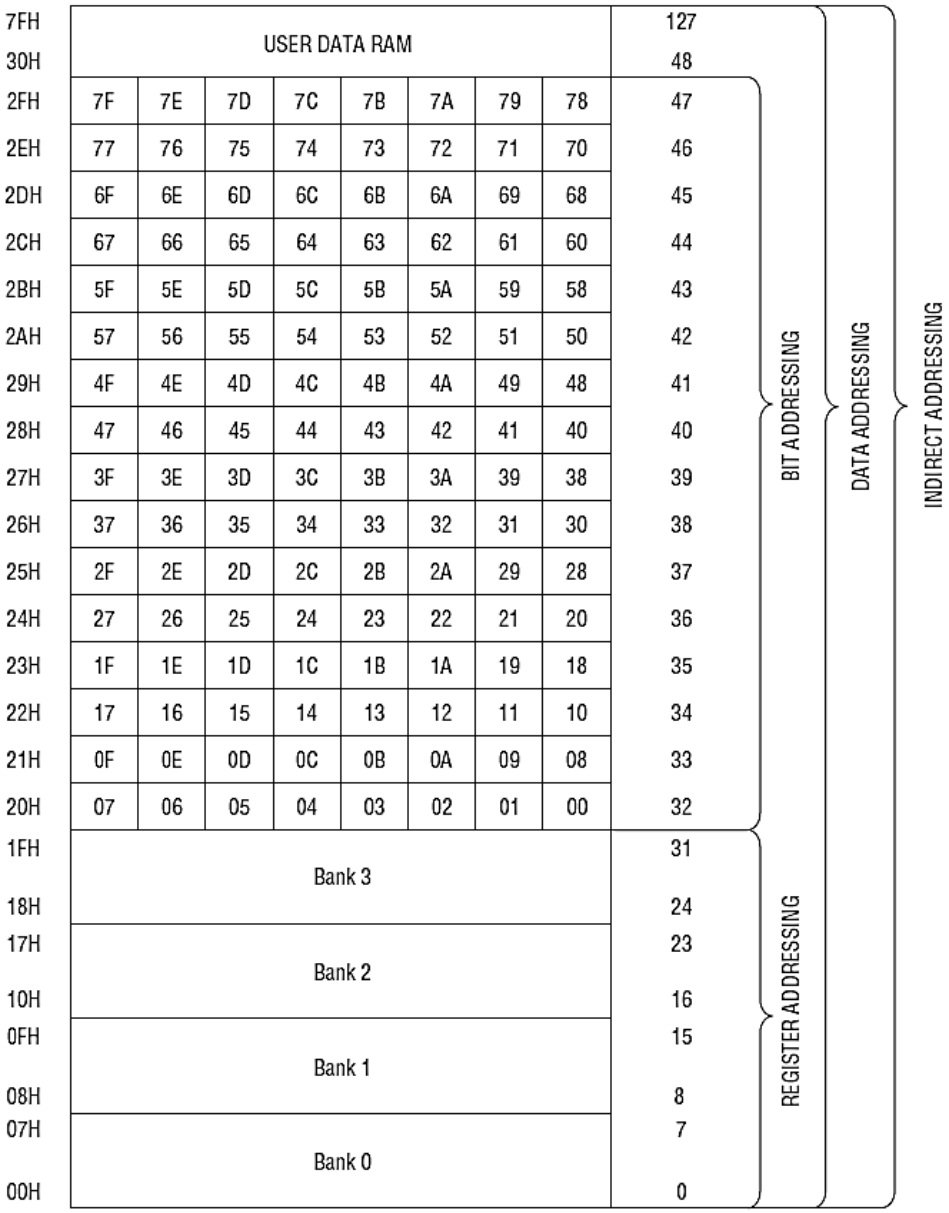
Fentiek figyelembevétele azt eredményezte, hogy a fejlesztők a belső RAM memória (IRAM) címkiosztásánál a következő „cselhez” folyamodtak: a 0..0x7F címtartományt (összesen 128 bájt) a következő részekre osztották fel:

- A 0..0x1F címtartományban (32 bájt) a CPU által közvetlenül kezelt 8 db regiszter (R0..R7) található meg 4-4 példányban, 4 db ún. regiszterbank formájában. Ezek közül mindig csak 1 bank látható a processzor számára, a PSW (Program Status Word), népszerűbb nevén a „flagek” 2 bitje szabja meg, melyik bankot használjuk éppen. Ily módon pl. megszakítás és a háttéralkalmazás, vagy taszkok közötti váltáskor egyetlen utasítással „kontextusváltást” tudunk csinálni a regiszterek fáradtságos mentése nélkül. Szükség esetén a regiszterek többszörözésére is használható a módszer.
- A 0x20..0x2F címtartomány 16 bájtját elérhetjük egyik címzési móddal a szokásos módon bájtonként is, de bitcímzéses címzési módban a 0..0x7F címeket

sorban hozzárendelve a memória 16 x 8 = 128 bitjéhez közvetlenül is manipulálhatjuk (lekérdezhetjük, állíthatjuk, negálhatjuk) őket. Ez a bitek kezelését legalább 3-4 szeresére gyorsítja fel, számuk a mikrokontrollerekben szokásos alkalmazások méreteihez elegendő.

- A 0x30..0x7F címtartományban található 80 bájt szokásos RAM-ként használható a programok számára.

A fentieknek megfelelő belső RAM memória (IRAM) címkiosztása tehát a következő:



A belső (IRAM) terület címkiosztása a 8051-es mikrokontrollerekben

Ez a „trükk” folytatódik a perifériák kezelésére hivatott SFR-ek esetében is. Az alapelv itt a következő: a címtartományuk 0x80..0xFF (128 bájt), de minden 0-ra és 8-ra végződő című regiszter (összesen 16 db lehetséges) bitcímzéssel történő elérése esetén bitenként is manipulálhatjuk őket a 0x80..0xFF (bit-)címtartományban.

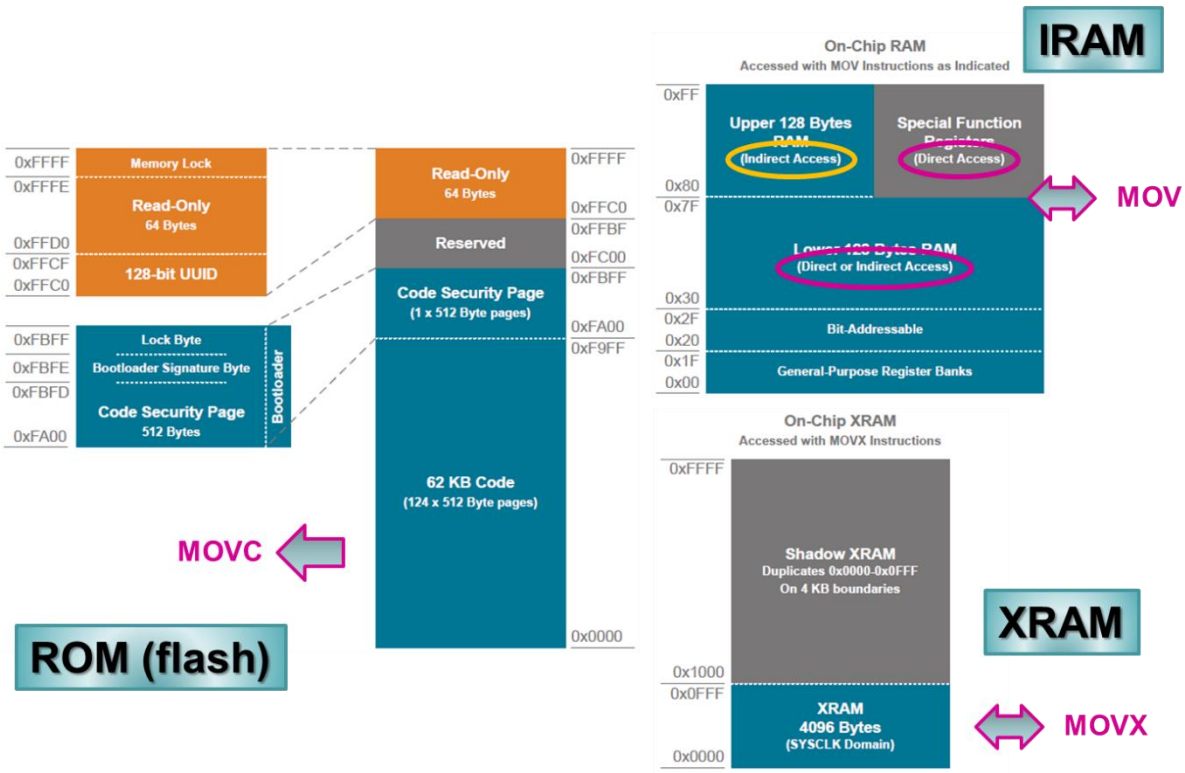
Data Address	(MSB)			Bit Address				(LSB)	Special Function Register Symbol
0F0H	F7	F6	F5	F4	F3	F2	F1	F0	B
0E0H	E7	E6	E5	E4	E3	E2	E1	E0	ACC
	CY	AC	F0	RS1	RS0	OV	F1	P	
0D0H	D7	D6	D5	D4	D3	D2	D1	D0	PSW
				PS	PT1	PX1	PT0	PX0	
0B8H	—	—	—	BC	BB	BA	B9	B8	IP
0B0H	B7	B6	B5	B4	B3	B2	B1	B0	P3
	EA			ES	ET1	EX1	ET0	EX0	
0A8H	AF	—	—	AC	AB	AA	A9	A8	IE
0A0H	A7	A6	A5	A4	A3	A2	A1	A0	P2
99H	Not Bit Addressable								SBUF
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI	
98H	9F	9E	9D	9C	9B	9A	99	98	SCON
90H	97	96	95	94	93	92	91	90	P1
8DH	Not Bit Addressable								TH1
8CH	Not Bit Addressable								TH0
8BH	Not Bit Addressable								TL1
8AH	Not Bit Addressable								TL0
89H	Not Bit Addressable								TMOD
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	
88H	8F	8E	8D	8C	8B	8A	89	88	TCON
87H	Not Bit Addressable								PCON
83H	Not Bit Addressable								DPH
82H	Not Bit Addressable								DPL
81H	Not Bit Addressable								SP
80H	87	86	85	84	83	82	81	80	P0

Az SFR címtartomány a 8051-es mikrokontrollerekben

Megjegyzések:

- A bitkezelő utasításoknál látni fogjuk, hogy összesen 256 bit címezhető meg az utasításokban: 128 IRAM-bit (0..7Fh) és 128 SFR-bit (80h-0FFh).
- Az SFR tartományban található a kontroller néhány fontos belső regisztere is, ezek a veremmutató (SP), a Data Pointer (DPH,DPL), a státusz (PSW), az akkumulátor (A) és annak segédregisztere (B).

Akár hisszük, akár nem, fentiek még bővíthetők is: az EFM8BB3 családban a jóval nagyobb memória- és perifériakészlet a következő módon illeszthető a fenti koncepcióba:



Az EFM8BB3 mikrokontroller család memóriatérképe

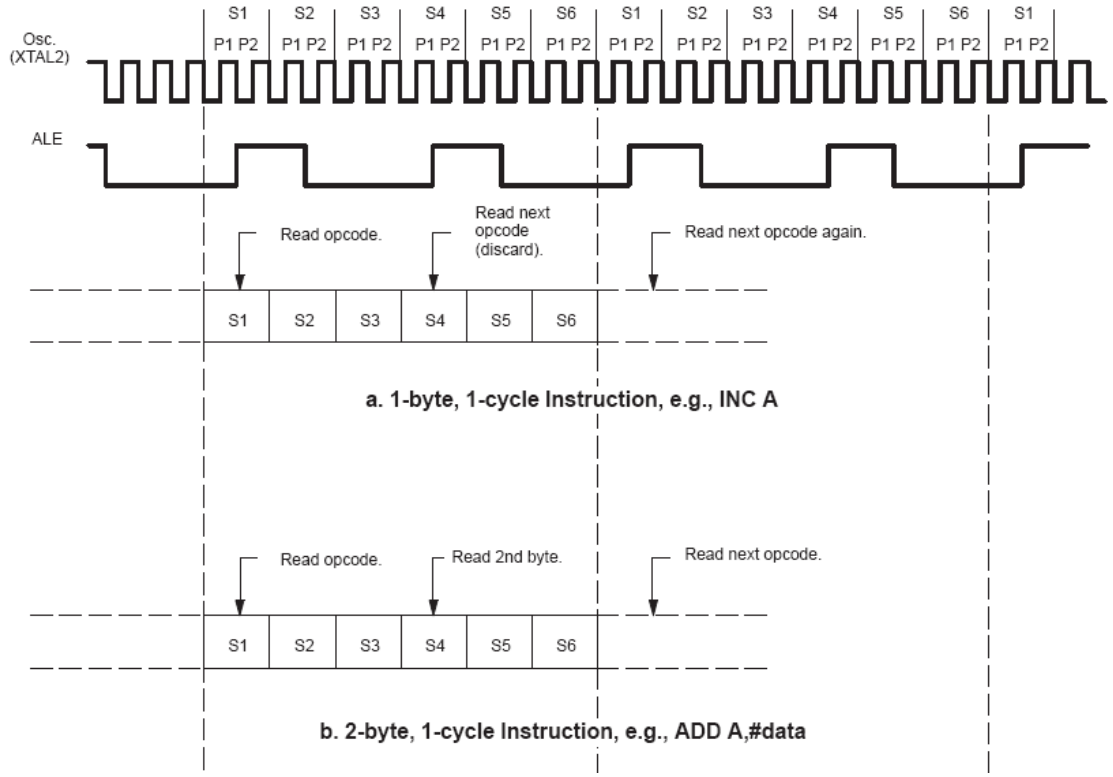
- A 16, 32 vagy 64 kbájt flash memória elfoglalja a teljes programkód címtartományt. A tetején található egy 1,5 kbájtos, speciális célokra használható nem felejtő memóriarész (gyári bootloader, védelmi, azonosító és titkosító kódok, a flashez hasonlóan MOVC utasítással olvasható).
- A beépített XRAM csak külső memóriaként (MOVX utasítással) érhető el.
- A belső RAM (IRAM) duplája az eredeti 8051-ben található RAM-nak, tehát átlapolódik az SFR-ekkel. A címzési mód (direkt/indirekt) dönti el, melyikhez fordulunk.
- A nagyszámú periféria több SFR regiszter használatát teszi szükségessé. Segédregiszterekkel és lapozási technikával megteremtették összesen 256 x 128 SFR regiszter használatának lehetőségét. Az eredeti 8051-es architektúrában található regiszterek kompatibilitási okokból valamennyi lapon (ugyanott) látszanak.

Az architektúrális áttekintés végén (de nem utolsó sorban) megemlítjük, hogy a 8051-es mikrokontrollerek is támogatják az **alacsony fogyasztású** alkalmazásokat. Ezt a CPU két speciális üzemmódja biztosítja (melyet a felhasználó tetszőleges időpontban SFR vezérlőbitekkel állíthat be):

- IDLE módban a kontroller „elalszik”, azaz leállítja a program futását úgy, hogy valamennyi regisztere és a RAM területek tartalma is változatlan marad. Valamennyi periféria tovább működik, de a processzor áramfelvétele lényegesen lecsökken (az eredeti 8051 esetében kb. az 1/10-re). Alvó állapotából a kontrollert valamennyi megszakításkérés (vagy külső reset) felébreszti.
- POWER DOWN (STOP) módban a kontroller még az oszcillátorait is leállítja, tehát ilyenkor a perifériák is leállnak. Áramfelvétele mikroamperekre csökken. Ebből az állapotból a kontrollert csak külső reset tudja felébreszteni.

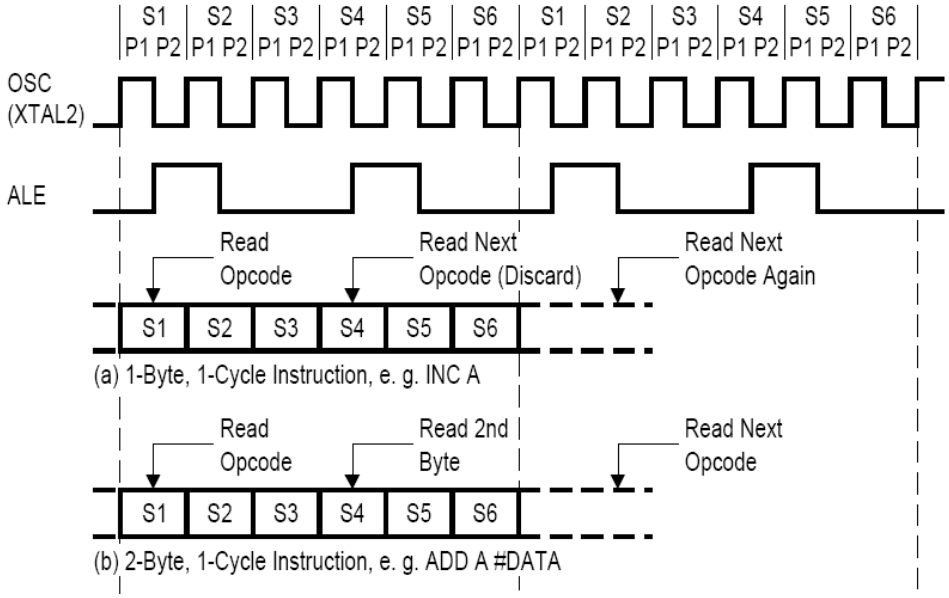
1.5.2 Buszciklusok

A 8051-es mikrokontroller megjelenésekor max. 12 MHz-es órajel-frekvenciával működött, memória hozzáférési ciklusait ún. gépi ciklusokba foglalták fejlesztői. Minden gépi ciklus pontosan 12 órajel-periódust foglalt magába, így 1 gépi ciklusos utasításait pontosan 1 μsec alatt (1 MIPS), 2 gépi ciklust igénylő utasításait 2 μsec alatt (0.5 MIPS) tudta elvégezni. Ezeken kívül összesen 2 db 4 gépi ciklusos utasítása volt (szorzás és osztás).



Az eredeti 8051-es gépi ciklus (M = 12T)

Minden gépi cikluson belül a mikrokontroller két memória hozzáférést tudott elvégezni. Egy bájtos utasítások esetén (pl. INC A) a második nem történt meg, az egy gépi ciklus alatt elvégezhető két bájtos utasítások (pl. ADD A,#data) operandusa ilyenkor került beolvasásra.
A műveletvégzés hamarosan lassúnak bizonyult. A kapuidők csökkenése és némileg bonyolultabb belső kialakítás révén hamarosan megjelent a 8051-es mikrokontrollerek „x2 módja”: egy vezérlő bit beállításával a kontroller duplájára volt gyorsítható azáltal, hogy S-ciklusaihoz már nem két, hanem csak egyetlen órajel-periódust igényelt (ld. az Atmel, Infineon, Philips cégek 8051-es mikrokontrollereinek adatlapjait):



A 8051-es gépi ciklus X2 módban (M = 6T)

A Silicon Laboratories cég még ezzel sem elégedett meg. Pipeline műveletvégző egységet tervezett a 8051-es magba, és ezzel elérte, hogy 50 MHz-es működési frekvencia mellett is utasításai 70%-a 1 vagy 2 órajel-periódus alatt befejeződik.

Végrehajtási idő (T)	1	2	2/3	3	3/4	4	4/5	5	8
Utasítások száma	26	50	5	14	7	3	1	2	1

Műveletvégzési sebesség az EFM8BB3 mikrokontroller családnál

1.5.3 Programozási modell

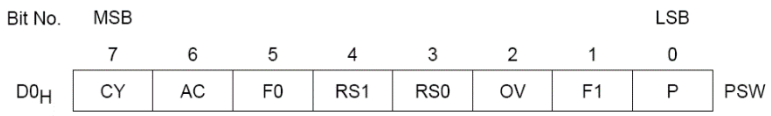
CPU regiszterek

A 8051-es alapvetően **akkumulátoros (A)** műveletvégző egységet tartalmaz, aritmetikai műveleteit mindig ebben a regiszterben végzi. Szorzáshoz és osztáshoz egy további **segédregisztert (B)** használ, amely az akkumulátorhoz hasonlóan az SFR tartományban található.

Az operandusok mozgatását és tárolását az **R0..R7 regiszterek** látják el. Ezekből 8-8 darab található a 4 regiszterbankban, melyek közül a státusszó vezérlőbitjeivel választhatunk.

További CPU regiszterek, melyekről tudnunk kell:

- a **státusszó (PSW)** az SFR tartományban található, felépítése a következő:



A 8051-es mikrokontroller státusszavának felépítése

Az egyes bitek jelentése a következő:

CY: Carry (átvitel)
AC: Auxiliary Carry – a tetrádok közötti átvitel (BCD műveletekhez)
F1, F0: General Purpose Flags – tetszőleges célra használhatjuk
RS1, RS0: Register Bank Select – a 4 regiszterbank közül választja ki az aktuálisat
OV: Overflow (túlcsordulás)
P: Parity – az akkumulátor mindenkori paritását jelzi, értéke = 1, ha az akkumulátorban az egyesek száma páratlan.

(Felhívjuk az olvasó figyelmét arra, hogy lényeges különbség van az előjel nélküli számokkal történő műveletvégzést támogató CY, és az előjeles, kettes komplemens számábrázolású számok aritmetikájához szükséges OV flagek között: 8 biten ábrázolt adatok esetében az előbbi jelzőbit a (255)↔(0) átfordulást, míg az utóbbi a (+127)↔(-128) közötti váltások jelzésére szolgál. Ezen túlmenően az AC flag a BCD számokkal történő műveletek elvégzéséhez szükséges. Beágyazott rendszerek alacsony szinten történő programozásához fentiek ismerete elengedhetetlen !)

- a 16 bites **Data Pointer (DPTR)** szintén az SFR tartományban található, a 16 bites címképzést segíti elő - elsősorban a kód- és a külső memóriák számára.
- a **veremmutató (SP)** 8 bites, az SFR tartományban található. Méretéből következik, hogy a verem csak a belső RAM memóriában építhető fel, töltéskor (PUSH) a mutató a növekvő címek irányába halad.

- a CPU **utasításszámlálója (PC)** a központi egységben található és az utasítások címképzésén túl a relatív ugrások, és a PC relatív adattöltés (MOVC) során is felhasználásra kerül.

Címzési módok

A mikrokontroller alapvetően 5 címzési módot használ. Ezek gyors megértéséhez foglaljuk össze az egyes címzési módok legfontosabb tulajdonságait:

- **Regiszter címzésnek** nevezzük azt a módot, amelyben a forrás- vagy a céloperandus a CPU regiszterek egyike. Lévén a memóriatérkép alapján nem egyértelmű, a regiszterekhez soroljuk az aktuális regiszterbank regisztereit (R0..R7), az akkumulátort és segédregiszterét (A és B) valamint a Data Pointert (DPTR).
- **Direkt címzés** használata esetén a műveleti kódot (opcode) követő második bájt az operandus (direkt) címe. A mindössze 256 cím lehetséges címzettjei az IRAM alsó 128 bájtja (0..0x7F) vagy az SFR regiszterek (0x8x..0xFF) lehetnek. Hasonlóan direkt módon címezzük a bitoperandusokat is, őket az opcode különíti el a fenti memóriaterületektől.
- **Közvetlen (immediate) címzésnek** az utasításba beépülő konstans adatértékek használatát nevezzük, melyek szintén az opcode-ot követő 1 vagy 2 bájt (DPTR töltése esetén) lehet.
- **Regiszter indirekt címzésre** vagy a két első 8 bites adatregisztert (R0 vagy R1), vagy a 16 bites Data Pointert (DPTR) használhatjuk. Segítségükkel mind a belső (MOV), mind a külső (MOVX) memória címzése lehetséges. 8 bites adatregiszter használata esetén a cím belső memória címzésére elegendő, külső memória esetén a magasabb helyiértéket továbbító P2 port állapota változatlan marad.
- **Bázis regiszter indexelt címzés** csak a MOVC utasításhoz használható (kódmemóriában tárolt konstansok elérésére).

Címzési mód	Mire alkalmazható	Példa
Regiszter címzés	R0..R7 (a kiválasztott regiszterbankban) A, B, CY, DPTR	MOV A,R0
Direkt címzés	IRAM alsó 128 bájt, SFR regiszterek	MOV A,7Fh
Immediate címzés	Program memória	MOV A,#100
Regiszter indirekt címzés	IRAM (@R1, @R0, SP) XRAM (@R1, @R0, @DPTR)	MOV A,@R0
Bázis regiszter + index regiszter címzés	Program memória (@A+DPTR, @A+PC)	MOVC @A+DPTR

A 8051-es mikrokontroller címzési módjai

1.5.4 Az utasításkészlet

A 8051-es mikrokontroller összesen 111 különböző utasítással programozható. Ez a szám önmagában sem túlzottan magas, főleg, ha figyelembe vesszük, hogy ebben a számban már a különböző címzési módok is benne foglaltatnak. Az utasítás mnenonikja 45 utasításnak különböző.

Aritmetikai utasítások (8 db):	ADD, ADDC, SUBB, INC, DEC, MUL, DIV, DA
Logikai utasítások (10 db):	ANL, ORL, XRL, CLR, CPL, RL, RLC, RR, RRC, SWAP
Bitmanipulációs utasítások (8 db):	CLR, SETB, CPL, JC, JNC, JB, JNB, JBC
Adatmozgató utasítások (6 db):	MOV, MOVC, MOVX, PUSH, POP, XCH
Vezérlésátadó utasítások (13 db):	ACALL, LCALL, RET, RETI, AJMP, LJMP, SJMP, JMP, JZ, JNZ, CJNE, DJNZ, NOP

(Bár az utasításkészlet kialakításakor valószínűleg nem ez volt az elsődleges szempont, az utasításstruktúra akár még a RISC elveket sem sértené alapjaiban.)

Fontos mindig szem előtt tartanunk, hogyan állítódnak a státusszó egyes bitjei a különböző utasítások hatására. Alapelv: az átvitelt jelző flageket (CY, OV, AC) az aritmetikai utasítások állítják, a szorzás/osztás az AC-t már nem állítja (nem alkalmas BCD műveletek elvégzésére), a léptető és a logikai utasítások értelemszerűen csak a CY-t változtatják. A bitmanipulációs utasítások (SETB, CLR, CPL, MOV) a CY flaget közvetlenül képesek kezelni. Külön ki kell hangsúlyozzuk: bár Z flag nincs a státusszóban, az akkumulátor (JZ, JNZ), vagy az utasításban szereplő egyéb regiszter vagy indirekten címzett adat (CJNE, DJNZ) 0 volta is lehet elágazási feltétel.

Instruction	Flag			Instruction	Flag		
	CY	OV	AC		CY	OV	AC
ADD	X	X	X	SETB C	1		
ADDC	X	X	X	CLR C	0		
SUBB	X	X	X	CPL C	X		
MUL	0	X		ANL C,bit	X		
DIV	0	X		ANL C,/bit	X		
DA	X			ORL C,bit	X		
RRC	X			ORL C,/bit	X		
RLC	X			MOV C,bit	X		
CJNE	X						

A státusz szó (PSW) jelzőbitjeinek kezelése (CY,OV,AC)

A következőkben egy rövid áttekintő táblázatot láthatunk a 8051-es mikrokontrollerek utasításairól:

Mnemonic	Description	Bytes	Clock Cycles
Arithmetic Operations			
ADD A, Rn	Add register to A	1	1
ADD A, direct	Add direct byte to A	2	2
ADD A, @Ri	Add indirect RAM to A	1	2
ADD A, #data	Add immediate to A	2	2
ADDC A, Rn	Add register to A with carry	1	1
ADDC A, direct	Add direct byte to A with carry	2	2
ADDC A, @Ri	Add indirect RAM to A with carry	1	2
ADDC A, #data	Add immediate to A with carry	2	2
SUBB A, Rn	Subtract register from A with borrow	1	1
SUBB A, direct	Subtract direct byte from A with borrow	2	2
SUBB A, @Ri	Subtract indirect RAM from A with borrow	1	2
SUBB A, #data	Subtract immediate from A with borrow	2	2
INC A	Increment A	1	1
INC Rn	Increment register	1	1
INC direct	Increment direct byte	2	2
INC @Ri	Increment indirect RAM	1	2
DEC A	Decrement A	1	1
DEC Rn	Decrement register	1	1
DEC direct	Decrement direct byte	2	2
DEC @Ri	Decrement indirect RAM	1	2
INC DPTR	Increment Data Pointer	1	1
MUL AB	Multiply A and B	1	4
DIV AB	Divide A by B	1	8
DA A	Decimal adjust A	1	1

	Logical Operations		
ANL A, Rn	AND Register to A	1	1
ANL A, direct	AND direct byte to A	2	2
ANL A, @Ri	AND indirect RAM to A	1	2
ANL A, #data	AND immediate to A	2	2
ANL direct, A	AND A to direct byte	2	2
ANL direct, #data	AND immediate to direct byte	3	3
ORL A, Rn	OR Register to A	1	1
ORL A, direct	OR direct byte to A	2	2
ORL A, @Ri	OR indirect RAM to A	1	2
ORL A, #data	OR immediate to A	2	2
ORL direct, A	OR A to direct byte	2	2
ORL direct, #data	OR immediate to direct byte	3	3
XRL A, Rn	Exclusive-OR Register to A	1	1
XRL A, direct	Exclusive-OR direct byte to A	2	2
XRL A, @Ri	Exclusive-OR indirect RAM to A	1	2
XRL A, #data	Exclusive-OR immediate to A	2	2
XRL direct, A	Exclusive-OR A to direct byte	2	2
XRL direct, #data	Exclusive-OR immediate to direct byte	3	3
CLR A	Clear A	1	1
CPL A	Complement A	1	1
RL A	Rotate A left	1	1
RLC A	Rotate A left through Carry	1	1
RR A	Rotate A right	1	1
RRC A	Rotate A right through Carry	1	1
SWAP A	Swap nibbles of A	1	1

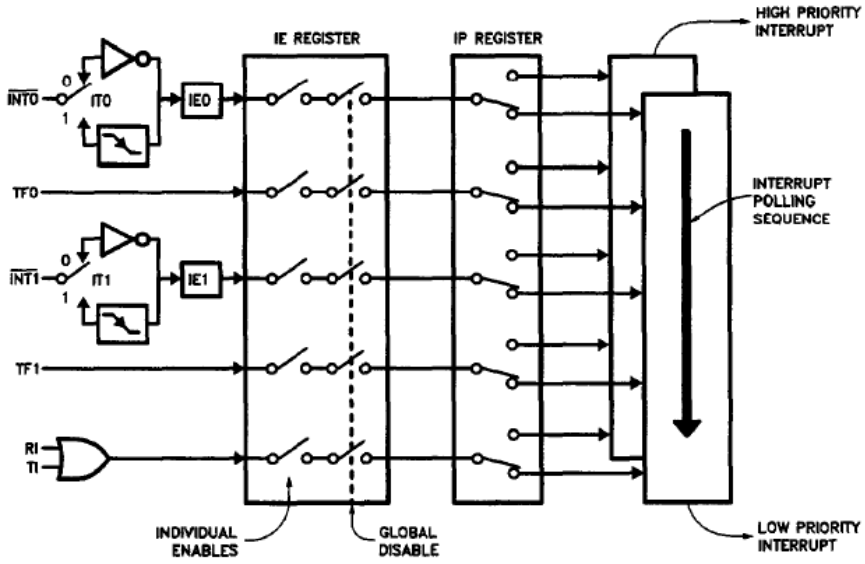
	Boolean Manipulation		
CLR C	Clear Carry	1	1
CLR bit	Clear direct bit	2	2
SETB C	Set Carry	1	1
SETB bit	Set direct bit	2	2
CPL C	Complement Carry	1	1
CPL bit	Complement direct bit	2	2
ANL C, bit	AND direct bit to Carry	2	2
ANL C, /bit	AND complement of direct bit to Carry	2	2
ORL C, bit	OR direct bit to carry	2	2
ORL C, /bit	OR complement of direct bit to Carry	2	2
MOV C, bit	Move direct bit to Carry	2	2
MOV bit, C	Move Carry to direct bit	2	2
JC rel	Jump if Carry is set	2	2/3
JNC rel	Jump if Carry is not set	2	2/3
JB bit, rel	Jump if direct bit is set	3	3/4
JNB bit, rel	Jump if direct bit is not set	3	3/4
JBC bit, rel	Jump if direct bit is set and clear bit	3	3/4

Mnemonic	Description	Bytes	Clock Cycles
Data Transfer			
MOV A, Rn	Move Register to A	1	1
MOV A, direct	Move direct byte to A	2	2
MOV A, @Ri	Move indirect RAM to A	1	2
MOV A, #data	Move immediate to A	2	2
MOV Rn, A	Move A to Register	1	1
MOV Rn, direct	Move direct byte to Register	2	2
MOV Rn, #data	Move immediate to Register	2	2
MOV direct, A	Move A to direct byte	2	2
MOV direct, Rn	Move Register to direct byte	2	2
MOV direct, direct	Move direct byte to direct byte	3	3
MOV direct, @Ri	Move indirect RAM to direct byte	2	2
MOV direct, #data	Move immediate to direct byte	3	3
MOV @Ri, A	Move A to indirect RAM	1	2
MOV @Ri, direct	Move direct byte to indirect RAM	2	2
MOV @Ri, #data	Move immediate to indirect RAM	2	2
MOV DPTR, #data16	Load DPTR with 16-bit constant	3	3
MOVC A, @A+DPTR	Move code byte relative DPTR to A	1	3
MOVC A, @A+PC	Move code byte relative PC to A	1	3
MOVX A, @Ri	Move external data (8-bit address) to A	1	3
MOVX @Ri, A	Move A to external data (8-bit address)	1	3
MOVX A, @DPTR	Move external data (16-bit address) to A	1	3
MOVX @DPTR, A	Move A to external data (16-bit address)	1	3
PUSH direct	Push direct byte onto stack	2	2
POP direct	Pop direct byte from stack	2	2
XCH A, Rn	Exchange Register with A	1	1
XCH A, direct	Exchange direct byte with A	2	2
Program Branching			
ACALL addr11	Absolute subroutine call	2	3
LCALL addr16	Long subroutine call	3	4
RET	Return from subroutine	1	5
RETI	Return from interrupt	1	5
AJMP addr11	Absolute jump	2	3
LJMP addr16	Long jump	3	4
SJMP rel	Short jump (relative address)	2	3
JMP @A+DPTR	Jump indirect relative to DPTR	1	3
JZ rel	Jump if A equals zero	2	2/3
JNZ rel	Jump if A does not equal zero	2	2/3
CJNE A, direct, rel	Compare direct byte to A and jump if not equal	3	3/4
CJNE A, #data, rel	Compare immediate to A and jump if not equal	3	3/4
CJNE Rn, #data, rel	Compare immediate to Register and jump if not equal	3	3/4
CJNE @Ri, #data, rel	Compare immediate to indirect and jump if not equal	3	4/5
DJNZ Rn, rel	Decrement Register and jump if not zero	2	2/3
DJNZ direct, rel	Decrement direct byte and jump if not zero	3	3/4
NOP	No operation	1	1

A 8051-es család utasításkészlete
(az utasítások végrehajtási ideje az EFM8BB3 család pipeline műveletvégzésének megfelelő)

1.5.5 A megszakítások kezelése

A 8051-es mikrokontrollerek kétszintű megszakításrendszerrel rendelkeznek, a következő alapelvek szerint:



A 8051-es mikrokontroller megszakításrendszere

A megszakításkérések származhatnak külső (INT0, INT1) vagy belső forrásból (Timer0, Timer1, UART). A külső megszakításkérések számára a TCON regiszterben adhatjuk meg, hogy azok szint-, vagy élvezéreltek legyenek-e. Minden megszakításkérő forrás külön-külön engedélyezhető/tiltható az IE regiszterben, illetve ennek legfelső bitje szolgál globális megszakításkérés engedélyezésként vagy tiltásként.

Az IP (Interrupt Priority) regiszterben adhatjuk meg, hogy a két lehetséges (magas és alacsony) megszakításkezelési szint melyikére kívánjuk helyezni megszakításkérésünket. Bármely megszakításkérést csak nála magasabb szintű megszakításkérés szakíthat meg (tehát csak a magas az alacsonyt). Az ugyanazon szinten szereplő egyidejűleg beérkező kérések sorrendjét a fix lekérdezési sorrend (polling sequence) dönti el.

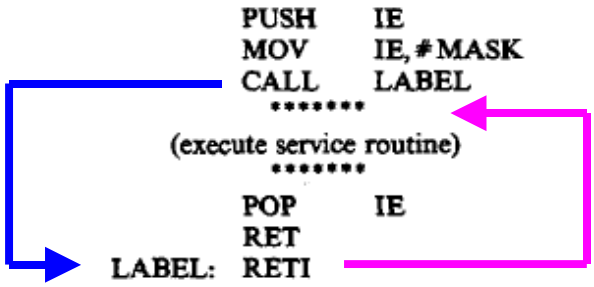
A megszakításkérés hatására a mikrokontroller elmenti a verembe a PC értékét - csak azt, a státusz és a regiszterek mentéséről a megszakítási rutinnak kell gondoskodnia. Itt használható fel rendkívül jól a bankok közötti egyszerű váltás lehetősége, ami feleslegessé teheti az egyes regiszterek mentését. A visszatérési cím mentését követően a megszakításnak megfelelő ún. megszakítási ugrótábla fix címeinek egyikére adja a vezérlést. Ide a felhasználói programnak olyan ugró utasítást kell elhelyeznie, ami a tényleges kiszolgáló rutinra mutat. A megszakítási rutin végén RETI utasításnak kell szerepelnie, ez törli a processzorban az adott prioritási szinthez tartozó foglaltság-jelző flip-flopot és visszatér a megszakítási pontot követő utasítás végrehajtásához.

Interrupt Source	Vector Address
IE0	0003H
TF0	000BH
IE1	0013H
TF1	001BH
RI & TI	0023H
TF2 & EXF2	002BH

Az „ős” 8051-es megszakítási ugrótáblája

Ez a megszakítási tábla a későbbi családtagok esetében – a megszakításkérő periféria elemek megnövekedett száma miatt – több elemet tartalmaz (az EFM8BB3 esetében már 20-at), kompatibilitási okokból azonban a fenti perifériák mindig ugyanezen a helyen kell szerepeljenek.

Maga a gyártó Intel cég is megemlíti, hogy a fenti kétszintű megszakításrendszere némi fáradtsággal még bővíthető. Amennyiben egy megszakítási rutin elejére elhelyezzük a következő kódrészletet:



A megszakítási szintek bővítése

Ekkor a kérdéses megszakítás a LABEL címkén található „dummy” RETI utasítás segítségével törli a mikrokontroller belső foglaltság jelző flagjét, ily módon megszakíthatóvá teszi saját magát a saját (vagy magasabb) megszakítási szintje számára. Azt, hogy ezt ne mindenki számára tegyük lehetővé, az IE regiszter mentésével és átállításával tudjuk biztosítani.

1.6 Az ARM mikrokontroller architektúra

Az ARM architektúra (az Advanced RISC Machine elnevezés kezdőbetűi, eredeti elnevezése az Acorn RISC Machine) egy 32-bites, az ARM Limited cég által fejlesztett RISC CPU architektúra, amely ma igen népszerű a beágyazott rendszerek körében. Energiatakarékosságuk miatt az ARM architektúrájú CPU-k a vezetők a hordozható elektronikai piacon – ahol az alacsony energiafogyasztás fontos tervezési szempont.

Az ARM architektúra fejlesztése 1983-ban kezdődött az Acorn Computers Ltd. (Cambridge, UK) laboratóriumában, ahonnan eredeti elnevezését is kapta. Ez a cég akkoriban az Acorn Proton (BBC Micro márkanéven értékesített) személyi számítógépek fejlesztőjeként és gyártójaként volt ismert, melyek egy az angol kormányzat által támogatott projekt keretében készültek azzal a céllal, hogy minden iskolába eljuttassák a számítógépeket. A tervezők egy, a MOS Technology Ltd. 6502-eséhez hasonló (emlékszünk? a 8 bites mikroprocesszorok egyik favoritja, a Commodore gépek központi egységének, a 6510-nek az elődje), de annál nagyobb teljesítményű mikroprocesszort kívántak kifejleszteni. Az ekkoriban fejlesztés alatt álló Berkeley RISC kutatási projekt publikációi nagy hatással voltak az Acorn szakembereire, és úgy döntöttek, hogy egy saját 32 bites processzor fejlesztésébe fognak. Az Acorn RISC Machine projekt 1983-ban indult el Sophie Wilson és Steve Furber vezetésével, szilíciumgyártó partnerként a VLSI Technology cég csatlakozott hozzájuk.Az első eredmények 1985 áprilisában láttak napvilágot ARM1 név alatt, de ez a típus nem jutott el a sorozatgyártásig sem. Az első, gyártásban is megjelent típus az ARM2-es volt (1986), amely 32 bites adatbusszal rendelkezett, 26 bites címtartományt tudott kezelni (64 Mbájt), 27 db 32 bites regisztere volt. Valószínűleg az ARM2 volt a legegyszerűbb, használható 32-bites mikroprocesszor a világon, huzalozott logikán alapuló vezérlőegysége mindössze kb. 30 000 tranzisztort tartalmazott, és az első RISC alapú, otthoni felhasználásra szánt kereskedelmi számítógép, az 1987-ben elkészült Acorn Archimedes központi egységeként szolgált.

Az Acorn Archimedes számítógép

Az ARM2 a mérések alapján jobb számítási teljesítményt nyújtott, mint az Intel 80286 CISC processzor a maga több, mint 130 ezer tranzisztorával (összehasonlításként: a Motorola 6 évvel korábban megjelent 68000-es mikroprocesszora kb. 70 000 tranzisztort tartalmazott és mikroprogramozott volt). Gyorsítótárat (cache) nem tartalmazott. Utódját, az ARM3-at, 4 Kbájt méretű cache-sel készítették, amely a teljesítményét tovább növelte.

Az 1990-ben az Apple Computer, a VLSI Technologies és az Acorn együttműködésbe kezdett az ARM új változatának kidolgozásához, így jött létre az Advanced RISC Machines Ltd. (innen a processzorok második, későbbi elnevezése, a céget ma ARM Holdings néven ismerjük). Az együttműködés hátterében az állt, hogy az Apple használni szerette volna az ARM technológiát, de nem akarta termékeit a személyi számítógépek piacán ekkor konkurensnek számító Acorn licenszére építeni. A cég tőkéjét az Apple finanszírozta, az Acorn adta a szakembereket (ez egy 12 fős mérnökcsapatot jelentett), a VLSI pedig az gyártási eszközöket biztosította. A közös munka eredményeként 1991-ben született meg az ARM6 mikroarchitektúra, amely számos újítást tartalmazott az utasításkészletben is (pl. a címbusz 32 bites lett), de a mérete csupán kis mértékben, 35 ezer tranzisztorra növekedett. Az Apple az ARM6 alapú ARM610-es chip köré építette az 1993-ban bemutatott Apple Newton nevű PDA-ját, amely kezdeti hibái miatt viszont nem lett sikeres termék.

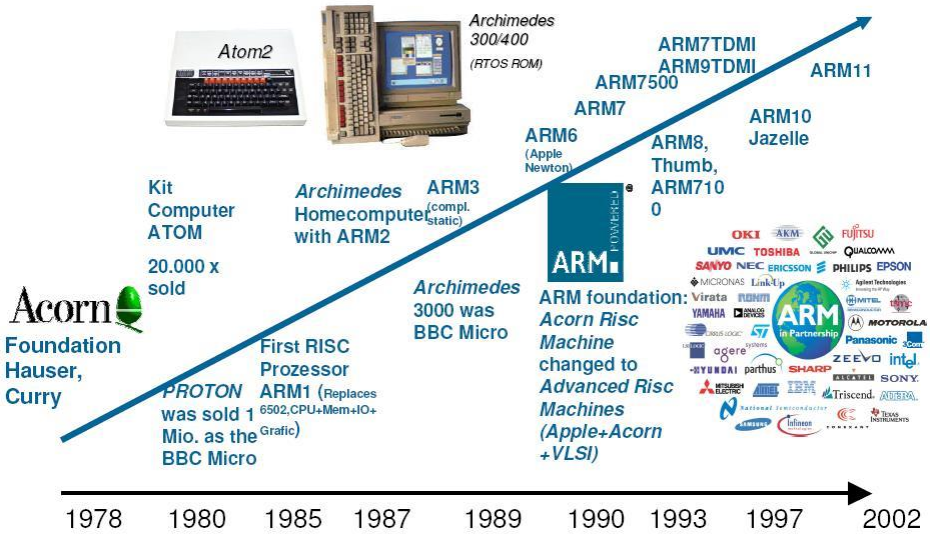
Ekkoriban döntött az ARM vezetése a saját chip-ek gyártása helyett az IP licenszek eladására épülő üzleti modell mellett, hogy ne egy-egy konkrét termék sikeressége határozza meg a jövőjüket. Az ARM processzorok licenszét számos chipgyártó cég kezdte vásárolni, amelyek egy egyszeri licenszdíjért és a legyártott lapkák után fizetett jogdíjért cserébe használhatták a kész processzorterveket. A partnerek számára ez a modell jelentősen lecsökkentette termékeik piacra jutási idejét, az ARM-nak pedig megalapozta az üzleti jövőjét. Az ARM Ltd. tehát saját maga nem gyárt mikrovezérlőket, hanem alapvetően két dolgot fejleszt:

1. utasításkészlet-architektúrákat (ezeket generációknak nevezik és ARMvX azonosítóval jelölik, pl. ARMv4, ARMv5, a legújabb ARMv9-ig),
2. az ezekre épülő processzor mikroarchitektúrákat (pl. az ARM7TDMI az ARMv4T-re, az ARM9-es processzorok típustól függően az ARMv4T vagy az ARMv5TE architektúrára épülnek).

Erre az időszakra esett az egyre kisebb méretben megvalósítható mobiltelefonok elterjedése. Az ARM egyik jelentős licenszpartnere a Texas Instruments volt, amely a Nokia beszállítójaként javasolta, hogy a következő Nokia telefont a TI ARM alapú rendszertervére építsék. A 32 bites architektúra memória-illesztésének sajátosságaiból fakadó magasabb várható előállítási költségek miatt a Nokia az ARM processzor alkalmazása ellen érvelt. Ekkor az ARM egy új, 16 bites utasításkészlet-verzióval állt elő (Thumb, erre később visszatérünk), ami csökkentette a memória alrendszerrel szemben támasztott követelményeket. Az új, kibővített architektúrára épülő processzor az 1994-ben megjelent ARM7TDMI volt, amely a TI által licenszelve az ikonikus Nokia 6110 telefon központi egysége lett, és ezt használták a következő, kiemelkedően népszerű Nokia 3210 és 3310 készülékekben is.

A Nokia 6110, 3210 és 3310 mobil telefonjai

Az ARM11-es processzorok az ARMv6 architektúrára épülnek (ezeket 2002-ben mutatták be). A később tárgyalásra kerülő Cortex-M4 mag az ARMv7E-M-et valósítja meg, a négymagos Cortex-A53 (ARMv8-A) a közismert Raspberry Pi 3 központi egysége, a Cortex-A72 pedig a Raspberry Pi 4 moduloké.



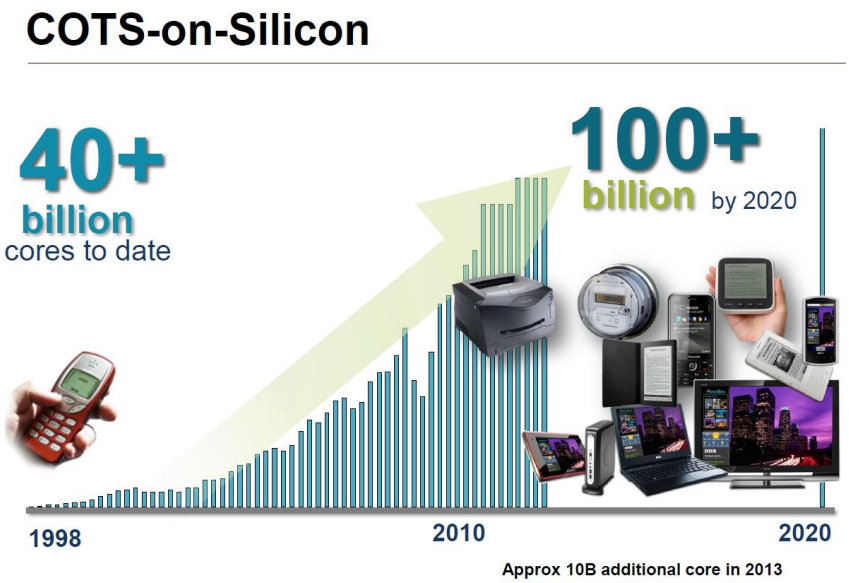
Az ARM család fejlődésének indulása az első két évtizedben

Az ARM utasításkészlet-architektúrákat generációkba sorolják, és ARMvX néven hivatkoznak rájuk, ahol X a verzió sorszáma. A historikusnak számító ARMv1 (ARM1 processzor), ARMv2 (ARM2, ARM3 processzorok) és ARMv3 (ARM6, ARM7 processzorok) architektúrák már nem támogatottak. Jelenleg az ARMv7, ARMv8 és ARMv9 architektúrákra épülnek a modern ARM processzormagok. Az alábbi táblázat áttekintést nyújt az egyes ISA változatokról, és a rájuk épülő, ARM Holdings és más cégek által fejlesztett processzormagokról (a teljesség igénye nélkül).

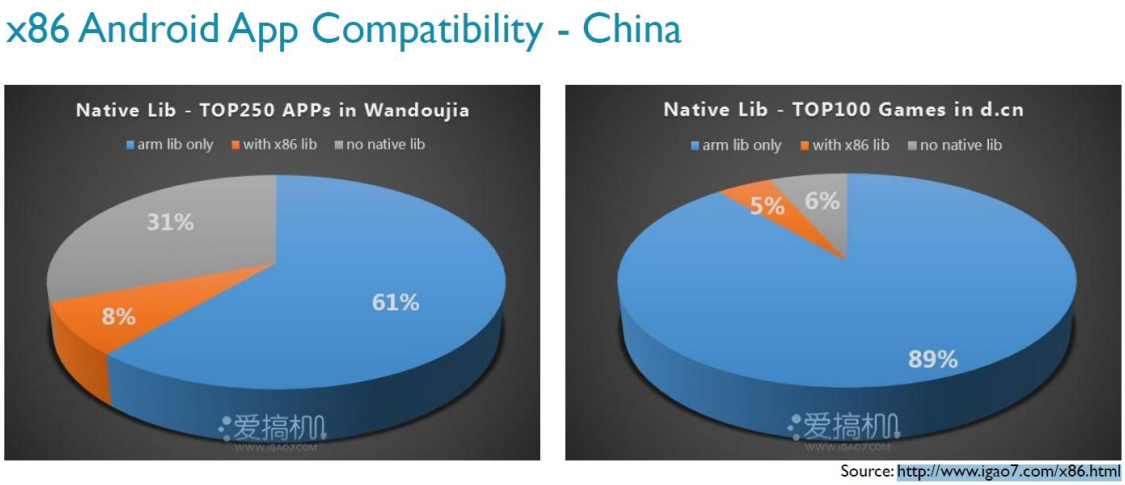
Év	ISA	Processzormagok (ARM Holdings)	Processzormagok (3rd party)
1985	ARMv1	ARM1	
1987	ARMv2	ARM2, ARM3	Amber
1991	ARMv3	ARM6, ARM7	
1994	ARMv4T	ARM7TDMI, ARM9TDMI	
1996	ARMv4	ARM8	DEC/Intel StrongARM, Faraday FA526
1999	ARMv5	ARM7EJ, ARM9E, ARM10E	Intel/Marvell XScale, Faraday FA626TE, Marvell Sheeva Feroceon, PJ1/Mohawk
2002	ARMv6	ARM11	
2004	ARMv7-M	ARM Cortex-M3	
2005	ARMv7-A	ARM Cortex-A5/A7, ARM Cortex-A8/A9, ARM Cortex-A12/A15/A17	Apple A6, Qualcomm Snapdragon Krait, Scorpion, Marvell Sheeva PJ4,
2007	ARMv6-M	ARM Cortex-M0/M0+/M1	
2010	ARMv7E-M	ARM Cortex-M4, Cortex M7	
2011	ARMv7-R	ARM Cortex-R4/R5/R7	
2012	ARMv8-A ARMv8.2-A	ARM Cortex-A35/A53/A57/A72/A73, ARM Cortex-A55/A65/A75/A76/A77, ARM Neoverse N1/E1	Apple A7/A8/A9/A10/A11, Qualcomm Snapdragon Kryo, Nvidia Denver Samsung Exynos M1/M2/M3/M4
2016	ARMv8-M	ARM Cortex-M23/M33	
2016	ARMv8-R	ARM Cortex-R52	
2017	ARMv8.2-A	ARM Cortex A55/A75	Samsung Exynos 850/9820
2018	ARMv8-M ARMv8.2-A	ARM Cortex M35 ARM Cortex A65AE/A76AE	
2019	ARMv8.2-A	ARM Cortex A77	Qualcomm Snapdragon 690/870 Samsung Exynos 1080/2100
2020	ARMv8-M ARMv8.2-A	ARM Cortex M55, Cortex R82 ARM Cortex A78, Cortex-X1	Qualcomm Snapdragon 695/880 Samsung Exynos 880/980
2021	ARMv9-A	ARM Cortex A510/A710, -X2	Qualcomm Snapdragon 7+
2022	ARMv9-A	ARM Cortex M85 ARM Cortex A715, Cortex-X3	Qualcomm Snapdragon 8
2023	ARMv9.2	ARM Cortex A520/A720, -X4	Qualcomm Snapdragon 8 Gen 2 Samsung Galaxy S23/S23+

ARM ISA generációk és processzormagok

A 2013-as VHPC (Virtualization in High-Performance Cloud Computing) konferencián hangzott el egy felmérés az ARM alapú eszközök elterjedtségére és ennek 2020-ig várható növekedésére az elektronikai eszközök piacán (COTS = Commercial on-the shelf). A legnagyobb elterjedtség a mobil piacon tapasztalható, 2015-ben a piac mintegy 81,6%-át uraló Android operációs rendszer alkalmazásai meghatározó mértékben az ARM architektúrára építettek. Például a mai mobil eszközökben elterjedt Qualcomm Snapdragon processzorok (Nokia/Microsoft Lumia sorozat, Samsung Galaxy S5, S7) az ARMv7-A és ARMv8-A architektúrákra, az Apple A8, A9 (pl. iPhone 6, 6S) és a Samsung Exynos (pl. Galaxy S6) processzorai az ARMv8-A architektúrára épülnek.



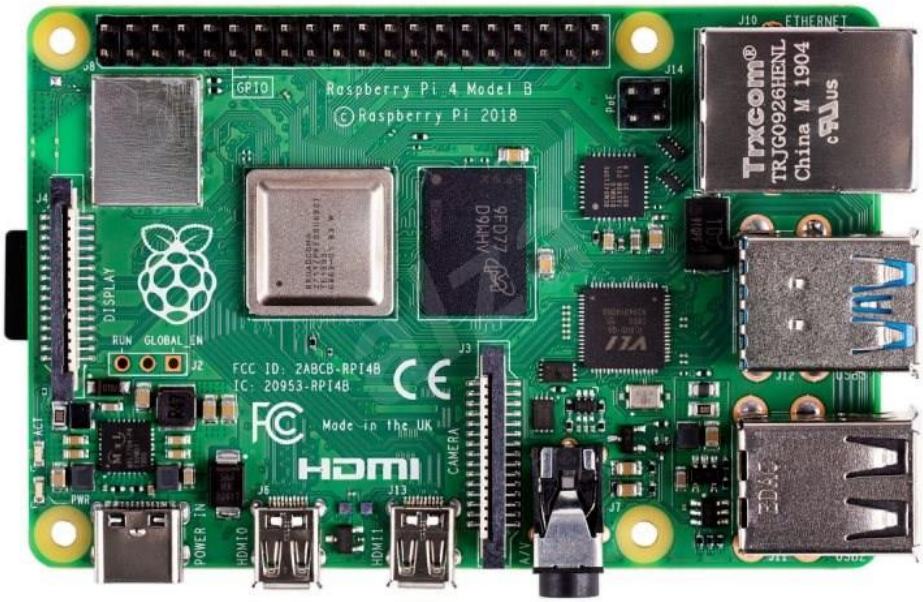
Az ARM család fejlődése (VHPC '13)



- Chinese Android app ecosystem is predominately optimized for ARM.

Az ARM architektúra elterjedtsége az Android alkalmazások körében

A Raspberry Pi 4 Model B egylapkás számítógépkártyája (1.5GHz 64 bites négymagos Cortex A72 CPU, 1/2/4/8 GB DDR4 SDRAM) Linux futtatási lehetőséggel mindössze 85,6 x 56,5 mm méretű.

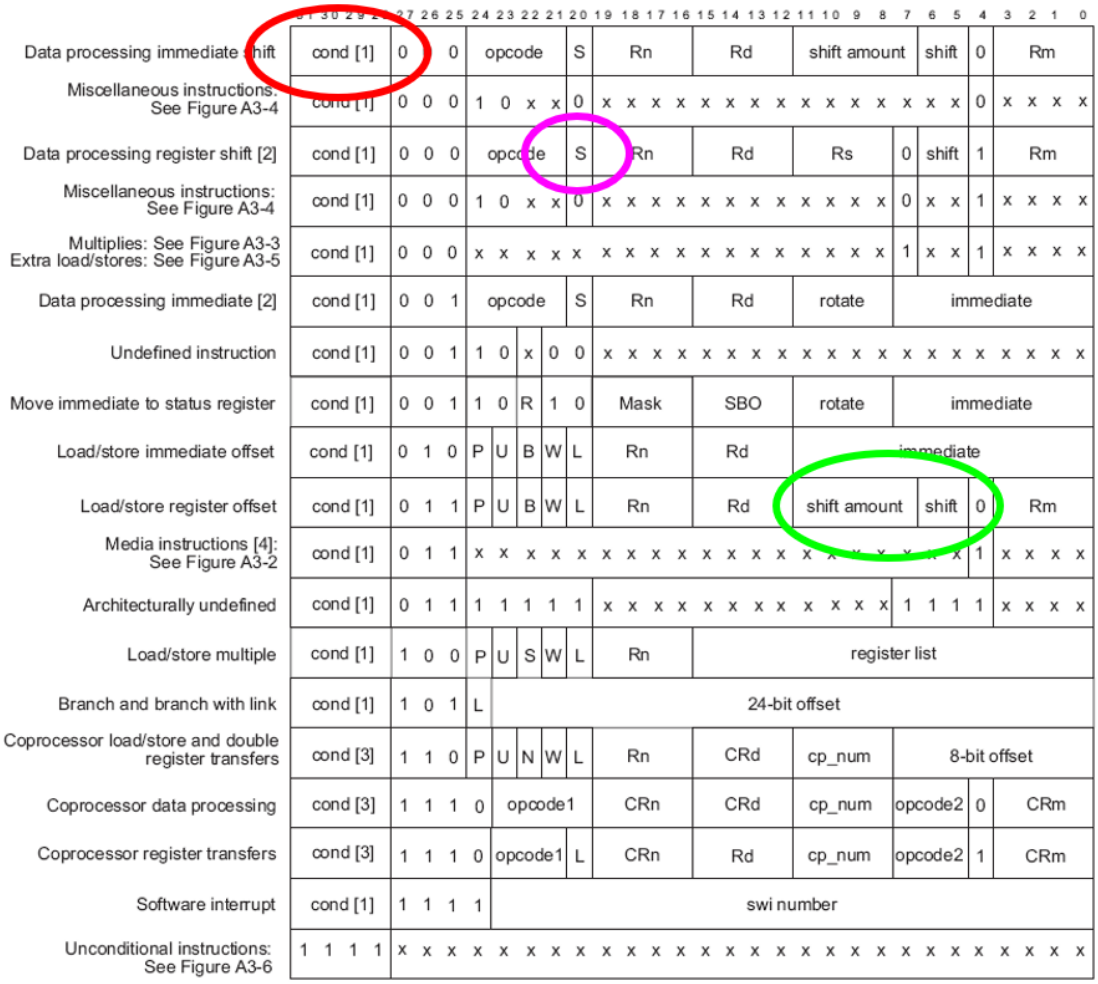


A Raspberry Pi 4 Model B

Belenézve a mikrokontroller-magok belsejébe a klasszikus ARM architektúra minden tagja egységesen a következő RISC tulajdonságokkal rendelkezik:

a) Load/store architektúra
b) Csak szóhatárra illeszkedő illesztett (aligned) memória hozzáférés az ARMv6-ig
c) Ortogonális utasításkészlet (bármely utasításban bármely címzési mód használható)
d) Nagy (31 db 32 bites) regiszterkészlet, amiből 16 látható egyszerre (R0..R15), a többi között a processzor aktuális üzemmódjai szerint automatikusan vált.
e) Egységes, 32 bites utasítások (egyszerűbb dekódolás és pipelining, csökkentett kódsűrűség)
f) Az utasítások többnyire egy órajel-periódus alatt történő végrehajtása

A fent felsorolt RISC elvekhez néhány figyelemreméltó további jellemzőt is hozzáadtak a fejlesztők. A következő táblázat az ARM processzorok utasításkészletének felépítését mutatja, kiemelve néhány jellegzetes részletet:



Az ARM processzorok utasításkészlete

1. A legtöbb utasításkód végrehajtása feltéthez köthető. Ezzel nagymértékben csökkenthető az elágazások száma a programokban, ami hatékonyabbá teszi a pipeline műveletvégzést (ezzel kompenzálva az elágazás predikátor hiányát).
2. Az aritmetikai utasításokban egy külön bit (az S bit) áll rendelkezésünkre annak vezérlésére, hogy az utasítás módosíthatja-e a feltételkód-regisztert (a flageket)
3. 1-32-bites eltolás (léptetés, shift) lehetősége egyetlen lépésben (ún. barrel shifter), amellyel rendkívül hatékonyan lehet aritmetikai műveleteket és cím-számolásokat végezni
4. Sok, a magas szintű nyelveket támogató címzési mód
5. 7 alapvető megszakítás típus, automatikusan váltott regiszterkészlettel

Code	Suffix	Flags	Meaning
0000	EQ	Z set	equal
0001	NE	Z clear	not equal
0010	CS	C set	unsigned higher or same
0011	CC	C clear	unsigned lower
0100	MI	N set	negative
0101	PL	N clear	positive or zero
0110	VS	V set	overflow
0111	VC	V clear	no overflow
1000	HI	C set and Z clear	unsigned higher
1001	LS	C clear or Z set	unsigned lower or same
1010	GE	N equals V	greater or equal
1011	LT	N not equal to V	less than
1100	GT	Z clear AND (N equals V)	greater than
1101	LE	Z set OR (N not equal to V)	less than or equal
1110	AL	(ignored)	always

Feltétel kódok (cond) az ARM utasításokban

Az első pontban említett érdekes ARM újítás a 4-bites *feltételes kód* minden utasítás elején, ezzel minden utasítás végrehajtása feltételhez köthető. Ez a tulajdonság egyik oldalról korlátozza ugyan a memória-elérési utasításokban az eltolás/indexelés mezőméretét, viszont az igen gyakori rövid ”`if-then-else`” szerkezetek esetén szükségtelenné teszi a feltételes elágazó utasításokat. Utóbbiak – mint már megtanultuk – a pipeline műveletvégzés legfőbb ellenségei. A standard példa erre a lehetőségre az Euklideszi (számelméleti) algoritmus megvalósítása ilyen tulajdonságokkal rendelkező processzorok esetén:

Tegyük fel, hogy **x** és **y** pozitív egész számok, és szeretnénk kiszámítani a legnagyobb közös osztójukat. Jelöljük ezt **(x,y)** –nal. Legyen **x>y**, ekkor igazak a következő állítások:

x = n * y + r, ahol **0 ≤ r < y**,

ha **r=0** (y osztója x-nek), akkor **(x,y)=y**

Euklidész algoritmusa szerint

(x,y) = (y,r)

Így a problémát visszavezettük két kisebb szám legnagyobb közös osztójának meghatározására. Ezt a lépést csak véges sokszor hajthatjuk végre, hiszen a számok mindvégig nem negatív egészek és folyamatosan csökkennek. Végül az utolsó, 0-tól különböző maradék a keresett legnagyobb közös osztó.
(Euklidesz eredeti algoritmusát úgy fogalmazta meg, hogy a nagyobbik számból kivonva a kisebbet a különbségnek is ugyanaz lesz az összes közös osztója, mint az eredeti számoknak. A fenti képletben az egész osztás használata csak felgyorsítja a számítás menetét, de ugyanazon a gondolaton alapszik.)

Könnyebb lesz az algoritmus megértése egy konkrét példán: keressük 280 és 175 legnagyobb közös osztóját

1. lépés:	280=1*175+105	(x=280, y=175, r=105)
2. lépés:	175=1*105+70	(x=175, y=105, r=70)
3. lépés:	105=1*70+35	(x=105, y=70, r=35)
4. lépés:	70=2*35+0	(x=70, y=35, **r=0**)

Tehát a keresett legnagyobb közös osztó a 35.

Ennek a feladatnak a megvalósítása C nyelven a következő (lévén **n** nem ismert, **r**-et az eredeti algoritmus szerint sorozatos kivonásokkal állítjuk elő):

```
int lnko(int i, int j) {
   while (i != j)    {          // ha x=y, akkor r=0, tehát vége
      if (i > j) i -= j;        // x=x-y annyiszor, amíg x>y
      else j -= i;              // y=y-x annyiszor, amíg y>x
   }                            // kilépünk, ha x=y
   return i;                    // ez lehetne return j is
}
```

Vegyük észre, hogy a fenti algoritmus minden egyes kivonást követően egy feltételes elágazást hajt végre – felborítva ezzel a pipeline műveletvégzés végrehajtást gyorsító hatását. Az ARM processzorok „nyelvén” fenti algoritmus így írható le (`i=Ri, j=Rj`):

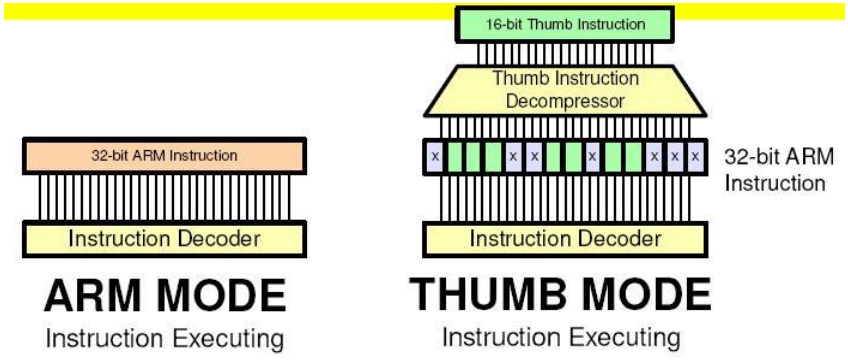
```
loop    CMP     Ri, Rj        ; beállítjuk a feltételt i-j szerint (S=1)
        SUBGT   Ri, Ri, Rj    ; ha i>j,  i = i-j (S=0)
        SUBLT   Rj, Rj, Ri    ; ha i<j,  j = j-i (S=0)
        BNE     loop          ; ha i<>j, akkor ugrás a loop-hoz
```

Így elkerülhetők az elágazások a `then` és az `else` kikötések körül. Vegyük észre az S bit használatának szükségességét, a második utasítás nem állíthatja a flageket ! A BNE utasítás után a legnagyobb közös osztó az Ri regiszterben található.

Az ARM kontrollerek utasításkészlete **egységes felépítésű és ortogonális** (valamennyi utasításban valamennyi címzési mód használható) – ahogyan azt a RISC alapelvek előírják. A sok beépített lehetőség azonban annyi variációt és lehetőséget takar, hogy az assembly szintű programozása még gyakorlott programozónak sem igazán ajánlható jó szívvel. Fogalmazzunk inkább úgy: ez a bonyolult utasításszerkezet jól megfogalmazható eszköztár az optimalizáló fordítóprogramok számára, melyek a tapasztalatok szerint az ARM processzorokra tömör és hatékony kódot képesek előállítani.

Az újabb ARM processzorok rendelkeznek egy tömörített utasításkészlettel, az ún. **Thumb**-bal, amely 16 bites utasításokat használ, de továbbra is 32 bites adatokkal dolgozik. (A „Thumb” jelző itt a kicsinyítést, tömörítést jelenti, ld. az angol Tom Thumb = Hüvelyk Matyi elnevezést). A Thumb mód lényege, hogy ez nem egy másik utasításkészlet, hanem az eredeti ARM utasítások tömörített változatai, bizonyos lehetőségek korlátozásával vagy elhagyásával a teljes utasításkódból. A rövidebb utasításkódokat egy dekompresszor egység „bővíti fel” a szabályos ARM utasításokra – a hiányzó részek alapértelmezett bitekkel való kitöltésével. Így kevesebb funkció érhető el egy-egy utasítás által, például csak az elágazások köthetők feltételhez és sok utasításkód nem lehet háromcímű, vagy nem érhet el minden CPU regisztert. A rövidebb

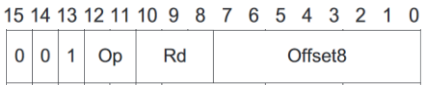
utasításkódok használatával növekszik a kódsűrűség, annak ellenére, hogy néhány művelet megvalósításához esetleg több utasítás szükséges. Az ötletet leginkább úgy lehetne jellemezni, hogy a bő (és sokszor ki nem használt) lehetőségek helyét megspóroljuk, viszont ha mégis szükség lenne a hiányzó lehetőségre, akkor azt csak két utasítás használatával tudjuk pótolni.



Utasítások dekódolása ARM és Thumb módban

Fentiek könnyebb megértésére egy konkrét példát mutat a következő ábra: egy konkrét 16 bites Thumb utasítás (ADD Rd,#konstans) „átalakítása” az ARM utasításkészlet 32 bites megfelelőjére a következőképpen történik:

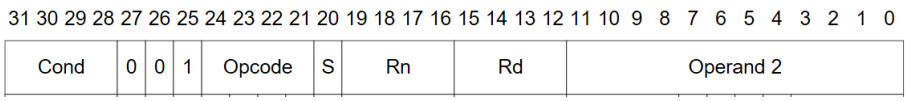
A Thumb utasítás első 3 bitje a művelet utasításcsoportjára utal, ezt követi 2 biten a csoporton belüli azonosító (Op: ADD=10). Ezután következik 3 biten az Rd regiszter azonosítója (Rd=0..7), majd az utasításba beépülő 8 bites konstans (Offset8).



Az ADD Rd,#konstans Thumb utasítás felépítése

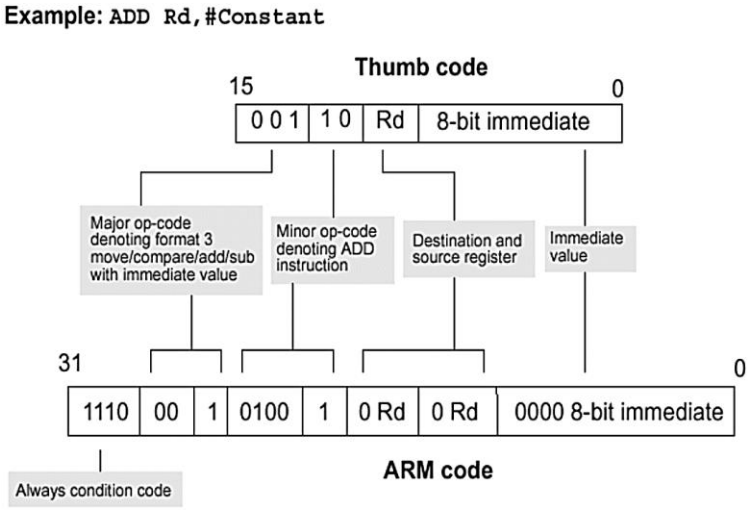
Az utasítás 32 bites megfelelője az ARM utasításkészletben a következő:

- A 31-28. biteken található a minden utasításra jellemző, a végrehajtás feltételét előíró feltételkód (Cond)
- A 27-25. biteken a művelet utasításcsoport-besorolása (Data Processing and FSR Transfer) található a Thumb kódhoz hasonlóan
- A 24-21. bitek azonosítják a csoporton belül a műveletet (ADD=0100)
- A 20. bit az S vezérlőbit (a művelet állítsa-e a flageket)
- A 19-16. és a 15-12. bitek a 16-16 db cél és forrás regisztereket azonosítják
- A 11-0. bitek tartalmazzák az utasításba beépülő 12 bites konstanst



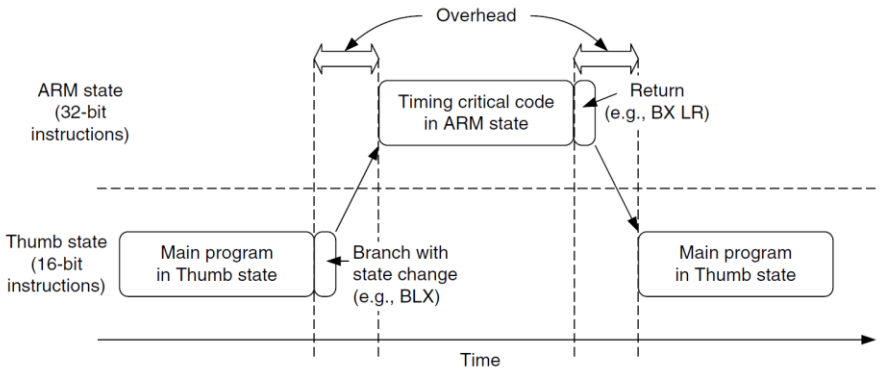
Az ADD {cond}{S} Rd, Rn, <Oprnd2> ARM utasítás felépítése

Az átalakítás a következő ábra alapján könnyen megérthető. A Thumb utasítások nem tartalmaznak feltételkódot, így az Always=1110 kód kerül behelyettesítésre, kevesebb utasítás áll rendelkezésünkre, így az Opcode elöl-hátul egy-egy 0 bittel egészül ki, a 3 címűség hiányában Rn = Rd, ami csak az alsó 8 regiszert címezheti a 16 regiszter közül, végül a beépülő konstans is csak 8 bites lehet, így 4 db 0-val egészül ki a tetején.



Utasítások dekódolása ARM és Thumb módban

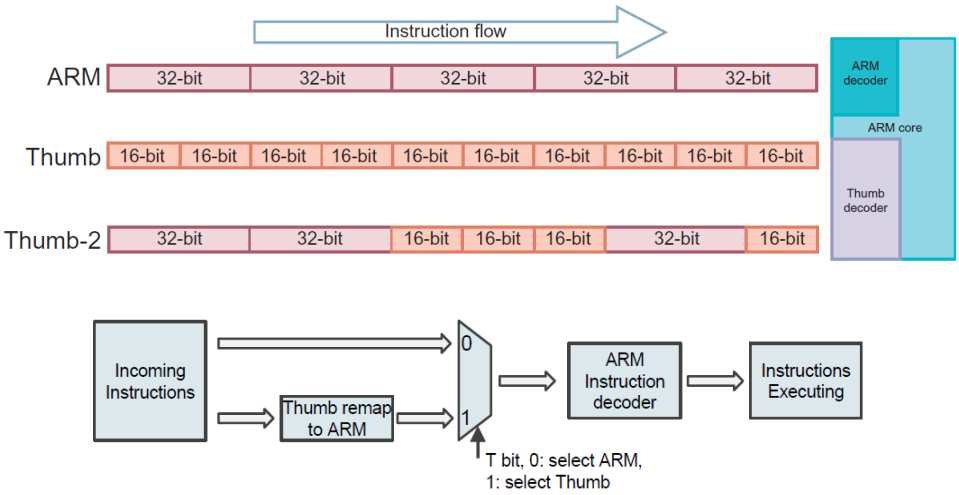
A Thumb utasítások és a normál ARM utasítások keverhetők egymással, a programszámláló (PC) a legalsó bitjén tárolja, hogy a kódok melyik utasításkészlet szerint értelmezendők. Normálisan a PC legalsó bitje a 16 vagy 32 bites utasításhossz miatt mindig 0, így ezt a bitet arra is használják, hogy amennyiben az értéke 1, az a Thumb-üzemmódot jelzi a processzor számára (T=1 a státusz regiszterben – ld. később). Természetesen ez csak állapotjelzés, a tényleges címzés során ezt a bitet az értékétől függetlenül 0-nak tekintik.



Áttérés Thumb utasítások és normál ARM utasítások között

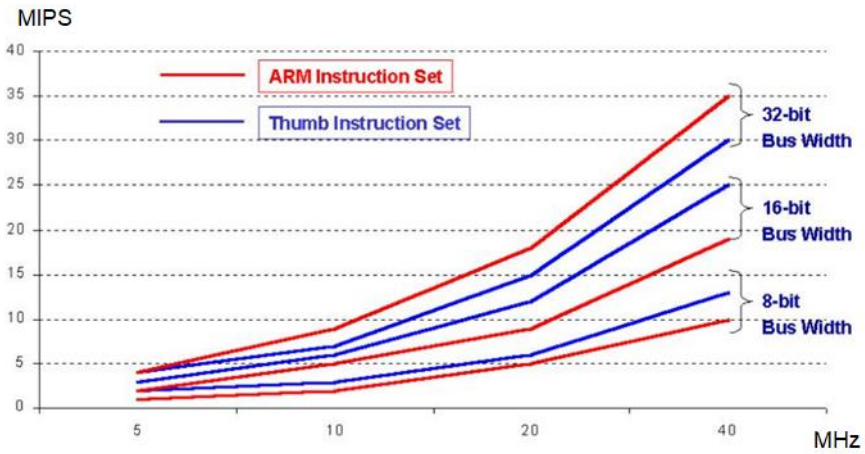
Az ARM processzor beolvasva a következő utasítást a T (Thumb) bit alapján dönti el, hogy az utasítás hogyan értelmezendő, a Thumb utasításokat azonnal kiterjeszti az eredeti

ARM utasítás formátumára, így a processzor végrehajtóművének ezzel már nem kell foglalkoznia.



Utasítások dekódolása ARM és Thumb módban

Különösen azokban a helyzetekben, amelyekben az utasítás memória buszszélessége kevesebb, mint 32 bit-re van korlátozva, a rövidebb Thumb utasításkódok lényegesen jobb teljesítményt nyújtanak, mint az eredeti 32 bites utasítások, a limitált memória-sávszélesség jobb kihasználása miatt. A beágyazott hardvereknek sok esetben nincs 32 bites adatbusza, inkább 16 bites, vagy még kevesebb (pl.: néhány mobiltelefon vagy gyerekjáték, mint pl. a Game Boy Advance). A rövidebb utasításokkal nagyobb kódtömörség érhető el, hiszen ha minden ARM utasítás helyett 2 db Thumb utasítást használnánk, akkor kapnánk meg az eredeti kódméretet. Ilyen esetekben van értelme a Thumb utasításkészletre fordításnak, esetleg a CPU-t inkább igénybe vevő részeket át kell írni a 32 bites ARM utasításkészletre, és utóbbiakat a 32-bites buszszélességű memóriába kell elhelyezni. Az első Thumb utasítás dekóderrel készült processzor az ARM7TDMI volt. Az ARM9-es és a későbbi családtagok már rendelkeznek ilyen értelmezővel.

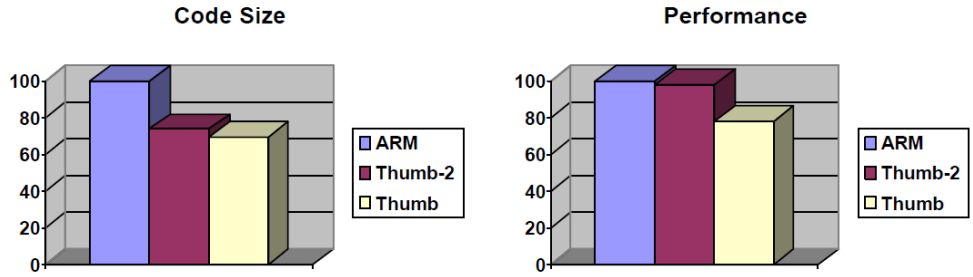


Utasítások végrehajtási sebessége az adatbusz méretének függvényében

A fenti diagramon is látszik, hogy az alkalmazott buszméret függvényében az ötlet beváltja a hozzá fűzött reményeket: 8 vagy 16 bites busz esetében a Thumb, 32 bites busznál az eredeti ARM utasításkészlet alkalmazható hatékonyabban. Ezt a megállapítást még tovább árnyalhatja a kód tömörsége (mérete), amit a fenti diagram nem tartalmaz.

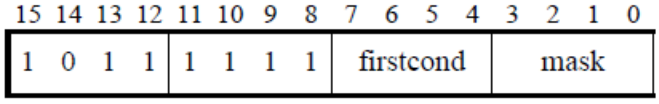
A Thumb utasításkészlet továbbfejlesztéseként 2003-ban vezették be a **Thumb-2** technológiát. Ez a Thumb 16 bites utasításkészletét egészíti ki 32 bites utasításokkal úgy, hogy az utasításkészlet átfogóbb legyen. Így a Thumb-2 megközelítőleg a Thumb kódsűrűségével (az ARM programkód méret 74%-a) és 32-bites memóriával rendelkező ARM utasítás-készlet sebességével (98%) bír. A Thumb-2 mind az ARM mind a Thumb utasítás-készletet kiegészíti új utasításokkal, így például bit-mező módosításokkal, ugrótáblákkal (elágazási tábla) és feltételes futtatással.

A következő diagram az ötlet tényleges programokon mért hatékonyságát mutatja mind a kódsűrűség, mind a teljesítőképesség függvényében:



Kódméret és teljesítmény összevetése
az ARM, a Thumb és a Thumb-2 utasításkészlet használata mellett

Külön említést érdemel a Thumb-2 utasításkészletben a feltételes utasítás végrehajtás visszahozása. A Thumb utasításkód rövidítése „magával vitte” az egyes utasítások feltételhez köthetőségét, ami pedig nagyon hatékonyan működik az ARM utasításkészletben. Bevezették ezért az ***if-then-else*** szerkezetet (utasítást). Az IT (if-then) utasítás szerkezete a következő:



Az IT (if-then) utasítás szerkezete a Thumb-2 utasításkészletben

ahol *firstcond* az ARM utasításoknál megismert 4 bites feltételkód (ami a teljes szerkezetre vonatkozik), míg a *mask* mező az utasítást követő 1..4 utasítás feltételes végrehajtását definiálja – egy speciális kódolással azt tartalmazza, hogy követi-e az IT utasítás után álló első feltételes utasítást második-harmadik-negyedik, és azokat a megadott feltételkód (then), vagy annak az ellenkezője (else) szerint kell végrehajtani. A könnyebb megértéshez nézzünk erre néhány rövid példát:

a) **R0 = abs(R1)**

```
        MOVS    R0, R1          ; beállítjuk a flageket R1 szerint
        IT      MI              ; (if) ha negatív
        RSBMI   R0, R0, #0      ; (then) akkor R0 = 0 - R0, különben skip
```

b) **ha R0 > R1 és R2 > R3, akkor R4 = R5**

```
        CMP     R0, R1          ; beállítjuk a flageket R0-R1 szerint
        ITT     GT              ; ha R0 <= R1, akkor két utasítás kimarad
        CMPGT   R2, R3          ; ha R0 > R1 akkor ha R2 > R3
        MOVGT   R4, R5          ; akkor R4 = R5, különben skip
```

c) **ha Z=1, akkor R0 = R3+[R1] különben R0=R4+[R2]**

```
        ITETE   EQ              ; ha Z=1, akkor/különben/akkor/különben
        LDREQ   R0, [R1]        ; ha Z=1, akkor R0 = [R1]
        LDRNE   R0, [R2]        ; ha Z≠1, akkor R0 = [R2]
        ADDEQ..R0, R3, R0...; ha Z=1, akkor R0 = R0 + R3
        ADDNE   R0, R4, R0      ; ha Z≠1, akkor R0 = R0 + R4
```

(A program jobb érthetősége érdekében az IT utasításban szereplő feltételt – vagy annak ellenkezőjét – a forráskódban meg kell ismételni a szerkezethez tartozó utasításkódokban is, jóllehet a Thumb-2 utasítások sem tartalmaznak utasításonként feltételkód-mezőt. Ezeket mutatják a példákban szereplő utasítások piros színnel jelölt feltétel-kódjai.
A szintaxis könnyebb érthetőségéhez néhány rövid magyarázat:

- Háromcímű utasításoknál az operandussorrend: <dest>, <src1>, <src2>
- RSB (reverse subtract): fordított operandussorrend annak érdekében, hogy a mindenképpen harmadik operandusként szereplő immediate konstans kisebbítendő is lehessen
- [Rn] = indirekt címzés)

A hatékonyság összevetésére nézzük az utóbbi feladatot mindhárom utasításkészlet használatával:

	ARM	Thumb	Thumb-2

Program-részlet	LDREQ R0, [R1]
	LDRNE R0, [R2]
	ADDEQ R0, R3, R0
	ADDNE R0, R4, R0
		BNE L1
		LDR R0, [R1]
		ADD R0, R3, R0
		B L2
		L1
		LDR R0, [R2]
		ADD R0, R4 R0
		L2
			ITETE EQ
			LDREQ R0, [R1]
			LDRNE R0, [R2]
			ADDEQ R0, R3, R0
			ADDNE R0, R4, R0

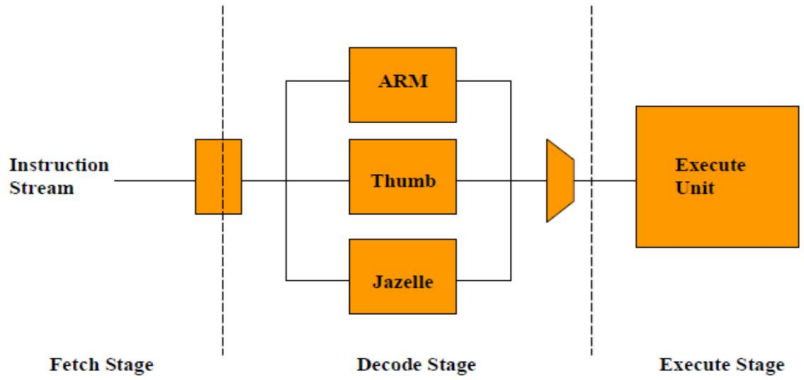
Kódméret	4 x 32 bit = 16 bájt	6 x 16 bit = 12 bájt	5 x 16 bit = 10 bájt
Ciklusok száma	4 T	4-20 T (elágazásbecsléstől függ)	5 T

Programrészletek az utasításkódok tömörségének és végrehajtási idejének összevetésére

Természetesen nem az adott műveletsor a tipikus a programokban, hanem általánosítva, egy előző művelet eredményétől (Z=1) függő két teljesen eltérő műveletsor if-then-else szerkezetben. Vegyük észre, hogy a fenti utasítássorozat végrehajtásához

- az ARM utasításkészlet esetében 4 db 32 bites utasítás (16 bájt, 4T),
- a Thumb utasítások esetében 6 db 16 bites utasítás (12 bájt, és mivel ugrások is szerepelnek benne, az adatoktól függően 4-20T), míg
- a Thumb-2 utasításkészlet használata esetén 5 db 16 bites utasítás (10 bájt, 5T) szükséges ugró utasítások nélkül !

Az ARM cég kifejlesztette a **Jazelle** nevű technológiát, ami lehetővé teszi a Java bájtkód hardver szintű végrehajtását az ARM processzorok számára. Ezt a lehetőséget általában a 'J' betű jelöli processzorok nevében. Mint ismeretes, a Java egy objektumorientált programozási nyelv, amelyet a Sun Microsystems fejlesztett ki a 90-es évek elején és fejleszt egészen napjainkig. A Java alkalmazásokat jellemzően egy platformfüggetlen bájtkód formátumra fordítják.

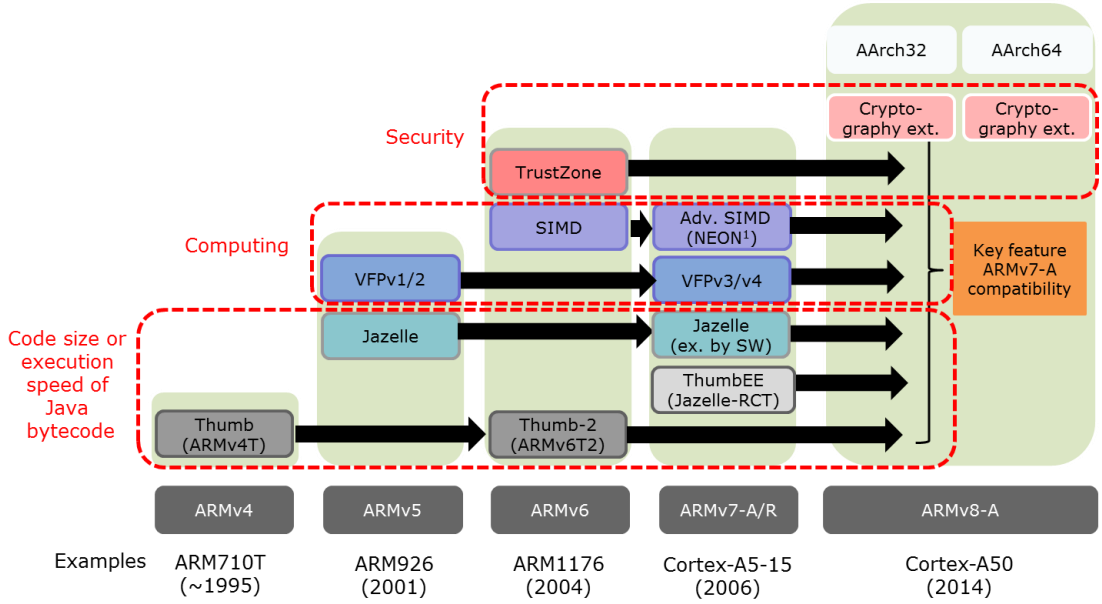


Java kód végrehajtása az ARM processzorokban

A bájtkód futtatása az ún. Java virtuális géppel (JVM) történik, ami vagy interpretálja a bájtkódot, vagy natív gépi kódot készít belőle és azt futtatja az adott operációs rendszer alatt. Léteznek közvetlenül Java bájtkódot futtató hardver eszközök is, (ezek az ún. Java processzorok) – ilyenek az ARM processzorok is a Jazelle képességükkel. A lehetőséget a mobiltelefonokban történő Java alkalmazások futtatására használják nagy előszeretettel.

A legújabb családtagok **NEON technológiája** egy többmagos (multicore), 64 és 128 bites adatokat kezelő SIMD (Single Instruction Multiple Data) társprocesszort takar, amely elsősorban multimédia és jelfeldolgozó alkalmazások részére biztosít hardveres gyorsítást. A SIMD architektúra párhuzamosan működő aritmetikai egységek segítségével egyetlen utasítással több adaton is képes azonos műveletet végezni – ilyenekre elsősorban a hang- és képfeldolgozó algoritmusok futtatása esetén van szükség. A NEON futtatni tudja az MP3 audio dekódert már 10 MHz-es processzorsebesség mellett (a processzormag áramfelvétele miatt nagyon fontos az alacsony sebesség !), a GSM AMR (Adaptive Multi-Rate) audio kodeket pedig 13 MHz-es CPU sebesség mellett is. Az adatok kezelését ilyenkor 32 db 64 bites és 16 db 128 bites (!) regiszter támogatja a processzormagban.

A **VFP** (Vector Floating Point) architektúrát a jelfeldolgozás, 3D grafika, valós idejű képfeldolgozás, mozgásszabályozások támogatására kínálják az ARM processzorok. A lebegőpontos szabvány (IEEE 754) szerinti formátumú egyszeres és duplapontosságú lebegőpontos számok műveletvégzését 1.3 MFLOP/MHz sebességgel garantálják az ARM9-es processzorok integrált lebegőpontos utasításai.



Az ARM generációk fejlődése

Az ábrán jól látható, hogy az ARM-család fejlesztői nagy hangsúlyt helyeztek az első generációk sikeres elterjedését követően

- a programméret és a végrehajtási sebesség optimalizálására (Thumb, Thumb-2, Jazelle);
- a számítási teljesítmény növelésére (SIMD, Neon, VFP);
- végül a megbízhatósági és titkosítási kérdésekre (TrustZone, Cryptography).

Végül a fejezet végén még bemutatjuk, hogy ARM családtagok egy-egy konkrét processzor típusjelzésében az egyes betűjelölések mit takarnak:

- T: az ARM-kontroller ismeri a tömörített Thumb utasításkészletet;
- D: a magba beépítésre kerültek a fejlesztést megkönnyítő JTAG alapú debuggoláshoz szükséges hardver egységek és tulajdonságok;
- M: a magba beépítésre került az a továbbfejlesztett szorzómű, ami a digitális jelfeldolgozást gyorsítja meg;
- I: a beágyazott ICE makrocella bővítmény valós idejű trace/debug funkciót tesz lehetővé. Segítségével (és 2-4 Mbyte nagysebességű buffer memória alkalmazásával) a célkörnyezetben valós időben működő processzor is megfigyelhetővé válik;

- J: Java bájtkód értelmező és hardvergyorsító bővítmény található a kontrollerben;
- E: DSP utasítások kerültek be a processzor utasításkészletébe, melyek a telítéses matematikai műveleteket alkalmazzák (túlcsordulás jelzése helyett a lehetséges maximális értékeket szolgáltatják). Ez egy, a jelfeldolgozásban használatos módszer a jelfeldolgozás gyorsítására.

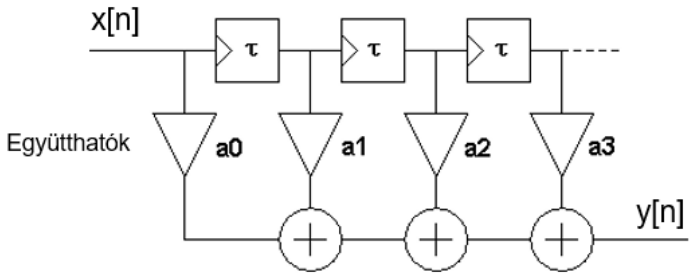
A jelprocesszor- tulajdonságok bevezetése az ARM processzorok lehetőségei közé ismét egy hatalmas piacot nyitott meg az ARM alkalmazások világában. Mint azt a következő fejezetünkben látni fogjuk, a jelfeldolgozó processzorok egy sor olyan egyedi tulajdonsággal rendelkeznek, melyek alkalmassá teszik őket a nagysebességű digitális jelfeldolgozás alkalmazásaiban. Ezen tulajdonságok jelentős részét képesek megvalósítani az ARM DSP-bővítmények is, és így komoly konkurenciát jelentenek a jelprocesszorok gyártóinak.

Instruction	Operation	Purpose
SMLAxy{cond}	16 x 16 + 32 → 32	Signed MAC
SMLAWy{cond}	32 x 16 + 32 → 32	Signed MAC wide
SMLALxy{cond}	16 x 16 + 64 → 64	Signed MAC long
SMULxy{cond}	16 x 16 → 32	Signed multiply
SMULWy{cond}	16 x 32 → 32	Signed multiply long
QADD Rd, Rm, Rs	SAT(Rm + Rd)	Saturating add
QDADD Rd, Rm, Rs	SAT(Rm + SAT(Rs x 2))	Saturating add double
QSUB Rd, Rm, Rs	SAT(Rm – Rd)	Saturating subtract
QDSUB Rd, Rm, Rs	SAT(Rm – SAT(Rs x 2))	Saturating subtract double
CLZ{cond} Rd, Rm	COUNTZ(Rm)	Count leading zeros

Jelfeldolgozást segítő utasítások az ARM kontrollerek utasításkészletében

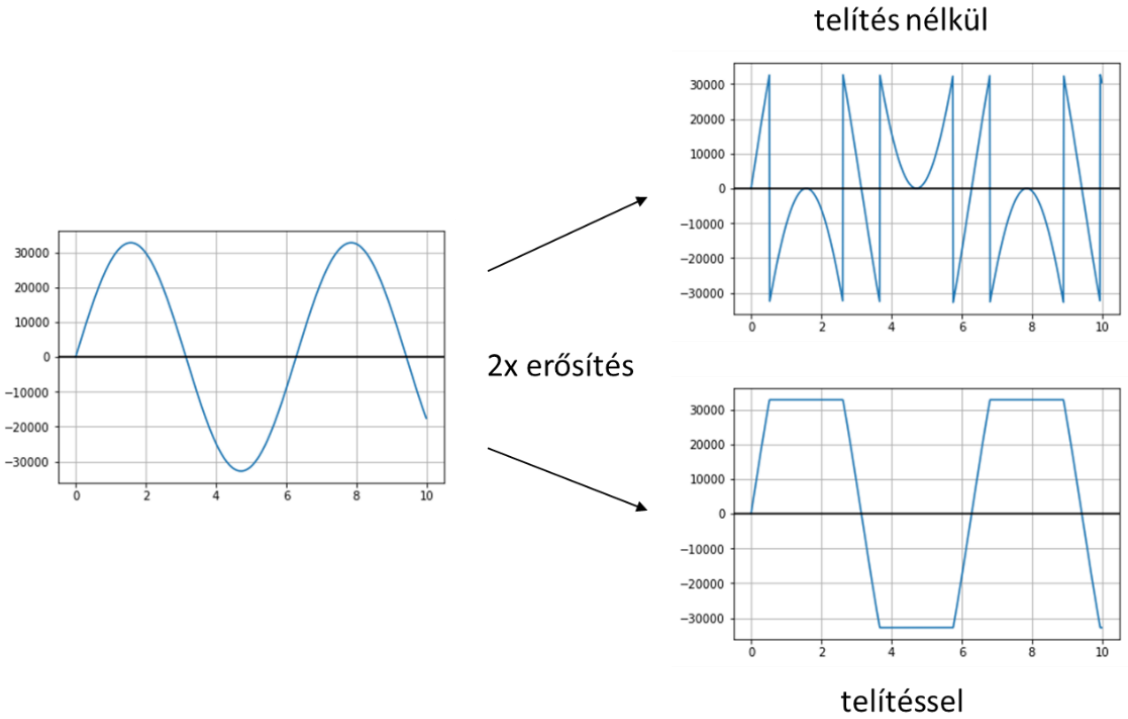
Ilyen tulajdonságok pl. az egyetlen utasítással elvégzett szorzás és összeadás (MAC) a digitális szűrőalgoritmusok (FIR, IIR) támogatására, melyek nagyon gyakoriak a digitális jelfeldolgozások algoritmusaiban.

$$y[n] = a_0x[n] + a_1x[n-1] + a_2x[n-2] + a_3x[n-3] + \cdots$$



FIR szűrő egyenlete és vázlata

Szintén a jelfeldolgozási algoritmusok hatékony eszközei a telítődést figyelembe vevő számtartomány-korlátos műveletek (saturation, az ún. „Q betűs” utasítások). Lényegük, hogy megakadályozzák egy aritmetikai művelet során az eredmény túl- vagy alulcsordulását, helyette az ábrázolható számtartomány legnagyobb, ill. legkisebb értékét szolgáltatják. A jelek feldolgozása során ugyanis számtalan esetben erősítjük vagy gyengítjük azokat (pl. audio vagy videojelekkel végzett műveletek esetén), a számábrázolás tulajdonságai viszont egy-egy aritmetikai művelet túlcsordulása esetén komoly anomáliákhoz vezethetnek. Pl. egy 16 biten ábrázolt előjeles szám legnagyobb lehetséges pozitív értékéhez (32 767 = 7FFFh) 1-et hozzáadva a szám túlcsordul (32 768 = 8000h), ami viszont a kettes komplemens számábrázolás szabályai szerint a tartományban ábrázolható legkisebb érték (8000h = -32 768).



A telítődés figyelembevétele a DSP-utasítások végrehajtása során

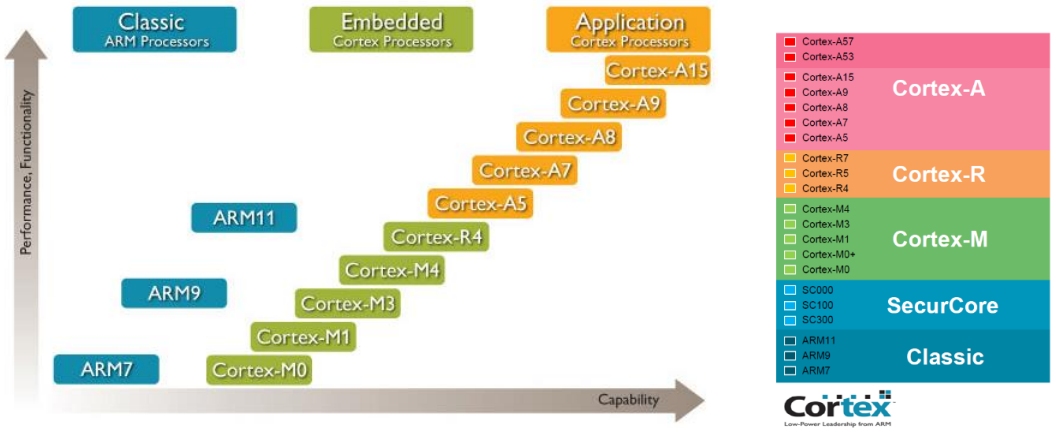
A telítődést figyelembe vevő aritmetikai utasítások „felütköztetik” a számértéket az ábrázolási tartomány alsó és felső határán, és a telítődés tényét a processzor Q státuszbitje jelzi a programozó számára.

1.6.1 Az ARM Cortex M4 család architektúra

Az ARM processzorok „klasszikus” mikroarchitektúráit a 2000-es évek közepétől a Cortex családtagok váltották fel. A Cortex („kéreg”) architektúra célja egy a korábbinál is szélesebb spektrumot lefedő piac meghódítása volt: kezdve az 1$ alatti ártól induló mikrokontrollertől egészen a 2 GHz-ig működő többmagos processzorokig kínál alternatívákat a piac többi résztvevőivel szemben.

A Cortex család tagjai 3 fő irányt kívánnak lefedni:

- A **Cortex-A** családtagok a nagyobb memóriaigényű és teljesítményű alkalmazások számára készültek (MMU, VFP, Neon). Tipikusan a nagy számítási teljesítményt igénylő ún. high-end alkalmazások processzorai. Sok elterjedten használt operációs rendszer platform készült ezen processzorok számára (többféle Linux, QNX, Symbian, Android, Windows-CE), fő vadászterületük a mobiltelefonok, mobil számítógépek, digitális TV-k, hálózati eszközök piaca.
- A **Cortex-R** családtagok a valós idejű (real-time) alkalmazások számára készített nagyteljesítményű processzorok, nagymegbízhatóságú memóriavédelmi egységgel kiegészítve. Legfőbb piacuk: mobil irányítórendszerek, nagy tömegű adattárlók vezérlői, GPS, motorvezérlő elektronikák, videokamerák, robotirányító rendszerek.
- A **Cortex-M** tagok a kifejezetten költségérzékeny mikrokontrollerek piacára készített egyszerűbb, kimondottan alacsony fogyasztású eszközök. Elterjedten alkalmazzák őket gépjárművek vezérlőiben, orvosi berendezésekben, elektronikus játékokban.



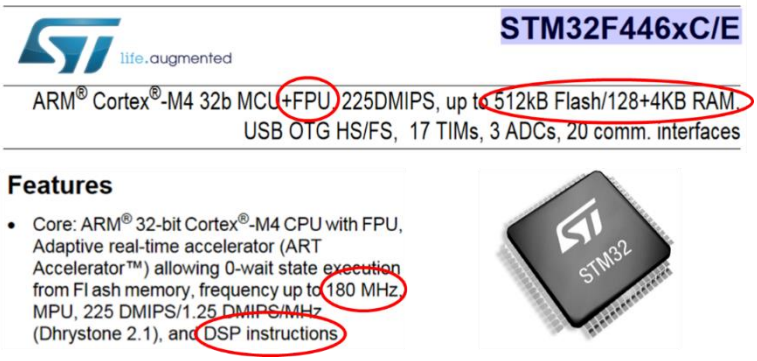
A Cortex processzorok

A Cortex processzorok hihetetlen népszerűek lettek: az A-sorozat az (okos) mobiltelefonok piacán 80% feletti részesedést ért el napjainkra, az M sorozat az általános célú beágyazott rendszerek világának egyik legelterjedtebb processzora. Az ARM cég által közzétett adatok szerint naponta kb. 16 millió (!) Cortex processzor kerül piacra a különböző gyártóktól („ARM truly is The Architecture for the Digital World®” – olvasható az ARM honlapján).

A következőkben egy jellegzetes Cortex M4 mikrovezérlő segítségével mutatjuk be a család legjellegzetesebb tulajdonságait. A kiválasztott képviselő az ST Microelectronics cég STM32F446-os mikrovezérlője, a tanult mikrokontrollerekkel való összevethetőség érdekében csak a legfontosabb tulajdonságait soroljuk itt fel:

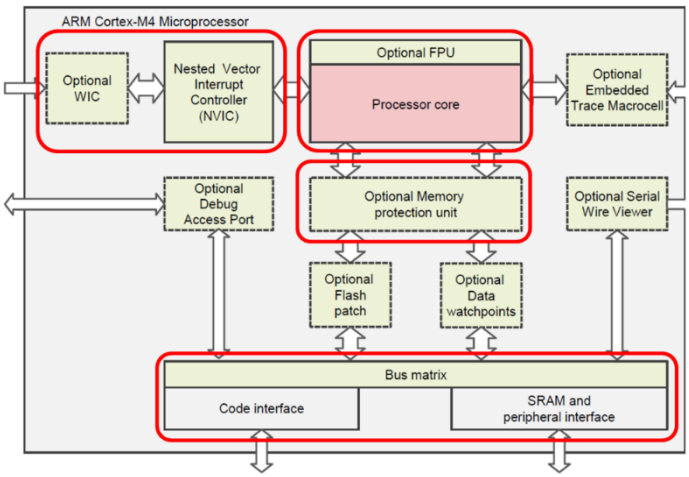
- 32 bites ARM architektúra
- 32/64 bites lebegőpontos műveletvégző egység (FPU)
- Jelprocesszor (DSP) kiterjesztés az utasításokban
- 180 MHz működési frekvencia 225 DMIPS műveletvégzési sebesség
- 512 kbájt flash és 128+4 kbyte RAM a működés késedelem nélküli kiszolgálására

(A Dhrystone MIPS vagy DMIPS a számítógépek teljesítménymérésére használt szám a 70-es évek DEC VAX 11/780 ipari mércének tekintett számítógépét véve alapul, melynek teljesítőképessége 1 MIPS volt. A teszt a számítógépek több jellegzetes teljesítménymeghatározó tulajdonságát figyelembe véve adja meg a tesztelendő egység minősítését.)



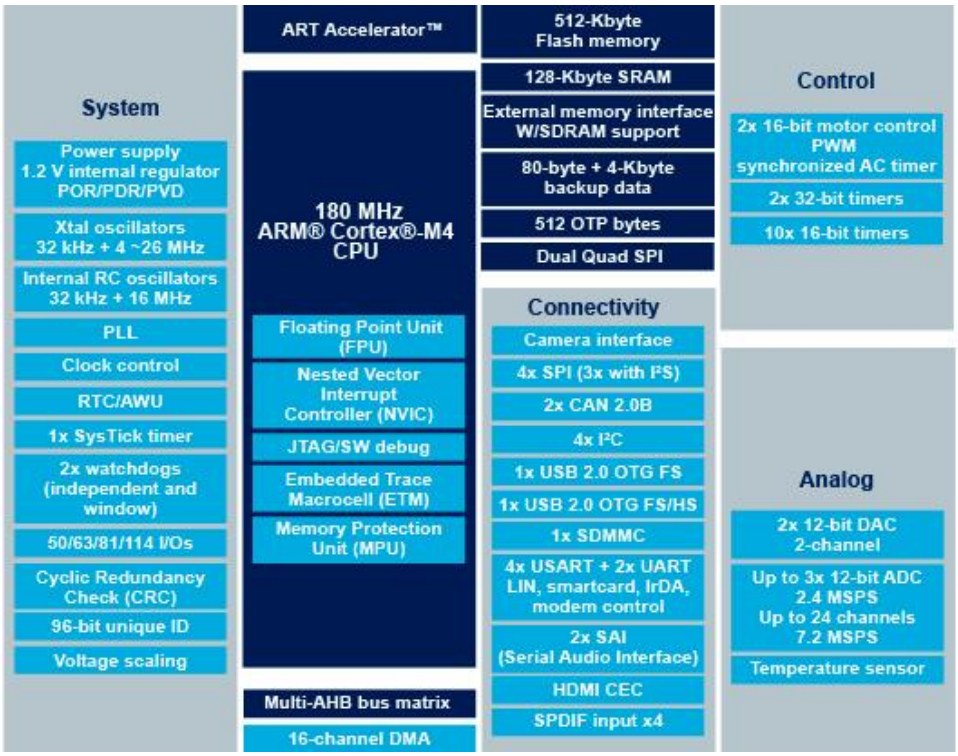
Egy Cortec M4 mikrokontroller főbb jellemzői

A következő ábra a Cortex-M4 mikroprocesszor általános felépítését mutatja (a beépített memória és periféria elemek nélkül):



Cortex M4 mikroprocesszor felépítése

A mikroprocesszor tartalmazza az ARM által fejlesztett processzor magot, szoros kapcsolatban az opcionális FPU egységgel. Az architektúra elengedhetetlen jellegzetessége a beépített vektoros megszakításvezérlő (NVIC) és az ahhoz kapcsolódó, a csökkentett áramfelvételt támogató alvó üzemmódokból való „felébresztést" támogató WIC (Wake-up Interrupt Controller). Egyes családtagok tartalmazzák a memóriatartományok hardver védelmét lehetővé tevő memóriavédelmi egységet (MPU), végül a nagysebességű mag a buszmátrix egységen keresztül kapcsolódik a fizikai memória és periféria egységekhez.

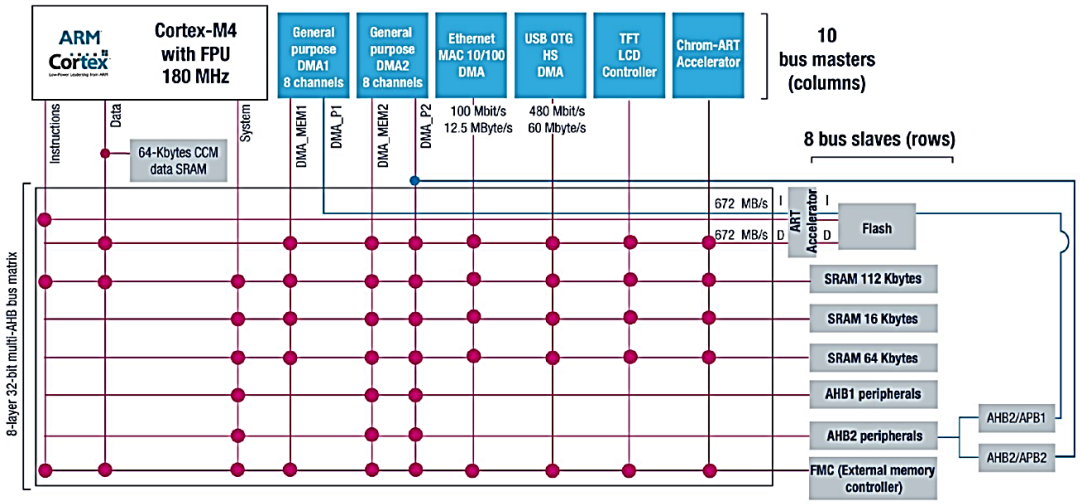


Az STM32F446RE Cortex M4 mikrovezérlő felépítése

A bemutatott Cortex M4 központi egységhez illesztőegységek, memóriák és perifériák sora kapcsolódik egy-egy konkrét mikrovezérlőben. Ezen egységek igen bő választékát demonstrálja az STM32F446RE mikrokontroller felépítését bemutató ábra.

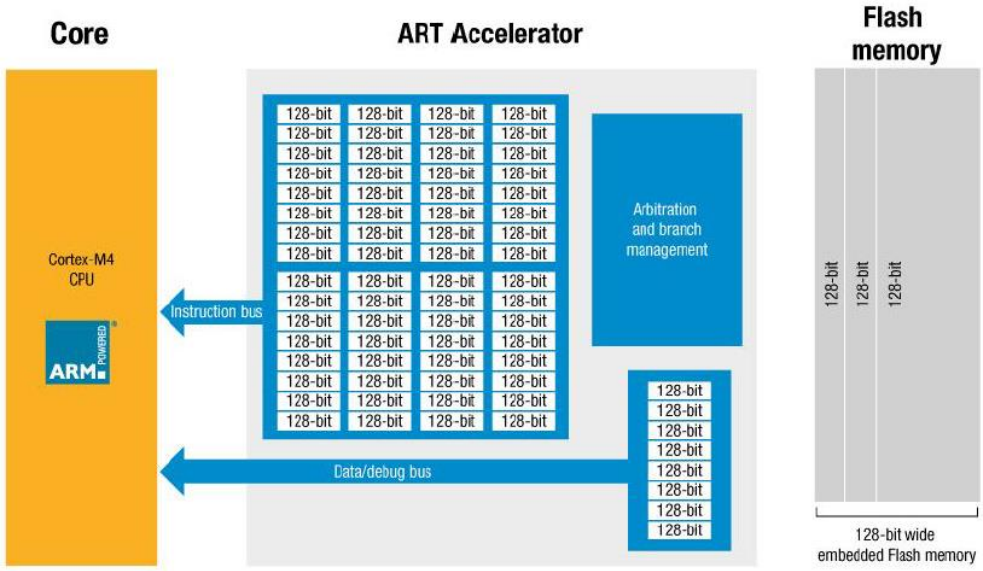
A processzor rendkívül erős buszrendszerrel rendelkezik a pipeline műveletvégzés miatt átlapolódó utasítások és operandusok számára a nagysebességű memóriák és perifériák azonos órajelciklusban való egyidejű elérhetősége érdekében. Ne tévesszük szem elől, hogy a központi egységnek órajelenként utasítást kell beolvasnia, adatokat kell írnia-olvasnia a belső regiszterekből-regiszterekbe, miközben a CPU-tól független két DMA vezérlő szintén buszműveleteket végez a központi egység tehermentesítése érdekében (pl. A/D vagy D/A átalakítók, soros kommunikációs interfészek kezelése). Ezek mellett nagysebességű Ethernet és USB vezérlő, színes képernyő vezérlő kommunikál mind a központi egységgel, mind a megfelelő memóriaegységekkel. Ennek a nagytömegű és nagysebességű adatforgalomnak a kezelésére fejlesztették ki azt a nagyteljesítményű

buszrendszer-mátrixot (AHB = Advanced High-speed Bus), amely különböző adatutakat képes kiépíteni az oszlopokban ábrázolt 10 vezérlő (master) és az egyes sorokban szereplő 8 kiszolgáló (slave) egység között. Az ábrán a lehetséges kapcsolódásokat jelzik a pontok, minden egyes pont mögé az alapvetően 32 bites cím- és a 32 bites adatbusz, valamint még egy sor vezérlőjel (írás, olvasás, engedélyezés) fizikai ki-be kapcsolását képzeljük.



AHB (Advanced High-speed Bus) mátrix

A 180 MHz órajel frekvencia mellett már külön problémaként jelenik meg a memóriák megfelelő sebességű elérhetősége (T_{CLK} = 5.55 ns). A technológia mai állása mellett az integrált statikus RAM memóriák még képesek ennek a sebességnek a kiszolgálására, a flash memóriák esetében azonban ez az olvasási sebesség már nem lenne megoldható. Az ST Microelectronics erre a célra fejlesztette ki az ART (Adaptive Real-Time) Accelerator gyorsítótárat, amely egy speciális cache-mechanizmusnak tekinthető.



Az ART gyorsító tár működési elve

A gyorsítótár a következő elven működik: a 128 bit szószélességű flash memória minden hozzáférésre max. 8 db (16 bites), min. 4 db (32 bites) utasításkódot olvas ki egyszerre. A processzor által ténylegesen címzett utasítás- vagy adatkód ebből kivágásra kerül, a kiolvasott 128 bites információ pedig utasításkód esetén egy 8x8=64 szó (szó alatt itt 128 bit értendő) ill. adat esetén 8 szó befogadására alkalmas tárolóba kerül. Szekvenciális programvégrehajtás mellett a módszer 4-től 8 utasításonként igényelne 1-1 memória hozzáférést a flash memória esetében (T_{ACC} = 22.2 - 44.4 ns). Ez már teljesíthető a mai flash memóriák sebessége mellett. A gyorsítótár a fel nem használt információk következő ütemben történő felhasználása, illetve a nem szekvenciális programvégrehajtás (ugrások, ciklusok) miatti címugrások áthidalására szükséges (természetesen egy eddig nem használt területről történő kódfelhozás esetében a késedelem nélküli olvasás nem garantálható – de ennek a valószínűsége normál programok esetében alacsony).

Table 5. Number of wait states according to CPU clock (HCLK) frequency

Wait states (WS) (LATENCY)	HCLK (MHz)			
	Voltage range 2.7 V - 3.6 V	Voltage range 2.4 V - 2.7 V	Voltage range 2.1 V - 2.4 V	Voltage range 1.8 V - 2.1 V Prefetch OFF
0 WS (1 CPU cycle)	0 <HCLK≤30	0 <HCLK ≤24	0 <HCLK ≤22	0 < HCLK ≤20
1 WS (2 CPU cycles)	30 <HCLK ≤60	24 < HCLK≤48	22 <HCLK ≤44	20 <HCLK ≤40
2 WS (3 CPU cycles)	60 <HCLK ≤90	48 < HCLK≤72	44 < HCLK≤66	40 < HCLK≤60
3 WS (4 CPU cycles)	90 <HCLK ≤120	72 < HCLK≤96	66 <HCLK ≤88	60 < HCLK≤80
4 WS (5 CPU cycles)	120 <HCLK ≤150	96 < HCLK≤120	88 < HCLK≤110	80 < HCLK≤100
5 WS (6 CPU cycles)	150 <HCLK ≤180	120 <HCLK ≤144	110 < HCLK≤132	100 < HCLK≤120
6 WS (7 CPU cycles)		144 <HCLK ≤168	132 < HCLK≤154	120 < HCLK≤140
7 WS (8 CPU cycles)		168 <HCLK ≤180	154 <HCLK ≤176	140 < HCLK≤160
8 WS (9 CPU cycles)			176 <HCLK ≤180	160 < HCLK≤168

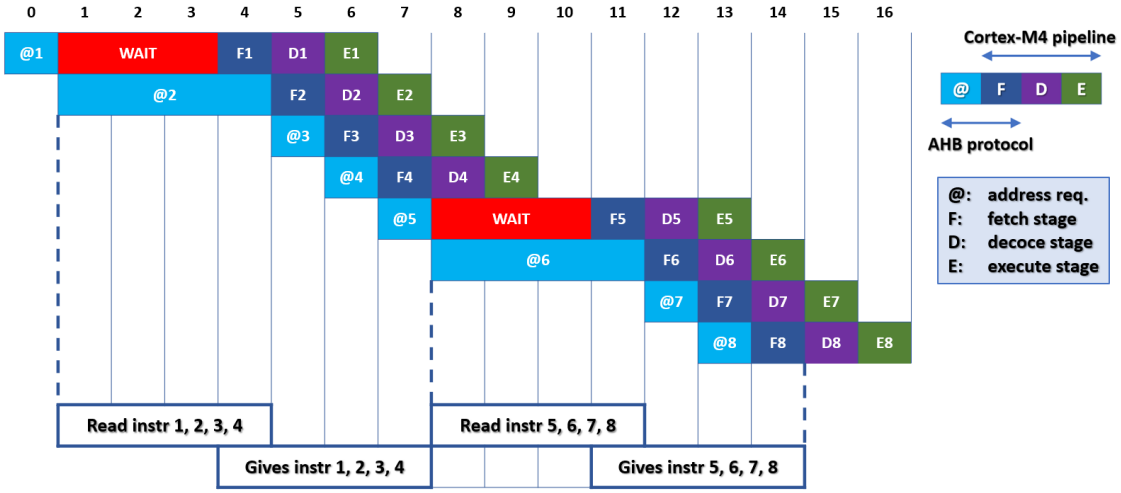
A kódmemória olvasásához szükséges váróciklusok száma az STM32F446RE esetében

A kódmemória hozzáférések esetében mind az alkalmazott CPU frekvencia, mind a mikrokontroller tápfeszültsége befolyásolja az alkalmazandó váróciklusok számát. Kisebb tápfeszültségek esetében a CMOS elemekre épülő logikák lassulnak, a szükséges váróciklusok számát (WS) a rendszer inicializáló programjának kell beállítani a kontroller vezérlő regiszterében.

A blokkos (burst-) olvasás, egyetlen olvasással 4 vagy 8 utasítás beolvasása még mindig nem oldja meg azt a problémát, hogy teljesen folytonos (szekvenciális) kód esetében is minden negyedik (nyolcadik) utasítás után nem fogjuk megtalálni a következő utasítást a gyorsítótárban, ez pedig a váróciklusok miatt biztosan időkiesést jelentene a folyamatos pipeline műveletvégzés során. A Cortex M4-es központi egységek 3 fokozatú műveletvégző egységet használnak:

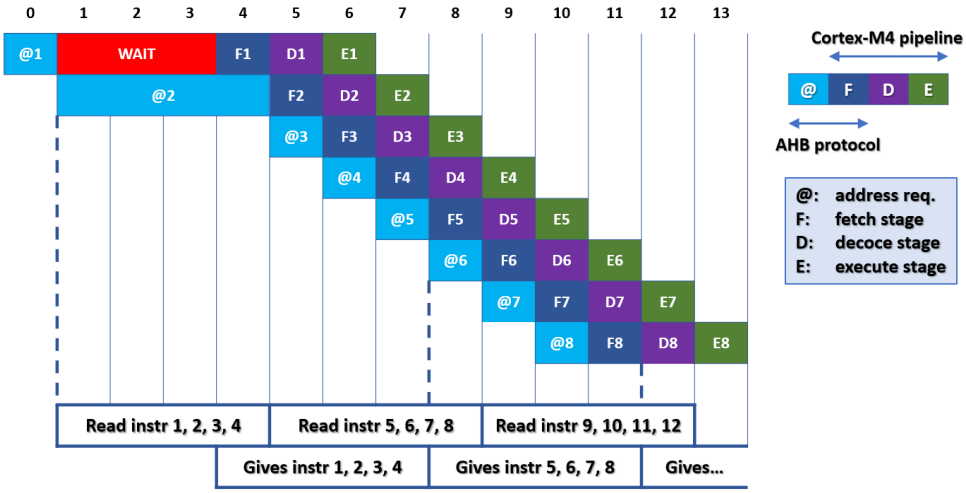
1. Utasítás beolvasó fázis (Fetch stage = F)
2. Dekódolási fázis (Decode stage = D)
3. Végrehajtási fázis (Execute stage = E)

A kialakuló helyzetet mutatja a következő ábra.



Kódmemória olvasása ART gyorsító tárral prefetch nélkül

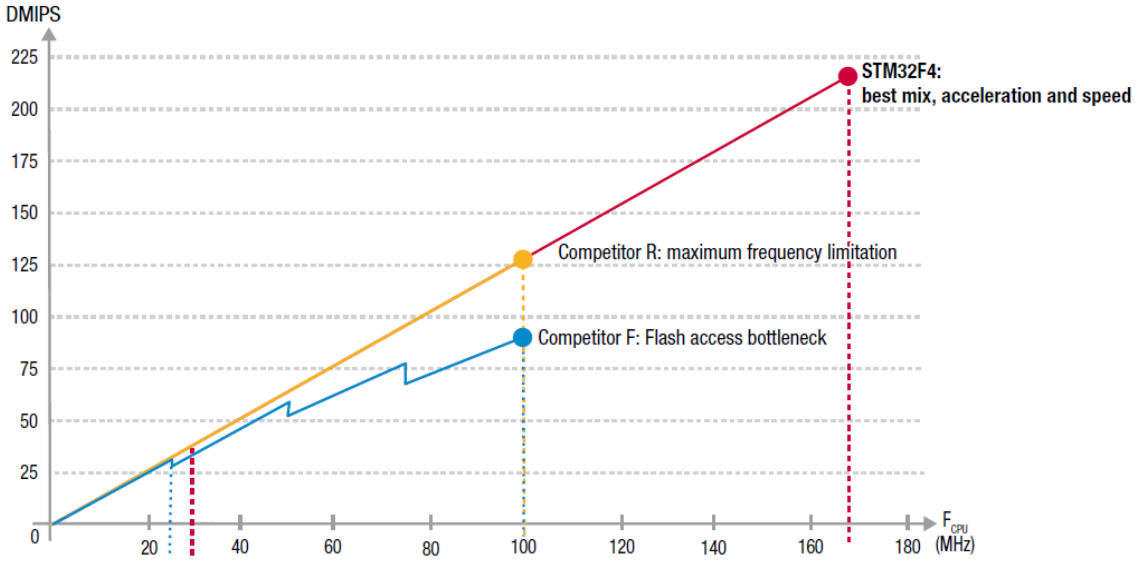
Az ART 128 bites beolvasási művelettel beolvassa a következő blokkot a gyorsítótárba. Ez azonban az ábrán bemutatott példa esetében csak 3 várociklussal lehetséges. Központi egységünk tehát várakozni kényszerül (WAIT), majd elkezdi az 1. utasítás egyes fázisainak a végrehajtását (F1-D1-E1). A 2.-4. utasítások már időkésés nélkül beolvashatók a gyorsítótárból, de az 5. utasításnál ismét elakadunk: az 5. utasítás 1 helyett csak 4 órajelperiódus alatt hajtódik végre. Ez ideális esetben (ugrások nélkül) is azt jelenti, hogy 4 utasítás 7 órajel alatt hajtódik végre, azaz a 100%-os (1 utasítás = 1 T) végrehajtási sebesség helyett csak 4 / 7 = 57.14%-os sebességgel működünk !



Kódmemória olvasása ART gyorsító tárral prefetch technikával

Az AHB azonban ennél többre képes: már az első utasítás beolvasása után megkezdi a következő utasítás beolvasását (prefetch – Read ins 5,6,7,8). Ily módon elérjük, hogy az utasítások CPU által történő beolvasása időkésés nélkül lehetséges, ily módon a működési sebesség 100%-os.

A következő diagram a gyorsítótár hatékonyságát mutatja (ismét DMIPS-ben). Jól látható, hogy az órajel frekvencia függvényében belépő várociklusok alkotják a működési sebesség szűk keresztmetszetét.



Az STM32F4 processzorcsalád teljesítőképessége ART gyorsító tárral

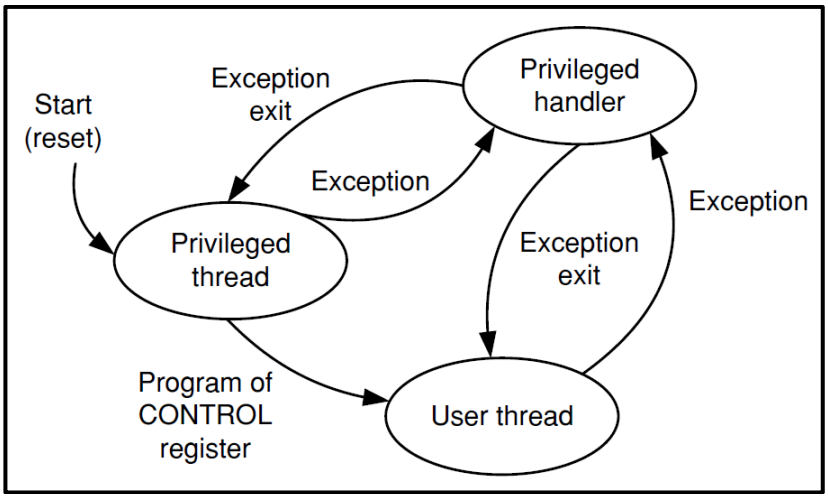
Az új ipari szabványnak tekintett Cortex-M processzorok egyszerűsített ARM architektúrával rendelkeznek, programozási modelljük sok tekintetben emlékeztet a korábbi 68000-es processzorcsalád tulajdonságaira (Motorola). Az egyszerűsített architektúra alulról kompatibilis a klasszikus ARM architektúrával, kevesebb üzemmód lehetséges (**thread** mód = alkalmazás szintje és **handler** mód = kivételeljárások szintje).

		Operations (Privilege out of reset)	Stacks (Main out of reset)
Modes (Thread out of reset)	Handler - Processing of exceptions	Privileged execution full control	Main stack used by OS and exceptions
	Thread - No exception is being processed - Normal code execution	Privileged or unprivileged	Main or process

A Cortex M4 processzorcsalád üzemmódjai

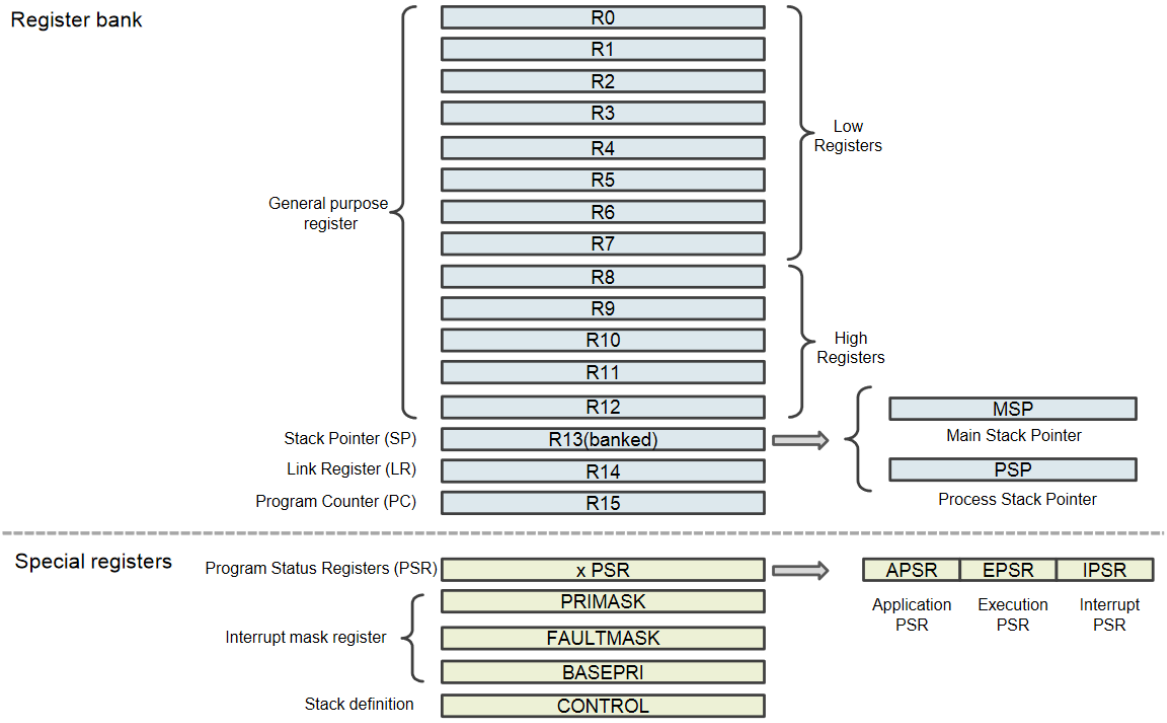
A mikrokontroller reset után privilegizált thread módban éled, ezzel lehetősége van a központi egység valamennyi erőforrásának elérésére. Normális működés esetén a rendszer inicializálását követően nem privilegizált thread módba kapcsol (felhasználói mód), ekkor már életbe lépnek a rendszer biztonságos működését garantáló korlátozások (memóriaterületek védelme, egyes rendszerszolgáltatások korlátozása). A kivételeljárások (megszakítások és bizonyos rendszerhibák kiszolgáló rutinjai)

természetesen ismét privilegizált üzemmódban futnak, biztosítva ezen rendszerelemek teljes jogú hozzáférését valamennyi komponenshez.



A Cortex M4 processzorok üzemmódjainak állapotgráfja

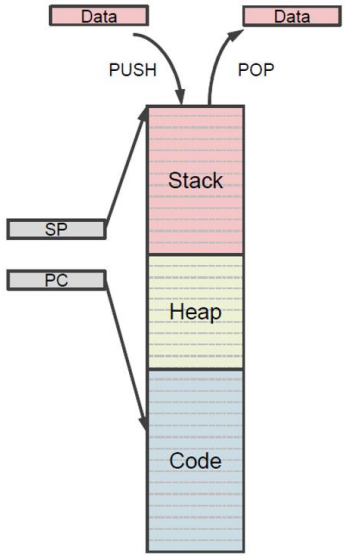
A központi egység 16 db 32 bites regisztert és néhány további vezérlőregisztert tartalmaz. Az R0..R12 regiszterek egyenértékűek, az R13 regiszter alkotja a veremmutatót (SP), az R14 an ún. Link regisztert (LR, ld. később), az R15 pedig a programszámlálót (PC).



A Cortex M4 processzorok regiszterei

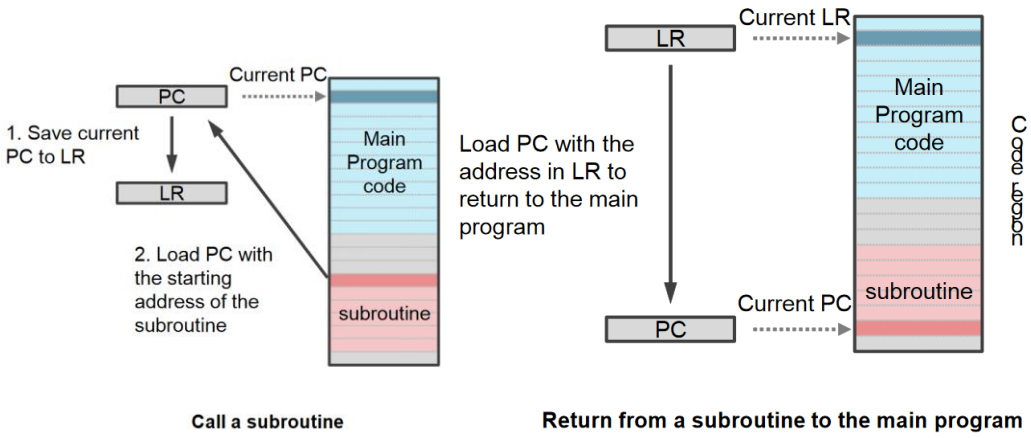
Az R0..R7 regiszterek (Low Registers) és az R8..R12 regiszterek (High Registers) még egy szempontból elkülönülnek: mint láttuk, a Thumb és Thumb-2 utasítások a kisebb utasításhossz miatt csak 8 regiszter kezelésére képesek, ezek mindig csak a kisebb sorszámú (Low) regiszterek lehetnek. Az R13-R15 regisztereknek (melyek azért saját speciális funkcióval bírnak) a normál regiszterek közé „keverése” számos előnnyel jár azok kezelése szempontjából (utasítások, címzési módok egységes szerkezetben kezelik).

A PC és az SP teljesen szokásos módon működnek az ARM architektúrában is:



A PC (R15) és az SP (R13) működése az ARM processzorokban

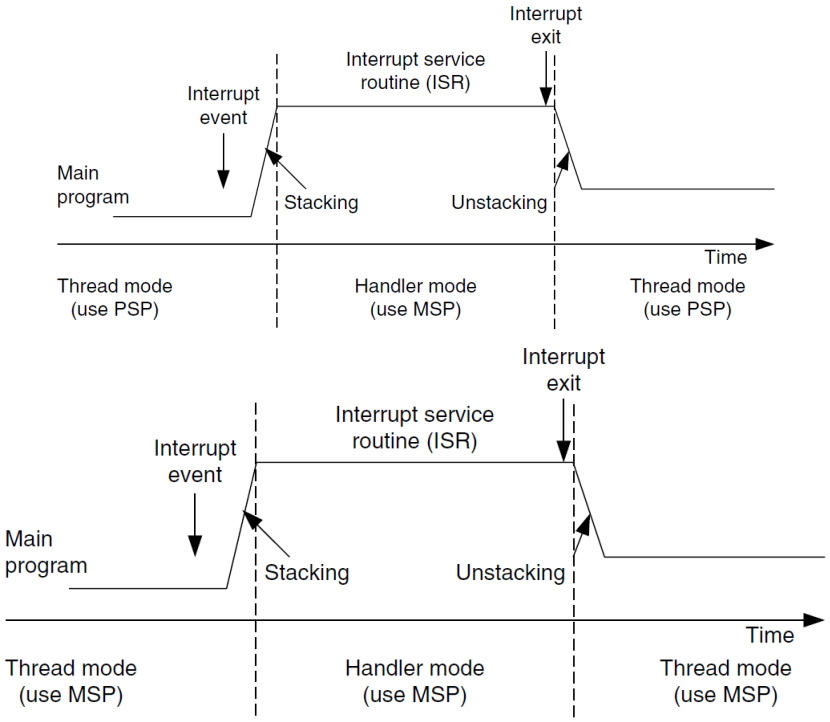
Külön ki kell térnünk a Link regiszter (R14) szerepére. Ez a regiszter egyszerűen egy rendkívül gyors szubrutin (függvény-) hívást tesz lehetővé, az LR hívás után mindig a visszatérési címet tárolja.



Az LR (R14) működése az ARM processzorokban

Mi az oka ennek a szokatlan megoldásnak a visszatérési címnek a verembe (stack) történő mentéssel szemben ? Egyrészt a megoldás gyorsabb (regiszterbe ment a memóriával szemben), rövid algoritmusok struktúrált programozási szempontok szerinti elkülönítése sokkal jobban megvalósulhat. Az LR használata ”levélszerű” szubrutinokat eredményez a program gráfjában (a gráfelméletben a fák elsőfokú csúcsait hívjuk levélnek). Az optimalizáló fordítók ezt a tulajdonságot előszeretettel használják itt nem részletezendő programtulajdonságok megvalósítására. Ugyanakkor a többszörös szubrutinhívás előtt már mentenünk kell a LR-t is a verembe (más regiszterekkel együtt), hiszen a következő hívás a tartalmát felülírná.

A veremmutató (SP) két különböző regiszter egyike az aktuális üzemmódtól függően. A biztonságos működésre törekvő rendszerek elkülönítik a felhasználói és a rendszer vermet, mivel a véletlenül vagy szándékosan helytelen működésű felhasználói program (taszk) kritikus rendszerváltozókat írhat felül a rendszer veremben, meghiúsítva ezzel a rendszer biztonságos működését. A PSP (Process SP) a felhasználói verem mutatója, ebbe dolgoznak a felhasználói taszkok, az MSP (Main SP) a rendszer verem mutató, ebbe a privilegizált programegységek dolgoznak. Mindkét regiszter elérhető privilegizált módban, de az R13 regiszter helyén mindig az üzemmód szerinti regiszter található.

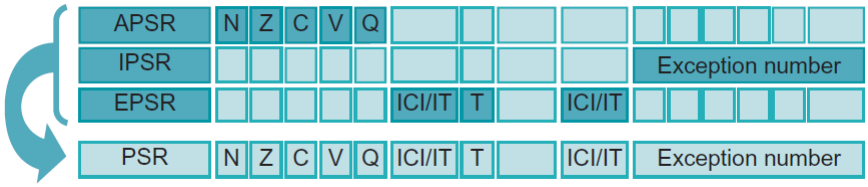


Az SP és az üzemmódok kapcsolata user thread (fent) és priv. thread (lent) módban

A speciális regiszterek közül megemlítjük a státuszregisztert (PSR), amely egyrészt a szokásos feltételbiteket (flageket) tartalmazza, azonban – lévén ez is 32 bites regiszter – ennél sokkal több feladatot is ellát. A regiszterben a következő bitek és bitcsoportok találhatók:

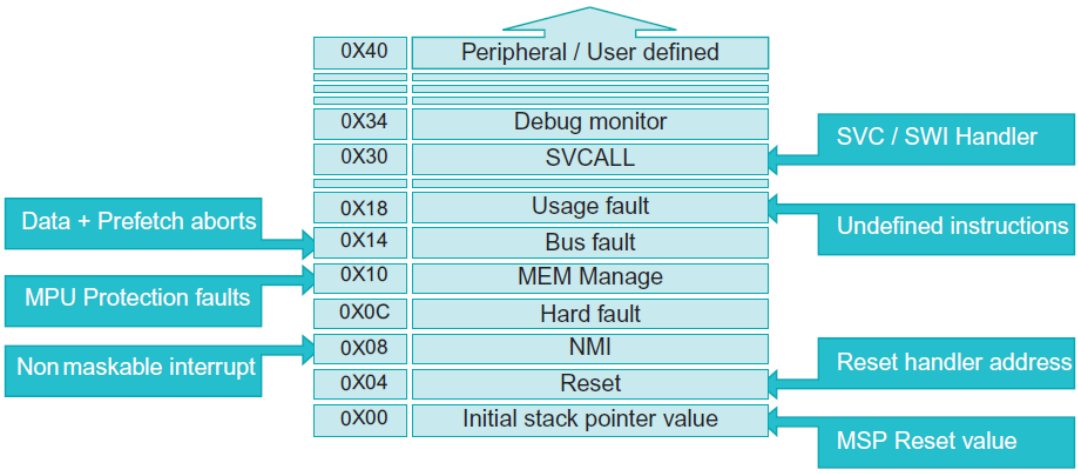
- Feltételbitek (N, Z, C, V, Q)
  - N = negatív
  - Z = zero
  - C = carry
  - V = overflow
  - Q = saturation
- Kivételeljárás száma (Exception number)
- Thumb üzemmód (T)
- Segédbitek többciklusos utasítások / Thumb-2 IT (if-then) utasítás megszakításának kezeléséhez (ICI/IT)

A PSR regiszternek három „másolata” (alias) is létezik APSR, IPSR és EPSR néven. A regiszterek ilyen néven is hivatkozhatók, és ilyenkor az előbb felsorolt bitek, bitcsoportok csak részlegesen jelennek meg, nem téve szükségessé a maszkolásukat lekérdezés vagy beállítás esetén.



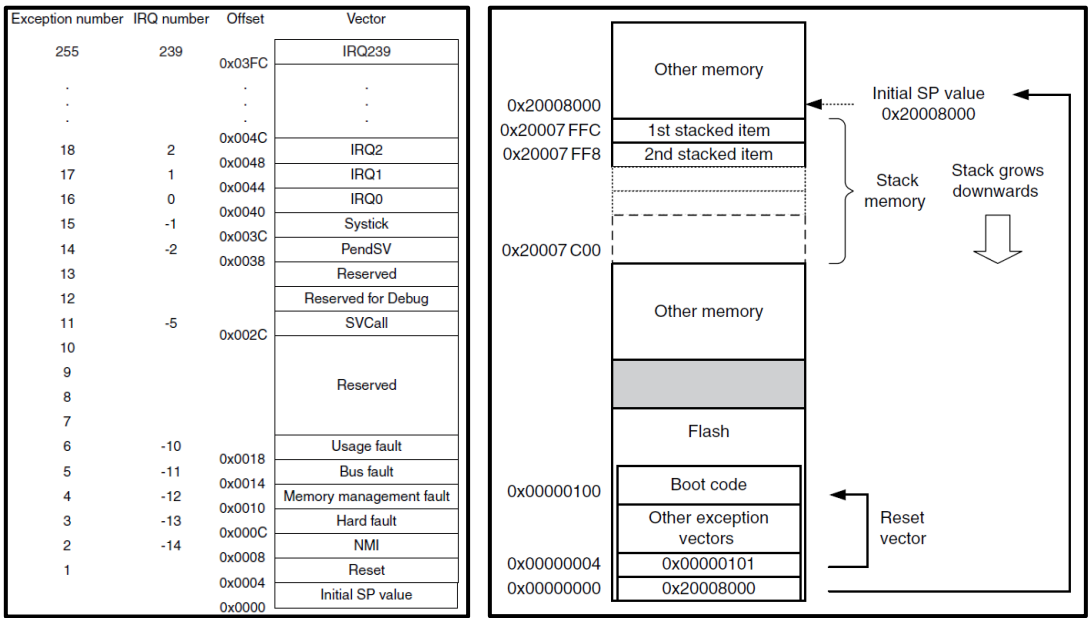
A státuszregiszter (PSR) és alias regiszterei

Bármely mikroprocesszor egyik legfontosabb jellemzője a megszakításrendszerének felépítése. Nagyobb programrendszerek, operációs rendszerek fontos igénye a megfelelő prioritási tulajdonságokkal rendelkező hardver megszakításvezérlő. Az ARM processzorok 256 prioritásszintet különböztetnek meg a kivételeljárások számára, vektortáblát használnak a beérkező eseményeket lekezelő rutinok eléréséhez. (A vektortáblák csak az adott szint rutinjának címét tartalmazzák, szemben pl. a 8051-nél tanult ugrótáblával, ahova az egyes rutinok számára komplett utasításokat kell elhelyeznünk.)



A megszakítási vektortábla felépítése

A 256 megszakítás közül az első 16 speciális feladatot lát el. Ezeket az ARM terminológia „negatív” megszakítási szinteknek nevezi és -1-től -16-ig számozza. A többi 240 szint (0..239) a felhasználói megszakítás szintjei. A 16 speciális szint részben a rendszer indulásakor automatikusan beállítandó értékeket (0x0000 = induló SP, 0x0004 = induló PC reset után), részben a speciális rendszer kivételeljárások és hibák (NMI, Hard fault, MMU fault, Bus fault, stb.) kiszolgáló rutinjainak címeit tartalmazza. Minden vektor 1 szó (4 bájt) hosszú, tehát a bájtszervezésű memóriában (ahol minden bájt saját címmel rendelkezik) a vektortábla címei négyesével növekednek.



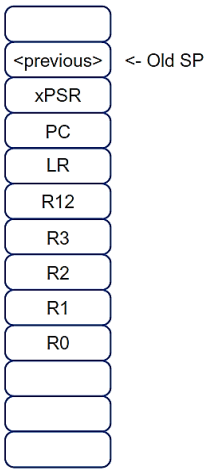
A megszakítási vektortábla – a mikrokontroller indulása reset után

Fontos kérdés a megszakítások prioritása, a prioritási szintek száma és a megszakítások egymás által történő megszakíthatósága. Ez a tulajdonság alapvetően befolyásolja a megszakítások reakcióidejét, a rutinok kialakítását és adott célra való felhasználhatóságukat. A Cortex ARM magok esetében a megszakítások lehetséges száma nagy (240+16), a prioritás minden egyes megszakításhoz megadható külön konfigurációs regiszterekkel (NVIC_IPRx), regiszterenként 4 db 8 bites prioritást definiálva. A prioritások értéke elvben 0..255 lehet, minél kisebb a számérték, annál nagyobb a megszakítás prioritása. Valójában a Cortex M4 processzorokban csak 16 prioritási szint értelmezett a 8 bites prioritások [7..4] bitjei alapján, az alsó 4 bit értéke ezekben a processzormagokban közömbös. A felső 4 bitet is két bitcsoportra osztják a Cortex processzorok: a magasabb helyiértékű bitekből kialakított csoport képezi a ténylegesen elkülönülő prioritási szinteket (Group priority bits), míg a maradék kisebb helyiértékű bitek a prioritási csoporton belüli alprioritásokat mutatják (Subpriority bits). A kívánt felosztás egy konfigurációs regiszterben (AIRCR, PRIGROUP bitek) adható meg összesen ötféle konfigurációban, ezzel 0/2/4/8/16 csoportot képezhetünk 16/8/4/2/0 alprioritási szinttel.

PRIGROUP [2:0]	Interrupt priority level value, PRI_N[7:4]			Number of	
	Binary point[1]	Group priority bits	Subpriority bits	Group priorities	Sub priorities
0b0xx	0bxxxx	[7:4]	None	16	None
0b100	0bxxx.y	[7:5]	[4]	8	2
0b101	0bxx.yy	[7:6]	[5:4]	4	4
0b110	0bx.yyy	[7]	[6:4]	2	8
0b111	0b.yyyy	None	[7:4]	None	16

A megszakítások 8 bites prioritásainak felosztása prioritási szintekre (Group priority) és alprioritásokra (Subpriority)

Az általános szabály: kisebb csoport prioritási szintű (nagyobb számértékű), vagy azonos csoport prioritási szintű megszakításkérések (melyek tehát csak az alprioritásuk számában különböznek) nem tudják egymást megszakítani, ezek a megszakítások csak egymás után – az előző megszakítás lezárultát követően – jutnak érvényre. A különböző prioritási csoportokba tartozó megszakítások közül a magasabb prioritású (kisebb számértékű) megszakítás meg tudja szakítani az alacsonyabb prioritásút. Azonos prioritási szinten belül az alprioritások értéke szabja meg a várakozó megszakításkérések egymás utáni érvényre jutásának sorrendjét.

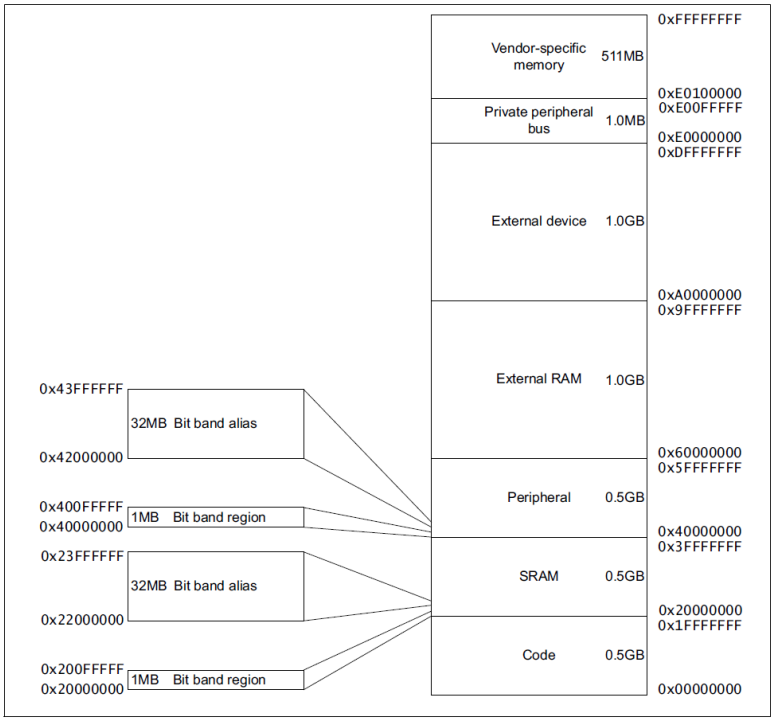


Exception frame

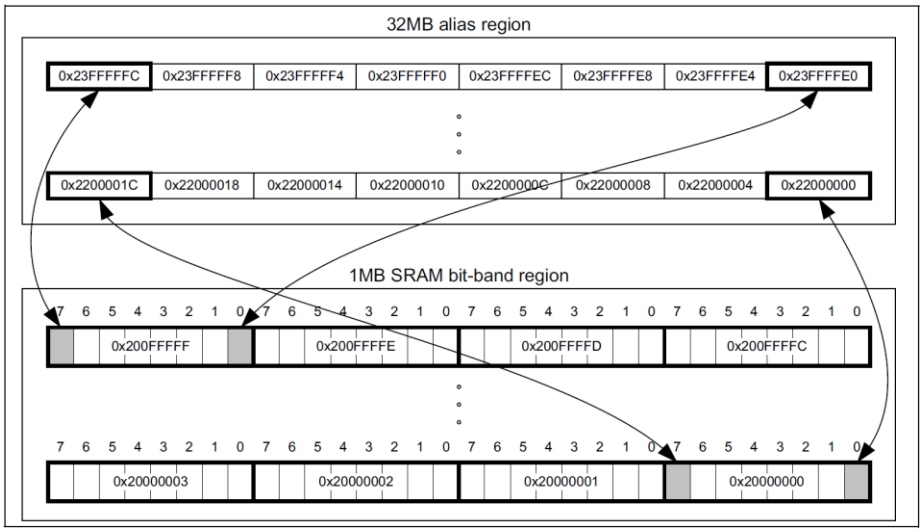
Ha egy megszakítás érvényre jut, automatikusan menti a legfontosabb regisztereket (exception frame). A mentés bővülhet FPU használata esetén még az FPU regiszterekkel is, ennek részleteivel itt terjedelmi okokból nem foglalkozunk.

A 8051-es családnál tanult, vezérlési célokra kiválóan használható bitváltozókat az ARM architektúra is bevezette és használja. A nagyobb címtartományoknak megfelelően itt két bitmezőt definiáltak: egyet az adatmemória, egyet pedig a perifériák számára lefoglalt

címterületen belül. Mindkét terület 8-8 Mbit (2 * 8 388 608 db) bitváltozót hoz létre (bit band region), melyek szavas hozzáféréssel is elérhetőek (2 * 256 kszó), vagy pedig 2 db 32 Mbájtos (8 megaszavas) ún. alias területen keresztül írhatók-olvashatók az egyes bitek, ahol minden bithez egy-egy 32 bites szó került hozzárendelésre. Az alias hozzáférések történhetnek bájt, félszó és szó művelettel is, az adott bit mindig az alias mező 0-dik bitjének felel meg.

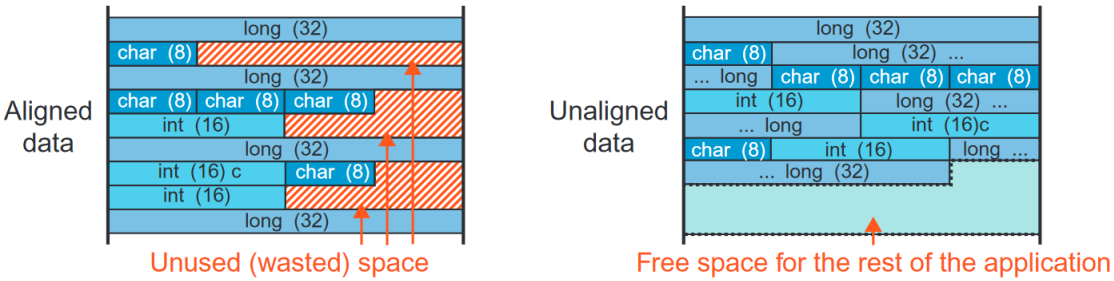


A bitmezö (bit-band) területek és alias elérhetőségük



A bitmező (bit-band) területek és alias területeik összerendelése

A Cortex processzorok adatoperandusai bájt (8 bit), félszó (16 bit) és szó (32 bit) hosszúságúak lehetnek, mint már említettük a memória bájt szervezésű (minden bájt önálló memóriacímmel rendelkezik). Az ARM architektúrában kezdetben szigorúan megkövetelt szóhatárra illesztettséget (félszó csak páros, szó csak 4-gyel osztható címen helyezkedhetett el, ún. word-aligned operandusok) a Cortex processzoroknál feloldották, és megengedik a szóhatárra nem illeszkedő (non word-aligned) operandusokat is bizonyos adathozzáférések esetében. A megoldás magyarázata egyértelműen a nagysebességű, drága memóriával való nagyobb takarékoskodás – szükség esetén egy-egy folytatóciklus beiktatásával szóhatárt átlépő operandus esetén.

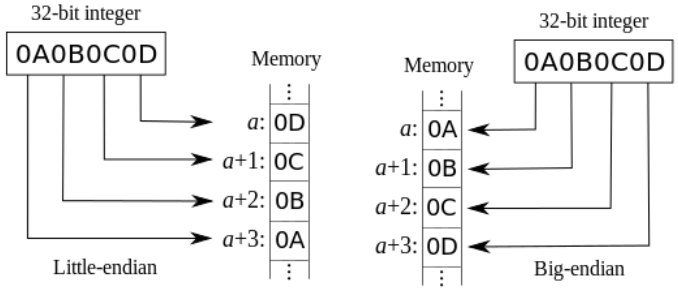


Illesztett és illesztetlen operandusok a Cortex processzorok memóriájában

Fontos megemlítenünk az adatok elhelyezkedését a memóriában (ún. bájtsorrend kérdése). A bájtszervezésű memóriában a bájtnál nagyobb méretű félszó (16 bit) és szó (32 bit) elhelyezése többféle módon is történhet: adott címtől (*a* = az operandus kezdőcíme) sorban tesszük le egymás után az *a*, *a+1*, *a+2*, *a+3* címekre a 32 bites operandus négy bájtját, de kérdés, hogy milyen sorrendben ? Két módszer terjedt el a mikroprocesszorok világában:

1. a „legkisebb elöl” (**little-endian**) sorrend, mikor a legkisebb helyiértékű bájt (LSB) kerül az *a* címre, és a legmagasabb helyiértékű (MSB) az *a+3* címre,
2. a „legnagyobb elöl” (**big-endian**) sorrend, amikor a legnagyobb helyiértékű bájt (MSB) kerül az *a* címre, és a legkisebb helyiértékű (LSB) az *a+3* címre.

(Jelen esetben az LSB és az MSB rövidítésekben „B” bájtot jelöl.)



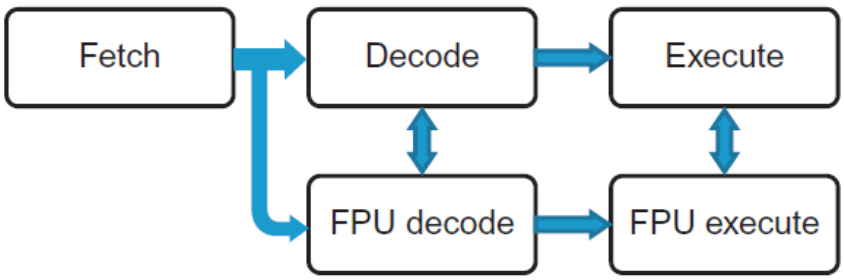
Használatos bájtsorrendek az operandusok tárolásánál a memóriában

A mikroprocesszor regisztereiben minden esetben az LSB kell legyen a 7..0 biteken, így nem mindegy tehát hogy a memóriában található 4 bájton elhelyezkedő szót milyen sorrendben töltjük be a regiszterbe.

Miért használatos egyáltalán ez a kétféle tárolási mód ? A több-bájtos aritmetikai műveletek esetében a little-endian mód használata előnyösebb, mivel az aritmetikai műveletek többségét LSB-MSB sorrendben (a növekvő helyiértékek irányában) hajtjuk végre, egy egyesével növekvő memória mutató pont ilyen sorrendben hozza fel az operandusok bájtjait a memóriából. Karakterekkel (8 bit) történő műveleteknél viszont a memóriában elhelyezkedő szöveg (ahol növekvő címek irányában jönnek egymás után a szöveg betűi) számára pont a big-endian sorrend a kedvezőbb a szövegben az elejétől a vége felé való haladás esetén (ilyenkor egy 32 bites szó 4 db karaktert tartalmaz egyszerre). Persze ezek inkább „belemagyarázó” okoskodások a sorrendek igazolására, egyértelműen egyik sorrend sem kiáltható ki jobbnak a másiknál.

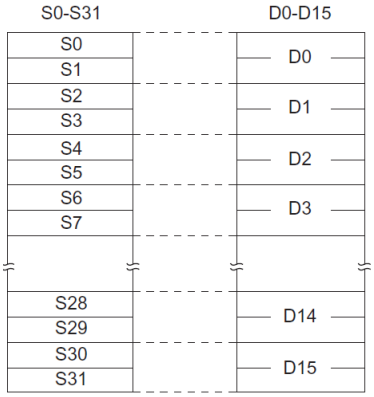
Az Intel processzorok kezdettől fogva a little-endian bájtsorrendet követik, a Motorola processzorok pont fordítva, a big-endiant. A leírtakból viszont következik, hogy a kérdés kódhordozhatóság-problémához vezet: ameddig egy processzor a saját maga által a memóriába „letett” adatokat olvassa vissza, a bájtsorrend fajtája közömbös. Amint azonban a memóriába előre „betöltünk” adatokat (konstansok, táblázatok, soros kommunikációval beolvasott bájtsorozatok), már nem mindegy, hogy milyen értelmezéssel olvassuk be azokat. Innen kezdve a legfontosabb kérdés: melyik bájtsorrendet használják az Cortex processzorok ? A dolog fontosságát felismerve ne csodálkozzunk a válaszon: egy konfigurációs bittől függően reset után mindkét sorrend beállítható a processzorok számára (természetesen vagy egyik, vagy másik a konfigurációs bit állásától függően). Processzortípustól függően ez a tulajdonság korlátozás alá eshet, így a Cortex-M4 mikrokontrollerek esetében is az AIRCR (Application Interrupt and Reset Control Register) 15. bitje (=ENDIANNESS) csak 0 értéket vehet fel, ami a little-endian bájtsorrendnek felel meg. Cserében viszont utasítások állnak rendelkezésre (REV, REV16, REVSH), melyekkel a beolvasott bájtsorrend megfordítható (pl. egy kommunikáció során kapott big-endian hosszúszó bájtsorrendjének megfordításához).

Ejtsünk pár szót a lebegőpontos műveletvégző egységről (FPU). Ez az opcionális (egyes Cortex-M4 processzorokban megtalálható, másokban nem) egység a törtszámokkal való műveleteket nagymértékben gyorsítja és egyszerűsíti. Az egység nem épül be a CPU magba, hanem azzal párhuzamosan dolgozik, az utasítás felhozatalakor dől el, hogy az a normál CPU vagy az FPU számára jelent-e végrehajtandó feladatot.



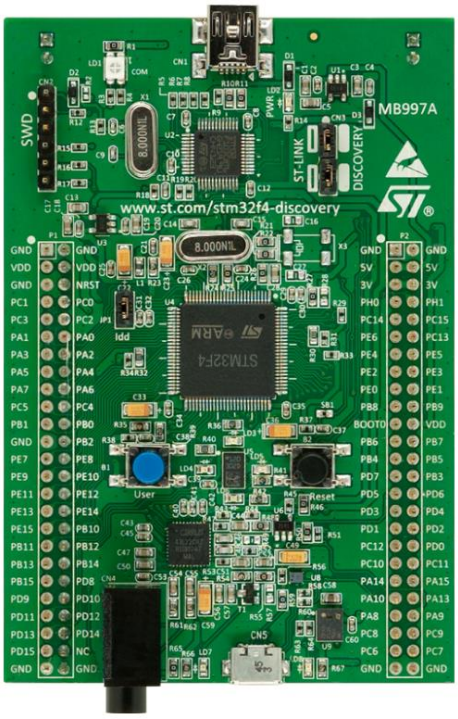
Normál és lebegőpontos utasítások végrehajtása a Cortex mikrokontrollerekben

Az elkülönülő FPU saját regisztereket használ (S0..S31), melyek egyszeres pontosságú lebegőpontos számok tárolására alkalmasak. 2-2 egymás melletti regiszter összefogható egy 64 bites regiszterré (D0..D15), amelyek viszont már duplapontosságú lebegőpontos számokként értelmezhetők, mindkét esetben az IEEE-754 lebegőpontos szabvány előírásainak megfelelően (részletesen ld. a 2.18 fejezetben). Ezek a regiszterek töltő-tároló utasítások segítségével tudnak adatot cserélni a normál CPU regiszterekkel.

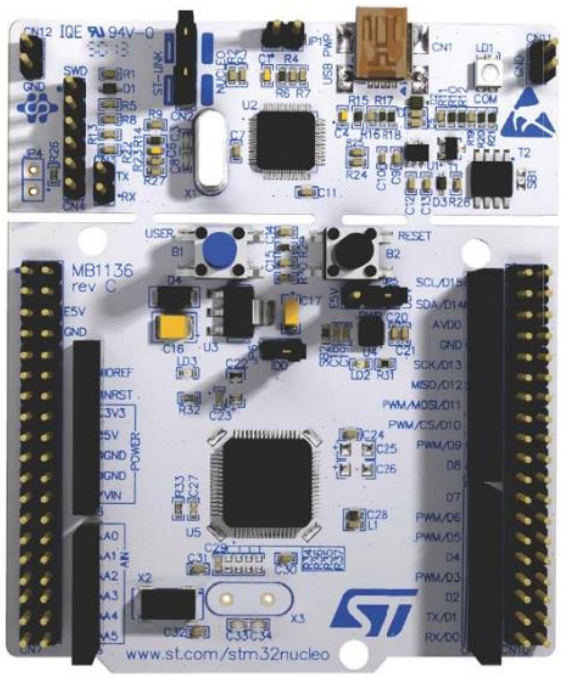


FPU egyszeres és duplapontosságú regiszterek

Végezetül egy-egy ábra a bemutatott Cortex-M4 mikrokontrolleren alapuló fejlesztő kártyákról, mellyel részletesebben a Mikrokontroller laboratórium tantárgy keretében folyó mérések során ismerkedhetnek meg.



STM32 F4 Discovery kit
STM32F407VG mikrokontrollerrel



STM32 Nucleo-64 kit STM32F446RE
mikrokontrollerrel

1.7 Jelfeldolgozó processzorok (kiegészítő tananyag)

(Jelen fejezet nem képezi a tananyag kötelezően elsajátítandó részét. Ajánljuk az érdeklődő – pl. IMSc programban résztvevő – hallgatóknak ismereteik bővítéséhez.)

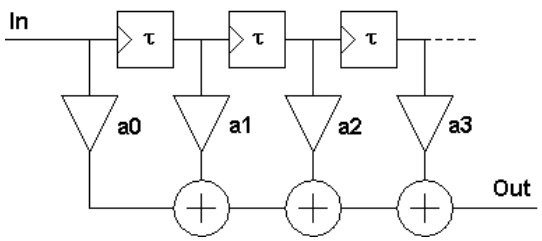
A digitális jelfeldolgozás céljaira készített processzorok az 1980-as évek elején jelentek meg a piacon. Mint arról már beszéltünk, ebben az időben a mikroprocesszorok bizonyos területeken (elsősorban a nagyobb sebességű jelek feldolgozása és az ezekhez kapcsolódó folyamatok irányítása) való alkalmazhatóságának egyik korlátja a technológiailag korlátos működési/műveletvégzési sebesség volt. Az elvégzendő – sokszor elég speciális műveletsorokat követelő – feladatok aritmetikai alapműveletekből való összeállítása tovább osztotta az egyébként is erősen ”véges” műveletvégzési sebességet. Ezeknek a területeknek a meghódítására indult el a jelfeldolgozó processzorok fejlesztése, melyek meghatározó tulajdonságai voltak:

- lehető legnagyobb műveletvégzési sebesség
- a műveletek elvégzéséhez megfelelő szóhosszúság
- speciális – de tipikus – műveletsorok összefogása egyetlen műveletbe
- speciális címzési módok ezen műveletek támogatására
- integrált nagysebességű elemek (elsősorban memória) a buszhozzáférések gyorsítására.

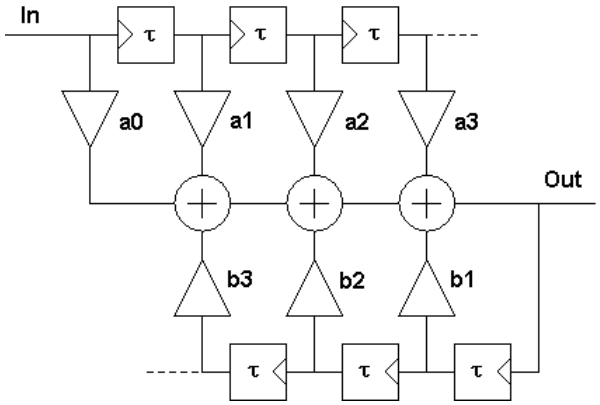
A digitális jelfeldolgozásra való képesség megkövetelte az architektúra és az utasításkészlet oly módon való kialakítását, hogy az lehetővé tegye minimum a három alapművelet (osztás hiányzik!), a logikai és eltolási műveletek (barrel shifter!) valóban nagysebességű hardverrel történő végrehajtását, amelynél az ez idő tájt uralkodó mikroprogramozott megoldások szóba sem jöhettek. A sebesség növelése érdekében kezdettől fogva megpróbálták átlapolni az utasítások elvégzését (pipeline műveletvégzés), ez azonban a technológia akkori állása szerint igen speciális – sok esetben kellemetlen – utasításstruktúrát eredményezett (pl. előző utasításoknak már hivatkozniuk kellett a következő címére, hogy az utasítás felhozatala “párhuzamosan” megtörténhessen). A belső buszrendszer vegyes architektúrát követelt meg a memória hozzáférések gyorsítása érdekében. (Az osztott buszrendszer Harvard architektúrát takar, az adat- és programmemória keveredése azonban Neumann jellegre utal.) Ez ebben az időben egyáltalán nem volt tipikus a processzorok világában.

A digitális jelfeldolgozás során elvégzendő tipikus műveletsorok tekintetében csak néhány példát említünk:

- a véges válaszidejű szűrők (FIR – Finite Impulse Response) csak a bemenőjelből vett mintákat használják a jelfeldolgozás során



- a végtelen válaszidejű szűrők (IIR: Infinite Impulse Response) ezzel szemben a kimenőjel előző állapotait is felhasználja a jelfeldolgozás során



- a Fourier transzformáció (folytonos/diszkrét/gyors) a jelfeldolgozás egyik leggyakoribb módszere, amely az időtartományt a frekvenciatartományba transzformálja és a jel spektrumának tulajdonságait használja a jel tulajdonságainak elemzésére/tárolására.

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2k\pi}{N}n}$$

A digitális jelfeldolgozó rendszerek tipikus elemei a szűrők. Ilyen szűrőket a rendszer tömegesen igényel, a megvalósító algoritmusok többnyire a fent említett digitális jelfeldolgozó algoritmusok. Megvalósításukhoz a hagyományos mikroprocesszorok túlságosan lassúak voltak. A szűrő algoritmus végrehajtása ugyanis mintavételi periódusonként nagyszámú szorzást és összeadást igényel. Egy véges impulzusválaszú (FIR) szűrő az alábbi egyenlet szerint működik:

$$y(n) = a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + \ldots$$

Vagyis elvégzéséhez olyan aritmetikai egységre van szükség, amely lehetőleg egy gépi ciklus alatt képes a szorzás és összeadás, esetleg még a tárolás összetett műveletének az elvégzésére. Emellett a műveletvégző mikroprocesszornak olyan címzési módokkal kell rendelkeznie, amelyekkel a mintavételezett értékek és a hozzájuk tartozó együtthatók gyorsan elérhetők. Nem utolsó sorban pedig legyenek ezek a processzorok olcsók, mivel nagy mennyiségben kerülnek alkalmazásra. E követelményeknek megfelelően fejlesztették ki a digitális jelfeldolgozó processzorokat.

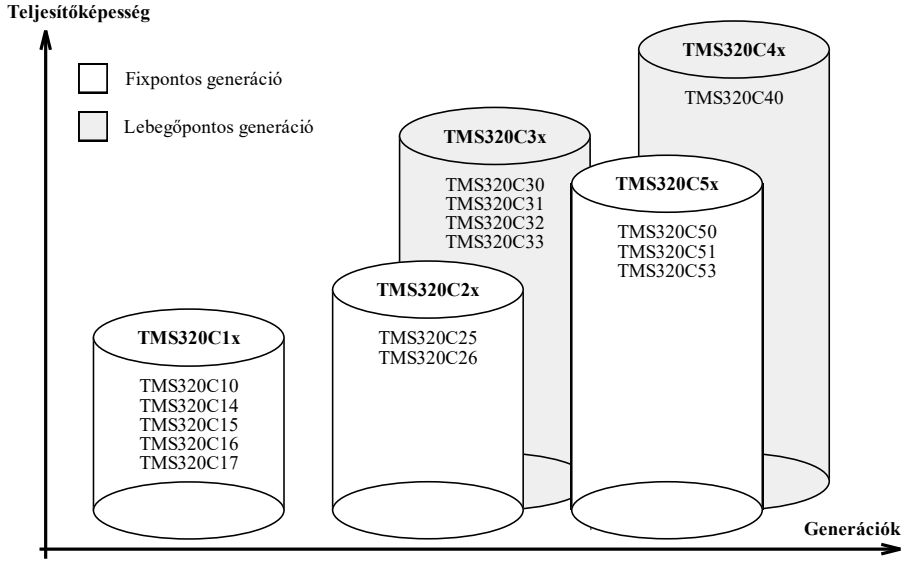
Az úttörő a folyamatban a Texas Instruments cég volt, akinek első jelfeldolgozó processzorát, a TMS32010-est 1983. április 8-án jelentették be. 1984-ben jelent meg a későbbi 16 bites fixpontos műveletvégző aritmetikával ellátott TMS320C10-es processzor (teljesen kompatibilis a TMS32010-zel), melyet digitális szűrők megvalósítására fejlesztettek ki, hatalmas sikert aratott és elnyerte az "év chipje" címet. Azóta a DSP technika önálló iparággá nőtte ki magát. Alkalmazásuk rendkívül gyorsan terjedt el az élet minden területén. Népszerűségük elsősorban a rendkívül kedvező

teljesítmény/ár viszonnyal és a rendelkezésre álló széles választékkal magyarázható. Ma már minden alkalmazáshoz megtalálható a megfelelő processzor az egyszerű 16 bites fixpontos áramkörtől a lebegőpontos multimédia processzorig.

A folyamathoz hozzátartozik még két igen fontos további szempont is:

- A jelfeldolgozó processzorok tervezői kezdettől fogva sok tekintetben hasonló célokat tűztek ki maguk elé, mint a RISC architektúrát kidolgozó szakemberek. Nem elemeznénk részletesen a folyamatot, a „ki kitől tanult többet" kérdés eldöntését bízzuk inkább a számítástechnika történetét feldolgozó szakemberekre. Számunkra a végeredmény a lényeges: a két architektúra igen sok hasonló tulajdonságot mutat – jóllehet a jelfeldolgozó processzorok bizonyos speciális tulajdonságai szóba sem jöhetnek a RISC architektúra előírásai között.
- A jelfeldolgozó processzorok is „átestek" a legfontosabb memória és perifériaelemek integrálásán – akár sebességi, akár alkalmazhatósági szempontok alapján nézzük a követelményeket, azok teljesen hasonlóak voltak a mikrokontrollerek világához. Az architektúrális kialakítás hasonlósága révén a közelmúltban a Texas Instruments (mint az egyik legnagyobb DSP gyártó) elkezdte használni a DSC (***D**igital **S**ignal **C**ontroller*) megjelölést jelprocesszoraira.

A TMS320 család tagjait a Texas cég generációkba sorolja (ez nem azonos a mikroprocesszorok osztályozásánál megismert generációkkal), így beszélünk a 10-es, 20-as, 30-as, 40-es és az 50-es generációról (vagy családtagokról), egy-egy generáció általában szintén több, azonos képességű de különböző kialakítású típust foglal magába. (Megjegyezzük, hogy a családnak vannak további generációi is, mint pl. a C80-as jelprocesszor. Ezek kialakítása azonban lényegesen eltér az említett családtagokétól, így ezek ismertetésével itt nem foglalkozunk.)



A Texas Instruments TMS320 jelfeldolgozó processzor családjának generációi

A “nagy ugrást” a családban a 30-as családtagok jelentették (1988), mivel itt jelent meg az integrált lebegőpontos műveletvégzés. Hozzáférhetősége és ára miatt sokáig ez a generáció volt a realitás (a 40-es széria túl drága volt, és nehezebben hozzáférhető). Természetesen azóta a Texas több nagyteljesítményű jelprocesszor-családdal is megjelent (C2000-C6000, DaVinci sorozat). Utóbbiak már a média feldolgozás területére kidolgozott extrém magas teljesítőképességű sorozatok 8000 MIPS teljesítőképesség körül. Mi ebben a fejezetben az alapelvek megértéséhez a 30-as generáció tulajdonságaival foglalkozunk.

(A TMS320C3x család tagjait napjainkban a Texas cég a családtagok eredeti változatainál már magasabb működési órajel-tartományban kínálja SM320C3x ill. katonai kivitelben SMJ320C3x típusjelölésekkel aktív státuszban, az eredeti változatokat új fejlesztésekhez már nem ajánlja.)

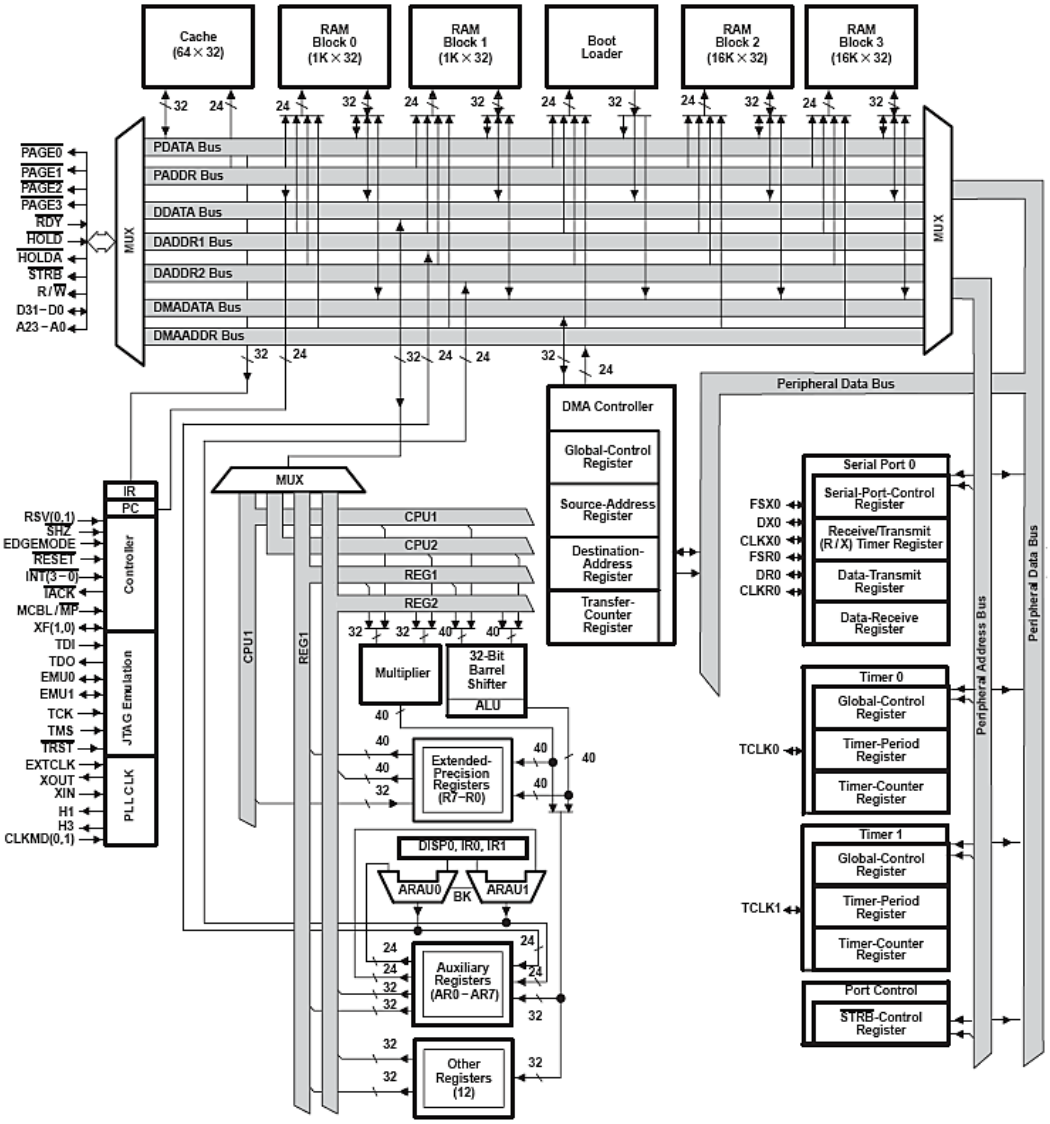
1.7.1 A TMS320C3x generáció belső architektúrája

A TMS320C3x jelprocesszor család "legáltalánosabb" tagja a C30-as. A C31-es ennek némileg egyszerűsített változata, az egyszerűsítésekkel a Texas az egyszerűbb alkalmazhatóságot és az alacsonyabb árat célozta meg. Az egyszerűsítés annyira "bejött", hogy 4-5 évvel később megjelent a C32-es jelprocesszor, a család talán legjobban elterjedt, legolcsóbb tagja, amellyel a Texas cég a C31-es tapasztalatai alapján az egyébként nem olcsó jelprocesszorok árát drasztikusan letörte. **A C33-as a család legfiatalabb tagja**, amely több mint 10 évvel a C30-as után látott napvilágot.

Ezzel a típussal a többit háttérbe is szorították: a technológia és az alacsonyabb tápfeszültségek adta lehetőségek alapján a Texas cég ebben a processzorban már tekintélyes méretű belső SRAM memóriát helyezett el (2 x 1 + 2 x 16 kszó = 34 kszó !). Belső PLL alkalmazásával a műveletek végrehajtási sebességét jelentősen megemelte (75 MIPS – Million Instructions Per Second, ill. 150 MFLOP – Million FLoating-point Operations Per Second), árát pedig annyira alacsonyra állították be (kb. 12$ ≈ 3000Ft/db), hogy kisebb képességű társait már ne legyen érdemes választani.

High-Performance Floating-Point Digital Signal Processor (DSP):
- **TMS320VC33-150**
  - **13-ns Instruction Cycle Time**
  - **150 Million Floating-Point Operations Per Second (MFLOPS)**
  - **75 Million Instructions Per Second (MIPS)**
- **TMS320VC33-120**
  - **17-ns Instruction Cycle Time**
  - **120 MFLOPS**
  - **60 MIPS**

A TMS320VC33 jelprocesszor sebességi jellemzői

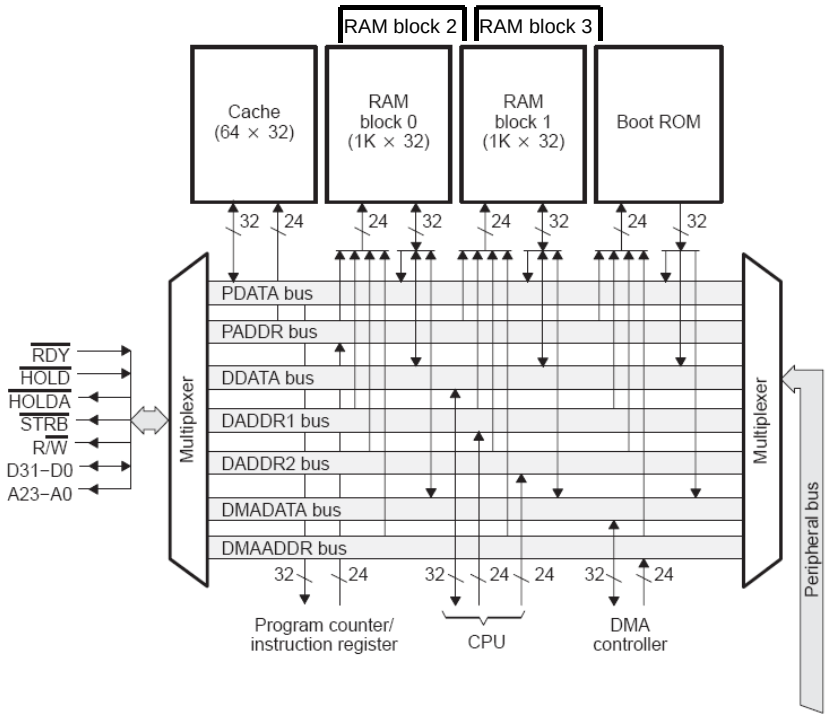


A TMS320VC33 jelprocesszor belső architektúrája

A blokkvázlat alapján leszűrhető fontosabb tulajdonságok a következők (a tulajdonságokat a továbbiakban a TMS320VC33-as processzor jellemzői alapján foglaljuk össze – a belső egységek mennyiségi jellemzői tekintetében a családtagok tulajdonságai ettől kis mértékben eltérhetnek):

1. A processzor 32 bites, ami egyben a legkisebb hozzáférési egység is mind a belső regiszterekhez, mind a memória és I/O egységekhez (megszűnik tehát az eddigi byte-okban való gondolkodás). Ezt az egységet a továbbiakban a processzor szóhosszának nevezzük és a "szó" kifejezést eszerint is fogjuk használni: 1 szó = 4 byte = 32 bit. A címvezetékek 1-gyel való növelése vagy csökkentése tehát a következő (vagy az előző) szó címzését eredményezi. A címvezetékek száma 24, tehát a teljes címtartomány 16 Mszó (64 Mbyte).

2. A processzor belső memóriákat is tartalmaz, boot ROM-ot, 2 x 1 kszó és 2 x 16 kszó (összesen 34 kszó = 136 kbájt) RAM-ot és 64 szó (256 bájt) utasítás cache-t. Ezek sebességüket tekintve időkésleltetés nélkül képesek kiszolgálni a processzor hozzáférési ciklusait.



A C33-as memória blokkja és 3 szintű buszrendszere

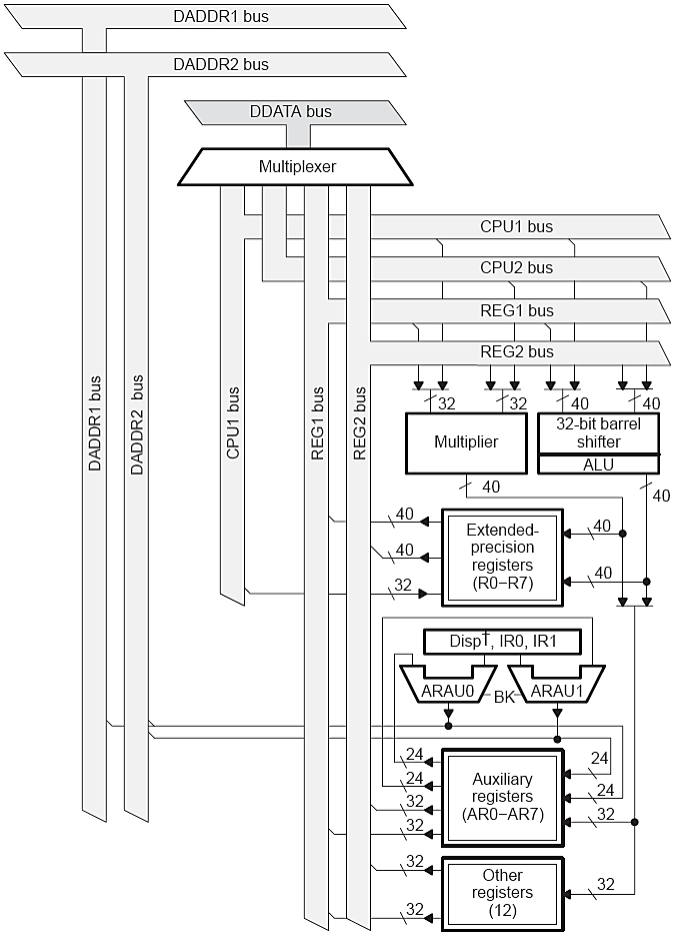
3. Ezen memóriák, a külvilág és a processzor belső adatbusza egy 3 szintű buszrendszerre kapcsolódik, melyek mindegyike saját cím- és adatbuszt foglal magába. Ezek: a kizárólag utasítások továbbítására szolgáló **program busz** (PADDR és PDATA), az adatok továbbítására szolgáló **adatbusz** (DADDR1, DADDR2 és DDATA), melyhez láthatóan 2 db független címbusz is kapcsolódik. Végül az utolsó az integrált DMA-vezérlő által használt **DMA-busz** (DMAADDR és DMADATA). Ezen 3 szintű buszrendszerrel és a RAM/ROM blokkok kialakításával érik el azt, hogy a 30-as család tagjai egyetlen CPU cikluson (= 2 órajel perióduson) belül képesek

   - bármelyik belső blokkhoz 2-szer hozzáférni
   - egyidejűleg utasítást olvasni, adatot olvasni és írni valamint ezenkívül DMA-átvitelt végezni (pl. a CPU 2 adatot tölt azonos RAM blokkból, közben utasítást olvas (kívülről vagy a cache-ből) és még mindig ugyanebben a ciklusban DMA-átvitelt végez a külvilág és a másik RAM blokk között).

   A blokkokhoz kapcsolódó nyilak azt mutatják, hogy az adott vonalak hány buszvezeték-csoporthoz tudnak kapcsolódni (pl. a címvezetékeknél a címet mind a PADDR, DADDR1, DADDR2 vagy a DMAADDR is meghajthatja, az adatvezetékek

ennek megfelelően a PDATA, DDATA vagy a DMADATA vonalakra kapcsolódnak). A számok a tényleges jelvezetékek számait mutatják.

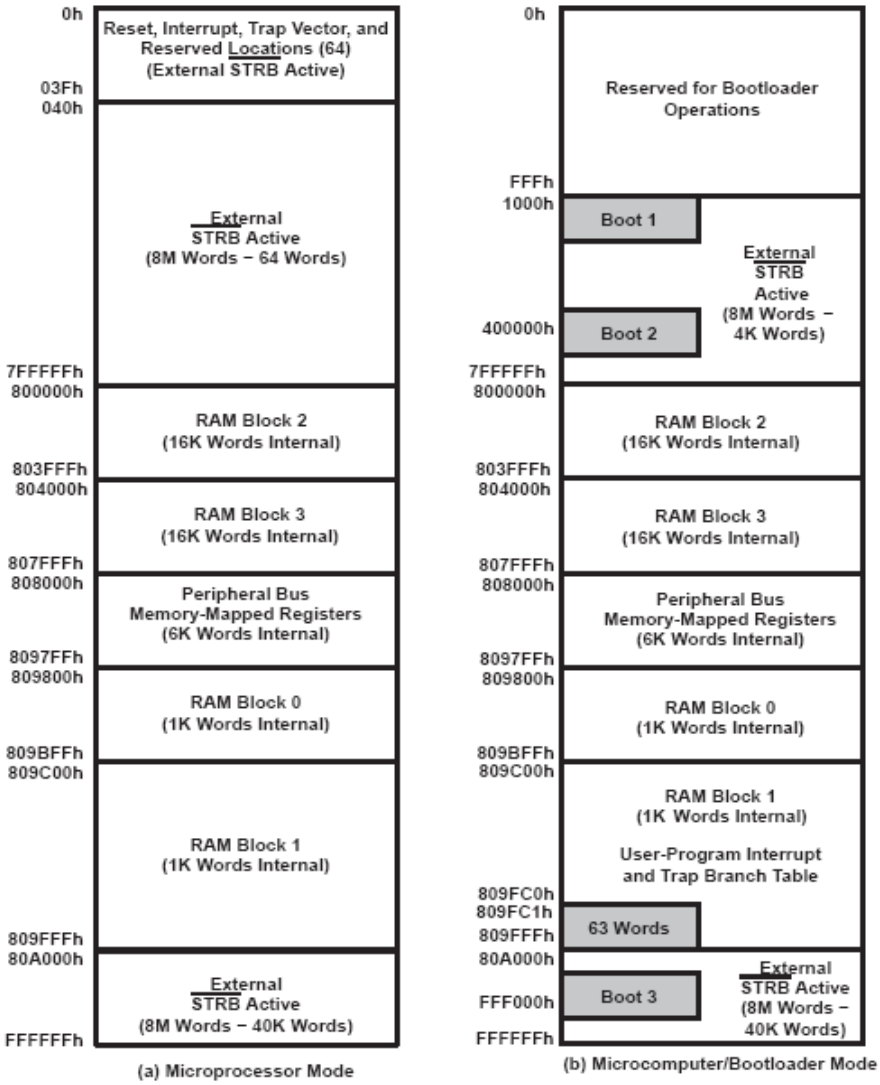
4. A 3 szintű buszrendszer valamelyike multiplexereken keresztül kapcsolódik a külvilágba vezető buszhoz (elsődleges busz), a periféria buszhoz és a processzor belső adatbuszához. Ez utóbbi 4 db további adatbuszt jelent: a CPU1 és a CPU2 buszok "csak" 32 bites adatok továbbítására képesek a memóriák felé míg a REG1 és a REG2 buszokon 40 bites adatok továbbíthatók a regiszterekből (a 40 bites adatok jelentőségével később foglalkozunk). Ily módon lehetőség nyílik az ALU/barrel shifter és a szorzó egység párhuzamos működésére valamint ezen kívül 2 db címképző aritmetika gondoskodik az operandusok címének előállításáról. Az aritmetikai egységek mind egész, mind lebegőpontos műveletek elvégzésére is képesek, azokat el tudják végezni egyetlen CPU ciklus (2 órajel periódus) alatt. Az elmondottak alapján ne csodálkozzunk a következő számokon: a 150 MHz-es TMS320VC33-as teljesítőképessége 75 MIPS és 150 MFLOP! (Utóbbi szám magyarázata: a szorzó egység és az ALU egymástól függetlenül képesek a műveleteket végezni, tehát a bevezetőben említett szorzás-összeadás műveletpárost és bizonyos további lebegőpontos műveletekből kettőt is képes elvégezni egyetlen utasítással.)



A C33-as CPU magja és belső buszrendszere

5. A 30-as család népszerűségének másik oka – sebességén kívül – az integrált perifériákban rejlik. Amint már említettük, a jelfeldolgozó processzorok sok tekintetben hasonlítanak a mikrokontrollerek világához: az alapvető, szinte mindenki számára szükséges perifériákat (timer, IT-kontroller, soros vonal, DMA-kontroller esetleg DRAM-kontroller) itt is beépítik a mikroprocesszorba, ily módon a felhasználónak már csak a memóriát és saját speciális perifériáit kell illesztenie a processzorhoz. Ez látható a 33-as esetében is: 2 db 32 bites időzítő, 1 db soros port, 1 db DMA-vezérlő és 4 db független külső megszakítás-kérési lehetőség áll a felhasználó rendelkezésére (az időzítők és a soros port megszakításain túl).

1.7.2 Memória modell



A TMS320VC33-as memória térképe

A 30-as család valamennyi tagjának van egy-egy olyan konfigurációs bemenete, amellyel beállíthatjuk, hogy használjuk-e a processzor belső ROM memóriáját (ezek az MC/-MP

ill. MCBL/-MP bemenetek). A C30-as esetében ez még felhasználói ROM volt, nyilvánvaló, hogy a megoldás csak nagyobb darabszámban gyártott termékek esetén jöhet szóba. A C31/C32/C33-as már csak egy kisebb kapacitású ROM memóriát tartalmaz, amit viszont a gyártó fix tartalommal, az ún. boot loader programmal szállít le. Ebben az esetben tehát a ROM ki-/bekapcsolása ezen boot loader program esetleges használatát jelenti (részleteire egy későbbi fejezetben térünk ki).

Valamennyi családtag utasításszinten nem különíti el a memória és I/O egységekhez való hozzáférést (memóriába ágyazott I/O). A belső memóriák és I/O elemek esetében a címtartomány egyes részei fixen hozzárendeltek ezekhez az egységekhez, ez a címkonfiguráció nem változtatható.

1.7.3 Programozási modell, utasításkészlet

A 30-as család jelprocesszorainak minden tulajdonságát áthatja a maximális sebességre való törekvés. Ez - mint látni fogjuk – egyrészről igen irigylésre méltó mérőszámokban (75 MIPS, 150 MFLOP) nyilvánul meg, másrészről viszont időnként olyan szokatlan megoldásokhoz vezet, ami a hagyományos CISC mikroprocesszorokhoz szokott programozót első ránézésre megriaszthatja. Szeretnénk rögtön a bevezetőben hangsúlyozni, hogy a 30-as család olyan igen figyelemreméltó tulajdonságokkal rendelkezik, mint:

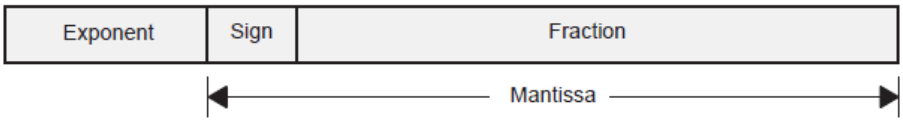
- minden utasítása pontosan 1 szó (32 bit), tehát nincs változó utasításhossz
- minden utasítását egyetlen CPU ciklus (az órajel frekvencia fele) alatt elvégzi, de lehet, hogy egyetlen ciklus alatt 2 utasítást is el tud végezni
- egységes utasításstruktúrában kezeli az egész és a valós (lebegőpontos) számokat, utóbbiakkal való műveletvégzés nem különítődik el az aritmetikai társprocesszoroknál "megszokott" lassabb műveletek formájában.

a) Adattípusok

A processzor alapvetően kétféle adattípust képes kezelni: az egész és a lebegőpontos típust. Konverziós utasítások segítségével ezen típusok között bármikor egyetlen CPU ciklus alatt átalakítást tud végezni.

Az **egész típus** alapértelmezésben 32 bites előjeles egész szám. Természetesen – lévén az előjel nélküli ill. az előjeles számok kezelése közismert konvenciók kérdése – mindkét fajta számkezelés lehetséges a programozó számára (a szükséges jelzőbitek rendelkezésre állnak). De vigyázat: mind a konverziós, mind a szorzó utasítások előjelesnek feltételezik a számokat! Ha valaki figyelmesen megnézi azonban a mikroprocesszor belső architektúráját feltüntető blokkvázlatot, fel fog tűnni neki, hogy az adatregiszterek 40 bitesek. A legfelső 8 bitet a jelprocesszor csak és kizárólag a lebegőpontos számok kezelésénél használja, minden egész típusú művelet (így az eltolás és elforgatás is) csak az alsó 32 biten történik, tehát az összes adatregisztert ilyenkor 32 bitesnek kell tekinteni! A **lebegőpontos típus** (sajnos?) egyedileg értelmezett a TMS-családban. Mint ismeretes, a lebegőpontos számok kezelését (mantissza és karakterisztika mérete, ábrázolási formája, pontossági és kerekítési előírások, stb.) 1985-ben az amerikai IEEE (Institut of

Electrical and Electronics Engineers) IEEE Std. 754-1985 néven szabványosította. Manapság a lebegőpontos számok kezelésében szinte mindenki ehhez a szabványhoz tartja magát (a későbbiekben ezzel a témakörrel foglalkozunk még). Lévén az előírások egy a 70-es évek végén, a 80-as évek elején kialakult állapotot tükröznek, az idő túlhaladt e szabvány előírásain és ma már egyértelmű, hogy bizonyos ott előírt számábrázolási formák nem optimálisak a hardver végrehajtási sebesség szempontjából. Ezért a Texas cég merész lépésre szánta el magát: egyedi formátumot definiált a lebegőpontos számok ábrázolására: a 2-es komplemens kódban ábrázolt exponenst a szintén kettes komplemens kódban ábrázolt mantissza követi, ily módon külön előjel tárolására nincs szükség. Az IEEE-754-es szabványból ismert "rejtett 1-es" itt is értelmezett a pontosság növelésére. A jelprocesszorban a lebegőpontos számok ábrázolása tehát a következők szerint történik:



ha $e = min, s = 0$ és $f = 0$: $x = 0$
$e > min, s = 0$: $x = 01.f * 2^e$ ($x > 0$)
$e > min, s = 1$: $x = 10.f * 2^e$ ($x < 0$)

	Rövid (Short)	**Egyszeres pontosságú (Single precision)**	**Bővített pontosságú (Extended precision)**
Bitszám (exp + mant)	16 (4 + 12)	32 (8 + 24)	40 (8 + 32)
Exponens minimum	-8	-128	-128
Legnagyobb pozitív szám	$2.5594 * 10^{2}$	$3.4028234 * 10^{38}$	$3.4028236683 * 10^{38}$
Legkisebb pozitív szám	$7.8125 * 10^{-3}$	$5.8774717 * 10^{-39}$	$5.8774717541 * 10^{-39}$
Legkisebb negatív szám	$-7.8163 * 10^{-3}$	$-5.8774724 * 10^{-39}$	$-5.8774717569 * 10^{-39}$
Legnagyobb negatív szám	$-2.5600 * 10^{2}$	$-3.4028236 * 10^{38}$	$-3.4028236691 * 10^{38}$

A lebegőpontos számok értelmezése a TMS320VC33-as jelprocesszorban

Az alapértelmezés a TMS családban a 32 bites egész- és a 40 bites lebegőpontos szám. Gondoljuk meg azonban a következőt: adataink egy részét a programokban memóriából olvassuk (direkt vagy indirekt címzés), jelentős részüket viszont közvetlen adatok formájában beépítjük az utasításokba (ún. implied vagy immediate címzés). Emlékezzünk viszont arra az alapkijelentésre, hogy a jelprocesszor minden utasítása pontosan 1 szó és máris adódik a szokatlan probléma: hogyan lehet egy 32 bites utasításszóba egy 32 vagy pláne egy 40 bites adatot beépíteni? A válasz egyértelmű: **sehogy**.

Emiatt a jelprocesszor tervezői a következő megoldáshoz folyamodtak:

- az adatokat teljes pontosságukban (előre) betölthetjük a memória adatszegmensnek tekintett területére és onnan direkt vagy indirekt címzéssel beolvashatjuk. A felhozatal nem lassabb, mint az adatnak közvetlenül az utasításba való beépítése
- immediate címzéssel csak rövidebb, max. 16 bites adatokat tudunk megadni (a vezérlésátadó utasítások címinformációi ez alól kivételek lesznek, ld. később). Megoldandó, hogy ezen adatok hogyan "válnak" 32 bites adatokká.

Egész számok esetén a töltő- és aritmetikai műveletekben a 16 bites konstans előjeles számként értelmeződik és kiterjesztődik 32 bitre. Logikai műveletek esetén nincs kiterjesztés, a felső 16 bit értéke 0 lesz.

Lebegőpontos számok esetén a helyzet bonyolultabb: a jelprocesszor a lebegőpontos számok tekintetében ugyanis eltérően kezeli saját regisztereit és a környezetét.

- a belső adatregiszterekben a lebegőpontos számok mindig kiterjesztett pontossággal tárolódnak, ezért 40 bites a jelprocesszor összes adatregisztere.
- az összes lebegőpontos műveletet a processzor kiterjesztett pontossággal végzi (egy megszorítás van a szorzásnál, ld. később), a művelet végeredménye mindig a 40 bites regiszterek valamelyikében tárolódik.
- a környezetét a jelprocesszor alapvetően egyszeres pontosságúnak tekinti. Ennek megfelelően a normál lebegőpontos töltő utasítással (LDF) 32 bites szót tölt a 40 bites regiszterbe a mantissza kiterjesztésével (ez a 2-es komplemens ábrázolás miatt nem probléma). A normál lebegőpontos tároló utasítás (STF) a 40 bites regiszter mantisszáját csonkolva tárolja a megadott címen. A csonkolás a 2-es komplemens ábrázolás miatt lefelé kerekítésnek felel meg, amennyiben ezt el akarjuk kerülni, tárolás előtt az RND utasítással a legközelebbi egyszeres pontosságú értékre kerekíthetjük a számot.
  Az eddig elmondottak alapfilozófiája: a processzor próbálja "nem halmozni" a hibát a lebegőpontos műveletek során, de "sajnálja" 2 szóban tárolni a kiinduló adatot és az eredményt. Lehetőséget teremt viszont erre az exponens és a mantissza külön utasításokkal (LDE, LDM) való kezelhetősége, de erről bővebben majd az utasításkészletnél beszélünk.
- amennyiben a lebegőpontos műveletek valamelyik forrás operandusa nem regiszter, a környezetből felhozott szám az előzőeknek megfelelően egyszeres pontosságú. Kivétel ez alól az immediate konstans, ami csak rövid (16 bites) lehet.

b) Regiszterek

A TMS320C30-as tulajdonságai szerint csoportosítva 3 regiszterblokkal rendelkezik. A blokkok a következők:

A 8 db 40 bites **adatregiszter** (eredeti nevén kiterjesztett pontosságú regiszter [***extended precision register***], R0..R7) általános adatregiszterként szolgál a műveletek forrásaként/eredményeként. A regiszterek 40 bitesek, aminek oka az egész és a lebegőpontos adatok korábban ismertetett kezelésében rejlik.
A 8 db 32 bites **címregiszter** (eredeti nevén segédregiszter [***auxiliary register***], AR0..AR7) elsődleges feladata a címképzésben való részvétel, de használhatók aritmetikai feladatokra is (még egész számok szorzásánál is!) vagy pl. ciklusszámlálókként.

További 13 db **egyéb regiszter** segíti a processzor és a programozó munkáját, ezekről igen röviden és a bitkiosztások mellőzésével csak a legfontosabb tulajdonságaikat mondjuk el:

PC, SP, ST : A szokásos programszámláló, veremmutató és státusregiszter. Egy nagyon lényeges megjegyzés: a veremmutató tároláskor (push) itt is a növekvő címek irányába halad !

IE, IF : Az IE regiszter teszi lehetővé az összes belső és külső megszakításkérés külön-külön történő engedélyezését és tiltását (az együttes tiltás/engedélyezés GIE bitje a státusregiszterben található). Bármelyik megszakításkérő vonal DMA-kérelmet is jelenthet a belső DMA-vezérlőnek, ez szintén az IE regiszterben állítható be. Az IF regiszter a megszakításkérő vonalak állapotának lekérdezését biztosítja engedélyezettségüktől függetlenül.

IOF : Multiprocesszoros alkalmazások szemafor-jellegű funkcióit a jelprocesszor két jelvezetékkel és megfelelő utasításokkal támogatja. Az IOF regiszter segítségével ez a két jelvezeték konfigurálható/manipulálható.

DP : Data Page Pointer, a címzési módoknál visszatérünk rá.

BK : Blokkméret-regiszter, a címzési módoknál visszatérünk rá.

IR0, IR1 : Indexregiszterek, a címzési módoknál visszatérünk rájuk.

RS, RE, RC : Ciklus kezdőcím, végcím, ciklusszámláló. Ennek a 3 regiszternek és 2 ezekre épülő utasításnak a segítségével a jelprocesszor igen hatékony támogatást nyújt a ciklusok szervezéséhez. Működésüket két rövid példán keresztül mutatjuk be:

```
          LDI     99,RC          ; Ciklusszám = 100 (=RC+1)
          RPTB    VEGCIM         ; Ez a ciklusszervező utasítás
          <Ciklustörzs>          ; Ez maga a ciklus
VEGCIM:   <Utolsó utasítás>      ; Nem lehet vezérlésátadás
```

illetve

```
          RPTS    199            ; Ciklusszám = 200
          <Ciklus>               ; Ez maga az 1 utasításos ciklus
```

c) Címzési módok

A jelprocesszorok egyik meghatározó tulajdonsága, hogy megfelelő címzési módokkal hatékonyan kell támogassa a gyors jelfeldolgozást. Ennek megfelelően a jelprocesszor meglehetősen sok (32) címzési módot különböztet meg, melyekből inkább csak szemelvényeket adunk a részletes ismertetés helyett:

- **Regiszter címzésről** beszélünk, ha a forrás vagy cél operandus valamelyik regiszter
- **Direkt címzésről** beszélünk, ha közvetlenül adjuk meg az operandus címét. Mivel az adatok direkt címei csak 16 bit erejéig építhetők be az utasításokba, a felső 8 bitet a Data Page Pointer (DP) szolgáltatja, ami külön utasítással tölthető. Az elmondottaknak megfelelően a “lapméret” 64 kszó.

```
     LDP  CIM,DP            ; CIM = 123456h esetén DP = 12h
     LDI  @BELSOCIM,R0      ; R0 = (120000h + BELSOCIM)
```

- **Indirekt címzés** esetén a címregiszterek szolgáltatják a címet. Ebben a csoportban található a legtöbb lehetőség, innen csak példákat mutatunk a lehetőségek érzékeltetésére.

```
LDI   *AR0,R0           ; R0 = (AR0)
LDI   *+AR0(DISP),R0    ; R0 = (AR0+DISP)
LDI   *++AR0(DISP),R0   ; R0 = (AR0+DISP)   és AR0 = AR0+DISP
LDI   *AR0++(DISP),R0   ; R0 = (AR0)        és AR0 = AR0+DISP
LDI   *--AR0(IR0),R0    ; R0 = (AR0-IR0)    és AR0 = AR0-IR0
```

  Ennek a címzési osztálynak a tagja a cirkuláris és a bittükrözéses címzés is, ezekre külön visszatérünk.
- **Rövid immediate** címzés esetén az utasításba építjük be az operandust. Ennek részleteit az adattípusoknál már ismertettük.
- **Hosszú immediate** címzést csak a vezérlésátadó utasítások használhatnak. Az ezekhez szükséges 24 bites címinformáció be tud épülni az utasításokba, ily módon ugrások és szubrutinhívások tekintetében a teljes memória közvetlenül címezhető (abszolút vagy relatív címzéssel).

Az indirekt címzésnél említett cirkuláris és bittükrözéses címzés speciális jelprocesszor címzési módok.

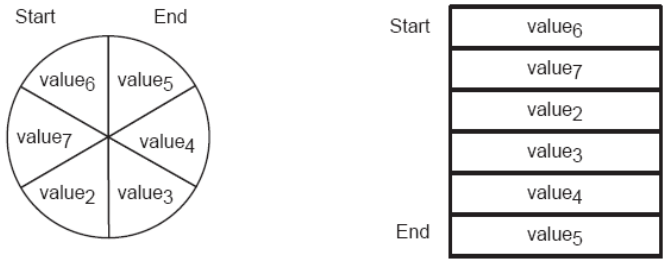
A **cirkuláris címzést** hatékonyan használhatjuk cirkuláris pufferek felépítésénél (pl. FIR szűrők, kommunikációs bufferek, ld. a PC-k klaviatúra buffere), végeredményben ez nem más, mint egy modulo N címzés a blokkméret-regiszter (BK) segítségével. Ismét egy rövid példa:

```
LDI   6,BK              ; Blokkméret = 6
LDI   100h,AR0          ; Puffer kezdőcíme = 100h
LDI   *AR0++(5)%,R0     ; R0 = (100h)       és AR0 = 105h
LDI   *AR0++(2)%,R0     ; R0 = (105h)       és AR0 = 101h
LDI   *AR0--(3)%,R0     ; R0 = (101h)       és AR0 = 104h
```

Kiegészítés a fentiek jobb megértéséhez: a cirkuláris puffer kezdőcíme alsó bitjeinek 0-nak kell lenniük oly mértékben, hogy az ezen bitek által meghatározható maximális memóriatartományba a puffer beleférjen. (pl. 5-8 bájtos puffer esetén a puffer kezdőcím alsó 3 bitjének, de 9 bájtos esetén már az alsó 4 bitjének kell 0-nak lennie). A használt mutatónak (esetünkben AR0) „bele kell mutatnia” ebbe a tartományba, mivel ezen alsó bitek – az ún. indexbitek - meghatározása minden esetben a következő algoritmus szerint történik (a többi felső bit változatlanul hagyása mellett):

```
IF (0 ≤ index + offset < BK) index = index + offset;
ELSE  IF (index + offset ≥ BK) index = index + offset – BK
      ELSE index = index + offset + BK
```

(ahol offset az utasításba foglalt eltolás – a fenti példában +5, +2 és –3).



A cirkuláris buffer logikai és fizikai képe

A **bittükrözéses címzést** igazából a gyors Fourier-transzformáció (FFT) támogatására fejlesztették ki, ami a nagysebességű digitális jelfeldolgozásban viszonylag gyakori feladat. Ennek részleteire itt az algoritmus ismeretének hiánya miatt nem térünk ki.

d) Az utasításkészlet

A TMS320VC33-as összesen 113 utasítással programozható. Az említett utasításszám ugyan soknak tűnik a RISC architektúra előírásaihoz képest, ennek oka viszont az, hogy azonos "jellegű" utasítások többször fordulnak elő az utasításkészletben, ez növeli meg az utasítások számát. Nem sok értelme lenne itt mind a 113 utasítás elsorolásának, inkább olyan szempontokat emelünk ki, melyek vagy eltérnek a más mikroprocesszoroknál megszokottaktól, vagy különös jelentőséggel bírnak a jelprocesszor működése szempontjából.

1. Mint azt már említettük, a C33-as minden utasítása 1 szó hosszúságú. Az utasításokat a jelprocesszor 4 fázisra bontja (utasítás felhozatal, dekódolás, operandus olvasás, végrehajtás és tárolás), pipeline műveletvégző egység biztosítja az utasítások oly módon való átlapolását, hogy **minden CPU ciklusban egy új utasítás** végrehajtása kezdődhessen meg.

2. Rendelkezik azonban a jelprocesszor egy sor ún. **párhuzamos utasítással** is, amikor az architektúrális felépítésnél látott nagymértékű belső párhuzamosítás lehetővé teszi két utasítás "egyidejű" végrehajtását is. Ezek a párhuzamos utasítások csak meghatározott utasítás-párok lehetnek, és erről a párhuzamosításról a programozónak kell rendelkeznie.

```
        ABSF    src1, dst1      ; Lebegőpontos szám abszolút értéke
     || STF     src2, dst2      ; Lebegőpontos szám tárolása
```

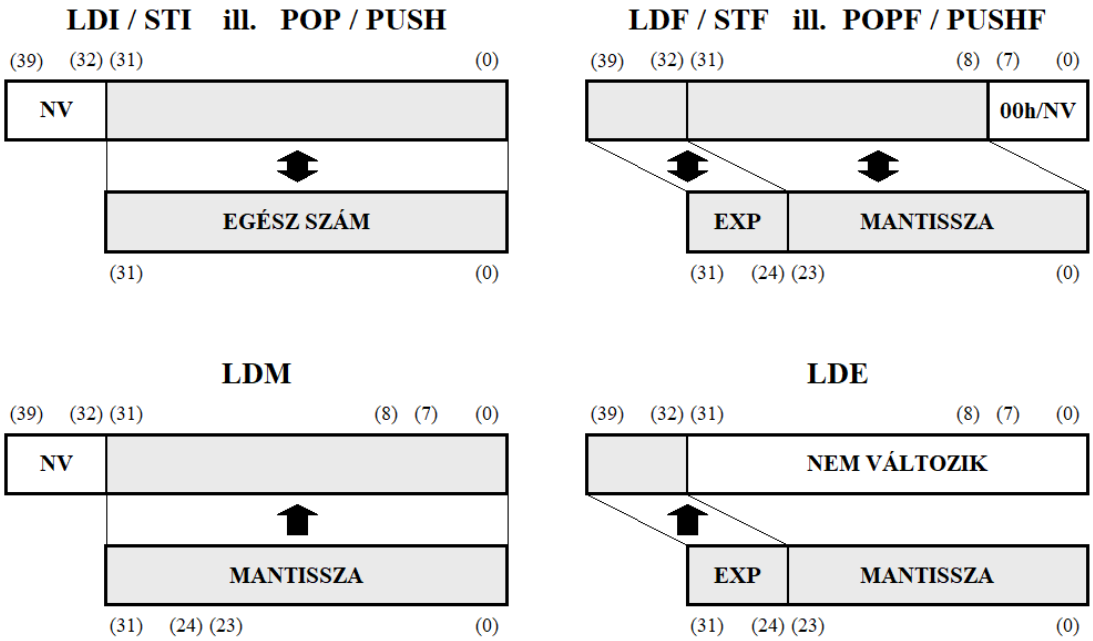
   Tehát a "||" jel jelöli a párhuzamosítást, ez a 2 utasítás természetesen két szóból is áll, a C33-as 2 hozzáféréssel olvassa be, de ez az egész egyetlen CPU ciklus alatt le tud zajlani. Vigyázat: Minden forrás operandus a ciklus elején kerül kiolvasásra, és minden cél operandus a ciklus végén tárolódik. Amennyiben tehát az első utasítás cél operandusa és a második forrása megegyezik, akkor nem a művelet eredménye, hanem az előző állapot kerül tárolásra!

3. A C33-as utasításainak jelentős része megtalálható **2 és 3 című** kivitelben is. Kutatások azt mutatják, hogy az a kétcímű gépekre jellemző tulajdonság, mely szerint az egyik forrás- és a cél operandus címe megegyezik, rengeteg utántöltési és mentési művelet forrása. Ebből kiindulva javasolják a kutatók a 3 című utasítások kialakítását. A C33-as utasításkészletében mindig az utasítások végén található 3-as szám mutatja a háromcíműséget. A korábban már említett 150 MFLOP-os teljesítőképességet pl. a következő utasítás-párok biztosítják:

```
        MPYF3   src1, src2, dst1   ; Lebegőpontos számok szorzása
     || ADDF3   src3, src4, dst2   ; Lebegőpontos számok összeadása
        MPYF3   src1, src2, dst1   ; Lebegőpontos számok szorzása
     || SUBF3   src3, src4, dst2   ; Lebegőpontos számok kivonása
```

4. Az utasításkészletben megtalálhatók ugyan a töltő- és tároló utasítások (mindig LDx jelenti a regiszter felé irányuló műveletet, STx pedig a memóriában való tárolást), mégis az aritmetikai/logikai utasítások legtöbbjének forrás operandusa lehet direkt vagy indirekt cím is, a cél operandus mindig regiszter. Ez a megoldás a gyorsítást szolgálja, még akkor is, ha ily módon az utasításkészlet nem felel meg a ma egyre inkább megkövetelt **load/store architektúra** követelményeinek (tisztán regiszterreferens aritmetikai és logikai utasítások). Az elmondottak igazak a 3 című és a párhuzamos utasításokra is.

5. Külön meg kell említenünk a **lebegőpontos számok töltését és tárolását**. A jelprocesszor adatmozgató utasításai a következő csoportokba oszthatók:



A lebegőpontos számok kezelése a TMS320VC33-as jelprocesszorban

A lebegőpontos számok töltését és tárolását első közelítésben az LDF és STF utasítások végzik, mint azt már említettük, oly módon, hogy a memóriában található adatot mindenképpen 32 bites egyszeres pontosságúnak feltételezi a jelprocesszor, tehát töltéskor a számot kiterjeszti (a mantissza alsó 8 bitje 0), tároláskor pedig az alsó 8 bitet egyszerűen levágja, ami mechanikus lefelé (-∞ irányába való) kerekítésnek felel meg.

A lebegőpontos számok teljes pontosságú (40 bites) manipulációját az LDE és az LDM utasítások segítik. Az LDE utasítás csak a megadott operandus exponens részét tölti be (azaz a legfelső 8 bitet), a mantissza változatlan marad. Az egyetlen kivétel az, ha az exponens értéke minimum (-128), ilyenkor a mantissza részt erőszakosan törli. Az LDM utasítás a 32 bites memória adatot teljes egészében a mantissza helyére tölti, az exponens változatlan marad. Ezzel a két utasítással tehát megoldható (2 művelettel) egy lebegőpontos szám kiterjesztett pontosságú betöltése is.
Mind a 4 utasítás speciális esete, ha a forrás immediate konstans, vagy ha mind a forrás, mind a cél regiszter. Előbbi esetben a forrás csak rövid formátumú lebegőpontos szám lehet (csak töltő műveletről lehet szó!), a mantissza és/vagy az exponens minden esetben kiterjesztésre kerül. Regiszterek esetében az adatok csonkítás nélkül töltődnek egyik regiszterből a másikba.

A teljes szimmetriából hiányoznak az STE és az STM utasítások. Megértve azonban az egész/lebegőpontos számkezelés filozófiáját beláthatjuk, hogy az STE utasítást helyettesíti az STF, amelyik az exponensen kívül a csonkolt mantisszát is tárolja (semmibe nem kerül), míg az STM utasítást tökéletesen helyettesíti az STI utasítás. (Fordítva ez már nem lenne igaz, és ennek okai ismét az immediate forrás operandusok: az LDI utasítás a 16 bites forrást a 0..15 biteken helyezi el és kiterjeszti az előjel szerint, ezzel szemben az LDM a 16 bites forrás 0..11 bitjeit a 23..12 biteken helyezi el és az alsó biteket nullázza.)

Felhívjuk viszont a figyelmet az elkülönített PUSHF és POPF utasításokra. Egy regiszter teljes (40 bites) mentése csak 2 utasítással lehetséges, ahol a sorrend sem mindegy: a helyes mentési/visszaállítási szekvencia PUSH, PUSHF / POPF, POP (lévén a POP nem bántja az exponens részt, ezzel szemben a POPF nullázza az alsó 8 bitet!). Amennyiben pl. egy megszakítási rutin csak egész számokat használ, elegendő a megszokott PUSH-POP szekvencia használata. Teljes mentés esetén viszont (lévén nem tudjuk, kit szakítottunk meg) 2-2 utasítás szükséges az adatregiszterek mentéséhez és visszaállításához.

6. Külön szót kell említsünk a szorzó utasításokról. A szorzásra elkülönített utasítás áll rendelkezésre egész- ill. lebegőpontos műveletek elvégzésére. Az alapszabály a következő: a szorzóművet 24 bites előjeles számok szorzására alakították ki. Ennek megfelelően:

   Egész típusú szorzás esetén a szorzómű a forrás operandusokat 24 bites előjeles egészként kezeli (a felső 8 bitet figyelmen kívül hagyja), a keletkező 48 bites

eredménynek az alsó 32 bitje kerül a célregiszterbe. Hibajelzést kapunk (overflow), ha az előjeles eredmény nem fér el 32 biten.

Lebegőpontos szorzás esetén a forrás operandusok csak egyszeres pontosságúak lehetnek (még abban az esetben is, ha a források adatregiszterekben vannak), a szorzat kiterjesztett pontossággal tárolódik a célregiszterben.

7. Már a bevezetőben is említettük, hogy a nagysebességű műveletvégzés már az alapműveletek egyikénél, az **osztásnál** is problémákba ütközik. Miért ?

   Csak olyan rekurzív osztó algoritmus áll rendelkezésünkre, amely úgy végezi el több lépésben az osztás műveletét, hogy az egyes fokozatok bemeneteit az előző fokozatok kimenetei alkotják. Ez azt jelenti, hogy még az ún. párhuzamos osztóművek is úgy működnek, hogy az egyes fokozatok jelterjedés szempontjából sorba kapcsolódnak. Hardver műveletvégzés szempontjából áramkörileg az ún. nem visszaállításos osztás megvalósítása a kedvezőbb, ennek elve röviden a következő:

   - az osztandó kijelölt részéből – pl. a legelső digitből – (A) kivonjuk az osztót (B), jelölje ezt

     $$A\text{-}B$$

   - Visszaállításos osztás esetén (ezt tanultuk az általános iskolában)
     - negatív eredmény esetén („nincs meg”) azt szoktuk meg, hogy visszaállítjuk az eredeti állapotot:

       $$(A\text{-}B) + B$$

     - ezt követően hozzávesszük az eddigi A-hoz (ez balra léptetést jelent, azaz kettes számrendszerben *A*-t szoroztuk 2-vel) a következő osztandó helyiértéket (*a*), majd újra megkíséreljük a kivonást:

       $$(2*A+a) - B$$

   - Nem visszaállításos osztás esetén észrevesszük, hogy felesleges a visszaállítás, mert amennyiben visszaállítás nélkül hozzávesszük az eredményhez a-t, de „nincs meg” esetén kivonás helyett összeadást végzünk, akkor vegyük észre, hogy

     $$[2*(A\text{-}B)] + a + B = (2*A+a) - B$$

Nézzük meg a fenti elvet egy egyszerű példán:

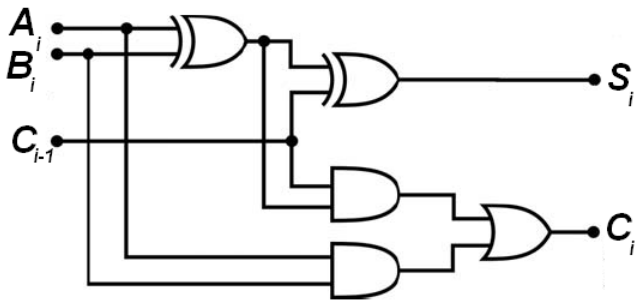
14:3 = 4|2

1110 : 0011 = 0100|0010

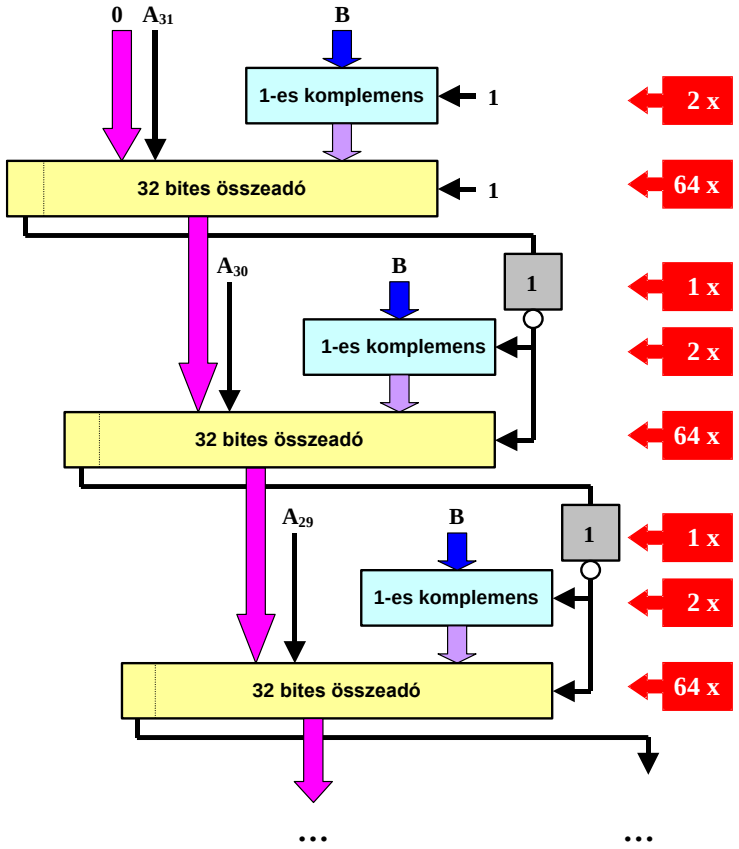
```
  A = 1110 B = 0011 -B = 1101

   0001      000          | A3
  +1101      -B
0⇐ 11101     Részeredmény | A2
   +0011     +B
 1⇐ 00001    Részeredmény | A1
   +1101     -B
  0⇐ 11100   Részeredmény | A0
   +0011     +B
  0⇐ 1111    Utolsó eredmény
   +0011     Visszaállítás
    0010  →  Maradék
0100  ———→   Hányados
```

Példa a nem visszaállításos osztás lépéseire

A megvalósításhoz 32 bites operandusok esetén összesen 32 fokozatban fokozatonként egy-egy 32 bites összeadóműre és az előző fokozat részeredménye alapján vezérelhető kettes komplemens képzőre lesz szükségünk. Amennyiben feltételezzük, hogy egy 1 bites teljes összeadót legalább a bitek közötti átvitel szempontjából max. 2 szintű logikai hálózattal megvalósítjuk, akkor a 32 bites összeadónk 64 szintű logikai hálózatot eredményez, vezérelt kettes komplemens képzőnket egyszerűen XOR kapuk sorozatából (1-es komplemens) és a következő összeadó legalsó bitjén 1 hozzáadásával alakíthatjuk ki.



1 bites teljes összeadó (4-4 logikai kapuszinttel)



A párhuzamos osztóm elve

Amennyiben fenti feltételezéseink teljesülnek, osztóművünk fokozatonként (bitenként) 67 kapuszintet tartalmaz, ez 32 biten összesen 2144 kapuszintet eredményezne. A helyzeten még javíthatunk gyors átvitelképző (carry look ahead) alkalmazásával. Ezek annyira bonyolult magas kapuszámot igénylő kapcsolások, hogy csak bitcsoportonként (4-8-16 bitenként) szokás kialakítani őket. Tételezzük fel, hogy még ezt is teljesen párhuzamosítjuk – így a kapuszintek számát kb. 128-ra redukálhatjuk. Az a 2T = 13.3 nsec (150 MHz) alatti műveletvégzéshez átlagosan 104 psec (128 kapuszint) ill. 6.2 psec (2144 kapuszint) jelkésleltetési időt enged meg kapunként. Előbbi ma már ugyan nem kivitelezhetetlen (10 GHz-nél járunk!), de annyira speciális áramköri megoldásokat igényel, ahol a tápáram nagysága és az integrálhatóság foka ellentmond a mikroprocesszorok gyártásához szükséges követelményeknek. (Ilyen áramkörök önálló integrált áramkörként léteznek – egy-egy jelprocesszor árával összemérhető borsos áron.)

Ennek megfelelően a mikroprocesszorok az osztást mindig mikroprogram ciklussal végzik el, mivel azonban a C30-as család tagjai huzalozott logikára épülő mikroprocesszorok, így mind az egész, mind a lebegőpontos osztás hiányzik az utasításkészletből. Maga a Texas cég is "érzi" az osztás hiányát, ezért a cég már a processzor leírásában közöl egy-egy optimalizált algoritmust (szubrutint) az osztás egész- vagy lebegőpontos számokon történő elvégzésére:

- 2 db 32 bites előjeles egész szám osztása 31..62 ciklus alatt (ez 413 .. 827 nsec-nak felel meg 150 MHz órajel-frekvencia esetén)
- Lebegőpontos szám reciprokának képzése 35 ciklus alatt (467 nsec / 150 MHz)

(Összehasonlításként: egy 486-os mikroprocesszor beépített osztó utasítása 2 db 32 bites egész számot 43 órajel periódus alatt oszt el, a lebegőpontos osztást 73 órajel periódus alatt tudja elvégezni.)

8. Aritmetikai műveletsorok elvégzése szempontjából kellemes lehetőséget kínálnak a jelprocesszor **tárolt jelzőbitjei** (latched flags). A processzor státusszava tartalmazza a szokásos jelzőbiteket (**C**arry, O**V**erflow, **Z**ero, **N**egative és Floating Point **U**nder**F**low), a túlcsordulás bitet mind az egész, mind a lebegőpontos műveletek állítják, az alulcsordulás bit (exponens = -128) csak a lebegőpontos műveletek hatására állítódik. Ezen kívül azonban megjelenik a következő két bit is: **LV** (Latched Overflow) és **LUF** (Latched Floating Point Underflow). Ezek a jelzőbitek akkor aktiválódnak, mikor a megfelelő nem tárolt bitek (V és UF), viszont szemben azokkal tartalmukat mindaddig megtartják, míg a processzor nem törli őket a státusszó írásával. Ezzel azt érjük el, hogy egy-egy műveletsor elvégzésekor elegendő a sor végén megvizsgálni a jelzőbiteket, hogy történt-e kritikus hiba a műveletsor elvégzése során, nem kell megtennünk azt minden egyes művelet után.

9. A **vezérlésátadási műveletek** lehetnek feltételesek és nem feltételesek. Összesen 20 feltétel építhető be a vezérlésátadási műveletek (ugrások, szubrutinhívás és visszatérés) mindegyikébe. A kombinációk minden lehetőséget lefednek a következő csoportokra:

   - előjel nélküli egészek kezelése (LO, LS, HI, HS, EQ, NE)
   - előjeles egész- és lebegőpontos számok kezelése (LT, LE, GT, GE, EQ, NE)
   - előjelvizsgálat (Z, NZ, P, N, NN)
   - jelzőbitek egyedi vizsgálata (minden flag és negáltja)
   - mindig igaz (U = Unconditional)

   Lényeges viszont azt megemlítenünk, hogy a vezérlésátadó műveletek típus szerinti osztályozása (feltételes vagy sem) egyben mást is takar. A nem feltételes ugró- és vezérlésátadó utasítások (BR ill. CALL) 24 bites címinformációt hordoznak, amely abszolút címként kerül értelmezésre. Ezekkel az utasításokkal tehát a jelprocesszor teljes címtartománya közvetlenül címezhető.

   Indirekt és relatív címzésre csak a feltételes utasítások adnak lehetőséget (Bcond és CALLcond), persze ezek is lehetnek "feltételtől mentesek" BU és CALLU formában. Indirekt címzésnél egy regiszter tartalma töltődik PC-be, relatív címzésnél pedig az utasításba beépült 16 bites konstans adódik a PC-hez (előjelesen). Ily módon a relatív címzéssel átfogható memóriatartomány ±32 kszó.

10. A vezérlésátadó műveletek még egy szempontból kivételt jelentenek a "normál" utasítások sorából: végrehajtásuk szemben minden eddig elmondottal 4 CPU ciklust igényel. Hozzá kell tennünk: mint minden más utasításé. Csakhogy a többi utasítás a pipeline műveletvégzés miatt átlapolódik, ily módon kívülről szemlélve minden CPU ciklusban véget ér (legalább) egy-egy utasítás, tehát számunkra úgy néz ki, mintha minden utasítás valóban 1 órajel periódus alatt hajtódna végre. A vezérlésátadó műveletek viszont bajt okoznak: az utánuk bekerülő utasítások előfeldolgozása feleslegessé válik, a csövet ki kell üríteni, ily módon "előtűnik" az eredeti kijelentés: 4 CPU-ciklus alatt csak egyetlen utasítás ért véget.

    A jelprocesszor felkínálja azonban annak a lehetőségét, hogy lehetőleg a gyakoribb ugró utasítások esetében ez ne forduljon elő: ezek a **késleltetett ugró utasítások** (delayed jump). A BcondD és a BRD utasítások kiadása után még a 3 következő utasítás végrehajtásra kerül és csak ezután történik meg a vezérlésátadás. Az eredmény: 1 CPU ciklus "alatt" végrehajtott vezérlésátadás. (Természetesen a 3 következő utasítás egyike sem lehet újabb vezérlésátadó utasítás.)

Mikrokontroller alapú rendszerek

Elektronikus jegyzet

2. fejezet (1. rész)

Készítette: Szabó Zoltán mérnöktanár
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar tudósok körútja 2.
Q épület B szárny II. em. QB218.
Tel: 463-2597
Fax: 463-2871 (adm.)
Mail: zszabo@aut.bme.hu

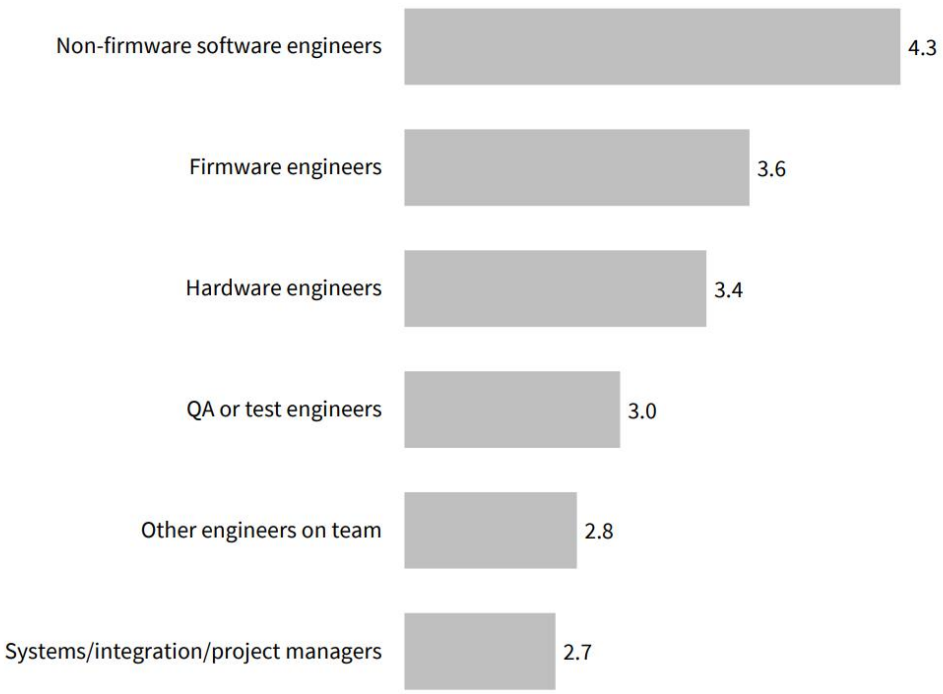
Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki alapszak (BSc)
III. évfolyam, 5. félév
Beágyazott és irányító rendszerek specializáció hallgatói

COPYRIGHT

TARTALOMJEGYZÉK

2. Szoftverfejlesztés beágyazott rendszerekben

A beágyazott rendszerek készítése szinte kivétel nélkül csapatmunkán alapul, a különböző szakterületeken tapasztalatot szerzett mérnökök együttműködésével készíthető csak el. Természetesen az egészen kicsi projektek esetében elképzelhető, hogy a teljes rendszer – beleértve a hardveres és szoftveres részeket is – egyetlen ember munkája, azonban nagyobb projektek esetében ez többnyire nem így szokott lenni.
A beágyazott rendszerek elkészítéséhez szükségesek egyrészt hardveres ismeretekkel rendelkező szakemberek, akik elkészítik magát a fizikai eszközt. A fizikai eszköz szolgáltatásainak figyelembe vételével el kell készíteni a rendszerszoftver azon részeit, melyek kifejezetten hardverfüggőek – ez a firmware engineer feladata. Az alapszintű hardver kezelő szoftverelemekre építve készíthetők el a rendszerszoftver magasabb szintű moduljai, illetve a – nem feltétlenül a beágyazott eszközön futó – egyéb kiegészítő szoftverkomponensek. Utóbbiak elkészítése általában a szoftver mérnökök feladata.
Egy 2023-ban készült reprezentatív online felmérés adatai alapján az alábbi ábrán látható létszámú és összetételű egy-egy átlagos projektben résztvevő fejlesztőgárda:



1. ábra Beágyazott fejlesztések szakember összetétele

Látható, hogy a fejlesztési feladatoknak csak kis része igényel hardveres szakembert, míg a maradék munka nagyrészt kifejezetten szoftveres jellegű.
Mint azt a fenti statisztika is alátámasztja a beágyazott szoftveres ismeretek elengedhetetlenek egy napjainkban végző villamosmérnök számára, még abban az esetben is, amennyiben kifejezetten hardveres területen szeretne a későbbiekben elhelyezkedni.

2.1 A szoftverfejlesztés folyamata *(kiegészítő anyag)*

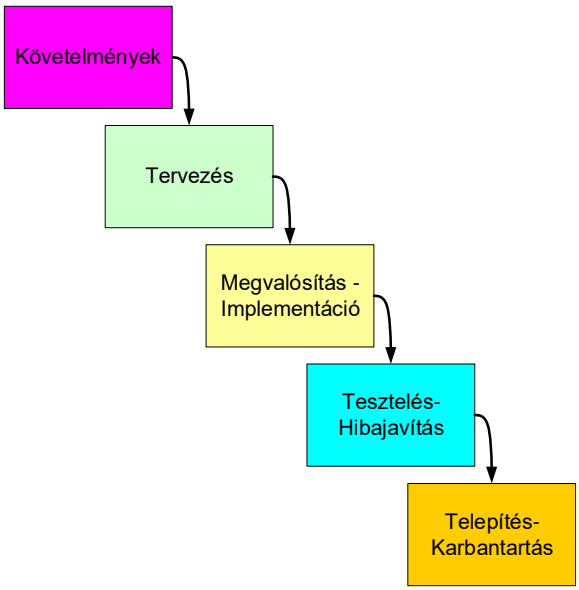
A szoftverfejlesztés fogalma alatt, az emberek jó része csak a programok fizikai megvalósítását – az implementációt – érti. A szoftverfejlesztés azonban ennél több részből áll, szerteágazóbb folyamat, melynek minden részterülete napjainkra külön szakággá fejlődött. Nézzük végig, hogy milyen részproblémákat kell megoldanunk egy szoftver sikeres elkészítéséhez.

- Követelmények meghatározása
- Tervezés
- Megvalósítás
- Tesztelés - Hibajavítás
- Telepítés - Karbantartás

Az említett részfeladatok megoldása lefedi a teljes szoftverfejlesztési folyamatot. Az említett feladatok egymásutánisága első megfontolás alapján triviálisnak tűnik, azonban ha jobban belegondolunk nem ilyen egyszerű a helyzet. Megfontolandó kérdés az is, hogy lehet-e az egyes munkafázisokat párhuzamosan végezni, valamint hogy létezik-e egyáltalán ideális sorrend.
A felmerülő kérdések megválaszolásához a szoftverfejlesztés folyamatát modelleznünk kellene, a modellek alapján próbáljunk választ találni a feltett kérdésekre. A szoftver életciklus modellezése népszerű téma, viszonylag sokakat foglalkoztat, ezért jelentős mennyiségű irodalom áll rendelkezésünkre a kérdéseink megválaszolásához. Nézzük, melyek azok a modellek, melyekkel a szoftverfejlesztés folyamatát több-kevesebb sikerrel le tudjuk írni.

2.1.1 Vízesés modell

A modell szerint az egyik fázis szekvenciálisan követi a másikat. Első lépésként specifikáljuk a követelményeket – ettől a későbbiekben nem térünk el. A kész specifikáció alapján megtervezzük a rendszert. A terv lényegében egy útmutatás, hogy hogyan készítsük el a rendszert. A minden részletre kiterjedő tervezés után a tervdokumentáció alapján zajlik a megvalósítás. Amennyiben az implementációt több fejlesztő végzi, akkor szükséges ezek integrálása. Az implementálás és integrálás után a rendszert minden részletre kiterjedően tesztelni szükséges, az esetleges hibákat is ebben a fázisban kell javítani. Ezek után lehet a terméket telepíteni. A telepítés után pedig az utógondozás keretében lehet további új funkcionalitást hozzáadni a termékhez, illetve az esetlegesen még előforduló hibákat is ebben a fázisban javítjuk ki.
A modellben az egyik fázisból csak akkor léphetünk a következő fázisba, ha az adott fázist teljesen befejeztük. Nincs visszafelé lépegetés, és a fázisok sem lapolódnak át. A modell nem írja le igazán jól a valós folyamatokat, azonban egy kezdeti prototípus előállításának folyamatát jól modellezhetjük vele.

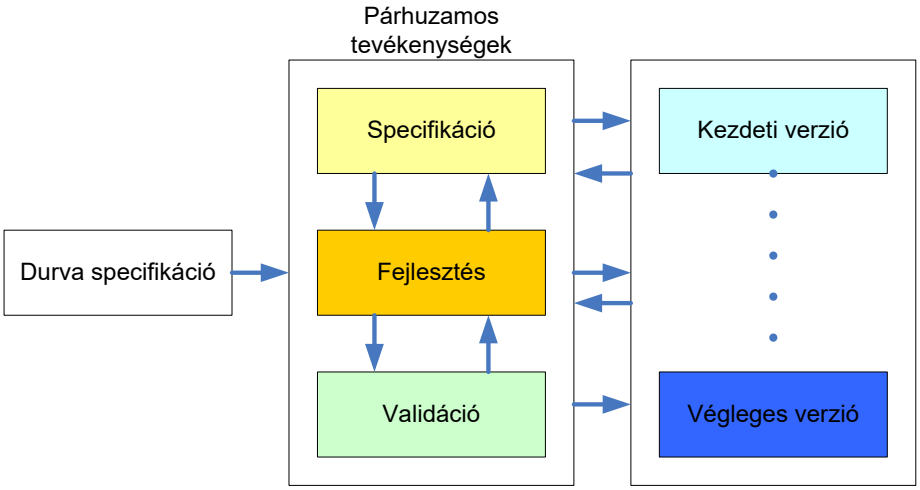


2. ábra A Vízesés modell

A modell legnagyobb hibája, hogy a folyamat elindulása után nem igazán lehet változtatni, valamint hogy az egyes fázisok szigorúan követik egymást, nem lehet közöttük átlapolódás sem.

2.1.2 Evolúciós fejlesztési modell

Az evolúciós fejlesztési modellt annak a helyzetnek a kezelésére dolgozták ki, hogy gyakran egy projekt kezdetén a végső követelmények nem teljes egészében specifikáltak. Gyakran előfordul, hogy egy termékkel kapcsolatosan csak a főbb követelmények, megvalósítandó célok fogalmazhatók meg, viszont a megvalósítás miként, részletei csak a projekt folyamán kerülhetnek rögzítésre.
Az egyik lehetséges út, hogy a megrendelővel együttműködve a kezdeti durva specifikáció alapján elkészítünk egy rendszert, melyet később az igények alapján tovább építünk. Ez a módszer abban az esetben célravezető, ha a megrendelőnek van konkrét, részletes elképzelése a tényleges igényeiről.
Amennyiben a projekt kezdetén csak homályos elképzeléseink vannak a rendszerrel szemben támasztott követelményekről, akkor célravezető lehet, ha elkészítünk a homályos ismeretek alapján egy prototípus rendszert – melynek lehet, hogy egyetlen darabját sem fogjuk felhasználni a végső rendszerben. A prototípus működését elemezve már jobb eséllyel meg tudjuk fogalmazni a tényleges igényeket.
Tehát a kezdeti durva specifikáció alapján elkészítünk egy működő rendszert, majd fokozatosan finomítjuk a specifikációt, és kis lépésekben eljutunk a végső rendszerig.



3. ábra Az Evolúciós fejlesztési modell

A modell legnagyobb hibája, hogy legtöbbször rosszul strukturált, nehezen újrafelhasználható kódot eredményez.

2.1.3 Komponens alapú szoftverfejlesztés

A szoftverrendszerünket már létező saját, vagy a piacon beszerezhető komponensekből integráljuk össze.

A szoftver elkészítésének lépései:
- Követelményspecifikáció
- Komponens analízis
- Követelmények módosítása
- Rendszertervezés újrafelhasználással
- Fejlesztés és integráció

A követelményeket felmérve áttekintjük a saját, ill. a piacon beszerezhető komponenseket. Megvizsgáljuk a rendelkezésre álló komponensek tulajdonságait, és kiválasztjuk a követelményeknek leginkább megfelelő komponenseket. A legtöbb esetben a rendelkezésre álló komponensek nem elégítik ki teljes egészében a követelményspecifikációban megfogalmazott elvárásokat, sőt akár az is elképzelhető, hogy a követelményeken is módosítani szükséges, hogy a kiválasztott komponensek használhatóak legyenek. A komponensek kiválasztása és a követelmények komponensek képességeihez illesztése után elkészítjük a rendszertervet, melyben a kiválasztott komponenseket használjuk. A tervezés után történik a rendelkezésre álló komponensek által meg nem valósított funkciók megvalósítása, valamint a rendelkezésre álló és elkészített komponensek rendszerbe integrálása.
A komponensszabványok elterjedésével a komponens alapú fejlesztés népszerűsége egyre növekszik, azonban beágyazott rendszerekben ez a megközelítés napjainkban még nem tipikus. A komolyabb teljesítményű processzorokat alkalmazó beágyazott rendszerekben a közeljövőben várhatóan egyre inkább terjedni fog ez a megközelítés.

2.1.4 Iteratív megközelítés

A fent említett modellek mindegyikéhez nagyon jól alkalmazható még az iteratív fejlesztési megközelítés, ami annyit jelent, hogy fokozatosan jutunk el valamilyen módon a végtermékig. A fokozatosság megnyilvánulhat abban, hogy szoftverünket részekben szállítjuk, tehát bizonyos funkciókat az első verzióból kihagyunk, majd a következő verziókban szállítjuk csak le azokat. Másrészt megnyilvánulhat úgy is, hogy a fejlesztésünket különböző fázisokra bontjuk, és az egyes fázisokban a fenti modellek valamelyikét alkalmazva elkészítjük az adott fázishoz szükséges részeket, amikor az összes fázison végigértünk, akkor készülünk el a szoftverrel. Az első megoldás előnye, hogy a vevőnek viszonylag gyorsan tudunk valami használhatót adni, viszont korlátozott funkcionalitással – beágyazott rendszerekben ezt a megközelítést nem igazán szokták alkalmazni. A második megközelítés nagy előnye, hogy jól definiálható az egyes fázisokban elérni kívánt cél, és sokkal strukturáltabb kód készíthető ezzel a megközelítéssel, mint a tiszta evolúciós modell segítségével. Az iteratív fejlesztési megközelítéssel az előrehaladás sokkal jobban mérhetővé válik, és ezáltal az üzleti kockázat is lényegesen könnyebben felmérhető.

2.1.5 Általános szabályok

Általános szabályok, melyeket célszerű betartani

1. Követelményspecifikáció:
   Jól mérhető követelményeket állítsunk fel, ill. egyeztessünk a megrendelővel. Vegyük észre, hogy ez már a hardverválasztást is erősen befolyásolhatja. Beágyazott rendszerekben általában a szoftver- és a hardverfejlesztés nem teljesen szétválasztható. Dokumentáljunk!
2. Tervezés:
   Olyan szoftver architektúrát használjunk, ami teljesíti a követelményeket és nem kifejezetten érzékeny az esetleges változtatásokra. Minél részletesebb a tervünk, annál kisebb eséllyel futunk bele nehezen megoldható problémákba az implementáció során.
   Válasszuk ki a megfelelő eszközöket, fejlesztési nyelvet… Dokumentáljunk!
3. Megvalósítás/Implementáció:
   A kiválasztott fejlesztőeszközzel készítsük el a szoftvert. Lehetőleg ne térjünk el a tervektől. Ha a tervezési fázisban mindent jól végig gondoltunk, akkor „csak kódolnunk” kell. Ha mégsem, akkor tervezzük át a kérdéses részt, gondoljuk át az esetleges mellékhatásait, és csak ezután implementáljunk. Dokumentáljunk!
4. Tesztelés:
   Vizsgáljuk meg, hogy az elkészített rendszerünk teljesíti-e a követelményeket. Ha nem teljesíti, menjünk vissza akár a tervezésig is. Dokumentáljunk!
5. Karbantartás:
   Miután elkészült az eszközünk, még mindig lehetnek benne hibák (lesznek is). Gondoskodjunk a hiba okának feltárásáról, javítsuk őket, tanuljunk belőlük, hogy a következő projektjeinkben már eleve kevesebb hibát ejtsünk. Dokumentáljunk!

Egyszerűbb rendszerek esetében a Tervezés-Megvalósítás-Tesztelés lépések nem igazán szétválaszthatók, általában ugyanaz az ember szokta ezeket a feladatokat elvégezni.

Aranyszabály: Előbb gondolkodj, aztán kódolj!

A félév folyamán tervezési és tesztelési módszertanokkal nem fogunk részletesen foglalkozni (MSC képzésben fogunk ezekkel részletesen találkozni), inkább a megvalósításra koncentrálunk.

2.2 Programozási nyelvek

Azzal bizonyára tisztában van mindenki, hogy szoftvert írni sokféle nyelven, sokféle metodika szerint lehet.
A nyelveket sokféleképpen lehet osztályozni, mi csak a legegyszerűbb osztályozást nézzük meg.

2.2.1 Alacsony szintű nyelvek

A lehető legalacsonyabb szintű nyelv a processzor gépi kódjában történő programozás. Ezt a módszert csak a legelső mikroprocesszorok esetén használták. Gépi kódban történő programozás esetén a programozó a memória tartalmát bájtonként adja meg. El tudjuk valószínűleg képzelni, hogy nagyobb szoftverek esetén mennyire lesz olvasható, ill. áttekinthető a programunk. A gépi kódban írt programokat nem kell fordítani, hiszen a processzorunk gépi kódot futtat. A magasabb szintű nyelvek mind gépi kódra fordulnak. Sokkal használhatóbb, mind a mai napig széles körben alkalmazott nyelv az assembly nyelv. Az assembly nyelv csak egy kicsivel magasabb szintű a gépi kódnál, lényegében az utasítások bájtkódját helyettesítjük rövid szöveges azonosítókkal, ún. mnemonic. Az assembly kódot a processzorunk nem képes közvetlenül futtatni, azt le kell fordítanunk a processzor gépi kódjára egy kifejezetten erre a célra készült fordítóval, az assemblerrel.
Az alacsony szintű nyelveken készült programok nem hordozhatóak, még forráskód szinten sem a különböző processzortípusok között. A korábbiakban hallottunk már róla, hogy a processzorokat általában a gyártók családokba szervezik, tehát legtöbbször ugyanannak a magnak a felhasználásával készítenek különböző képességű modelleket. A processzorcsalád tagjai között az alacsonyszintű nyelven íródott kódok – néhány kivételtől eltekintve (pl. olyan periféria kezelését végző kód, mely nincs meg a család minden tagjában) – általában hordozhatóak.

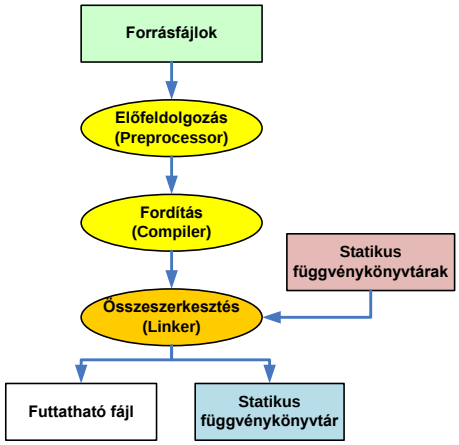
2.2.2 Magasszintű nyelvek

A processzorok fejlődésével, és a rengeteg különböző processzor megjelenésével egyre komolyabb problémát okozott az, hogy az elkészült kódokat nem lehetett egyszerűen különböző processzortípusok között átvinni. A magasszintű nyelvek kialakulásának másik mozgatója volt, hogy az alacsonyszintű nyelven történő programozáshoz elengedhetetlen a processzor működésének és felépítésének alapos ismerete, mivel ennek hiányában még egy egyszerű feladatot sem lehet assemblyben elkészíteni. Szükség volt olyan programozási lehetőségre, mely a processzor típusától lehetőség szerint függetlenül

egy magasabb absztrakciós szinten teszi lehetővé a programok elkészítését. A magasabb absztrakciós szint közelebb áll az emberi gondolkodáshoz, amivel a programozó munkáját nagyban megkönnyíti.
Általánosságban elmondható, hogy az összetettebb programok elkészítése magasszintű nyelven egyszerűbb, az alacsonyszintű nyelveken viszont – legtöbbször – hatékonyabb kódot írhatunk.
A magasszintű nyelven íródott programkódot gépi kódra kell fordítani ahhoz, hogy a processzor futtatni tudja azt.
A legtöbb esetben magasszintű nyelven készítjük el a programjainkat, azonban számtalan olyan eset van, amikor kénytelenek vagyunk assembly nyelven dolgozni, ill. keverni a magas és az alacsonyszintű nyelvek használatát. Természetesen a magas és az alacsony szintű nyelvek együttes használatát támogatnia kell a magasszintű nyelvnek.

2.3 Szoftverfejlesztés PC-re

Nézzük milyen lépésekben jutunk el a kész szoftverhez, illetve milyen eszközökre lesz ezekhez szükségünk – első körben egy kommersz személyi számítógépre íródott szoftver esetén.
A forráskódot elkészítjük valamilyen szövegszerkesztővel, mivel komplex rendszert készítünk, a jobb strukturáltság elérése érdekében több forrásfájlt készítünk. Az elkészített forrásfájlokon különböző előfeldolgozási feladatokat végzünk (include, define). Amikor a forrásfájljaink átestek az előfeldolgozási fázison, jöhet a tényleges fordítás. A fordítás során a forrásnyelvi fájlból egy objektumfájlt készítünk, melyben gépi kódú részek, valamint hivatkozások találhatóak. A gépi kódú részek a forrásfájlban explicite jelenlevő kódból jönnek létre, a hivatkozások feloldása nem a fordító feladata.
A fordítás után a keletkező objektumfájlokat a linker szerkeszti össze egyetlen gépi kódú bináris állománnyá a hivatkozások feloldásával.



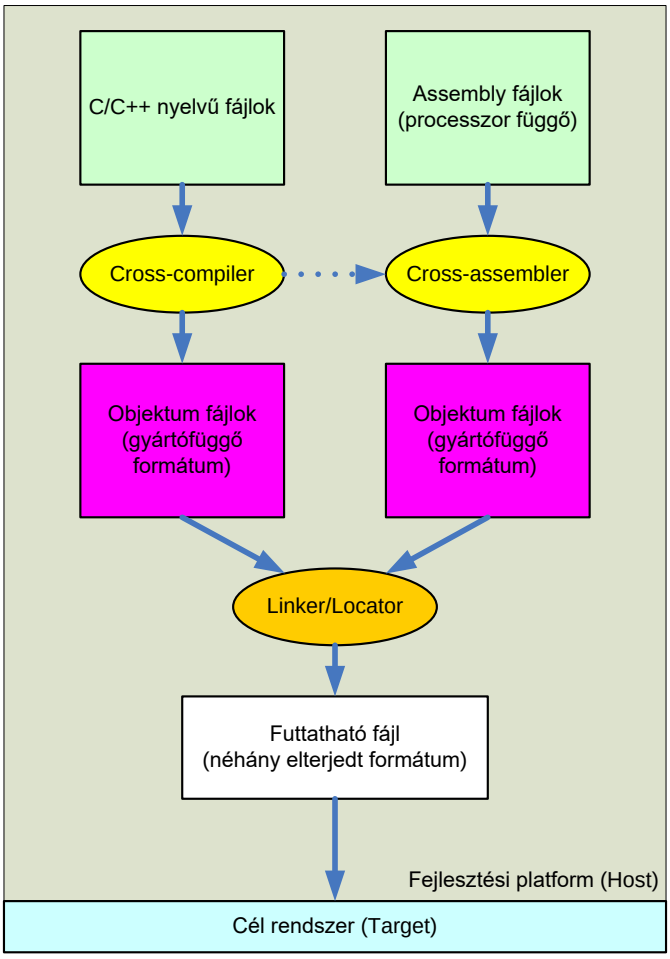
4.ábra Fordítás személyi számítógépre

PC környezetben a fenti lépések elvégzésével előáll a futtatható kód, amit ott helyben ki is próbálhatunk, semmi extra eszköz nem szükségeltetik hozzá.

A legelterjedtebb fejlesztőeszközökben közös kezelőfelületben integrálva megtalálható a szövegszerkesztő, a compiler és a linker.

2.4 Szoftverfejlesztés beágyazott rendszerre

Beágyazott rendszerre történő alkalmazásfejlesztés esetén a feladatunk úgyszintén az, hogy forrásfájlokból valamilyen módon futtatható állományt készítsünk. Általában a céleszközünk nem alkalmas a helyben történő fejlesztésre, ezért a fejlesztést nem itt végezzük, hanem egy személyi számítógépen. Mivel a fejlesztési-, és a cél platform különböző, ezért speciális eszközöket kell használnunk a fejlesztés során. Egyrészt szükségünk van olyan speciális fejlesztőeszközökre, melyek PC-n futnak, azonban a beágyazott rendszer processzorának megfelelő kódot készítenek, ezeket nevezzük cross-platform eszközöknek.



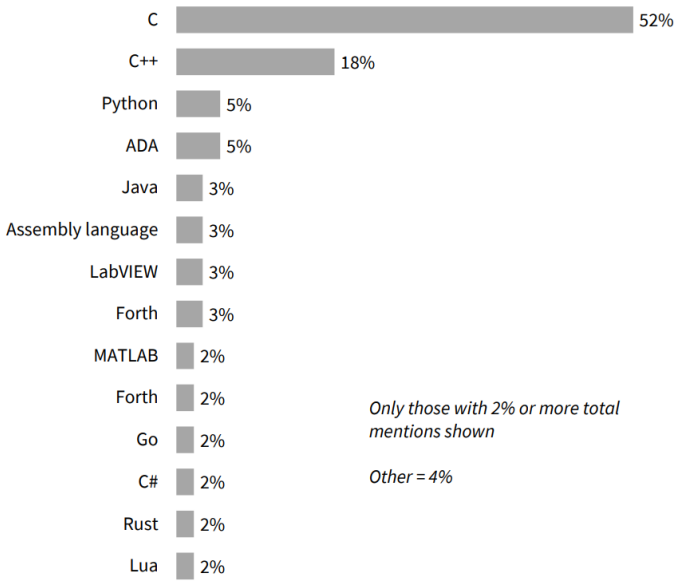
5. ábra Fordítás folyamata beágyazott rendszerre

Beágyazott rendszeren futóképes kód elkészítése után ki is kell azt valahogy próbálnunk. Az egyik lehetőség, hogy fizikailag a céleszközön próbáljuk ki a kódot, ehhez szükségünk van olyan hardver eszközökre, melyek segítségével a beágyazott eszközbe betöltjük a

kódot (flash programozó, in circuit debugger,…). A másik lehetőség, ha nem áll rendelkezésre a beágyazott rendszer, akkor korlátozott funkciókkal PC-n futó emulátor segítségével is kipróbálhatjuk az elkészített szoftverünket. A későbbiekben bővebben fogunk foglalkozni ezekkel az eszközökkel.
Beágyazott rendszerekben – mivel a rendszernek általában szigorú ütemezés szerint kell működnie – olyan magasszintű nyelven készítjük az alkalmazásainkat, melyekhez rendelkezésünkre áll egy olyan fejlesztőeszköz, melynek segítségével sebességben és méretben is a lehető legoptimálisabb kódot tudjuk elkészíteni. Természetesen előfordulnak olyan helyzetek, amikor mindenképpen assembly kódot is kell készítenünk, mert egyrészt esetleg az adott magasszintű nyelv nem tartalmaz olyan utasítást, ami a kívánt funkciót elvégzi, másrészt közel sem biztos, hogy a fordító a magasszintű nyelven megírt kódunkat sebesség szempontjából a legoptimálisabb gépi kódra fordítja le. Ez nagyon fontos szempont – főleg a kisebb teljesítményű kontrollerek esetében.
A fenti szempontok, és megfontolások alapján olyan nyelvet kell választanunk a beágyazott szoftverünkhöz, amely:

- széles körben ismert, és elterjedt
- létezik hozzá az adott eszközhöz fordító
- támogatja az assembly kód beágyazását

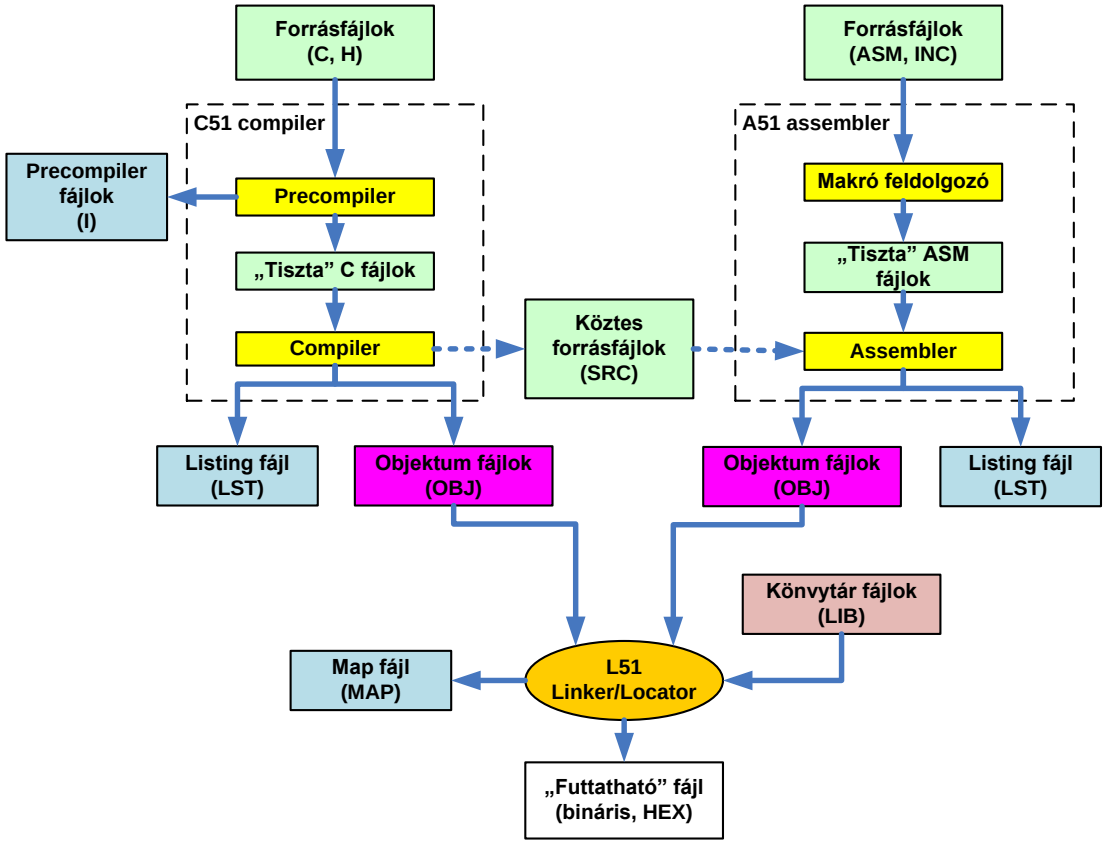
Ezeknek a feltételeknek tökéletesen megfelel a C nyelv, ami a beágyazott rendszerekben széles körben használt, a kisebb teljesítményű processzorokon szinte kizárólagos magas szintű nyelv. A nagyobb teljesítményű processzorokon, amikor már lehetőségünk nyílik különböző beágyazott operációs rendszerek és futtatókörnyezetek használatára, ott már megjelenik a C++, Java, C#, VB és más magas szintű nyelvek is.



6. ábra Szoftverfejlesztéshez használt nyelvek használati arányai

A fenti ábrán a korábban már említett felmérésből származó adatokat láthatunk, ezek az adatok is alátámasztják választásunkat – hiszen a leggyakrabban C nyelven történő alkalmazásfejlesztést választották a felmérésben résztvevő szakemberek is.

A korábbiakban megismerkedhettünk egy, az iparban széles körben elterjedt, rengeteg különböző alkalmazásban használt kontrollercsaláddal, az Intel 8051 alapú kontrollerekkel. A következőkben megnézzük, hogy milyen lépésekben, milyen eszközök felhasználásával juthatunk el a forráskódtól az elkészült bináris formátumú alkalmazásig. Azért ezt a kontroller családot választjuk mintának, mivel a beágyazott szoftverfejlesztés minden fontos részlete bemutatható rajta, azonban kellően egyszerű ahhoz, hogy ne vesszünk el a processzorcsaládd al kapcsolatos részletekben. A fordítás lépéseit és a keletkező fájlokat a Keil cég fordítócsomagjához illeszkedően mutatjuk be. Ez a fordítócsomag jelenleg a 8051 mikrokontrollerek világában piacvezető termék, melyet a legtöbb 8051 alapú mikrokontrollert gyártó cég támogat és ajánl a termékeihez.
Tegyük fel, hogy elkészítettük az alkalmazásunk forráskódját C vagy assembly nyelven, tehát eljutottunk odáig, hogy szeretnénk lefordítani a forráskódot. A 7. ábrán láthatjuk a beágyazott rendszerekre történő fordítás lépéseit, az ábrát követve végignézzük, hogy az egyes lépésekben mit dolgozunk fel, és mi lesz az adott lépés eredménye.



7. ábra Fordítás menete a Keil C51 fordítóval

Preprocesszor

Amennyiben C nyelven készítettük a forrásállományainkat, akkor első lépésben mindenképp szükséges a forrásállományokon egy előfeldolgozást végeznünk. Erre a lépésre azért van szükség, mivel szinte minden esetben szerepeltetünk olyan utasításokat a forrásállományainkban (pl. #include), melyek befolyásolják a lefordítandó állomány

tényleges tartalmát. A preprocesszor a compiler szerves része, tehát nem egy különálló alkalmazás. A preprocesszor feladata, hogy az állományunkat átalakítsa teljesen „tiszta" C nyelvű kóddá, ez a gyakorlatban annyit jelent, hogy a forrásállományunkba behelyettesíti (#include, #define), illetve a forrásállományból elhagyja (#ifdef, #ifndef) a preprocesszor utasításoknak megfelelő tartalmat. A fordító megfelelő opciójának bekapcsolása esetén ezt a preprocesszor utasításoktól megtisztított „tiszta" C nyelvű állományt elmenti. A keletkező állomány a preprocesszor kimeneti állománya, a keletkező fájl *.i néven kerül elmentésre.

Cross-compiler

Az előfeldolgozási fázis után következhet a tényleges fordítás. A fordítás során a C nyelven megírt forrásfájlból elsődleges kimenetként egy objektumfájlt készít a fordító. Az objektumfájl lényegében nem más, mint a C nyelven megírt kódsorok gépi kódra lefordított megfelelője, kiegészítve még a forrásfájlok összeszerkeszthetőségéhez szükséges információkkal. Az objektumfájlban szerepelnek az előzőeken kívül még a hatékony hibakeresést lehetővé tevő részletek, ezek lényegében azt írják le, hogy az adott gépi kódú programrészlet melyik forráskódú utasításnak felel meg, és az hol található a forrásállományban. Az objektumfájlok *.obj néven kerülnek elmentésre.

Az objektumfájl tartalmáról, valamint a fordító működésével kapcsolatos eseményekről eseménynapló is készül a fordítás során, ez az eseménynapló általában *.lst fájlnéven kerül elmentésre. Az eseménynaplóból a fordítással kapcsolatos minden lényeges információ kinyerhető. Az eseménynapló, vagy más néven listing fájl a következő információkat tartalmazza:

- Compiler parancssor
- Forráskód
- Forráskód assembly-re fordítva
- Szimbólumok listája
- Modul információk (memóriafoglalás típusonként)

A compiler az objektum és a listing fájlokon kívül – beállítástól függően – képes még előállítani a forrásfájlunk assembly nyelvre lefordított változatát, ez általában *.src fájlnéven kerül elmentésre. A compilernek ezt a beállítását több célból szoktuk használni, egyrészt kötelezően használni kell akkor, ha a C forrásunkban assembly betéteket helyezünk el. Mivel a compiler nem képes assembly kódot gépi kódra fordítani, így első lépésben ilyenkor assembly nyelvű fájlt készíttetünk a C forrásunkból, amiben az assembly betéteink változtatás nélkül megtalálhatóak lesznek, aztán az assembly fájlt lefordítjuk a cross-assembler segítségével. Amennyiben a compilernek ezt az opcióját használjuk, akkor nem keletkezik objektum fájl, azt majd a cross-assembler fogja elkészíteni.

Cross-assembler

Amennyiben a forrásfájljaink valamelyike assembly nyelven íródott, illetve a C nyelvű forrásainkba assembly betétet helyeztünk el, szükségünk van a cross-assemblerre, hogy elsődleges kimenetként az objektumfájlt előállíthassuk. Az assembler által előállított

objektum fájl lényegében ugyanazt tartalmazza, mint a compiler objektum fájlja – nem is lehetne ez másként, hiszen ugyanannak a linkernek kell bemenetet szolgáltatniuk. Az assembler is készít naplófájlt az általa elvégzett feladatokról, ugyanazzal a tartalommal, mint a C compiler.

Linker

A linker feladata, hogy az objektumfájlokból összeszerkessze a programmemória tartalmát, és minden egyes függvénynek, változónak, konstansnak meghatározza a fizikai helyét a memóriában. A linker elsődleges kimenete a programmemória végleges bináris tartalmát leíró bináris fájl, ez a fájl általában kiterjesztés nélküli nevet szokott kapni. Az összeszerkesztés és elhelyezés folyamatáról a linker naplófájlt készít, melyet map fájlnak is nevezünk. A naplófájlt a linker általában *.map, vagy *.m51 néven szokta elmenteni.

A map fájlban a következő információk találhatók meg:

- Linker parancssor
- Bemeneti modulok listája
- Memória térkép
- Hívási fa
- Publikus szimbólumok listája
- Hivatkozások listája

A felsorolás első két eleme nem szorul magyarázatra, a következőkről viszont érdemes pár szót ejteni. A memóriatérkép mutatja meg, hogy az egyes globális változóink, valamint az általunk megírt függvények hova kerülnek a memóriában, és mekkora helyet foglalnak. A hívási fa azt mutatja meg, hogy az alkalmazásunk forráskódjában, és az esetleg más függvénykönyvtárakban szereplő függvények milyen láncolatokon keresztül hívják egymást. A publikus szimbólumok listájában megtalálhatjuk az összes olyan szimbólumot, melyek a forrásfájljaink közötti együttműködésben használhatóak. A hivatkozások listája pedig megmutatja, hogy a különböző forrásfájljainkban szereplő függvények milyen módon kapcsolódnak egymáshoz.

A linker által elkészített bináris kódot valamilyen eszköz segítségével el kell juttatnunk a célrendszer programmemóriájába. Ehhez a feladathoz általában valamilyen külső flash programozó eszközre van szükségünk. A külső eszközök nem minden esetben tudják értelmezni a linker által készített bináris adatfolyamot, ezért a bináris információt elő kell állítani olyan formátumban, hogy azt a flash programozó eszköz fel tudja dolgozni, és el tudja végezni a céleszköz beprogramozását. Az iparban több, erre a célra alkalmas formátum létezik – ezeket a nagyobb processzorgyártó cégek dolgozták ki (Motorola S-record, Intel Hex). A linker – mivel egy eredetileg Intel gyártmányú processzorral foglalkozunk – az Intel Hex fájlformátumában is képes előállítani a végleges memóriatartalmat, az előállított fájl neve *.hex lesz.
A legtöbb programozó eszköz ismeri az Intel Hex fájlformátumot, mivel kvázi ipari szabvány.
Az Intel Hex fájl egy bináris információkat ASCII kódolással leíró szöveges fájl, ebből következően könnyen, különleges segédeszköz nélkül is megtekinthető a tartalma.
A hex fájlban tetszőleges számú, azonban meghatározott formátumú sorok - rekordok szerepelhetnek. A sorok öt mezőt tartalmazhatnak a következő formátumban:

:llaaaatt[dd...]cc

Minden betűcsoport egy-egy mezőhöz tartozik, és minden betű egy hexadecimális digitet jelöl. Minden mező legalább 2 hexadecimális digitet tartalmaz – ez pont egy bájtnyi adat. A sorok jelentése a következők szerint értelmezendő:

- minden sor kötelezően kettősponttal kezdődik
- az ***ll*** mező a rekordban szerepeltetett adatbájtok számát adja meg
- az ***aaaa*** mező az első adatbájt memóriacímét adja meg
- a ***tt*** mező a rekord típusát adja meg – lásd alább
- a ***dd…*** mező nem minden rekordtípusban szerepel, az adatbájtokat adhatjuk meg vele, mivel a hossz mező két digit hosszú lehet, ezért egy rekordban maximum 255 bájtnyi (hexadecimálisan FF) adatot tudunk leírni.
- a ***cc*** mező pedig egy ellenőrzőösszeg, melyet úgy képzünk, hogy vesszük az összes megelőző bájt összegét 1 bájton, majd az összeg kettes komplemense lesz az ellenőrzőösszegünk.

A Hex fájlban 4 féle rekordtípus szerepelhet:

- tt = 00 Adatrekord
- tt = 01 Fájlvége rekord
- tt = 02 Kiterjesztett szegmenscím rekord – x86 specifikus
- tt = 04 Kiterjesztett lineáris cím rekord

Az adatrekordok írják le a memória tényleges tartalmát, ebben a rekordtípusban szerepeltetjük kizárólag az adatmező értékeit.
A fájlvége rekorddal jelezzük, hogy véget ér a hex fájl, nincs több feldolgozandó adat – a fájl végét jelezheti egy olyan adatrekord is, melyben nincsenek adatbájtok.
A kiterjesztett szegmenscím rekord x86 specifikus, itt nem foglalkozunk vele.
A kiterjesztett lineáris cím rekord lehetővé teszi, hogy ne csak 64 kBájtnyi memóriát tudjunk megcímezni – négy hexadecimális digiten pont ekkora memóriaterület címezhető meg – hanem ki lehessen a megcímezhető tartományt terjeszteni egészen 4GBájtig. A kiterjesztett lineáris cím rekord adatbájtjaiban szereplő cím, és az ezt követő adatrekordokban szereplő címek együttesen fogják meghatározni a tényleges fizikai címet. A kiterjesztett lineáris cím rekordban szereplő cím lesz a tényleges cím felső két bájtja, az adatrekordban szereplő pedig az alsó két bájtja. Mindaddig, míg újabb kiterjesztett lineáris cím rekordot nem találunk a fájlban, addig ez a címzési mód lesz érvényes.

Nézzünk egy mintát, hogy hogyan is nézhet ki egy hex fájl:

```
:1000030075810775892075 8DFDD28E759852C20052
:0200000400FFFB
:03000000020003F8
:0B00130090001E12003612002B80F53A
:00000001FF
```

Az első sor egy adatrekord, mely a 0003-as címtől kezdődően tartalmaz 16 bájtnyi adatot.
A második sor egy kiterjesztett lineáris cím típusú rekord, melyben 2 adatbájt található.

A 2 adatbájt értéke 00FF, ez azt jelenti, hogy az összes következő adatrekordban szereplő cím és a 00FF cím együttesen fogják meghatározni a fizikai címet. A harmadik sor ismét egy adatrekord, melyben 3 bájtnyi adat található. Mivel a második és a harmadik sor együttesen határozzák meg a címet, így a harmadik sorban szereplő első adatbájt a 00FF0000 címre fog kerülni. A negyedik sor ismét egy adatrekord, mivel van érvényes kiterjesztett lineáris címünk, így a 11 adatbájt a 00FF0013 címtől kezdődően kerül elhelyezésre a memóriában. Az utolsó sorban pedig a fájl végét jelző rekordot találhatjuk meg.
Egy – C nyelven megírt – alkalmazás fordítási folyamata során keletkező kimeneti fájlok tartalmát tekinthetjük meg a 2.16 fejezet első mintapéldájában.

2.5 Fejlesztőeszközök

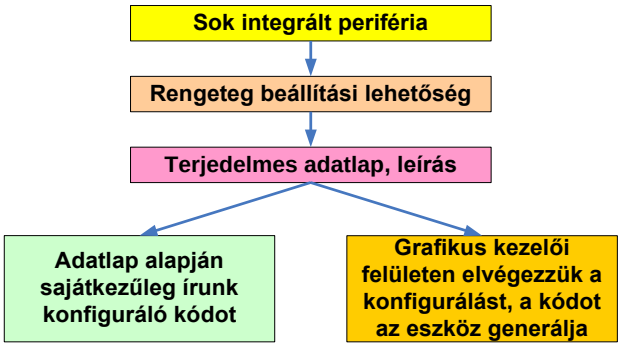
A korszerű fejlesztőeszközök egyetlen integrált kezelőfelületen teszik elérhetővé a forráskód elkészítéséhez, és az elkészült forráskód lefordításához szükséges eszközöket. Ahhoz, hogy hatékonyan tudjuk elkészíteni a programunk forráskódját, és a forráskód áttekinthető legyen, szükségünk van első lépésben egy nagyon jó szövegszerkesztőre, melyben az olyan gyakran használt funkciók, mint a csoportos tabulálás, vagy a különböző címkék, adattípusok színezése megoldott. Az ilyen, kifejezetten forráskód készítésre fejlesztett szövegszerkesztők használatával a fejlesztési folyamat lényegesen felgyorsulhat.
Nagyon fontos, hogy a nagyobb projektek esetében a projekt keretében készített forrásfájlok között hatékonyan tudjunk navigálni, ill. esetleg ha az elkészítendő szoftverrendszer több projektet tartalmaz, a projektek közötti navigáció is kényelmes legyen. A korszerű fejlesztőeszközök általában egy workspace – munkaterület jellegű fogalom segítségével teszik lehetővé, hogy a szoftverrendszerünkhöz több projektet adhassunk, a projekteken belül pedig a forrásfájljainkat kezelhetjük.
Természetesen, mivel a fordítási folyamat több lépésből áll, több különböző, egymásra épülő alkalmazást kell kezelnünk, azokon különböző beállításokat kell elvégeznünk a hatékony fordítás elérésének érdekében. A korszerű fejlesztőeszközökben általában grafikus felületen keresztül állíthatjuk be az egyes eszközeink paramétereit – természetesen ezek majd az adott eszköz parancssori paramétereit fogják beállítani.
Nagy projektek esetében hasznos, ha a fejlesztőeszköz képes különböző forráskód menedzsment eszközökhöz, verziókezelő rendszerekhez csatlakozni. A korszerű fejlesztőeszközökben ehhez is van manapság már támogatás.
Nagyon fontos, hogy a fejlesztőeszköz hatékony támogatást biztosítson az elkészült alkalmazásban történő hibakeresésre.

A hibakereséshez szükségünk van:
- Töréspontok – Breakpoint elhelyezésére a forráskódban
- Léptetésre utasításonként, utasításcsoportonként
- Változók értékének nyomonkövetésére – Watch
- Memóriatérkép megtekintésére – Memory map
- Regiszterek állapotának nyomonkövetésére – Register watch
- Hívási folyamat megtekintésére– Call stack
- Az eredeti forrás és a lefordított assembly egyidejű megtekintésére - Disassembly

A korszerű fejlesztőeszközök általában tartalmazzák ezeket a hibakereséshez nélkülözhetetlen lehetőségeket.

2.6 Kiegészítő eszközök

Az integrált áramköri technológia fejlődésével a mikrokontroller gyártók egyre növekvő számú perifériát integrálnak a legolcsóbb mikrokontrollerekbe is. Ez egyrészről nagyon hasznos, mivel a mikrokontrollert tartalmazó nyomtatott áramkör jelentősen egyszerűsödik. Az integrált perifériák számának növekedésével együtt jár, hogy a mikrokontroller leírása, adatlapja jelentősen meghízik. A legkisebb mikrokontrollerek esetében is néhány száz oldalt tesz ki a különböző integrált perifériák beállítási lehetőségeinek, és működésének a leírása. Mivel a leírás elég terjedelmes, ezért sokszor nem könnyű az összes – adott feladat megvalósításához szükséges – periféria konfigurálási adatainak előkeresése a dokumentációból. A szoftverfejlesztők munkáját megkönnyítendő a legtöbb mikrokontroller gyártó cég különböző kiegészítő szoftver eszközöket szokott biztosítani a megvásárolt mikrokontrollerek mellé, melyek segítségével az adott eszköz konfigurálásához szükséges forráskódot egy grafikus felhasználói felületen össze lehet állítani.



8. ábra Mikrokontroller konfigurációs kódjának elkészítési lehetőségei

A konfigurációs szoftver felületén viszonylag könnyedén összeállíthatjuk a mikrokontroller konfigurációs kódját, azonban azt tartsuk mindvégig szem előtt, hogy a konfigurációs szoftverek sem tökéletesek, az általuk generált kód nem mindig azt a konfigurációt állítja elő, amit elvárnánk. A konfigurációs kód elkészítésének legbiztosabb módja, ha saját kezűleg az adatlap alapján megírjuk. Amennyiben konfigurációs szoftvert használunk, legalább annyit tegyünk meg, hogy a generált kód főbb részeit az adatlap alapján ellenőrizzük.
A legtöbb mikrokontroller gyártó a konfigurációs szoftverek mellett, vagy helyett kínál függvénykönyvtárakat, melyek segítségével a hardver inicializálása és használata nagymértékben leegyszerűsödik. Nagyteljesítményű mikrokontrollerek használata esetén – főleg, ha még nem vagyunk tapasztaltak az adott családdal kapcsolatban – erősen javasolt ezek használata.

2.7 Assembly programozás 8051-re

A 8051 alapú rendszerekre történő fejlesztést lehetővé tévő, széles körben használt fejlesztőeszköz a Keil cég uVision szoftvercsomagja. A szoftvercsomag része az A51 cross-assembler, melynek segítségével az assembly nyelven megírt kódot gépi kódra tudjuk lefordítani. Az assembler objektum fájlokat készít, melyeket ezután az L51 linker segítségével tudunk egyetlen bináris fájlba összefűzni. A következőkben az assembly programok felépítését, és az assembly programozással kapcsolatos legfontosabb tudnivalókat szeretnénk ismertetni.

2.7.1 Assembly kifejezések

Az assembly programok forrásfájljaiban használható kifejezéseket három nagy csoportba sorolhatjuk. Használhatjuk egyrészt a 8051 processzor utasításkészletének megfelelő utasításokat (mnemonikok), másrészt használhatunk különböző vezérlő utasításokat, valamint direktívákat. Nézzünk egy egyszerű assembly nyelvű programot:

```
$INCLUDE (SI_EFM8BB3_Defs.inc)
       CSEG   AT  0
       LJMP   Main
Main:  LJMP   Main
       END
```

A fenti assembly programban összesen négy kifejezést láthatunk. Az első kifejezés a $INCLUDE, ami egy assembler direktíva, a CSEG és az END egy vezérlő utasítás, az LJMP pedig egy 8051 assembly utasítás.

Az assembly forrásfájlok bármilyen szövegszerkesztővel elkészíthetőek feltéve, ha betartjuk a következő egyszerű szabályokat:

- Egy sorban egyetlen assembly utasítás, vezérlő utasítás, illetve direktíva szerepelhet.
- Egy kifejezésnek egyetlen sorban el kell férnie, a többsoros kifejezések nem engedélyezettek.
- A kifejezések neve és az operandusaik között legalább egy space-t kell hagynunk.
- Használhatunk bármilyen tabulálást, az assembly kifejezések nem érzékenyek a tabulálásra. A programkód olvashatósága érdekében célszerű konzekvensen használni a tabulálást.
- Minden assembly programban szerepelnie kell pontosan egy helyen az END vezérlő utasításnak. (A legegyszerűbb assembly program egyetlen END utasításból áll). Az END utasítás jelzi az assembly programunk végét, amennyiben a programunkban több END utasítás található, az assembler a legelső END utasításig dolgozza fel a forráskódunkat.

Vezérlő utasítások

A vezérlő utasítások az assembler számára nyújtanak információt arról, hogy miképp dolgozza fel a különböző kifejezéseket, tehát az assemblert vezérlik. A vezérlő utasítások használhatóak például konstansok létrehozására, változóknak történő helyfoglalásra, és még sok hasonló feladatra. A későbbiekben részletesen lesz szó a vezérlő utasításokról.

Direktívák

A direktívák segítségével az assembler kimeneti fájljainak – az objektum és listing fájlok – tartalmát tudjuk befolyásolni. A direktívák általában nem befolyásolják az assembler által generált kódot. Kivételt képeznek ez alól a feltételes fordítást lehetővé tevő direktívák. A direktívákat egyrészt szerepeltethetjük az assembly forrásfájlunkban, valamint az assembler parancssorában is – kivéve a feltételes fordítás direktíváit, azok csak a forrásfájlban szerepelhetnek.

Assembly utasítások

Az assembly utasításokkal írjuk le azt a programot, amit az assembler majd ténylegesen gépi kódra lefordít, és az objektum fájlba elhelyez.

Az assembly utasítások általános formátuma a következők szerint alakul:

```
[cimke:] mnemonic [operandus][,operandus][,operandus][;komment]
```

Ahol a

Címke	egy szimbólum, amivel az adott sor címére hivatkozhatunk,
Mnemonic	a processzor valamely utasításának ASCII kódban leírt neve
Operandus	a mnemonic által meghatározott paraméter(ek)
Komment	Komment☺

A 8051 processzor utasításkészletéről volt már szó, valamint a processzorok adatlapján részletes leírás található róluk.

2.7.2 Szimbólumok

A szimbólum egy név, melynek segítségével egy adott címre, egy értékre, egy regiszter nevére, vagy szövegblokkra hivatkozhatunk. A szimbólumok használhatóak még konstansok és kifejezések elnevezésére is.

A szimbólumokkal kapcsolatos legfontosabb szabályok:

- Maximum 31 karakter hosszú
- A szimbólumnévben használhatunk betűket, számokat, aláhúzást, és kérdőjelet.
- A szimbólumnév nem kezdődhet számmal
- Egy szimbólumot csak egyszer szabad definiálni

Szimbólumok definiálására számos lehetőségünk van. Egyrészt használhatjuk az EQU és a SET vezérlőutasításokat, ezek használatával különböző kifejezésekhez tudunk nevet rendelni – ezekről a későbbiekben részletesebben is olvashatunk. Például:

```
NUMBER_FIVE    EQU  5
TRUE_FLAG      SET  1
FALSE_FLAG     SET  0
```

A címkék is egyfajta szimbólumok, melyeket a következőképpen tudunk definiálni:

```
LABEL1: DJNZ R0, LABEL1
```

A címkéket arra használjuk, hogy a programunk egy adott „helyét” elnevezzék. Lényegében a programunk adott sorára tudunk velük hivatkozni. Mint az assembly utasítások általános formátumából is látható, a cimkének az első szövegnek kell lenni az adott sorban, és kötelezően egy kettőspont követi a címke nevét. A címke neve előtt és a kettőspont után tetszőleges számú tabuláló karakter szerepelhet. Egy sorhoz egyetlen cimkét rendelhetünk. A cimkét lényegében a helyszámláló (location counter) adott szegmensben érvényes értékéhez rendeljük hozzá. Címkékre láthatunk még néhány példát az alábbiakban:

```
LABEL1: DS 2
LABEL2: ;címke önmagában
NUMBER: DB 27, 33, 'STRING', 0
```

Szimbólumot létrehozhatunk még olyan célból is, hogy egy változó címére hivatkozni tudjunk, erre láthatunk egy példát.

```
SERIAL_BUFFER DATA 99h
```

Szimbólumokat az assembly programokban kifejezetten sűrűn szoktunk használni, mivel sokkal olvashatóbbá teszik a programkódot. Sokkal könnyebb egy szimbólum jelentését megérteni, mint egy numerikus azonosítót.

2.8 A 8051 utasításkészlete

Az assembly nyelvű programjaink legfontosabb célja, hogy a processzorunk utasításait használva elkészítsünk egy az adott processzoron futni képes alkalmazást. Az utasításkészlet leírását a korábbiakban az 1.5.4 pont alatt már részletesen bemutattuk, ezért ismételt bemutatásától eltekintünk.

2.9 A 8051 memóriatérképe

A mikrokontroller memóriatérképét a korábbiakban az 1.5.1 pont alatt már részletesen bemutattuk, ezért ismételt bemutatásától eltekintünk.

2.10 Vezérlőutasítások

Az A51 cross-assembler számos vezérlőutasítással rendelkezik, melyek segítségével szimbólumokat definiálhatunk, memória területet foglalhatunk, inicializálhatunk, valamint a kódunk memóriában történő elhelyezését befolyásolhatjuk.
A vezérlőutasítások hatására nem keletkezik gépi kód, a DB vezérlőutasítás kivételével a kódmemória tartalmát nem befolyásolják. A vezérlő utasítások az assemblert vezérlik, meghatározzák, hogy hogyan dolgozza fel a forrásfájlban található assembly utasításokat. Ne keverjük össze a vezérlőutasításokat az assembly utasításokkal!

A vezérlőutasítások a következő csoportokba sorolhatók:
- Szegmens vezérlés
- Szimbólum definíció

- Memória inicializáció
- Memória foglalás
- Linkeléssel összefüggő vezérlőutasítások
- Címkezelés
- Egyebek

2.10.1 Szegmensek kezelése

A 8051 mikrokontroller esetében – a harvard jellegű architektúra következtében – különös figyelmet kell szentelnünk annak, hogy a rendelkezésre álló memóriában mit hová helyezünk. Korábban bemutatásra került a mikrokontroller memóriájának felépítése. Mindenképp érdemes tudnunk – ez a harvard architektúrából is következik – hogy a mikrokontroller egymástól teljesen függetlenül kezeli a ROM-ot (kód szegmens), valamint a RAM-ot. Röviden ismételjük át, hogy milyen részekre (szegmensekre) osztható fel a memória. Egyrészt rendelkezésünkre áll 64 kByte kód memória (ROM), melyet kódszegmensnek nevezünk (CSEG). Másrészt rendelkezésünkre áll 256 Byte belső adatmemória, melyből 128 byte direkt címezhető (adat szegmens, DATA), a maradék 128 byte indirekt címezhető (IDATA). A direkt címezhető tartomány alján (00-1F) találhatóak a regiszterek 4 bankban, felette pedig (20-28) a bit címezhető memória (BDATA). A belső memórián kívül rendelkezésünkre áll még egy külső memóriarész is (XDATA), mely 64kByte méretű. A címtartomány alján a belső memória érhető el.

Az assembly programunkban minden esetben meg kell határoznunk, hogy mit melyik memóriarészre szeretnénk elhelyezni. Alapszabályként azt érdemes követni, hogy minden kód kerüljön a kódmemóriába – máshová nem is tudnánk tenni, mivel a processzor csak a kódmemóriában lévő kódot tudja futtatni. A kódon kívül kerüljenek mindenképp a kódmemóriába a konstansok. Minden egyéb adatot helyezzünk az adatmemória megfelelő részére.
A szegmensvezérlő utasítások segítségével tudjuk meghatározni, hogy mit hová helyezzen az assembler. Az assembler minden szegmensre vonatkozóan kezel egy helyszámlálót (location counter). A helyszámláló lényegében egy mutató, mely az aktív szegmens címterületére mutat. Amikor egy szegmenst először aktiválunk, a helyszámláló értéke 0 lesz. Minden egyes utasítás után az utasítás méretének megfelelően a helyszámláló értéke növekszik. A memória foglalására szolgáló vezérlőutasítások is megváltoztatják a helyszámláló értékét. A helyszámláló értékét explicit módon is módosíthatjuk az ORG vezérlőutasítás segítségével. A helyszámláló aktuális értékére a $ szimbólummal hivatkozhatunk.

Abszolút szegmensek

Alapértelmezés szerint a processzor memóriatérképének megfelelően minden tartományra létezik egy-egy saját szegmens, melyeket abszolút szegmenseknek nevezünk. Az abszolút szegmensek kiválasztása a szegmens típusának megfelelő vezérlőutasítás segítségével történik.
Az abszolút szegmensek kiválasztására a következő vezérlőutasítások használhatóak:
CSEG – kódszegmens kiválasztása (CODE tartomány)
DSEG – adatszegmens kiválasztása (DATA tartomány)

BSEG – bitcímezhető tartomány kiválasztása (BIT tartomány)
ISEG – belső indirekt címezhető szegmens kiválasztása (IDATA tartomány)
XSEG – külső memória szegmens kiválasztása (XDATA tartomány)

A kiválasztó utasítások szerepelhetnek paraméter nélkül, ekkor az adott memóriatartomány kezdőcímének a kiválasztása történik meg. Valamint meghívhatóak paraméter megadással is a következő formátum szerint:

```
xSEG AT address
```

Ebben az esetben a paraméterben megadott cím kerül kiválasztásra az adott szegmensen belül. Természetesen a címnek illeszkednie kell az adott memóriatartomány méretéhez, túlcímzés esetén az assembler hibaüzenetet generál.
Nézzünk néhány példát, hogy hogyan tudunk abszolút szegmenseket kiválasztani:

```
CSEG
BSEG AT 30h  ; absolute bit  segment @ 30h
CSEG AT 100h ; absolute code segment @ 100h
DSEG AT 40h  ; absolute data segment @ 40h
```

Általános szegmensek

Az abszolút szegmensek segítségével lehetőségünk nyílik a memória tetszőleges címének kiválasztására, azonban ez a megoldás nem a legkényelmesebb, mivel mindig a fizikai címre kell hivatkoznunk, amikor például a kódtartomány bizonyos részét szeretnénk kiválasztani. A könnyebb kezelhetőséget, és a jobb átláthatóságot biztosíthatjuk a programunkban, amennyiben nem csak abszolút szegmenseket, hanem ún. általános célú szegmenseket is használunk. Az általános célú szegmensek használatával lényegében címkéket ragaszthatunk a memóriánk bizonyos részeihez, és ezeket a címkéket kiválasztva a memória tetszőleges helyére ugorhatunk. Ha ilyen módon partícionáljuk a rendelkezésre álló memóriánkat, akkor a programunk esetleges későbbi módosítása esetén sokkal könnyebben át tudjuk szervezni a memóriánk tartalmát, amivel rengeteg időt és munkát takaríthatunk meg.

Általános célú szegmenseket a SEGMENT vezérlőutasítás segítségével hozhatunk létre. A SEGMENT vezérlőutasításnak meg kell adnunk, hogy mi legyen a címke neve, valamint, hogy melyik memóriatartományban szeretnénk az adott szegmenst létrehozni. A SEGMENT utasítást soha nem használjuk önmagában, mivel csak egy címkét hoz létre. A létrehozott cimkét az RSEG utasítás segítségével ki kell választanunk. Az RSEG utasítás paraméterében meg kell adnunk a kiválasztandó szegmens nevét.
Nézzünk egy példát általános szegmens létrehozására, és kiválasztására:

```
MYPROG SEGMENT CODE      ; deklaráljuk a szegmenst
RSEG MYPROG              ; kiválasztjuk a szegmenst
MOV A, #0                ; a kód a szegmensbe kerül
```

A fenti mintapéldában létrehoztunk egy általános memória szegmenst a kódmemóriában és kiválasztottuk azt. A szegmens kiválasztását követő assembly utasítások a kiválasztott szegmensbe kerülnek.

2.10.2 Szimbólum definíció

A korábbiakban olvashattunk már róla, hogy szimbólumokat általában azért hozunk létre, hogy a programunk olvashatóbb, áttekinthetőbb legyen. Arról is volt már szó, hogy a szimbólumok egy konstans számot, egy regisztert, vagy pedig egy címet szimbolizálnak. Szimbólumok például az assembly programunkban lévő címkék, valamint a memóriaszegmenseink nevei is. Természetesen szükségünk van olyan lehetőségre, aminek segítségével explicit módon tudunk el tudunk nevezni egy konstanst, vagy egy regisztert, vagy pedig egy memóriacímet.
Számot vagy regisztert szimbolizáló szimbólumok létrehozásához használhatjuk az EQU ill. a SET vezérlőutasításokat. A két vezérlőutasítás szintaktikája megegyezik, a szimbólum neve után a vezérlőutasítás következik, majd pedig a szimbólumhoz hozzárendelő értéket szerepeltetjük.
A SET vezérlőutasítással létrehozott szimbólumok bármikor felüldefiniálhatóak, tehát újra adhatunk nekik értéket. Az EQU vezérlőutasítással létrehozott szimbólumok nem lesznek felüldefiniálhatóak, értéküket csak egyetlen egyszer állíthatjuk be.
Szám ill. regiszter szimbólumok létrehozására láthatunk az alábbiakban néhány példát:

```
LIMIT EQU 1200
VALUE EQU LIMIT – 200 + 'A'
SERIAL EQU SBUF
ACCU EQU A
COUNT EQU R5
VALUE SET 100
VALUE SET VALUE / 2
COUNTER SET R1
TEMP SET COUNTER
TEMP SET VALUE * VALUE
```

Mint korábban említettük, nem csak számokat és regisztereket jelentő szimbólumok hozhatók létre az assembly programban, hanem olyan szimbólumok is, melyek egy adott memóriacímet jelentenek. A memóriacímet jelentő szimbólum létrehozására szolgálnak a BIT, CODE, DATA, IDATA, XDATA vezérlőutasítások. Segítségükkel bármely memóriatartományban létrehozhatunk egy címre mutató szimbólumot. Az alábbiakban láthatjuk a címet szimbolizáló szimbólum létrehozási lehetőségei közül néhányat:

```
P1 DATA 90H
GREEN_LED BIT P1.2
VALUE XDATA 0xFF00
```

A mikrokontroller memóriatérképének van egy különleges részlete, melynek kezeléshez külön vezérlőutasítások állnak rendelkezésünkre. Ez a címtartomány a Speciális funkcióregiszterek tartománya. A tartomány kezeléséhez az sfr, sfr16, és az sbit vezérlőutasítások használhatóak. Az alábbi példában definiáljuk a P1 speciális funkcióregiszter szimbólumot a címének megadásával, valamint a P0_0 bit szimbólumot, úgyszintén a címének megadásával.

```
sfr  P1   = 0x90;
sbit P0_0 = P0^0;
```

Ha jobban megnézzük a mintapélda szintaktikáját, akkor megállapíthatjuk, hogy nem igazán illeszkedik az eddig megszokott assembly szintaktikába. Ez azért van, mert az sfr, sfr16 és az sbit vezérlőutasítások C szintaktikával rendelkeznek, mégpedig azért hogy ez a speciális memóriatartomány egységesen legyen kezelhető mind assembly, mind pedig C nyelven megírt kódból.

2.10.3 Memória allokáció

Az assembly programjainkban természetesen szükségünk lehet változókra, melyeknek helyet kell foglalnunk a memóriában. A változóinkat általában a direkt címezhető memóriatartományban szoktuk létrehozni. Amennyiben csak egyetlen bitnyi információt szeretnénk a változóban tárolni, abban az esetben a bitcímezhető tartományban is foglalhatunk neki helyet.

A DS vezérlőutasítást használhatjuk adott bájtnyi memóriaterület lefoglalására, egyszerűen meg kell adnunk a vezérlőutasítás paraméterében, hogy hány bájtra van szükségünk. A helyfoglalás önmagában azonban nem elegendő, valamilyen módon erre a területre hivatkoznunk is kell, ezért általában egy címkét szoktunk hozzárendelni a lefoglalt terület kezdetéhez. A címkére történő hivatkozással a lefoglalt terület tetszőleges bájtja elérhető lesz. A DS vezérlőutasítást a kódmemória kivételével minden memóriaszegmensben használhatjuk.

A DS vezérlőutasítás használatára mutatnak példát az alábbi kódsorok:

```
TIME: DS 8
GAP: DS (($ + 16) AND 0FFF0H) – $
DS 20
```

Az első mintában 8 bájtnyi helyet foglalunk egy változónak, aminek a TIME nevet adjuk. A második mintában egy kifejezés segítségével adjuk meg a lefoglalandó terület nagyságát. A kifejezésben szereplő $ szimbólum a location counter aktuális pozícióját jelenti. Tehát ha a location counter aktuális pozíciója 0, akkor a lefoglalt terület nagysága 16 bájt lesz, amennyiben a location counter értéke 8, akkor 8 bájtnak fogunk helyet foglalni. A location counterről a későbbiekben lesz még szó részletesebben.
A harmadik sor egyszerűen lefoglal 20 bájtnyi memóriát, azonban nem rendel hozzá cimkét. Ilyenkor a memóriaterületre valamely korábbi címkéhez viszonyított eltolással hivatkozhatunk.
Amennyiben bitcímezhető memóriában szeretnénk helyet foglalni a változónknak, ezt megtehetjük a DBIT vezérlőutasítás használatával. A DBIT vezérlőutasítás paraméterében meg kell adnunk, hogy hány bitnyi információt szeretnénk az adott változóban tárolni.
Az alábbi kódsorok a DBIT használatára mutatnak példát.

```
ON_FLAG:  DBIT 1 ; 1 bitnyi helyet foglalunk
OFF_FLAG: DBIT 1
```

A fenti példában 1-1 bitnyi helyet foglaltunk az ON_FLAG és az OFF_FLAG változóinknak.
Mind a DBIT, mind pedig a DS utasításra igaz, hogy csak annyi helyet foglalhatunk velük, amennyi az adott memóriaterületre elfér, tehát nem tudjuk velük túlcímezni a memóriaterületünket. Túlcímzés esetén az assembler hibaüzenetet generál.
Mindig tartsuk szem előtt, hogy a belső memória viszonylag korlátos méretű, ezért legyünk körültekintőek, ne foglaljunk feleslegesen helyet belőle. Természetesen amihez hely kell, azt nem tudjuk megspórolni, viszont előfordulhat olyan szituáció, ami megoldható egyetlen bájt foglalással is kettő helyett, akkor a kevesebb helyet igénylő megoldást válasszuk. Ha lehet, akkor az assembly rutinok közötti paraméterátadást is regiszterekben végezzük, ne pedig memóriában.
Törekedjünk arra, hogy lehetőség szerint olyan kódot írjunk, amitől a memóriaszervezésünk átlátható lesz. Ez nagyban segíthet abban, hogy ha az elkészült programunkban esetleg évek múlva valamit módosítanunk kell, azt gyorsan el tudjuk végezni, és ne a korábbi programunk bonyolult memóriaszervezésének kinyomozásával töltsük el az időt.
Nézzünk egy ROSSZ példát, hogy hogyan nem célszerű helyet foglalni, és azt használni:

```
myvars:     DS 4
…
            MOV A, myvars
…
            MOV A, myvars + 3
```

Ezzel a megoldással egy nagyon gyengén szervezett helyfoglalást valósítottunk meg, amennyiben a 4 bájtnyi információ nem összefüggő. Egy sokkal jobb megoldás lehet, ha az egyes változóknak külön-külön foglalunk helyet és külön-külön el is nevezzük őket:

```
alma:       DS 1
szilva:     DS 1
barack:     DS 1
korte:      DS 1
…
            MOV A, alma
…
            MOV A, korte
```

Amennyiben a második megoldást választjuk, a memóriánk jól szervezett lesz, és nem lesz érzékeny arra, ha esetleg az egyes változók helyét a memóriában fizikailag megváltoztatjuk. Az első megoldás kifejezetten érzékeny a változók helyének megváltoztatására.

2.10.4 Memória inicializálás

Az alkalmazásainkban nem csak változókra, hanem konstansokra is szükségünk lehet. Konstanson nem az alkalmazás assembly kódjába beleforduló immediate adatokat értjük, hanem valamilyen konstans adatokkal feltöltött memóriaterületet. A konstansokat – csak olvasható jellegükből ez következik is – a kódmemóriában szoktuk eltárolni. Konstansok létrehozására a DB vezérlőutasítás áll rendelkezésünkre. A DB vezérlőutasítás tetszőleges számú paraméterrel rendelkezhet. A paraméterek lényegében a konstans memória tartalmát határozzák meg.
Természetesen a konstansainkat is lehetséges, és célszerű címkével ellátni, hogy a programunkban könnyen tudjunk hivatkozni rájuk.
A DB vezérlőutasítás használatára láthatunk néhány mintát az alábbiakban:

```
REQUEST:    DB 'PRESS ANY KEY TO CONTINUE', 0
TABLE:      DB 0,1,8,'A','0', LOW(TABLE),';'
ZERO:       DB 0, ''''
CASE_TAB:   DB LOW(REQUEST), LOW(TABLE), LOW(ZERO)
```

A konstansok kezelésénél is törekedjünk az átlátható memóriaszervezésre, és a logikus partícionálásra.

2.10.5 Címkezelés

Mint azt a korábbiakból is tapasztalhattuk, az assembly program készítése során a memóriatartalom, és a tartalom memórián belüli elhelyezkedése mindvégig az irányításunk alatt marad. Megismerhettük, hogy hogyan tudunk a rendelkezésre álló memóriában szegmenseket létrehozni, hogyan tudjuk az egyes szegmenseket kiválasztani. Ezek segítségével már egész jól tudjuk befolyásolni a kódrészleteink memórián belüli elhelyezkedését. Nem kaptunk azonban igazi megoldást arra, hogy hogyan lehetséges egy adott szegmensen belüli pozícionálás. A pozícionálás működésének megértéséhez szükségünk van egy eddig csak érintőlegesen említett fogalom – a location counter – megismerésére.
Az assembler minden memóriaszegmenshez létrehoz egy mutatót, mely arra a címre mutat, ahol a memóriában éppen állunk, ez a mutató a location counter. A location counter értékére a programunkban a $ szimbólummal hivatkozhatunk. A location counter értéke automatikusan növekszik minden olyan utasítás hatására, mely az adott memóriaszegmensben történő helyfoglalást befolyásolja. Tehát például a kódmemóriában a location counter értéke automatikusan növekszik egy assembly utasítás hatására, természetesen az utasítás méretének megfelelően. Úgyszintén növekszik a location counter értéke konstansok létrehozása esetén is.
Fontos, hogy a location counter nem globálisan, hanem memóriaszegmensenként értelmezett, tehát minden memóriaszegmensre – a SEGMENT vezérlőutasítással létrehozott általános célú szegmensekre is – saját location counter kerül hozzárendelésre.
A location counter egy lehetséges felhasználási lehetőségére láthattunk példát a DS vezérlőutasításnál. A location counter értéke fordítási időben érdekes, az assembler és a linker valójában majd a tényleges címet fogja az végleges kódba belefordítani.
Sok esetben szükségünk lehet arra, hogy a location counter értékét explicit módon beállítsuk – a memóriában ugorjunk egyet. A location counter értékének beállítására

szolgál az ORG vezérlőutasítás. Az ORG vezérlőutasítás paraméterében meg kell adnunk egy szegmensen belül értelmezhető abszolút címet, erre a címre fog átugrani a location counter.

Az ORG vezérlőutasítás használatára láthatunk alább néhány példát:

```
ORG   100H
ORG   RESTART
```

Az ORG vezérlőutasítást szinte minden esetben használjuk, mégpedig a programunk elején az ugrótábla elkészítésénél.

2.10.6 Linkeléssel összefüggő vezérlőutasítások

A komplexebb alkalmazásokban szinte minden esetben előfordul, hogy a feladatot több assembly forrásfájlban valósítjuk meg. Ebben az esetben valamilyen módon szabályoznunk kell, hogy milyen átjárást engedélyezünk a különböző forrásfájlok között. Gyakran előfordulhat az is, hogy kevernünk kell a C nyelvű és az assembly nyelvű forrásfájlokat, melyek együtt kell hogy működjenek. Ehhez természetesen szükséges többek között az assembler támogatása is, mivel egyrészt láthatóvá kell tennie a C nyelven megírt rutinok számára az assembler által elkészített szimbólumokat, változókat. Másrészt szükséges támogatás ahhoz is, hogy assembly fájlokból elérhetővé váljanak a C nyelvű fájlokban létrehozott változók és függvények. A különböző forrásfájlok összekapcsolását, a közöttük fennálló kapcsolatok feloldását – mint azt a korábbiakban már megtanultuk – a linker feladata elvégezni, azonban a forrásfájlokban valamilyen módon jeleznünk kell a fájlok között engedélyezett kapcsolatokat.

A forrásfájljainkból egyrészt exportálni szeretnénk szimbólumokat, melyeket majd más forrásfájlokban használhatunk. Az exportálásra szolgál a PUBLIC vezérlőutasítás, melynek tetszőleges számú paraméterében vesszővel elválasztva felsoroljuk az exportálandó szimbólumokat.

Az alábbiakban mintát láthatunk a PUBLIC vezérlőutasítás használatára:

```
PUBLIC PUT_CRLF, PUT_STRING, PUT_EOS
PUBLIC ASCBIN, BINASC
PUBLIC GETTOKEN, GETNUMBER
```

A szimbólumok importálását az EXTRN vezérlőutasítással végezhetjük el, melynek használata megegyezik a PUBLIC vezérlőutasításéval. Az EXTRN vezérlőutasítás használatára láthatunk példákat az alábbiakban:

```
EXTRN CODE (PUT_CRLF), DATA (BUFFER)
EXTRN CODE (BINASC, ASCBIN)
EXTRN NUMBER (TABLE_SIZE)
```

2.11 Direktívák

Mint azt már korábban is említettük, a direktívák segítségével az assembler kimeneti fájljainak – az objektum és listing fájlok – tartalmát tudjuk befolyásolni. A direktívák általában nem befolyásolják az assembler által generált kódot, hanem csak az assembler működését. Kivételt képeznek ez alól a feltételes fordítást lehetővé tevő direktívák. A direktívákat egyrészt szerepeltethetjük az assembly forrásfájlunkban, valamint az assembler parancssorában is – kivéve a feltételes fordítás direktíváit, azok csak a forrásfájlban szerepelhetnek. Ne keverjük össze a direktívákat a vezérlőutasításokkal, a direktívák általában globális jellegű beállításokat módosítanak, szemben a vezérlőutasításokkal, melyek az assembler pillanatnyi állapotát befolyásolják.
A direktívákat forrásfájlban mindig $DirektívaNév formában kell szerepeltetni.

Nézzünk néhány direktívát, melyeket szinte minden alkalmazásunkban használni szoktunk:
Az $INCLUDE direktívát használhatjuk a különböző definíciós fájlok assembly forrásunkhoz csatolásához. A direktíva paraméterében meg kell adnunk a csatolni kívánt állomány nevét. Az INCLUDE direktívával kapcsolatosan érdemes tudni, hogy nem csak $INCLUDE formában használható, hanem engedélyezett a C nyelvben megszokott #include forma is.
A fordító – mivel kifejezetten 8051 alapú fejlesztéshez lett felkészítve – alapértelmezés szerint ismeri a standard 8051 regisztereit. Általában ezt a tulajdonságát nem használjuk ki, mivel a mikrokontroller gyártók szinte minden esetben rendelkezésünkre bocsátanak egy, az adott variáns minden regiszterét tartalmazó definíciós fájlt. Ahhoz, hogy ne legyen névütközés, szükséges a standard 8051 regiszterdefiníciók kikapcsolása, ezt a NOMOD51 direktíva segítségével tehetjük meg.
Több esetben szükségünk lehet arra, hogy az elkészült alkalmazásunk bizonyos részeit csak meghatározott feltétel teljesülése esetén fordítsuk bele az objektumfájlba. Képzeljük el például, hogy készítünk egy eszközt, melyet több vásárlónak szállítunk, a vásárlók az eszközünkhöz ugyanolyan funkciójú, azonban eltérő lábkiosztású perifériát kapcsolnak. Szeretnénk, ha az eszköz programjának belső működését nem kellene módosítani, hanem csak a portkiosztást szeretnénk megváltoztatni a lehető legegyszerűbb módon. Ehhez használhatóak a feltételes fordításra szolgáló direktívák.
A feltételes fordításhoz szükségünk van speciális szimbólumokra, melyek értékét megvizsgálva a programunk egyes részeit lefordítjuk, másokat pedig nem. Ezen speciális szimbólumok létrehozására szolgálnak a SET és RESET direktívák. A SET direktívával tetszőleges értéket adhatunk a szimbólumnak, a RESET direktívával pedig 0 értéket rendelhetünk az adott szimbólumhoz. A SET és a RESET direktívák parancssorból is elérhetőek.
A speciális szimbólumok értékvizsgálata az IF, ELSE, ELSEIF, ENDIF direktívák értelemszerű használatával történhet, ezek csak a forrásfájlban szerepelhetnek.

Egy mintapéldát láthatunk az alábbiakban feltételes fordításra.

```
$SET (CUSTOMER = 1)
$IF  (CUSTOMER = 1)
   BTN0      EQU   P1.0
   BTN1      EQU   P1.1
   BTN2      EQU   P1.2
$ELSEIF (CUSTOMER = 2)
   BTN0      EQU   P1.0
   BTN1      EQU   P2.2
   BTN2      EQU   P1.3
$ELSE
   BTN0      EQU   P1.2
   BTN1      EQU   P1.5
   BTN2      EQU   P2.3
$ENDIF
```

Attól függően, hogy melyik vásárlónak szállítjuk az eszközt más-más portkiosztást rendelünk az egyes perifériákhoz.

2.12 Egyéb tudnivalók és szabályok

Kifejezések és operátorok

Az operátorokat az assembly programokon belül az operandusok összehasonlítására, kombinálására, transzformálására használhatjuk. Az operátorok nem assembly utasítások, és nem is fordíthatók gépi kódra. Az operátorok csak fordítási időben ismert értékű operandusok értékeivel tudnak dolgozni.
A kifejezések számok, karaktersorozatok, operátorok és szimbólumok olyan kombinációi, melyeket fordítási időben egyetlen 16 bites számmá lehet átalakítani.
A kifejezésekben szereplő számokat megadhatjuk decimális (100), bináris (1100100b), hexadecimális (064h v. 0x64), valamint oktális(144o v. 144q) formában.
A karaktereket valamint a karaktersorozatokat mindig szimpla idézőjelek (’) között szerepeltetjük.

A kifejezésekben használhatjuk a helyszámláló (location counter) értékét is, erre a $ karakterrel hivatkozhatunk. A helyszámláló mindig az adott memóriaszegmensen belül az aktuális pozíciót tartalmazza. A helyszámláló értéke alapértelmezés szerint soronként növekszik. Lehetőségünk van a helyszámláló értékének a módosítására az ORG vezérlőutasítás segítségével.

A kifejezéseinkben különböző operátorokat használhatunk, ezek lehetnek aritmetikai operátorok (+, -, , /, MOD), bináris operátorok (NOT, SHR, SHL, AND, OR, XOR), összehasonlító operátorok (GTE v. >=, LTE v. <= , NE v. <>, EQ v. =, LT v. < , GT v. >), osztály operátorok (BIT, CODE, DATA, IDATA, XDATA), ill. egyéb operátorok (HIGH, LOW).

Nézzünk néhány példát kifejezésekre:

```
MSK   EQU 0F0H
VALUE EQU MSK - 1

MOV   R3,#(0x20 AND MSK) ;
MOV   R7,#LOW (VALUE + 20H)
MOV   R6,#1 OR (MSK SHL 4)
```

2.13 A jól struktúrált assembly program

Minden szoftverfejlesztő általában arra törekszik, hogy az általa elkészített kód áttekinthető legyen, ezért hoztunk létre szimbólumokat is, és ezért érdemes a programunkat helyesen tördelni.

Egy jól struktúrált assembly program mindenképp az alábbi felépítést követi:

a) Include - ok
b) Szimbólumok importálása
c) Szimbólumok exportálása
d) Adat szegmens allokációk
e) Kódszegmens kiválasztása
f) Ugrótábla
g) Assembly függvények
h) END

Amennyiben ilyen módon tördeljük az alkalmazásunkat, egyértelmű lesz, hogy mit hol kell keresnünk, és hibakeresésnél jelentősen leegyszerűsödik a dolgunk.
A Keil fordító mellé kapunk egy ugyanezt a felépítést követő alkalmazássablont – template.a51 – melyet használhatunk az assembly nyelvű fájljaink elkészítéséhez.

2.14 Egyszerűbb mintapéldák

Nézzünk néhány egyszerű feladatot megvalósító assembly kódrészletet. A mintapéldák legtöbb esetben nem valósítanak meg értelmes funkciót, csak az adott programozási szerkezet bemutatását szolgálják.
Az egyik legalapvetőbb feladat a ciklusszervezés, nézzük hogyan valósíthatjuk ezt meg assemblyben.

```
$NOMOD51
$INCLUDE (SI_EFM8BB3_Defs.inc)
            cseg  AT 0
            ljmp        Main
Main:  mov        A, #1
Loop:
            inc         A
            sjmp        Loop
end
```

A mintaprogram az akkumulátorban található számot növeli egy végtelen ciklusban – tehát semmi kifejezetten értelmeset nem csinál. A ciklust az sjmp Loop ugrással valósítottuk meg.
Nézzünk egy másik példát ciklusszervezésre, melyben már nem végtelen ciklust, hanem fix iterációszámú ciklust valósítunk meg.

```
$NOMOD51
$INCLUDE (SI_EFM8BB3_Defs.inc)

       cseg       AT 0
       sjmp Main

Main:  mov  DPTR, #0x1234
       mov  R1,   0x40
       mov  R2,   #20
Loop:  movx A, @DPTR
       mov  @R1, A
       inc  DPTR
       inc  R1
       djnz R2, Loop
       sjmp Main
```

A program a külső memóriából a belső memóriába másol adatokat. A ciklust a djnz utasítással valósítottuk meg, ami addig iterál, míg az R2 regiszter értéke nulla nem lesz, tehát az R2 regiszter tartalma határozza meg az iterációk, és ezzel a másolandó bájtok számát - esetünkben ez 20.

Nézzünk egy mintapéldát, hogyan lehet kódszegmensben található konstans karakterláncot átmásolni esetünkben egy regiszterbe.

```
$NOMOD51
$INCLUDE (SI_EFM8BB3_Defs.inc)
Constants SEGMENT CODE

       rseg Constants

Str:   DB         "HELLO!"

       cseg AT 0

Main:  mov        DPTR, #Str
       mov        R0,   #6

Loop:  mov        A, #0
       movc       A, @A + DPTR
       mov        R1, A
       inc        DPTR
       djnz       R0, Loop
       sjmp       Main
```

A sztring tartalmát esetünkben az R1 regiszterbe másoljuk. A táblázatok kezelése is hasonlóképpen zajlik.

Nézzük a fenti példát automatikus sztringhossz figyeléssel kiegészítve.

```
Constants SEGMENT CODE
       rseg Constants
Str:   DB   "HELLO!"
StrEnd:
       cseg       AT 0
       sjmp Main
Main:  mov  DPTR, #Str
       mov  R0,   #StrEnd-Str

Loop:  mov        A, #0
       movc       A, @A + DPTR
       mov        R1, A
       inc        DPTR
       djnz       R0, Loop
       sjmp       Main
```

Nézzünk egy mintapéldát alapvető aritmetikai műveletekre.

```
Byte2BCD:
       mov        A, Source
       mov        B, #100
       div        AB
       mov        Digit1, A
       mov        A, B
       mov        B, #10
       div        AB
       mov        Digit2, A
       mov        Digit3, B
       ret
```

A megvalósított függvény egy a Source nevű változóban lévő számot alakítja decimális számmá, az eredményt a Digit1-3 szimbólumok által meghatározott helyre másolja.
Szinte minden mikrokontrolleres alkalmazásban előforduló feladat, hogy megszakításokat kezeljünk, nézzük hogyan szoktuk ezt megvalósítani.

```
cseg  AT 0
       ljmp Main
       org  0x002B
       ljmp Timer2IT
Main:  ...
Timer2IT:
       push PSW
…
ExitTimer2IT:
       pop  PSW
       reti
```

Először készítünk egy ugrótáblát, mely a különböző megszakítások bekövetkezése esetén egy adott pontjára ugrik az alkalmazásunknak. Ezután elkészítjük a megszakítás kezelő függvényeket. A megszakítás kezelő függvényekkel kapcsolatos alapszabály, hogy a függvényen belül használt regisztereket elmenti belépéskor, kilépéskor pedig

visszaállítja, tehát a megszakítás kezelő függvény ugyanolyan állapotban hagyja a regisztereket, mint meghívása előtt voltak. A megszakítás kezelő függvényből mindig reti utasítással térünk vissza.

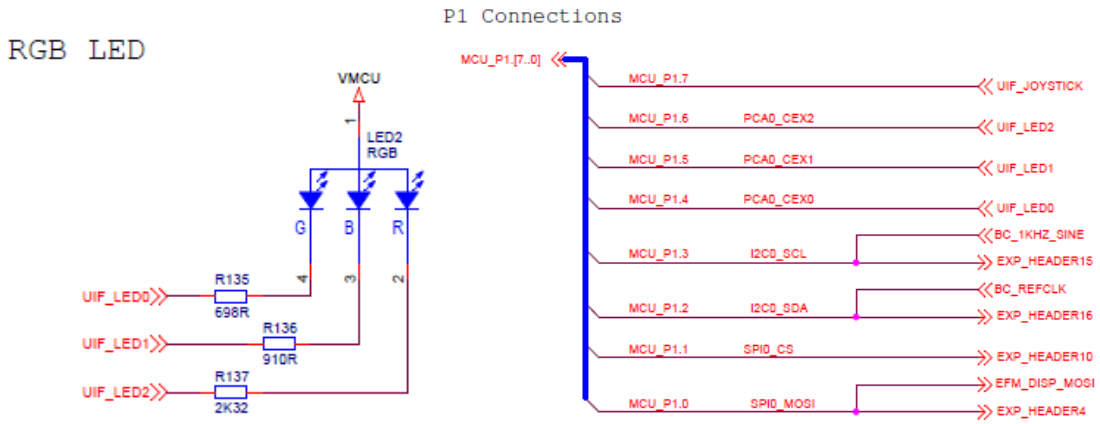
Előfordulhat, hogy szükségünk van szoftveresen megvalósított időzítésre, nézzük hogyan lehet ezt megvalósítani.

```
Wait1Sec:
            mov   R4, #125
Loop1:
            mov   R5, #0x00
Loop2:
            mov   R6, #0x00
            djnz  R6, $
            djnz  R5, Loop2
            djnz  R4, Loop1
            ret
```

Az időzítést egymásba ágyazott ciklusokkal valósítjuk meg, ezzel teljesen leterheljük a processzort. Az ilyen módon történő időzítés természetesen függ a processzor órajel frekvenciájától. Az időzítést elronthatják természetesen a bekövetkező megszakítások is. Alapvetően szoftveres megoldással nem lehet pontosan időzíteni.

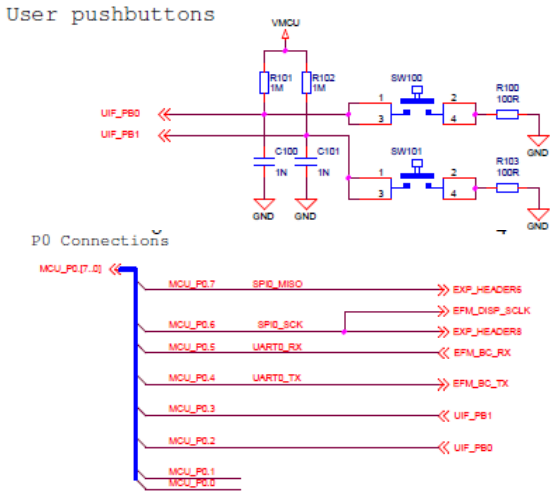
2.15 Összetettebb mintapélda – fejlesztőkártyára

Az egyszerűbb mitapéldákban megismert megoldásokat felhasználva elkészíthetünk egy összetettebb alkalmazást is. Az alábbi mintapéldában a Silicon Labs EFM8 Busy Bee Starter Kit feljesztőkártyára készítünk el egy egyszerű mintapéldát.
A mintapéldában a fejlesztőkártyán található mikrokontrollerhez csatlakoztatott RGB LED-et vezéreljük úgy, hogy a LED a fejlesztőkártyán lévő gomb megnyomásának hatására váltson színt. A gombnyomást megszakításos módon fogjuk kezelni.
A feladat megoldásához szükséges, hogy a fejlesztőkártya kapcsolási rajzán megkeressük a LED-ek és a nyomógombok bekötését.



9. ábra LED-ek bekötése az EFM8 Busy Bee fejlesztőkártyán

Mint az a kapcsolási rajz részletből láthatjuk, a LED-ek a mikrokontroller P1.4, P1.5, P1.6 lábaira vannak kötve, és null aktív vezérlésűek.
A kapcsolási rajzon meg kell keresnünk a nyomógombok bekötését is.



10. ábra LED-ek bekötése az EFM8 Busy Bee fejlesztőkártyán

Láthatjuk, hogy a nyomógombok a mikrokontroller P0.2 és a P0.3 lábaira csatlakoznak, és rendelkeznek hardveres pergésmentesítéssel – tehát eltekinthetünk a szoftveres pergésmentesítéstől.
A fentiek ismeretében nekikezdhetünk az alkalmazás elkészítéséhez.

```
$nomod51
$include (SI_EFM8BB3_Defs.inc)

;data allocations
dseg at 0x40
  LedVal: DS 1

;jump table
  cseg at 0
  ljmp main
  org 0x0003
  ljmp ext0it
```

Első lépésben a mikrokontroller regiszterkészletének definíciós állományát csatoljuk a forrásfájlunkhoz, majd elvégezzük az adatmemóriában szükséges allokációkat. Az adatmemóriában helyet foglalunk egy egybájtos LedVal változónak, melyben az RGB LED-ek állapotát fogjuk tárolni.
Ezek után elkészítjük az ugrótáblát, melyben szokás szerint a 0-s címre helyezzük a main címkére ugrást. A külső megszakítás kezeléséhez a mikrokontroller adatlapja alapján a 3-as címre helyezzük az ext0it megszakításkezelőre történő ugrást.
Az ugrótábla elkészítése után nekiláthatunk az alkalmazásunk tényleges funkcionalitását megvalósítani.

```
myprog segment code ;declare own segment
rseg myprog ;select own segment
main:
   CLR IE_EA ; disable all interrupts
   MOV WDTCN,#0DEh ; disable software watchdog timer
   MOV WDTCN,#0ADh


   ; init variables
   mov LedVal, #0x10 ; 4th bit set to 1

   ; setup ports
   mov XBR2, #0x40 ; enable crossbar
   MOV P1MDOUT, #0x70 ; set P1 port mode bits to output
   MOV P1, #0x70 ; turn off the LEDs
   MOV P0MDIN, #0x04 ; set p0.2 bit to input

   ; setup external interrupts
   mov IT01CF, #0x02 ;set P0.2 to generate external interrupt
   setb TCON_IT0 ; edge triggered interrupt
   setb IE_EX0 ; enable external interrupt
   SETB IE_EA ; enable interrupts

   jmp $
```

A saját kódszegmens létrehozása és kiválasztása után elkészítjuk az alkalmazásunk központi részét. A központi részben főleg hardver incializációs lépések, illetve egy végtelen ciklus lesz megtalálható, a tényleges funckionalitás a külső megszakítás kezelő rutinban kerül majd megvalósításra. Az inicalizációs lépések a watchdog kikapcsolásával kezdődnek – a mikrokontroller adatlapja alapján az interruptok tiltása után a WDTCN regiszterbe két speciális értéket kell beírni közvetlen egymás után.
Következő lépésben a korábban létrehozott LedVal változót inicializáljuk, mégpedig úgy, hogy a negyedig bitjében legyen egyes, a többi bit pedig legyen nulla. A LedVal változót fogjuk használni a LED-ek kijelzendő értékének tárolására – a változóban azon a biten lesz mindig egyes, amelyik LED-nek világítani kell.
Következő inicializációs lépésünk a portok beállítása. Ehhez először engedélyzenünk kell a crossbart – ezzel engedélyezzük, hogy a mikrokontroller belső regisztereinek változása a mikrokontroller fizikai lábain is megjelenjen. Be kell állítanunk, hogy a mikrokontroller azon lábai, melyekhez a LED-ek vannak csatlakoztatva, kimenetek legyenek. Ehhez a P1MDOUT regiszterben a 4,5,6 biteket egyesbe kell billentenünk. Miután ezt megtettük, inicializáljuk a lábak állapotát – a szükséges 3 bitet egyes állapotra billentjük, tehát a LED-eket lekapcsoljuk.
Következő lépésben a P0 port 2-es lábát a P0MDIN regiszter második bitjének egyesbe állításával beállítjuk, hogy a mikrokontroller láb legyen digitális bemenet.
A port beállítások után a külső megszakítás tulajdonságait kell beállítanunk, egyrészt az IT01CF regiszter értékét állítjuk be – ezzel rendeljük össze a kiválasztott portlábat a külső megszakítással, másrészt a TCON_IT0 bit beállításával kiválasztjuk, hogy a élváltásra generálódjon megszakítás, majd a külső megszakítást engedélyezzük az IE_EX0 flagen keresztül, végezetül a globális interrupt engedélyező flag-et (IE_EA) is beállítjuk.

Mint azt már korábban említettük, az alkalmazás tényleges funckionalitását a megszakítás kezelő rutinban valósítjuk meg.

```
ext0it:
      push psw
      push acc
      using 0 ; set register bank
      push ar0

      mov a, ledval; load the running light variable to the acc
      rl a ;shift it to the left
      mov ledval, a ; load it back
      anl a, #0x80 ;check if we shifted out of the three leds range
      jz displayonleds
      mov ledval, #0x10
displayonleds:
      anl ledVal, #0x70 ; mask out unneeded bits from the ledVal variable
      mov a, ledval ; move the ledVal value to the acc
      cpl a ; invert the ledval
      anl a, #0x70 ; mask out the unneeded bits
      mov r0, a ; save the calculated value to the r0
      mov a, p1 ; save the port state to the acc
      anl a, #0x8f ; mask out the 3 led bits
      orl a, r0 ; write the ledval bits from the r0 to the acc

      mov p1, a ; write back the changed value to the port
endtext0it:
      using 0 ; set register bank
      pop ar0
      pop acc
      pop psw
      reti
```

A megszakítás kezelő rutinoknál alapvető követelmény, hogy a mikrokontroller regisztereit a megszakításkezelő rutin végén olyan állapotba kell beállítani, amilyen állapotba voltak a rutin meghívásakor. Ennek érdekében a megszakításkezelő rutinokat a rutin törzsében használt regiszterek mentésével kezdjük, és azok visszaállításával fejezzük be a megszakításkezelő rutinok megírását. A megoldásunkban a mikrokontroller push és a pop utasításait használjuk mentésre és visszaállításra, melyek a stackre ill. stackről másolnak adatokat. Mivel a 8051 mikrokontroller utasításkészletében a push és a pop csak adott memóriarekeszek tartalmát tudja a stackre feltenni, ill. onnan visszaállítani, és mivel az általános célú regiszterek a bank beállítástól függő címen érhetőek el, ezért az általános célú regisztereket a using és push, ill. using és pop utasításokkal tudjuk csak kezelni.

A megszakításkezelő rutin működésének kulcsa, hogy a LedVal változóban található értéket balra léptetjük, és amennyiben az érték kilépett a változó legfelső bitjére, akkor visszatöltjük a változó értékét úgy, hogy az értékes egyes a negyedik bitpozícióban legyen. A léptetés után már más dolgunk nincs is, mint a LedVal változó értékét kiírni a P1 portra, azonban arra figyelnünk kell, hogy a porton csak a 4,5,6 lábak állapotát módosíthatjuk – hiszen ezekhez csatlakoznak a LED-jeink, az össze többit akár a

programunk más része vezérelhetné. A displayonleds címkével jelölt részben elsőként kimaszkoljuk a LedVal változónk összes olyan bitjét amiben nem szabadna 1-esnek lennie, tehát csak a 4,5,6 biteket hagyjuk meg. ezek után a LedVal változó értékét átmásoljuk az akkumulátorba – hiszen, ha a mikrokontroller utasításkészletét megnézzük, csak az akkumulátoron tudunk komplemens képzés műveletet elvégezni, direkt memória címen nem. Ezek után elvégezzük a komplemens képzést – erre azért van szükségünk, mert a kapcsolási rajzon láttuk, hogy a LED-ek null aktív vezérlésűek – majd kimaszkoljuk azokat a biteket, melyek nem a LED-ekhez tartoznak. A kapott eredményt átmásoljuk az r0 regiszterbe – amit átmeneti tárolóként használunk. Következő lépésben a port aktuális értékét betöltjük az akkumulátorba, és kitöröljük a 3 LED-nek megfelelő bitet, majd beírjuk az átmeneti tárolóban tárolt LED értéket az akkumulátor 3 bitjére. Természetesen itt nem három bitet írunk, hanem mind a nyolcat, viszont úgy alakítottuk ki egyrészt az akkumulátor tartalmát, hogy a felülírni kívánt bitekben 0 legyen, az r0 regiszterben pedig csak azokon a biteken lehet 1, amelyek a LED-ekhez kapcsolódnak, így a logikai vagy művelettel csak ezt a 3 bitet fogjuk módosítani. A portra kiírandó érték előállt az akkumulátorban, már csak ki kell írnunk azt a portra.
Végezetül a használt regiszterek állapotát visszaállítjuk, és a szubrutin végén reti-vel visszatérünk.

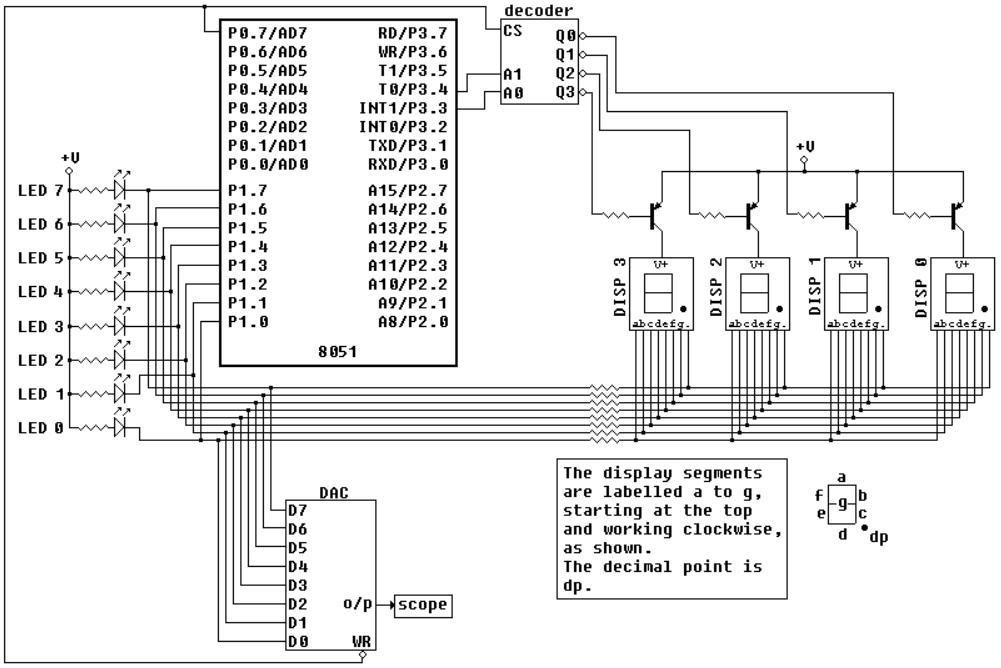
2.16 Összetettebb mintapélda - szimulátorra

Az egyszerűbb mitapéldákban megismert megoldásokat felhasználva elkészíthetünk egy összetettebb alkalmazást is. Az alkalmazást az edsim51 szimulátorra készítjük el, melyet a http://www.edsim51.com/ webcímről szabadon le lehet tölteni, és el lehet vele kezdeni 8051 mikrokontrollerre assembly nyelven fejleszteni. A szimulátor nem csak önmagában egy mikrokontrollert tartalmaz, hanem a mikrokontrollerhez csatlakoztatott számos perifériát is, így elég szerteágazó feladatokat lehet megvalósítani segítségével.

Az alábbi mintapéldában a 8051 mikrokontrollerhez csatlakoztatott hétszegmenses kijelző soron jelenítünk meg 4 jegyű hexadecimális számlálót.A feladat megoldásához első lépésként meg kell vizsgálnunk a kapcsolási rajzot, melyen meg kell keresnünk azokat a jeleket, melyek segítségével a hétszegmenses kijelzők vezérelhetőek.

Az alábbi ábrán mutatjuk be a kapcsolási rajz kapcsolódó részét. Láthatjuk, hogy a négy hétszegmenses kijelző közös adatvezetékeket használ. A megfelelő adatvezetéket logikai 0 szintre húzva fognak világítani a hétszegmenses kijelzőben található LED-ek. Azt is láthatjuk, hogy csak annak a hétszegmenses kijelzőnek a LED-jeit tudjuk vezérelni, amelyre az ábra felső részén látható dekóder tápfeszültséget juttat egy tranzisztor-kapcsolón keresztül. Tehát ahhoz, hogy a hétszegmenses kijelzőinkre bármilyen értéket tudjunk kiírni egyrészt megfelelően be kell állítanunk a dekóder vezérlőjeleit – kiválasztva ezzel az aktív kijelzőt –, másrészt az adatvonalakat kell a megjelenítendő karakter képének megfelelően vezérelnünk.

A kapcsolási rajz alapján a mikrokontroller P1 portjára kapcsolódnak a hétszegmenses kijelzők adatvezetékei, a P0 port 7-es lábára a dekóder CS jele, a P3 port 3-4 lábára pedig a dekóder címjelei – ezeket kell majd az alkalmazásunkban meghajtanunk.



11. ábra Az edsim51 szimulátor kapcsolási rajza (részlet)

A hétszegmenses kijelzőhöz tartozó adatvonalak vezérléséhez célszerű készíteni egy táblázatot, mely az egyes karakterekhez tartalmazza a hozzá tartozó vezérlőjel kombinációt. A táblázat a hétszegmenses kijelző LED pozícióinak ismeretében az alábbiak szerint tölthető ki.

IN	dp	g	f	e	d	c	b	a	OUT
0	1	1	0	0	0	0	0	0	C0
1	1	1	1	1	1	0	0	1	F9
2	1	0	1	0	0	1	0	0	A4
3	1	0	1	1	0	0	0	0	B0
4	1	0	0	1	1	0	0	1	99
5	1	0	0	1	0	0	1	0	92
6	1	0	0	0	0	0	1	0	82
7	1	1	1	1	1	0	0	0	F8
8	1	0	0	0	0	0	0	0	80
9	1	0	0	1	0	0	0	0	90
A	1	0	0	0	1	0	0	0	88
B	1	0	0	0	0	0	1	1	83
C	1	1	0	0	0	1	1	0	C6
D	1	0	1	0	0	0	0	1	A1
E	1	0	0	0	0	1	1	0	86
F	1	0	0	0	1	1	1	0	8E

A fentiek ismeretében nekikezdhetünk az alkalmazásunk elkészítéséhez. Első lépésben szimbólumokat definiálunk az egyes vezérlőjelekhez, illetve az alkalmazásban használt változókhoz.

```
; Szimbólum definíciók
   SEG7DATA        EQU   P1
   SEG7SEL         EQU   P3
   SEG7EN          EQU P0.7
   DigitCounter    EQU 0x20
   ValueCounterH   EQU 0x40
   ValueCounterL   EQU 0x41
; Ugrótábla
   ORG  0
   LJMP Init
; Konstansok
DecTable:
   DB 0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0x88,
       0x83, 0xC6, 0xA1, 0x86, 0x8E
```

Az alkalmazásban két globális változót fogunk használni, az egyik a 16 bites számláló értékét fogja tárolni (ValueCounter), a másik pedig a dekóder címvezetékeire kiírandó értéket fogja tárolni (DigitCounter). A korábbiakban említettük, hogy a Keil fordító BS és DBIT vezérlőutasításaival lehet helyet foglalni a memóriában változóinknak, viszont az edsim51 szimulátorba beépített fordító nem támogatja ezeket a vezérlőutasításokat, így a memóriacímre hivatkozással fogjuk ezeket kezelni. Vegyük észre, hogy a DigitCounter változó címe a mikrokontroller memóriájában a bitcímezhető tartományára esik, így a címző vezetékek értékét meghatározó változónk bit műveletekkel is manipulálható lesz, melyet ki is fogunk használni a későbbiekben.

Következő lépésben elkészítjük az ugrótáblát, melyben egyetlen bejegyzés lesz megtalálható – a 0 címről elugrunk az Init címkével jelölt címre. Ezek után a kódmemóriába betöltjük a Db vezérlőutasítás segítségével a hétszegmenses kijelző vezérléséhez szükséges táblázatot (DecTable).

A vezérlő táblázat elkészítése után el kell készítenünk az alkalmazásunk központi részét, mely egyrészt inicializálásából másrészt egy végtelen ciklusból áll.

```
Init:
   ;Érték számláló inicializálása
   MOV       ValueCounterL, #0
   MOV       ValueCounterH, #0
   ;Digit számláló inicializálása
   MOV       DigitCounter, #0

MainLoop:
   ;Érték számláló inkrementálása
   MOV       A, ValueCounterL
   ADD       A, #1
   MOV       ValueCounterL, A
   MOV       A, ValueCounterH
   ADDC      A, #0
   MOV       ValueCounterH, A
   ; Digitek kiírása
   MOV       R1, #4
DisplayLoop:
```

```
   CALL      GetNumber
   MOV       R7, DigitCounter
   CALL      WriteDigit
   CALL      Delay
   INC       DigitCounter
   DJNZ      R1, DisplayLoop
   LJMP      MainLoop
```

Az Init címkével jelzett inicializáló rész kezdőértéket ad a globális változóinknak. Az alkalmazás fő ciklusa a MainLoop. Ebben a ciklusban egyrészt növeljük a 16 bites számlálónk értékét, majd pedig kijelezzük a számláló értékét a kijelzőkön a DisplayLoop ciklussal. Láthatjuk, hogy a DisplayLoop ciklus egy fix iterációszámú ciklus, mely a 4 kijelzőt megcímezve egyesével kiírja a számláló egyes digitjeit. A DisplayLoop magjában első lépésben meghívjuk a GetNumber szubrutint, mely a ValueCounter és a DigitCounter értéke alapján az R6 regiszterben visszaadja a 16 bites számláló megfelelő helyiértékű számjegyét. Ezek után a WriteDigit szubrutin meghívásával kiírjuk az R6 regiszterben található értéket az R7 regiszterben megadott sorszámú kijelzőre. Ahhoz, hogy az R7 regiszterben a megfelelő érték legyen, be kell azt állítanunk a WriteDigit szubrutin meghívása előtt. A WriteDigit szubrutin meghívása – azaz a számjegy kiírása – után várnunk kell egy rövid ideig (különben nem látnánk az értéket a kijelzőn). A várakozást a Delay szubrutin meghívásával valósítjuk meg.

A főciklusban meghívott WriteDigit szubrutin implementációja az alábbi.

```
WriteDigit:
   CLR       SEG7EN
   MOV       A, R7
   ANL       A, #3
   RL        A
   RL        A
   RL        A
   MOV       R5, A
   MOV       A, SEG7SEL
   ANL       A, #0xE7
   ORL       A, R5
   MOV       SEG7SEL, A
   CALL      ConvertToDisplay
   MOV       SEG7DATA, R6
   SETB      SEG7EN
   RET
ConvertToDisplay:
   MOV       DPTR, #DecTable
   MOV       A, R6
   MOVC      A, @A+DPTR
   MOV       R6, A
   RET
```

Első lépésben a kijelzőket címző dekódert letiltjuk – erre azért van szükségünk, mert a címzés, vagy az adatvezetékek módosítása azonnal megjelenne a kijelzőn és ez helytelen kijelzést okozna. A következő lépésben az R7 regiszter tartalmát – kijelző címzés – átmásoljuk az akkumulátorba, és kimaszkoljuk a felső hat bitjét hiszen összesen két

címvezetékünk van, tehát nekünk csak az alsó két bit értékével kell dolgoznunk. Ezek után – mivel a címzővezetékek a mikrokontroller P3.3 és P3.4 lábára vannak kötve – az akkumulátor tartalmát balra léptetjük háromszor, hogy az értékes bitek a megfelelő pozícióra kerüljenek. Ezek után kiírjuk a P3 portra a címkiválasztó értéket – gondoskodva arról, hogy a port egyéb lábainak értéke ne változzék. A címző jelek előállítása után az adatvezetékek értékét állítjuk elő – ehhez meghívjuk a ConvertToDisplay szubrutint, mely az R6 regiszterben várja a kiírandó számjegyet és ugyanebben a regiszterben adja vissza az adatvezetékre kiírandó karakterkódot. A kiírandó karakterkód előállítása után az adatvonalakat beállítjuk és a címző dekódert engedélyezzük - megjelenítve ezzel a kívánt karaktert.
A ConvertToDisplay szubrutin a kódmemóriában eltárolt táblázatot megcímezve előállítja az R6 regiszterben megadott számjegy karakterkódját. A kódmemóriából adatot betölteni a MOVC utasítással lehetséges, a 16 bites kódmemóriacímet az akkumulátor és a 16 bites DPTR regiszter értéke fogja meghatározni. A DPTR regiszter értékét a táblázat kezdőcímére állítjuk, az akkumulátort pedig a táblázaton belüli indexeléshez használjuk. A kívánt karakterkód az R6 regiszterben áll elő.

```
GetNumber:
   JNB DigitCounter.1, LowerByte
   MOV  A, ValueCounterH
   JNB DigitCounter.0, RetVal
   SWAP A
   JMP  RetVal
LowerByte:
   MOV  A, ValueCounterL
   JNB DigitCounter.0, RetVal
   SWAP A
RetVal:
   ANL  A, #0x0F
   MOV R6, A
   RET
```

A kiválasztott kijelzőre kiírandó hexadecimális számjegy értékét a GetNumber szubrutin állítja elő, melyet a DisplayLoop ciklusból hívunk meg. A szubrutin a DigitCounter értékének megfelelően a 16 bites ValueCounter egy hexadecimális számjegyét írja be az R6 regiszterbe – ez a szubrutin visszatérési értéke. A szubrutin megvalósításához kihasználjuk, hogy a DigitCounter változót bitcímezhető területre helyeztük, így a DigitCounter két bitjének állapotai szerint egy a C nyelvben megszokott if-else jellegű konstrukciót valósítunk meg. Az első feltételes ugrást a digit számláló magas helyiértékű bitjének vizsgálatával végezzük. Ha a magasabbik helyiértékű bit 0 értékű (JNB), akkor a 16 bites érték számlálónk alsó bájtjával kell foglalkoznunk – elugrunk a LowerByte címkére. Ha a magasabbik helyiértékű bit 1, akkor nem ugrunk, hanem betöltjük az akkumulátorba a ValueCounter magas helyiértékű bájtját. Következő lépésben megvizsgáljuk a DigitCounter alacsony helyiértékű bitjét. Ha a bit értéke 1 - az akkumulátorban található szám alsó és felső 4 digitjét felcseréljük – hiszen ekkor a ValueCounter kiválasztott bájtjának felső 4 bitje tartalmazza a kiírandó karaktert. Ugyanezeket a műveleteket elvégezzük a LowerByte ágban is, majd a szubrutin végén a RetVal címkével jelölt helyen maszkolással gondoskodunk arról, hogy az R6 regiszterben csak az alsó 4 biten legyen érték.

```
Delay:
  MOV  R0, #0xFF
  DJNZ R0, $
  RET
```

A kijelző ciklusban a várakozáshoz a Delay szubrutint hívtuk meg, ez egy fix iterációszámú ciklust valósít meg.

2.17 Mintapélda - fejlesztőkártyára

A korábbiakban megismerkedhettünk a mikrokontroller alapú rendszerekre történő alkalmazásfejlesztés elméleti lehetőségeivel elsősorban assembly nyelven, illetve betekintést nyerhettünk a fordítás folyamatába is. A következőkben egy egyszerű mintapéldán keresztül megnézzük, hogy a gyakorlatban hogyan zajlik a fordítási folyamat, milyen fájlok keletkeznek a fordítás folyamán, hogyan kell értelmezni a keletkező fájlok tartalmát.

Általában egy mikrokontroller családdal történő ismerkedés során elsők között egy olyan egyszerű feladatot szoktunk megvalósítani, mellyel a leggyakrabban használt mikrokontroller perifériát – az általános ki-bemeneti portokat – próbáljuk ki, megismerkedünk annak működésével, tulajdonságaival.
Az általános ki-bementi portok kezeléséhez a következő egyszerű feladatot szeretnénk megvalósítani: adott egy 8051 alapú mikrokontroller, melyhez egy nyomógomb és egy LED csatlakozik. A nyomógomb a P3.7 lábra, a LED pedig a P1.6 lábra van csatlakoztatva. A nyomógomb megnyomásakor logikai 0-ra húzza le a lábon lévő jelszintet, a LED pedig akkor világít, amikor a lábra logikai 1-et küldünk. Azt szeretnénk megvalósítani, hogy a LED addig világítson, amíg a nyomógombot nyomjuk.

Készítsük el a feladatot megvalósító programot először assembly nyelven:

```
$NOMOD51
#include <C8051F040.inc>

LED         BIT   P1.6
BUTTON BIT  P3.7

            cseg at 0
            sjmp Startup

Startup:    mov   SFRPAGE, #CONFIG_PAGE
            mov   WDTCN, #0xDE
            mov   WTDCN, #0xAD
            mov   XBR2, #0xC0 // Enable crossbar
            mov   P1MDOUT, #01000000b // P1.6 as output
Main:       mov   c, BUTTON
            cpl   c
            mov   LED, c
            sjmp  Main
end
```

A feladat, mint a fenti megoldásból láthatjuk roppant egyszerűen megvalósítható, nézzük, hogy hogyan működik a megvalósított assembly program.

A programunk az elsődleges direktívák meghívásával kezdődik. A NOMOD51 utasítja a fordítót, hogy az alapértelmezett módon definiált 8051 regiszterneveket ne használja. Erre azért van szükségünk, mert egy saját – processzor specifikus – definíciós fájlt szeretnénk használni. Az include direktíva segítségével ezt a processzor specifikus definíciós fájlt csatoljuk hozzá az alkalmazásunkhoz.

Az elsődleges direktívák meghívása után a következnek a globális szimbólum definíciók. Két szimbólumot hozunk létre, az egyik a LED, a másik a BUTTON, melyekhez a feladatleírásban szereplő Port lábakat rendeljük hozzá.

A globális szimbólumdefiníciók után elkészítjük az ugrótáblát, mely esetünkben egyetlen elemet tartalmaz. Az ugrótáblában legalább azoknak a speciális címeknek kell szerepelniük, melyekre a programunknak szüksége van a megszakítási rutinok meghívásához. Esetünkben a kódszegmens 0-s címére – mely a Reset megszakításhoz tartozik – elhelyezünk egy ugró utasítást, mellyel a Startup címkére ugrunk.

A Startup címkével kezdődő utasítássorozat engedélyezi a Priority Crossbar Decodert, letiltja a watchdogot és beállítja, hogy a mikrokontroller azon lába, melyhez a LED csatlakozik, viselkedjen kimenetként – az összes többi láb alapértelmezés szerint bemenet lesz.

A Main címkével kezdődő utasítássorozat végzi a programunk tényleges funkcióját, beolvassa a nyomógomb állapotát a carry flagbe, aztán negálja a carry flaget – mivel a nyomógomb és a LED inverz működésű – aztán beállítja a LED állapotát, majd visszaugrik a Main cimkére.

Készítsük el ugyanezt a feladatot C nyelven, és nézzük végig a fordítás folyamatát.

```
#include <c8051f040.h>

sbit LED    = P1^6;
sbit BUTTON = P3^7;

void InitDevice( )
{
       SFRPAGE   = CONFIG_PAGE;
       XBR2      = 0xC0; // Enable crossbar
       WDTCN     = 0xDE;
       WDTCN     = 0xAD;
       P1MDOUT   = 0x40; // Set Port1 bit 6 as output
}
void main (void)
{
       InitDevice();
       while (1)
```

```
            LED = !BUTTON; // Write BUTTON State to LED
}
```

A megvalósított alkalmazás működése és felépítése nagyban hasonlít az assemblyben megírt alkalmazáshoz. Első lépésben becsatoljuk a processzor specifikus definíciós fájlt, majd pedig létrehozunk két sbit típusú változót. Az sbit típusú változókkal a bitcímezhető területen létrehozott változók bitjeire hivatkozhatunk. Esetünkben arra használjuk ezeket, hogy a mikrokontroller meghatározott portjainak egyes bitjeit elnevezzük.
A 8051 mikrokontrollerre történő C nyelvű alkalmazásfejlesztés sajátosságaival itt most nem foglalkozunk részletesen. A mikrokontrollerek C nyelven történő programozását majd a későbbiekben részletesen bemutatjuk a 32 bites Cortex mikrokontrollerek programozásával foglalkozó fejezetben.

Az eszköz bekapcsolásakor a main függvényt fogja meghívni, ami első lépésben meghívja az InitDevice függvényt.
Az InitDevice függvényben engedélyezzük a Priority Crossbar Decodert, letiltjuk a watchdogot, majd pedig beállítjuk, hogy a mikrokontroller azon lába, melyre a LED van csatlakoztatva, legyen kimenet.
Ezek után a main függvényben készítünk egy végtelen ciklust, melynek magjában a nyomógomb aktuális állapotát negáljuk, és erre az értékre beállítjuk a LED állapotát.

Fordítsuk le az elkészített alkalmazásunkat, és vizsgáljuk meg a keletkező fájlok tartalmát. A fordítás során a következő fájloknak kell keletkezniük:

- Preprocesszor kimeneti fájl - *.i
- Compiler listing vagy napló fájl - *.lst
- Compiler kimeneti fájl - *.obj
- Linker log fájl - *.m51 v. *.map
- Bináris kódfájl
- Intel Hex fájl - *.hex

A preprocesszor feladata, többek között, hogy a külső hivatkozásokat feloldja. Ez a feloldás lényegében annyit jelent, hogy a külső hivatkozások tartalmát a forrásfájlunkba másolja. Nézzük, hogy mit láthatunk az alkalmazásunk fordítása során keletkezett preprocesszor kimeneti fájlban:

```
#line 1 "ea1.c" /0#line 1 "C:\KEIL\C51\INC\CYGNAL\C8051F040.H" /0
sfr P0       = 0x80;
sfr SP       = 0x81;
sfr DPL      = 0x82;
/*.......a header fájl további tartalma ....... */
#line 1 "ea1.c" /0
sbit LED    = P1^6;
sbit BUTTON = P3^7;
void InitDevice( )
{
SFRPAGE = 0x0F;
XBR2    = 0xC0;
/*.......a forrás fájl további tartalma ....... */
```

Láthatjuk, hogy szerepel, milyen külső hivatkozások kerültek beszerkesztésre, majd a külső hivatkozások tartalma, ill. az eredeti forrásunk találhatók még meg az i fájlban.

Az előfeldolgozási lépést követően a compiler elvégzi a fordítást, és egyrészt elkészíti az objektumfájlt, ami bináris formátumú, valamint készít egy szöveges listing vagy más néven naplófájlt, melyben az objektumfájl készítéséhez felhasznált információk találhatóak meg.

A listing fájl elején minden esetben szerepel a listing fájl létrehozásának időpontja, valamint a compiler futtatásához használt parancssor.

```
C51 COMPILER V8.08   EA1
03/19/2008 16:45:20 PAGE 1

C51 COMPILER V8.08, COMPILATION OF MODULE EA1
OBJECT MODULE PLACED IN .\out\ea1.obj
COMPILER INVOKED BY: C:\Keil\C51\BIN\C51.EXE ea1.c OPTIMIZE(0,SIZE)
BROWSE NOAREGS DEBUG OBJECTEXTEND CODE SYMBOLS PRINT
                    -(.\out\ea1.lst) PREPRINT(.\out\ea1.i)
OBJECT(.\out\ea1.obj)
```

Ezt követi a forráskódunk, melyet le szerettünk volna fordítani.

```
line level    source

   1          #include <c8051f040.h>
   2
   3          sbit LED        = P1^6;
   4          sbit BUTTON     = P3^7;
   5
   6          void InitDevice( )
   7          {
   8   1              SFRPAGE   = CONFIG_PAGE;
   9   1              XBR2      = 0xC0; // Enable crossbar
  10   1              WDTCN     = 0xDE;
  11   1              WDTCN     = 0xAD;
  12   1              P1MDOUT   = 0x40; // Set Port1 bit 6 as output
  13   1      }
  14          void main (void)
  15          {
  16   1              InitDevice();
  17   1              while (1)
  18   1              LED = !BUTTON; // Write BUTTON State to LED
  19   1      }
```

A következő részben a forráskódunk assembly nyelvre lefordított változatát láthatjuk.

```
ASSEMBLY LISTING OF GENERATED OBJECT CODE
```

```
             ; FUNCTION InitDevice (BEGIN)
                                                ; SOURCE LINE # 6
                                                ; SOURCE LINE # 7
                                                ; SOURCE LINE # 8
0000 75840F              MOV     SFRPAGE,#0FH
                                                ; SOURCE LINE # 9
0003 75E3C0              MOV     XBR2,#0C0H
                                                ; SOURCE LINE # 10
0006 75FFDE              MOV     WDTCN,#0DEH
                                                ; SOURCE LINE # 11
0009 75FFAD              MOV     WDTCN,#0ADH
                                                ; SOURCE LINE # 12
000C 75A540              MOV     P1MDOUT,#040H
                                                ; SOURCE LINE # 13
000F           ?C0001:
000F 22                  RET
             ; FUNCTION InitDevice (END)

             ; FUNCTION main (BEGIN)
                                                ; SOURCE LINE # 14
                                                ; SOURCE LINE # 15
                                                ; SOURCE LINE # 16
0000 120000      R       LCALL   InitDevice
0003           ?C0002:
                                                ; SOURCE LINE # 17
                                                ; SOURCE LINE # 18
0003 A2B7                MOV     C,BUTTON
0005 B3                  CPL     C
0006 9296                MOV     LED,C
0008 80F9                SJMP    ?C0002
000A           ?C0003:
                                                ; SOURCE LINE # 19
000A           ?C0004:
000A 22                  RET
             ; FUNCTION main (END)
```

Láthatjuk, hogy a lefordított sorok előtt különböző hexadecimális számok szerepelnek. Az első szám azt jelenti, hogy az adott assembly kódsor hol helyezkedik el a programrészlet (pl. függvény) kezdetétől számítva. Ez az ún. relatív cím minden önálló programblokk esetében 0-tól indul újra, mivel a függvény tényleges helye majd csak egy későbbi fázisban kerül meghatározásra, az egyes függvények sorrendje még változhat. A második számsor pedig az assembly kódsor gépi kódra lefordított értéke. Ha előveszünk egy utasítás-kódtáblázatot, könnyen rájöhetünk, hogy ezek a gépi kódok is egy viszonylag egyszerű rendszer szerint épülnek fel: a programrészlet elején vegyük észre, hogy a minden sor elején szereplő 75h érték a MOV utasítások gépi kódja, ezt követi az első (cél-) operandus címe (pl. SFRPAGE = 84h, XBR2 = E3h, WDTCN = FFh stb. – ellenőrizzük fentieket a 8051F040 speciális funkcióregisztereit tartalmazó táblázatban), végül az az utasításba beépülő konstans, amit tölteni kívánunk az előbbi regiszterekbe. Hasonló elvek szerint az SJMP ?C0002 utasítás gépi kódja (80F9h) azon egyszerű filozófia szerint épük, hogy a rövid ugrás (short jump) gépi kódja 80h, a végrehajtandó ugrás „hossza” pedig -7 (=F9h), azaz pontosan 7 bájtnyi utasításkódot kívánunk visszafelé – ezért negatív - átugrani az ugró utasításunk segítségével. Senki ne értse félre:

nem célunk ezen gépi kódok emberi munkával történő előállítása, erre valók a fordítóprogramok. De ne is nézzünk érthetetlen hieroglifákként ezekre a kódokra, hiszen bármikor egyszerűen kitalálhatók – megértésük a hibák keresésekor nagy segítségünkre lehet !

Ha összehasonlítjuk a lefordított kódot az általunk korábban elkészített asssembly kóddal, észrevehetjük, hogy szinte teljes egészében megegyeznek. A különbség csak annyi, hogy míg mi emberi fejjel tudtunk értelmezhető cimkeneveket készíteni, addig a fordító automatikusan generált cimkéi nem annyira beszédesek.

Ezek után a szimbólumok listáját láthatjuk a fájlban.

```
NAME                            CLASS    MSPACE   TYPE     OFFSET   SIZE
====                            =====    ======   ====     ======   ====

P1 . . . . . .                  SFR       DATA    U_CHAR    0090H   1
WDTCN. . . . . .                SFR       DATA    U_CHAR    00FFH   1
P3 . . . . . .                  SFR       DATA    U_CHAR    00B0H   1
ADC0CN . . . . . .              SFR       DATA    U_CHAR    00E8H   1
SFRPAGE. . . . . .              SFR       DATA    U_CHAR    0084H   1
ADC2CN . . . . . .              SFR       DATA    U_CHAR    00E8H   1
P1MDOUT. . . . . .              SFR       DATA    U_CHAR    00A5H   1
IE . . . . . .                  SFR       DATA    U_CHAR    00A8H   1
CAN0CN . . . . . .              SFR       DATA    U_CHAR    00F8H   1
PCA0CN . . . . . .              SFR       DATA    U_CHAR    00D8H   1
IP . . . . . .                  SFR       DATA    U_CHAR    00B8H   1
XBR2 . . . . . .                SFR       DATA    U_CHAR    00E3H   1
SMB0CN . . . . . .              SFR       DATA    U_CHAR    00C0H   1
main . . . . . .                PUBLIC    CODE    PROC      0000H   -----
CPT0CN . . . . . .              SFR       DATA    U_CHAR    0088H   1
CPT1CN . . . . . .              SFR       DATA    U_CHAR    0088H   1
CPT2CN . . . . . .              SFR       DATA    U_CHAR    0088H   1
SPI0CN . . . . . .              SFR       DATA    U_CHAR    00F8H   1
TCON . . . . . .                SFR       DATA    U_CHAR    0088H   1
TMR2CN . . . . . .              SFR       DATA    U_CHAR    00C8H   1
TMR3CN . . . . . .              SFR       DATA    U_CHAR    00C8H   1
TMR4CN . . . . . .              SFR       DATA    U_CHAR    00C8H   1
LED. . . . . .                  ABSBIT    -----   BIT       0096H   1
BUTTON . . . . . .              ABSBIT    -----   BIT       00B7H   1
SCON0. . . . . .                SFR       DATA    U_CHAR    0098H   1
SCON1. . . . . .                SFR       DATA    U_CHAR    0098H   1
InitDevice . . . . . .          PUBLIC    CODE    PROC      0000H   -----
CAN0STA. . . . . .              SFR       DATA    U_CHAR    00C0H   1
PSW. . . . . .                  SFR       DATA    U_CHAR    00D0H   1
```

Ha a fájl tartalmát jobban megnézzük, láthatjuk hogy a függvényeinkhez nem került hozzárendelésre érvényes kezdőcím – ez majd a linker feladata lesz.

Végezetül a modullal kapcsolatos memóriafoglalási információkat láthatjuk a fájlban.

```
MODULE INFORMATION:   STATIC OVERLAYABLE
   CODE SIZE        =     27    ----
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
```

```
   PDATA SIZE       =   ----    ----
   DATA SIZE        =   ----    ----
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
```

Láthatjuk, hogy a lefordított kódunk összesen 27 bájtnyi helyet foglal a kódmemóriából. Mivel egyetlen forrásállományunk volt csak, így a linker dolga igen egyszerű, csak el kell helyeznie a memóriában a függvényeinket. Nézzük, hogy hogyan végezte ezt el a linker a map fájl tartalma alapján.

A map fájlban először a linker meghívásához használt parancssort, valamint a használt memóriamodell beállítást láthatjuk.

```
BL51 BANKED LINKER/LOCATER V6.05, INVOKED BY:
C:\KEIL\C51\BIN\BL51.EXE .\out\ea1.obj TO .\out\ea PRINT
(.\out\ea.m51) IXREF RAMSIZE (256) CODE (0X0000-0XFFFF) XDATA (
>> 0X0000-0X0FFF) DATA (0X40)

MEMORY MODEL: SMALL
```

Ezek után a feldolgozott objektum és könyvtár fájlok listáját láthatjuk.

```
INPUT MODULES INCLUDED:
  .\out\ea1.obj (EA1)
  C:\KEIL\C51\LIB\C51S.LIB (?C_STARTUP)
```

Láthatjuk, hogy egyrészt feldolgozásra került az előző lépésben előállított objektum fájl, másrészt feldolgozásra került a memória modellhez illeszkedő könyvtárfájl.

Ezek után láthatjuk a memóriatérképet, ami többek között megmutatja, hogy melyik függvényünk hová került fizikailag a memóriában.

```
LINK MAP OF MODULE:  .\out\ea (EA1)

        TYPE    BASE      LENGTH    RELOCATION   SEGMENT NAME
        -----------------------------------------------------
        * * * * * * *   D A T A   M E M O R Y   * * * * * * *
        REG     0000H     0008H     ABSOLUTE     "REG BANK 0"
        IDATA   0008H     0001H     UNIT         ?STACK
        * * * * * * *   C O D E   M E M O R Y   * * * * * * *
        CODE    0000H     0003H     ABSOLUTE
        CODE    0003H     0010H     UNIT         ?PR?INITDEVICE?EA1
        CODE    0013H     000CH     UNIT         ?C_C51STARTUP
        CODE    001FH     000BH     UNIT         ?PR?MAIN?EA1
```

Figyeljük meg, hogy a linker folyamatosan tölti a memóriát, láthatjuk, hogy a 0-s címtől kezdődően osztja ki az egyes függvényeinkhez használt szegmenseket. A kódmemória első 3 bájtja lesz az ugrótábla, aminek kötelezően itt kell szerepelnie, a függvényeinket tartalmazó szegmensek pedig az ugrótáblát követően kerülnek elhelyezésre.
A memóriatérkép után a hívási fát láthatjuk a map fájlban, ami megmutatja, hogy az egyes szegmenseink között milyen hívott-hívó viszony áll fenn.

```
OVERLAY MAP OF MODULE:   .\out\ea (EA1)

SEGMENT
  +--> CALLED SEGMENT
---------------------
?C_C51STARTUP
  +--> ?PR?MAIN?EA1

?PR?MAIN?EA1
  +--> ?PR?INITDEVICE?EA1
```

Láthatjuk, hogy a memóriamodellhez illeszkedő startup szegmensben található függvény meghívja a mi programunkban található main függvény szegmensét, ez pedig hívja az InitDevice függvényhez tartozó szegmenst.

A hívási fa után a szimbólumok listája található meg.

```
VALUE           TYPE          NAME
  ----------------------------------

  -------         MODULE        EA1
  C:0000H         SYMBOL        _ICE_DUMMY_
  D:0090H         PUBLIC        P1
  D:00FFH         PUBLIC        WDTCN
  D:00B0H         PUBLIC        P3
  D:00E8H         PUBLIC        ADC0CN
  D:0084H         PUBLIC        SFRPAGE
  C:001FH         PUBLIC        main
  B:0090H.6       PUBLIC        LED
  B:00B0H.7       PUBLIC        BUTTON
  C:0003H         PUBLIC        InitDevice
  D:00C0H         PUBLIC        CAN0STA
  D:00D0H         PUBLIC        PSW
  -------         PROC          INITDEVICE
  C:0003H         LINE#         6
  C:0003H         LINE#         7
  C:0003H         LINE#         8
  C:0006H         LINE#         9
  C:0009H         LINE#         10
  C:000CH         LINE#         11
  C:000FH         LINE#         12
  C:0012H         LINE#         13
  -------         ENDPROC       INITDEVICE
  -------         PROC          MAIN
  C:001FH         LINE#         14
  C:001FH         LINE#         15
  C:001FH         LINE#         16
  C:0022H         LINE#         17
  C:0022H         LINE#         18
  C:0029H         LINE#         19
  -------         ENDPROC       MAIN
  -------         ENDMOD        EA1
```

A listából láthatjuk, hogy az egyes szimbólumok mely memóriatípusra és azon belül milyen memóriacímekre mutatnak.

Végezetül a map fájl tartalmazza a modulok közötti kereszthivatkozások listáját, és egy összesítést a memóriahasználatról.

```
INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------
NAME . . . . USAGE    MODULE NAMES
---------------------------------

?C_START . . CODE;    ** L51 GENERATED **  ?C_STARTUP
?C_STARTUP . CODE;    ?C_STARTUP  EA1
INITDEVICE . CODE;    EA1
MAIN . . . . CODE;    EA1
Program Size: data=9.0 xdata=0 code=42
```

Láthatjuk, hogy összesen 9 bájtnyi adatmemóriát használtunk – a regiszterekhez és a stackhez, valamint összesen 42 bájt kódmemóriát a függvényeinkhez.

A linker map fájljából láthattuk a különböző szegmensek elhelyezkedését a memóriában, a compiler listing fájljából pedig láthattuk, hogy az egyes szegmensekbe milyen tartalom kerül. Ezen információk alapján akár kézzel is elő lehetne állítani a bináris kódfájlt, valamint az ennek ASCII reprezentációját jelentő Intel Hex fájlt. Szerencsére természetsen ezeket nem kézzel kell elkészítenünk, mivel a linker elkészíti őket. Nézzük meg azonban, hogy ténylegesen azok az információk szerepelnek-e a Hex fájlban, amit mi beleírnánk a compiler listing és a linker map alapján.
A keletkező Intel Hex fájl tartalma a következők szerint alakul:

```
:1000030075840F75E3C075FFDE75FFAD75A54022DE
:0B001F00120003A2B7B3929680F922F2
:03000000020013E8
:0C001300787FE4F6D8FD75810702001F1D
:00000001FF
```

Első lépésben ellenőrizzük, hogy tényleg 42 bájtnyi kódmemóriát próbálunk-e feltölteni a hex fájlban. Ehhez össze kell adnunk az adathosszt jelentő bájtok értékeit. 0x10+0xB+0x3+0xC = 0x2A, ami decimálisan tényleg 42 bájtnyi információt jelent.
Második lépésben válasszuk ki az első sort és nézzük meg a kezdőcímet jelentő bájtok értékeit: 0x0003. A map fájlban található memóriatérkép alapján ez az InitDevice függvényünk kezdőcíme. Nézzük, hogy milyen gépi kódnak kell ezen címtől kezdődően szerepelnie a memóriában. A compiler listing alapján 0x75840F bájtokkal kell kezdődnie, és 0x22 bájttal kell végződnie. Ez valóban teljesül.

Már csak azt kell ellenőriznünk, hogy az ellenőrzőösszeg valóban 0xDE értékű-e. Ehhez össze kell adnunk az értékes bájtokat 8 biten, és végül ki kell vonnunk a kapott összeget 256-ból. A bájtok értékeit összegezve 0x22 értéket kapunk, melyet 0x100-ból kivonva tényleg 0xDE értékű kell hogy legyen az ellenőrzőösszegünk.

Az Intel Hex fájl további sorainak tartalmát hasonlóképpen ellenőrizhetjük vagy állíthatjuk elő.

Mikrokontroller alapú rendszerek

Elektronikus jegyzet

2. fejezet (2. rész)

Készítette: Dr. Tevesz Gábor c. egyetemi tanár
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
Fax: 463-2871 (adm.)
Mail: tevesz@aut.bme.hu

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki alapszak (BSc)
III. évfolyam, 5. félév
Beágyazott és irányító rendszerek specializáció hallgatói

COPYRIGHT

TARTALOMJEGYZÉK

2. HARDVERKÖZELI PROGRAMOK FEJLESZTÉSE

2.18 Számábrázolási formátumok

Ebben a fejezetben áttekintjük azokat a legfontosabb számábrázolási formátumokat, melyeket a mikrokontrollerekben hardver vagy szoftver műveletvégző egységeink használnak. A hardverközeli programozás során gyakran térünk át a különböző számábrázolási formátumok között (sokszor egy egyszerű kasztolással), lényeges tehát pontosan ismernünk, hogy a kezelt adategységekben mely bitcsoportok milyen célt szolgálnak ill. befolyásolnak – hasonlóan fontos ismernünk a számábrázolási formátumok pontosságát, és korlátait.

A mikrokontrollerekben a műveleteket alapvetően a számok kétféle nagy csoportján kívánjuk elvégezni: sok esetben elegendőek a számításokhoz az egyszerűbb, egész típusú (**integer**) számok, de sokszor szükségünk van a nem egész (tört) fogalom kezelésére is – ezen kívül a számok nagysága vagy a pontosság megkívánja a valós (**real**) számok alkalmazását. Szintén a törtszámok alkalmazása szükséges, ha matematikai függvényeket alkalmazunk (négyzetgyök, szinusz, stb.) programrendszerünk algoritmusaiban. Az egész számokkal való műveletvégzés általában gyorsabban elvégezhető, az egész aritmetikát valamennyi mikrokontrollernek támogatnia kell, ezért sokszor használjuk őket még törtszámok kifejezésére is úgy, hogy a számba "beleképzelünk" valahova egy kettedes (bináris) pontot, és ennek mozgását a műveletvégzések során állandóan egyszerű eltolási műveletekkel korrigáljuk (ún. **fixpontos számábrázolás**). Ezzel a módszerrel – bizonyos korlátok között – át lehet hidalni a felbontás és a pontosság kérdését, de nem segít azon az alapvető gondon, hogy a számok nagysága igen tág határok között változhat, és az ábrázolás pontosságát leginkább a mindenkori tényleges számnagysághoz kellene igazítanunk (tehát az abszolút „1/100" pontosság semmitmondó a szám nagyságának ismerete nélkül). Ezt a matematikában a számok normál alakjával hidaljuk át, ennek a megfelelője a számítástechnikában az ún. **lebegőpontos számábrázolás**.

Az első mikroprocesszorok – melyek alapvetően 8 bites mikroprocesszorok voltak – többnyire még hardver szorzó- és osztó egységgel sem rendelkeztek, ezeknél bizony még problémát jelentettek akár a nagyobb egész számokkal való alapműveletek is. Műveleteink operandusai akkor sem voltak "kisebbek", mind ma: a 8 vagy 16 bites ábrázolásból adódó -128..+127 ill. -32768..+32767 számtartomány legtöbbször nem elegendő számítási műveletsoraink elvégzéséhez. Aritmetikai szubrutincsomagok végezték a nagyobb (32 bites) számok összeadását, kivonását, nem is beszélve a szorzási vagy a még időigényesebb osztási műveletekről (ld. 1.7 fejezet).

A 16/32 bites mikroprocesszorok megjelenése nagyot változtatott ezen az állapoton, különösen, ha figyelembe vesszük azt, hogy a szóhossz növekedése mellett megjelentek a processzorok utasításkészletében az (egész, de már előjeles) szorzó-osztó műveletek (ld. 68000 vagy 80x86). Ezekkel az egész típusú műveletvégzési problémáink zöme meg is oldódott.

A lebegőpontos számokkal végzendő műveletek számítása még alapműveletek esetén is igen komoly időigénnyel jár, nem beszélve a függvények többnyire polinomiális sorba fejtett alakjainak közelítő számításairól. Ezekre a feladatokra speciális hardver áramköröket fejlesztettek ki, melyek a processzorok "társaként" az ilyen típusú műveletvégzésekre specializálódtak. Ezek a **társprocesszorok** (ún. coprocesszorok) a lebegőpontos aritmetikai műveletvégzés mellett egész típusú műveleteket is el tudtak végezni – 32, sőt 64 biten – áthidalandó a már említett hossz- és szorzási/osztási problémákat. Ez a tulajdonságuk azonban a korszerűbb 16/32 bites mikroprocesszorok megjelenésével egyre inkább háttérbe szorult.

A törtszámok kezelése egy "külön gondolatvilágot" alkot a számítógépek világában. Alkalmazásuk során ugyanis olyan problémák lép(het)nek fel, melyek az egész típusú műveletvégzés során eszünkbe sem jutnak. Gondoljuk meg ugyanis, hogy

- a törtszámokat gondolatban a matematika valós számfogalmával azonosítjuk. Mivel azonban bármely számítógépben egy törtszámot csak **véges számú biten** tudunk ábrázolni, az ábrázolásnak van egy véges pontossága és számábrázolási nagysága – és itt mindkettő véges voltán van a hangsúly, ami a matematikai valós szám fogalmában nincs benne.
- jól megszokott **véges törtrészű** tizedes törtek a 2-es számrendszerben való ábrázolásból adódóan nem biztos, hogy véges bitszámon ábrázolhatók kettedes törtként is (nem ugyanazok lesznek a „kerek" törtszámaink, mint pl. az 1/10).
- a számítógépek azon legnagyobb erénye, hogy a műveletvégzésük igen gyors az ember műveletvégzési sebességéhez képest, arra ösztönözte felhasználóit, hogy olyan számításokat végeztessenek vele, amely műveletek millióit vagy milliárdjait jelenti – „kivárható" véges idő alatt. Igen ám, de amikor minden egyes műveletvégzésünk elvileg sem pontos (a számábrázolás pontatlansága miatt), milliónyi műveletet tartalmazó és egymásra épülő műveletsorok végén a végeredmények pontatlansága már akkora lehet, ami korábban az emberek számára elképzelhetetlen volt – mivel több emberélet is kevés lett volna az adott műveletsor elvégzésére. Megjelent a **numerikus labilitás** fogalma (egy bizonyíthatóan konvergens matematikai műveletsor kimenete azért kezd oszcillálni, mert számábrázolásunk és műveletvégzésünk pontossága nem elegendő).

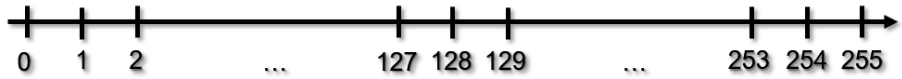
Az elmondottak miatt a lebegőpontos számok kezelését (mantissza és karakterisztika mérete, ábrázolási formája, pontossági és kerekítési előírások, stb.) egyre inkább "uniformizálták", míg 1985-ben az amerikai IEEE (Institut of Electrical and Electronics Engineers) **IEEE Std. 754-1985** néven szabványosította azokat (másik ismeretes elnevezése IEC 60559:1989, Binary floating-point arithmetic for microprocessor systems). Általában aki ma lebegőpontos számokról beszél – legyen az PC vagy mikrokontroller –, ehhez a szabványhoz tartja magát (eltekintve olyan kivételektől, mint amiket a jelfeldolgozó processzoroknál tanultunk). A szabvány kiterjesztése **IEEE Std. 754-2008** néven jelent meg, ez megtartotta az eredeti formátumokat, de hozzáigazította az ábrázolási méreteket a mai buszméretekhez (128 bit a korábbi 80/96 bit helyett).

2.18.1 Az egész típus

Az egész számok mindig vagy előjel nélküli, vagy előjeles (kettes komplemens kódú) számábrázolással értendők, az ábrázolható számok a következők:

Bits	Name	Range	Decimal digits
8	byte, octet	*Signed:* −128 to +127 *Unsigned:* 0 to +255	3 3
16	halfword, word	*Signed:* −32,768 to +32,767 *Unsigned:* 0 to +65,535	5 5
32	word, doubleword, longword	*Signed:* −2,147,483,648 to +2,147,483,647 *Unsigned:* 0 to +4,294,967,295	10 10
64	doubleword, longword, long long, quad, quadword	*Signed:* −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 *Unsigned:* 0 to +18,446,744,073,709,551,615	19 20
128		*Signed:* −170,141,183,460,469,231,731,687,303,715,884,105,728 to +170,141,183,460,469,231,731,687,303,715,884,105,727 *Unsigned:* 0 to +340,282,366,920,938,463,463,374,607,431,768,211,455	39 39

Már az egész számoknál is fontos kiemelni, hogy ábrázolásuk minden esetben (a rendszert tervező mérnök döntésétől függő) adott bitszámon történik, ami egy nem elfelejthető jelenséget hoz maga után. Vegyük az egyszerűség kedvéért az egy bájton (8 biten) ábrázolható pozitív egész számokat ! Mindenki tudja fejből, hogy 8 biten 256 különböző számértéket tudunk ábrázolni, 0-val kezdve ezek a 0..255 számok. Egy szokásos számegyenesen ábrázolva ezt:



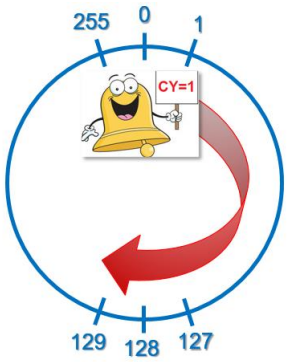
Ábrázolható pozitív egész számok 1 bájton (8 biten)

A számegyenes azonban mindkét irányban végtelen, megszokott matematikai ismereteink „becsapnak” a tartomány határain, hiszen a 255-ös értékhez hozzáadva 1-et mindenki azonnal rávágná, hogy az eredmény 256. Pedig nem így van, a 256 nem ábrázolható 8 biten, az összeadás után ez eredmény „túlcsordul” és az előbb említett

$$256_{10} = 1\ 0000\ 0000_2$$

eredményből csak az alsó 8 bit marad meg, azaz 0 lesz az eredmény és a mikrokontroller státuszregiszterében a CY=1 flag hívja fel a figyelmünket arra, hogy az eredmény túlcsordult, az eredményregiszterben kapott számérték „nem helyes”.

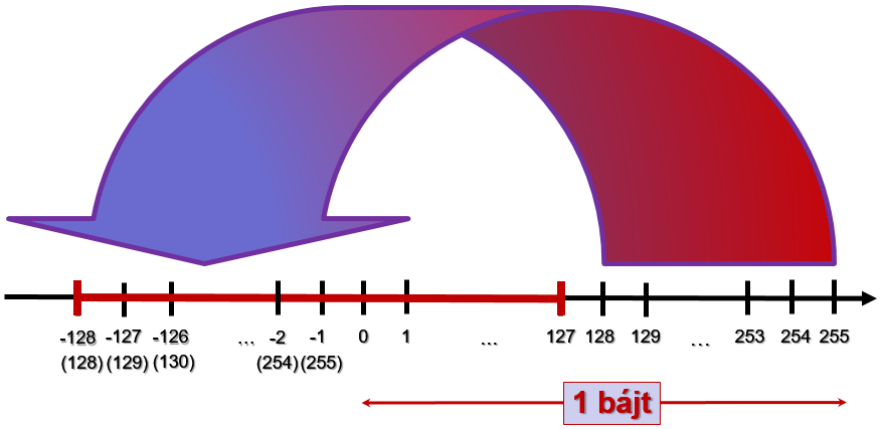
Az adott bitszámon ábrázolható számértékek fentiek miatt sokkal szemléletesebben ábrázolhatók a számegyenes helyett egy kör kerületén („feltekertük a számegyenest”), mivel a körvonal véges volta miatt a 255-ös végérték a 0-s kezdőérték szomszédja lesz, így senki sem fog csodálkozni a következő (természetesen matematikailag helytelen) kifejezéseken:




```
255 + 1 = 0    (és CY = 1)
0 - 1 = 255    (és CY = 1)
```

A számábrázolásunk véges voltát a lehetséges értékek egy körvonalon való elhelyezése jobban mutatja

Még bonyolultabb a helyzet az előjeles egész számok körében. Ismert mindenki számára, hogy ilyenkor a rendelkezésre álló tartomány felében ábrázoljuk a pozitív számokat (8 biten a 0..127 tartományban), a tartomány másik fele a negatív számoké (-1..-128). Az aritmetikát támogató kettes komplemens számábrázolás az eredeti tartomány felső felét „transzformája” a számegyenesen a negatív számok tartományába az alábbi ábra szerint:

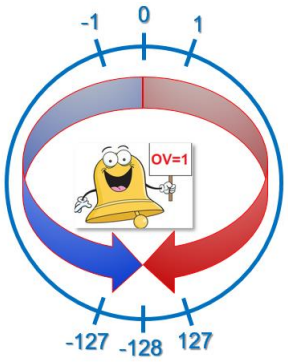


Ábrázolható előjeles egész számok 1 bájton (8 biten)

Itt is igaz viszont az adott bitszámon ábrázolható számok véges volta, csak a túlcsordulás máshova kerül: a 0 ↔ 255 (kettes komplemens értelmezésben a 0 ↔ -1) átmenet teljesen természetes és üzemszerű, semmilyen túlcsordulás esete sem forog fent. Hol van ilyenkor az „átfordulás”, a túlcsordulás ? Jól látható, hogy a 127 ↔ 128-as értékek között, hiszen kettes komplemens kódban a 128 már a -128-as értéket reprezentálja !

Akkor most itt kellene jeleznie a CY státuszbitnek ? Jól legyezzük meg: mikrokontrollerünknek fogalma sincs arról, hogy mi a számokat előjeles vagy előjel nélküli értelmezéssel fogjuk kiolvasni ! A 128-as számérték pontosan ugyanazt a bitkombinációt jelenti, mint kettes komplemens értelmezésben a -128, tehát mikrokontrollerünk nem tud máshogyan reagálni, mint hogy a 0 ↔ 255 átmenetnél állítja továbbra is a CY bitet 1-be. Előjeles számoknál viszont ez egy „téves” jelzés, hiszen a

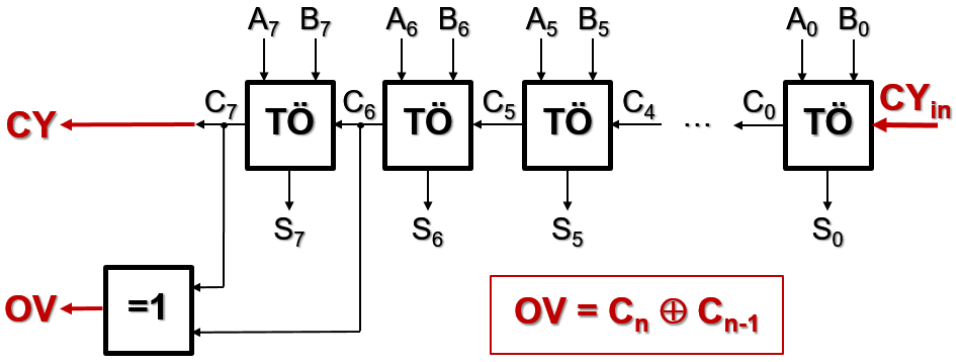
túlcsordulás valójában a 127 ↔ -128 átmenetnél következik be. Ezt egy másik státuszbit, az OV (Overflow) jelzi számunkra a mikrokontroller státuszában:




```
127 + 1 = -128 (és OV = 1)
-128 - 1 = 127 (és OV = 1)
```

A számábrázolásunk véges voltát a lehetséges értékek egy körvonalon való elhelyezése ebben az esetben is jobban mutatja

A két státuszbit előállítása rendkívül egyszerű (a Digitális technikában tanult egy bites teljes összeadó kapcsolást ismertnek feltételezve:



Ábrázolható előjeles egész számok 1 bájton (8 biten)

Miért rendkívül fontos erről tudnunk ? Mert a jelzések figyelembevétele a programozó döntése kell legyen (milyen feltétel szerint ágazik el a program) aszerint, hogy hogyan értelmezi az adott regiszterben található biteket: előjeles vagy előjel nélküli számokként. A mikrokontroller mit sem tud erről, ha rossz feltételt vizsgálunk, a lehető legnehezebben felderíthető hibát építjük be a programunkba: az esetek többségében az algoritmus jól fog működni (mikor nincs túlcsordulás), néha pedig nem (amikor van).

A megjegyzendő alapszabály tehát:

Előjel nélküli számokkal végzett aritmetikai műveleteknél a CY bit, előjeles számokkal végzett aritmetikai műveleteknél az OV bit jelzi a túlcsordulást.

Az egész számok kezelésénél még meg kell említsük a számábrázolási méretek közötti áttérést (pl. 8 bit helyett 16 biten ábrázoljuk a számokat – ilyen pl. a típuskonverzióknál fordul elő, vagy perifériák kezelésénél, ha a 10 vagy 12 bites előjeles A/D értéket 16 biten akarjuk ábrázolni, mivel 8 biten nem fér el.)

A számábrázolási hossz növelése esetén ún. **előjel-kiterjesztést** kell alkalmaznunk. Ne tévesszük ugyanis szem elől, hogy kettes komplemens számábrázolásunk bitszámhoz kötött fogalom ($2^n - A$), azaz az átalakítás során igazodnunk kell ahhoz, hogy a számot hány biten ábrázoljuk.

Pozitív számok esetén az áttérés egyszerű: megszoktuk, hogy egy szám elé tetszőleges számú 0-t írva a szám értéke nem változik.

Negatív számok esetében gondoljuk végig a következőt: $N > n$ bitesre szeretnénk átírni A számunkat. Ekkor igazak a következő összefüggések:

$$\mathbf{x = 2^n - |A| \quad ill. \quad y = 2^N - |A|}$$

azaz y értékét szeretném meghatározni x-ből.

⇐ N bit ⇒			
11111111	**11111111**	**=**	**2^N-1**
	11111111	**=**	**2^n-1**
11111111	**00000000**	**=**	**$(2^N-1)-(2^n-1)$**
	⇐ n bit ⇒		

Mivel

$$\mathbf{y = 2^N - |A| = 2^N - 2^n + 2^n - |A| = (2^N - 1) - (2^n - 1) + 2^n - |A|}$$

$$\mathbf{y = (2^N - 1) - (2^n - 1) + x}$$

Azaz: negatív szám esetén a változatlan szám elejét egyesekkel kell feltöltenünk.
Mivel egy szám pozitív, legfelső (előjel-) bitjének értéke mindig 0, negatív szám esetén ez 1, a fenti szabályt általánosíthatjuk:

Előjel-kiterjesztés esetén a szám elejéről hiányzó bitek a szám saját előjelbitjének értékével töltendők fel.

A számábrázolási hossz csökkentése esetén fentiek alapján egyszerűen megfogalmazható a fenti szabály fordítottja:

A szóhossz csökkentése esetén a szám elejéről egyszerűen elhagyhatjuk (levághatjuk) a legfelső biteket mindaddig, amíg
a) a levágott bitek azonosak (csak 0-k vagy 1-ek) ÉS
b) megegyeznek a lerövidült szám elején szereplő bittel

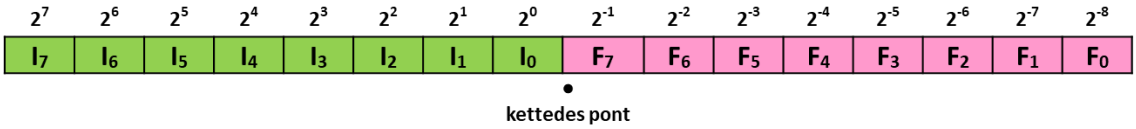
Fentiekre néhány példa:

	8 biten	12 biten	16 biten
+5 =	0000 0101	0000 0000 0101	0000 0000 0000 0101
-5 =	1111 1011	1111 1111 1011	1111 1111 1111 1011
+20 =	0001 0100	0000 0001 0100	0000 0000 0001 0100
-20 =	1110 1100	1111 1110 1100	1111 1111 1110 1100

2.18.2 A valós típus

A valós számokkal való műveletvégzéshez fix- vagy lebegőpontos számábrázolási formátumot használunk.

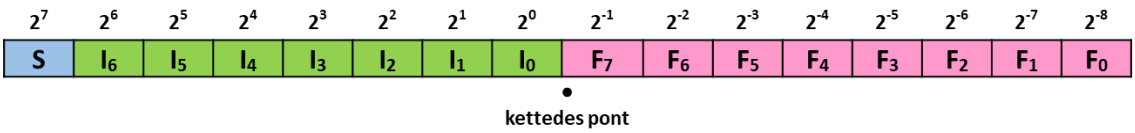
Fixpontos számábrázolás során az ábrázolt számba „beleképzelünk” egy kettedes pontot, és onnan kezdve a számmal ugyanúgy végezzük a műveleteket, mint az egész számokkal. A helyiértékeken balról jobbra a súlyozó faktorok folyamatos feleződés nem csak az egész részre igaz (16 – 8 – 4 – 2 – 1), hanem folytatódik a törtrészben is (1/2 – 1/4 – 1/8 stb.) Egy 16 biten ábrázolt, 8 bites egészrész és 8 bites törtrészt tartalmazó fixpontos törtszámot úgy képzeljünk el, mintha a szám helyett az ő 256-szorosát ábrázoltuk volna, de értelmezéskor a számot gondolatban elosztjuk 256-tal a kettedes pont odaképzelésével.



Az így ábrázolt számokon végzett műveletek minden további nélkül helyesek lesznek: ha két 16 bites számnak (A és B) tekintjük a fenti számokat, akkor az

$$A/256 + B/256 = (A+B)/256$$

értelmezés szerinti művelet is helyes eredményt ad. Ugyanez igaz előjeles számok esetére is:



Ebben az esetben az előbbi művelet:

$$(2^N\text{-}A)/256 + (2^N\text{-}B)/256 = (2^{N+1}\text{-}A\text{-}B)/256$$

(Ne zavarjon meg senkit a 2^N tag 2^{N+1}-re változása, az csak a kettes komplemens számábrázolás üzemszerű túlcsordulásának eredménye.)

Példa 1:

8 biten ábrázolt 4 egész + 4 törtbitet tartalmazó szám esetén

$$00100100 = 0010.0100_2 = 2.25_{10}$$

A megoldandó feladat:

$$5 - 2.25 = 2.75$$

A szokásos 2-es komplemens képzéssel

$$-2.25 = 11011100$$

Végezzük el a műveletet, mintha egész számokkal dolgoznánk:

```
  01010000    (+5.00)
+ 11011100    (-2.25)
  00101100
```

Ez pedig a konvencióink szerint

$$0010.1100_2 = 2 + ½ + ¼ = 2.75$$

Példa 2:

Oldjuk meg a következő műveletet

$$100 - 3.14 = 96.86$$

16 biten ábrázolt 8 egész + 8 törtbitet tartalmazó fixpontos számokkal.

Először is alakítsuk át a decimális számokat bináris megfelelőjükké !

Minden lépésben 2-vel osztva

100	0	LSB
50	0	
25	1	
12	0	
6	0	
3	1	
1	1	MSB
0		

Minden lépésben 2-vel szorozva

0.14	0.28	0	MSB
0.28	0.56	0	
0.56	1.12	1	
0.12	0.24	0	
0.24	0.48	0	
0.48	0.96	0	
0.96	1.92	1	
0.92	1.84	1	LSB

$100_{10} = 1100100.00000000_2$ $3.14_{10} = 00000011.00100011_2$

Jól látható a számábrázolás véges pontossága, mivel

$$00000011.00100011_2 = 3.13671875_{10}$$

A szokásos 2-es komplemens képzéssel

$$-3.14_{10} = 11111100.11011101$$

(Szokás szerint jobbról balra haladva az első 1-ig mindent változatlanul hagyunk, onnan kezdve minden az ellenkezőjére változik. Ebben a műveletben a kettedes pontot figyelmen kívül hagyjuk, az a fixpontos számot ábrázoló regiszterben sincs ott.)

Végezzük el a műveletet, mintha egész számokkal dolgoznánk:

```
  01100100.00000000    (+100.00)
+ 11111100.11011101    (  -3.14)
  01100000.11011101    ( +96.86)
```

Az egészrész egyszerűen ellenőrizhető:

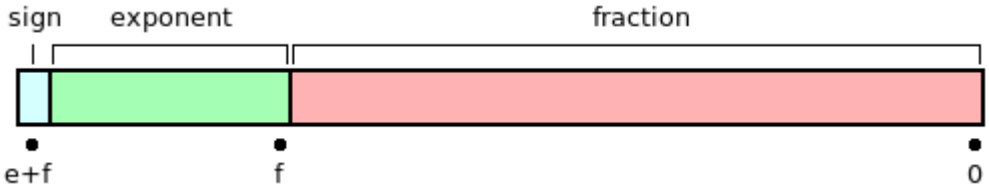
$$64 + 32 = 96,$$

a törtrész pedig - ne bajlódjunk a sok kettő hatvány reciprokával, léptessük gondolatban 8 helyiértékkel balra a számot (egész), majd léptessük ugyanennyit jobbra (/256) :

$$(255-32-2)/256 = 221/256 = 0.86328125$$

A fixpontos számábrázolás az egyszerűsége miatt rendkívül gyors (szinte az egész számokéhoz hasonló) műveletvégzést tesz lehetővé. Arra kell nagyon vigyázni, hogy a magasabb rendű műveletek (szorzás, osztás) esetén a kettedes pont visszakerüljön a helyére (ugyanis az első példa szerinti 4+4 bites ábrázolásban a 16-szoros számérték szorzás után már 256-szoros lenne, azaz a számot „vissza kell tolnunk” 4 bittel jobbra, hogy a számábrázolás helyes legyen). Elsősorban a jelprocesszorok körében elterjedt a fixpontos ábrázolás és az ezekhez kapcsolódó aritmetikai könyvtárak.

A lebegőpontos számok a matematikai normál alak fogalmához kapcsolódnak. A világ többnyire igazodik a bevezetőben hivatkozott IEEE-754 szabvány (sajnos néhol már elavult) előírásaihoz. A számok ábrázolása itt a következőképpen néz ki:



A lebegőpontos számábrázolás általános alakja

A szám értelmezése pedig:

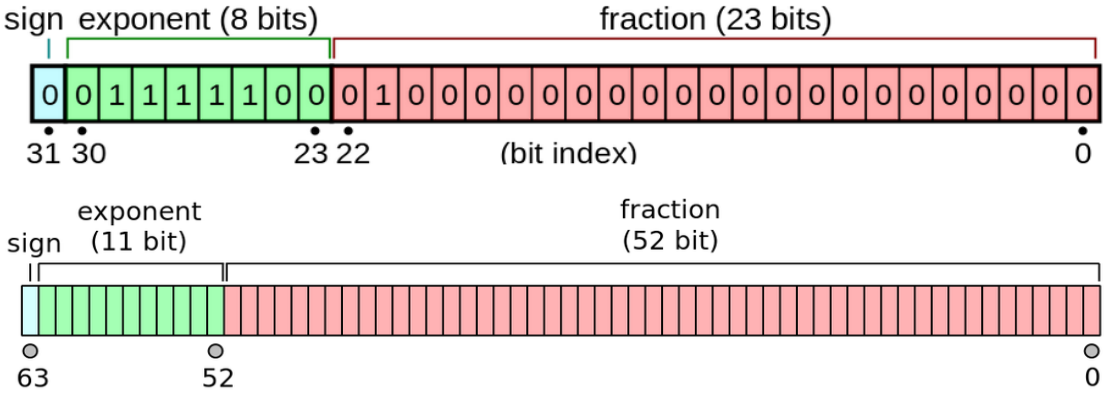
$$x = \pm m * 2^k$$

(Már a tárgyalás elején felhívjuk az olvasó figyelmét arra, hogy a fenti képletben szereplő **m** mantissza nem azonos az ábrázolásban „*fraction*” névvel jelölt törtrésszel, ugyanígy a **k** karakterisztika értékét tekintve nem azonos az „*exponent*” mező értékével.)

A **mantissza** az [1.0 .. 2.0) balról zárt intervallumba normált szám együttható része. Ezt az alakot a valós szám ún. normalizált valós alakjának nevezzük. Az elkülönülő előjel miatt a mantissza mindig pozitív. Mivel a mantissza egész része az intervallumból adódóan mindig 1, így ezt a szabvány előírásának megfelelően nem is ábrázoljuk (ún. rejtett 1-es). A *fraction* mezőben ábrázolt mantissza (*f* db bit) tehát valójában a tényleges mantissza törtrésze (m=1.törtrész).

Az *exponens* (*e* db bit) mindig egy pozitív szám, a "valódi" kitevő (karakterisztika) 0 értékéhez a mező méretéből adódó félérték–1 (pl. 8 bit esetében 127) tartozik. Ezt a nullponteltolást a továbbiakban offsetnek nevezzük (offset = n jelöléssel a kapott kódolást n többletes kódnak nevezzük). Az exponens ábrázolásából adódó minimális (0) és maximális értékeket (pl. $FF) speciális célokra tartották fent (ld. később).

A szabvány többféle lebegőpontos típust (méretet) definiál:



Az IEEE-754 szabvány szerint értelmezett lebegőpontos számábrázolások

Fentieken kívül az 1985-ös szabványban még egy kiterjesztett (extended) pontosság is értelmezett, ez viszonylag ritkábban használatos igen terjedelmes mérete (12 bájt = 96 bit) miatt. A tényleges számábrázolás itt 80 biten történik, mivel azonban a 10 (bájt) nem osztható 4-gyel, a 32 bites szóhosszúsághoz igazodva a szám 3 x 4=12 bájton való ábrázolását írja elő a szabvány. A 2008-as revízió ezen ábrázolási forma helyett már a 128 bites ábrázolási formátumot vezeti be.

	Előjel	Exponens	Törtrész	Összesen
Single	**1**	**8**	**23**	**32**
Double	**1**	**11**	**52**	**64**
Extended	**1**	**15**	**64**	**80 (+16 bit nem használt)**
128 bit	**1**	**15**	**112**	**128**

Látható, hogy a szabvány 2008-as revíziója nem a számábrázolás nagyságát növelte meg (az exponens mezőszélessége változatlan maradt), hanem a pontosságát (a törtrész mezőhossza nőtt). A tényleges számábrázolási határokat később még látni fogjuk, azt azonban már itt is leszögezhetjük, hogy a 15 bites exponens ($2^{16382} \approx 10^{4932}$) által lehetővé tett számábrázolási nagyságot nem sok értelme volna tovább növelni – viszont azon el

kell gondolkoznunk, hogy a 23 bites törtrész ~7, az 52 bites ~16, a 64 bites ~20, a 112 bites pedig ~34 hasznos számjegy ábrázolását teszi lehetővé. Ezen pontosságok értékelésénél mindig azt tartsuk szem előtt, hogy alkalmazott számítási algoritmusunk milyen pontatlanság halmozását eredményezheti – szélsőséges esetben egy egymillió műveletet tartalmazó ciklusos iteráció után a 6-7 számjegy kiindulási pontosságú operandusunk már csak $10^{-7} * 10^{6} = 10\%$ pontosságú !

Külön kell szólni a számábrázolás speciális eseteiről: a denormalizált alakról, a 0, a $\pm \infty$ ábrázolásáról és az ún. nem definiált alakról.

SM	Exponens (e)	Törtrész (f)	Értelmezés	Elnevezés
0/1	0 < e < Max	Tetszőleges	$\pm 1.f \cdot 2^{e-off}$	Normalizált szám
0/1	0	nem 0	$\pm 0.f \cdot 2^{0-off+1}$	Denormalizált szám
0/1	0	0	$\pm 0.0 \cdot 2^{0-\dots}$	Egzakt 0
0/1	Max	0	-	$\pm \infty$
0/1	Max	nem 0	-	NaN (Not-a-Number)

Számtartományok értelmezése az IEEE-754 szabvány szerint

A denormalizált alak szerepe igen lényeges: minden számábrázoláshoz egy véges nagyságú ún. felbontás rendelhető. Ez az a legkisebb "lépcső", amivel az adott pontosságú szám növelhető vagy csökkenthető, értéke értelemszerűen függ a számábrázolás pontosságától (S/D/X/128), pl. egyszeres pontosságú számábrázolás esetén

$$\textbf{Felbontás} = 1.00..0 * 2^{1-127} = 1.2 * 10^{-38}$$

Ezzel a pontossággal "együtt kell élnünk", hiszen az általunk választott számábrázolás következménye. Igazán kritikussá a dolog a 0 közelében válik, hiszen ott a 0 és az ábrázolható legkisebb szám (=felbontás) között nincs további érték, a számábrázolás hibája végtelen naggyá válhat. Vegyük azonban észre, hogy "lenne még lehetőség" a számban pontosításra – ha feladjuk azt az alapelvet, hogy a számnak a [1.0 .. 2.0) balról zárt intervallumba kell normálva lennie: a

$$\textbf{Felbontás} = 0.10..0 * 2^{1-127} (=0.10...0 * 2^{0-126})$$

szám még mindig ábrázolható, sőt az 1-es elvihető a mantissza legvégéig is; ezzel növeltük a pontosságot, de megszűnt az eddig alapszabályként őrzött ún. rejtett 1-es.

A denormalizált alak tehát a 0 közelében a felbontás pontosságát növeli azzal, hogy áttér az [1.0...2.0) intervallumról a (0.0...1.0) intervallumra (így tehát még a törtrész

helyiértékeinek számával növelhető a pontosság – csökkentve ezzel viszont az ábrázolt szám hasznos jegyeinek számát). Nézzük a fizikailag ábrázolható számokat pl. az egyszeres pontosságú (single) számábrázolási formátumnál:

Sign	Exp	Frac	Mant	Kar	Típus	s	e	f
0	11111111	11111111111111111111111	1.11..11	$2^{(255-127)}$	NaN	0/1	Max	≠ 0
0	11111111	11111111111111111111110	1.11..10	$2^{(255-127)}$				
		...						
0	11111111	00000000000000000000010	1.00..10	$2^{(255-127)}$				
0	11111111	00000000000000000000001	1.00..01	$2^{(255-127)}$				
0	11111111	00000000000000000000000	1.00..00	$2^{(255-127)}$	+ ∞	0/1	Max	=0
0	11111110	11111111111111111111111	1.11..11	$2^{(254-127)}$	Normalizált számok	0/1	0 < e < Max	bármi
0	11111110	11111111111111111111110	1.11..10	$2^{(254-127)}$				
		...						
0	00000010	00000000000000000000001	1.00..01	$2^{(2-127)}$				
0	00000010	00000000000000000000000	1.00..00	$2^{(2-127)}$				
0	00000001	11111111111111111111111	1.11..11	$2^{(1-127)}$				
0	00000001	11111111111111111111110	1.11..10	$2^{(1-127)}$				
		...						
0	00000001	00000000000000000000001	1.00..01	$2^{(1-127)}$				
0	00000001	00000000000000000000000	1.00..00	$2^{(1-127)}$				
0	00000000	11111111111111111111111	0.11..11	$2^{(0-126)}$	Denormalizált számok	0/1	= 0	≠ 0
0	00000000	11111111111111111111110	0.11..10	$2^{(0-126)}$				
		...						
0	00000000	00000000000000000000010	0.00..10	$2^{(0-126)}$				
0	00000000	00000000000000000000001	0.00..01	$2^{(0-126)}$				
0	00000000	00000000000000000000000	0.00..00	$2^{(0-126)}$	Egzakt 0	0/1	= 0	= 0

Számtartományok egyszeres pontosságú számábrázolás esetén

Vegyük észre azt a nagyon fontos tényt, hogy a normalizáltról a denormalizált tartományra történő áttérésnél az exponens offsetje 1-gyel csökken (a mantissza egészrésze nincs ábrázolva!).

Fentiek alapján valamennyi típusra értelmezhetjük az ábrázolható legnagyobb szám, a normalizált és a denormalizált felbontás, ill. hasznos számjegyek fogalmát (utóbbit természetesen csak a normalizált alak esetében):

Típus	Ábrázolható legnagyobb szám	Normalizált felbontás	Denormalizált felbontás	Hasznos (dec.) jegy
Single	$3.4028*10^{+38}$	$1.1754*10^{-38}$	$1.4012*10^{-45}$	6-7
Double	$1.7976*10^{+308}$	$2.2250*10^{-308}$	$4.9406*10^{-324}$	15-16
Extended	$1.1897*10^{+4932}$	$3.3621*10^{-4932}$	$3.6451*10^{-4951}$	19-20
128 bit	$1.1897*10^{+4932}$	$3.3621*10^{-4932}$	$1.2950*10^{-4965}$	33-34

A NaN-tipus vagy hibás számmegadás útján jöhet létre, vagy a lebegőpontos műveletvégző egységek generálják meghatározott esetekben. Itt kell megjegyeznünk, hogy a végtelen "definit" használata nem okoz problémát a műveletek számára, a szabvány előírásai szerint, pl.:

Művelet	Eredmény
n / ±∞	0
±∞ * ±∞	±∞
±[nemnulla] / 0	±∞
∞ + ∞	∞
±0 / ±0	NaN
∞ - ∞	NaN
±∞ / ±∞	NaN
±∞ * 0	NaN

Nézzünk néhány példát a fentiek könnyebb megértésére (a könnyebb áttekinthetőség érdekében csak az egyszeres pontosságú számábrázolást használjuk):

Példa 1:

Legyen x = 5. Ez a szám az [1..2) intervallumba normálva: $x = +1.25*2^2$.

- a keletkező szám előjele + (=0),
- az exponens értéke 2 (+127=129),
- a mantissza binárisan 1.01_2, amelyből a rejtett 1-es levágása után $.01_2$ marad.

s	exponens								törtrész																			
0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1. bájt									2. bájt								3. bájt								4. bájt			

Példa 1: ***x = 5*** *egyszeres pontosságú ábrázolása (0x40A00000)*

Példa 2:

Legyen x = -0.75. Ez a szám az [1..2) intervallumba normálva: $x = -1.5*2^{-1}$.

- a keletkező szám előjele - (=1),
- az exponens értéke -1 (+127=126),
- a mantissza binárisan 1.1_2, amelyből a rejtett 1-es levágása után $.1_2$ marad.

s	exponens								törtrész																			
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1. bájt									2. bájt								3. bájt								4. bájt			

Példa 2: ***x = -0.75*** *egyszeres pontosságú ábrázolása (0xBF400000)*

Példa 3:

Legyen x = 1/3. Ez a szám az [1..2) intervallumba normálva: $x = +4/3*2^{-2}$.
(Segítség: 1/3-**1/4**=1/12, 1/12-**1/16**=1/48, 1/48-**1/64**=1/192, 1/192-**1/256**=1/768…)

- a keletkező szám előjele + (=0),
- az exponens értéke -2 (+127=125),
- a mantissza binárisan $1.010101\ldots_2$, amelyből a rejtett 1-es levágása után $.010101\ldots_2$ marad.
- a véges bitszámból visszaszámított pontos eredmény: *x = 0.3333333134651184*

s	exponens								törtrész																							
0	0	1	1	1	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1. bájt									2. bájt								3. bájt								4. bájt								

*Példa 3: **x = 1/3** egyszeres pontosságú ábrázolása (0x3EAAAAAA)*

Példa 4:

Legyen x = 2014. Először alakítsuk át bináris számmá pl. a tanult felezéses módszerrel (a számot osztjuk 2-vel, a hányados lesz a következő osztandó, a maradék pedig a bináris szám legkisebb helyiértékű bitje – a folyamatot addig ismételjük, míg 0 eredményt nem kapunk):

Osztandó/hányados	Maradék	Helyiérték
2014	0	LSB
1007	1	
503	1	
251	1	
125	1	
62	0	
31	1	
15	1	
7	1	
3	1	
1	1	MSB
0		

Tehát $2014 = 11111011110 = 1.1111011110 * 2^{10}$

(Segítség: 2 kitevője megegyezik az eredetileg a szám végén található kettedes pont balra léptetéseinek számával.)

- a keletkező szám előjele + (=0),
- az exponens értéke +127+10 = 137 (az egyszerűbb átalakítás érdekében: 128+9)
- a mantissza binárisan 1.111101111 (a legalsó 0-t nem írtuk ki)

s	exponens								törtrész																							
0	1	0	0	0	1	0	0	1	1	1	1	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1. bájt									2. bájt								3. bájt								4. bájt								

Példa 4: 2014 egyszeres pontosságú ábrázolása (0x44FBC000)

Példa 5:

Legyen x = 2.014 (A példa tanulsága: mit is okoz egy egyszerű decimális eltolás ?)

Először is alakítsuk az egész és a törtrészt is bináris számmá. A szám egész része ránézésre megmondható (nagyobb számok esetén a 6. példában alkalmazott felezéses módszert alkalmazzuk), a törtrészt pl. a tanult kétszerezéses módszerrel számoljuk át. (Most az átváltandó törtrészt szorozzuk 2-vel, az eredményként kapott szám egészrésze lesz a keresett bináris törtrész legmagasabb helyiértékű bitje, a kapott törtrész pedig a következő lépés szorzandója. A műveletsor akkor hagyható abba, ha a törtrész 0-nak adódik – innen kezdve minden további bit 0 –, vagy ha a lépések száma elérte az ábrázolni kívánt bináris törtrész szükséges bitjeinek számát.)

Törtrész	Egészrész	Helyiérték
0.014	0	törtrész MSB
0.028	0	
0.056	0	
0.112	0	
0.224	0	
0.448	0	
0.896	1	
0.792	1	
0.584	1	
0.168	0	
0.336	0	
0.672	1	
0.344	0	
0.688	1	
0.376	0	
0.752	1	
0.504	1	
0.008	0	
0.016	0	
0.032	0	
0.064	0	
0.128	0	törtrész LSB

Tehát 2.014 = 10.000000111001010110 = $1.00000001110010101 10 * 2^1$

- a keletkező szám előjele + (=0),
- az exponens értéke +127+1=128
- a mantissza binárisan 1.00000001110010101 10… (ez még folytatódna !)

s	exponens								törtrész																			
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	1	0	1	1	0	0	0	0	0
1. bájt									2. bájt								3. bájt								4. bájt							

Példa 5: 2.014 egyszeres pontosságú ábrázolása (0x4000e560)

A 4. és az 5. példa eredményének összevetése jól mutatja, hogy egy szám decimális eltolása (10 hatványával való szorzása vagy osztása) teljesen átstrukturálja a bináris számot, nem csak az egyes bitek változnak meg, de a szám is átalakul „kerek” törtrészből akár végtelen szakaszos kettedes törtté.

Példa 6:

Legyen x = a minimális ábrázolható pozitív szám (denormalizált alak)

- a keletkező szám előjele + (=0),
- az exponens értéke -126 (k=0, a denormalizált alak miatt az offset 1-gyel csökken),
- a mantissza binárisan 0.00…1 (nincs rejtett 1-es, 0 nem lehet, mert az a 0 szám)
- $x = 2^{-126} * 2^{-23} = 2^{-149} = 1.4013 * 10^{-45}$

s	exponens								törtrész																			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1. bájt									2. bájt								3. bájt								4. bájt			

Példa 6: A legkisebb szám egyszeres pontosságú ábrázolása (0x00000001)

Példa 7:

Legyen x = a maximális ábrázolható pozitív szám

- a keletkező szám előjele + (=0),
- az exponens értéke +127 (+127=254), 255 nem lehet, mert a maximum a ∞ lenne
- a mantissza binárisan 1.11…1 (=2 minimális kerekítési hibával)
- $x = 2 * 2^{+127} = 2^{+128} = 3.4028 * 10^{+38}$

s	exponens								törtrész																			
0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1. bájt									2. bájt								3. bájt								4. bájt			

Példa 7: A legnagyobb szám egyszeres pontosságú ábrázolása (0x7F7FFFFF)

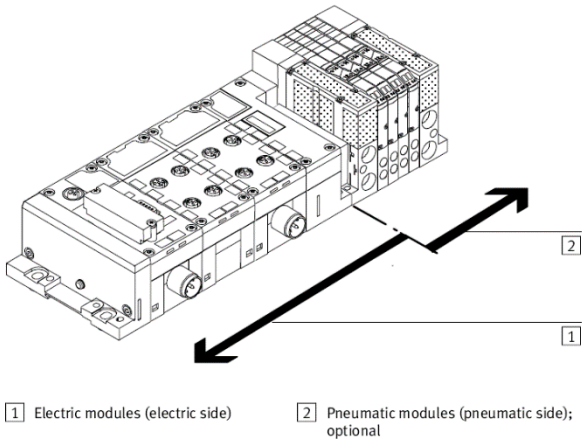
2.19 Esettanulmány: egy szoftver rendszerterv elemei

Ebben a fejezetben egy ténylegesen lezajlott gyártmányfejlesztési folyamat első lépéseit mutatjuk be, azokat a lépéseket, mikor a fejlesztő team a feladat megértésétől, megfogalmazásától kezdődően lebontja annak elemeit hardver és szoftver által megoldandó részekre, felállítja a szoftver rendszer vázát és elkészíti az ún. szoftver rendszertervet.

Bármely feladatban a firmware előállítása „egyszeri” befektetés, szemben a minden példányban újra és újra megépítendő hardver részegységekkel a szoftver úton megvalósított megoldás már csak minimális újabb költségekkel (szoftver karbantartása, javítása, update) járó reprodukciót jelent. Érthető tehát, hogy különösen azokban a jelfeldolgozási feladatokat megkövetelő ipari irányítórendszerekben, melyekben a változtatható (paraméterezhető) tulajdonságokkal rendelkező jelfeldolgozást bonyolult állapotgépeket feltételező hardver kapcsolások helyett szoftver által megvalósított programozott funkciók tehermentesítik a hardver úton megvalósítandó feladatokat, fokozottan oda kell figyelnünk a feladatok szétosztására, a hardver és a szoftver szoros kapcsolatára.

Bármilyen hihetetlennek tűnik is, a szoftverrel történő jelfeldolgozás egyik legkomolyabb ellenfele a mai napig az idő, ill. a sebesség: egyszerűnek tűnő részfeladatok nagy tömegben és nagy sebességgel szoftver úton történő elvégzésére óvatosan, előzetes számolások alapján vállalkozzunk csak. Ne értsük félre a problémát: nem a mikrokontrollerek maximális teljesítőképességével, sebességével van gondunk, hanem egy költségoptimált megoldást kell előállítanunk: „túl nagy” kontroller kiválasztása drágább megoldást jelenthet, mint egy egyszerű elemekkel realizálható hardver megoldás – az optimum megtalálása a fejlesztő igazi felelőssége.

Az ipari automatizálási elemeket gyártó Festo cég egy gyártmánycsaládjának (CPX) alapvető felépítése a következő: a terepi buszra (Profibus DP) illeszkedő lokális érzékelő/beavatkozó egységek modulárisan, tetszőleges konfigurációban egymás mellé építhetők, egy-egy ún. ”sziget” kialakítása a következő:

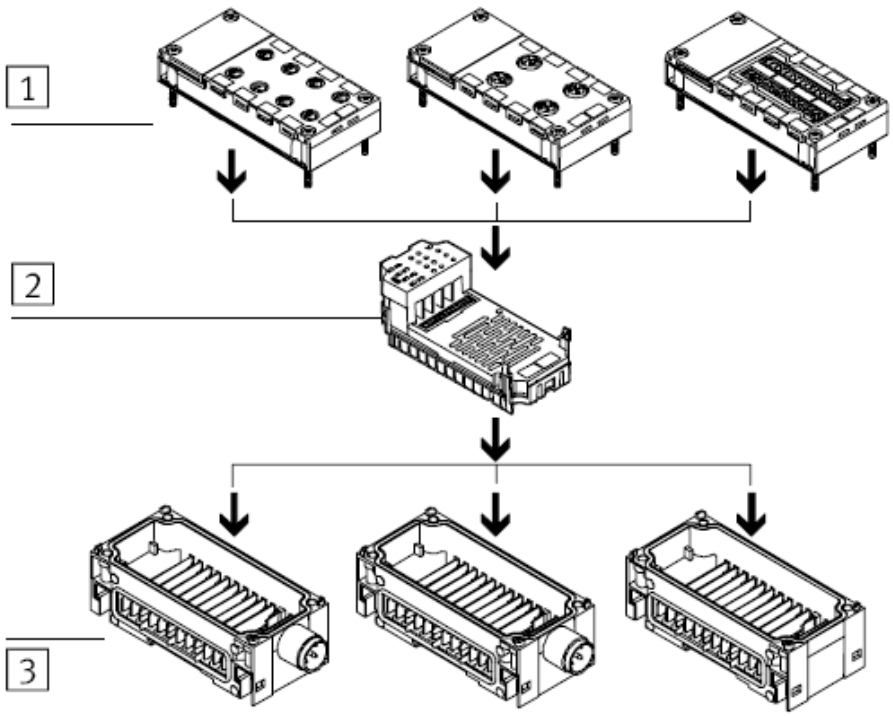


A Festo cég CPX folyamatirányító modulrendszere

A sziget baloldali részén az elektronikus modulok (a terepi busszal kommunikáló ún. master egység, digitális és analóg be- és kimeneti modulok) találhatók, míg a jobb oldalon a cég fő profiljához illeszkedő miniatűr pneumatikus végrehajtó egységek (kétállapotú és ún. arányszelepek) helyezkednek el, melyek vezérléséről szintén az elektronikus rész gondoskodik.

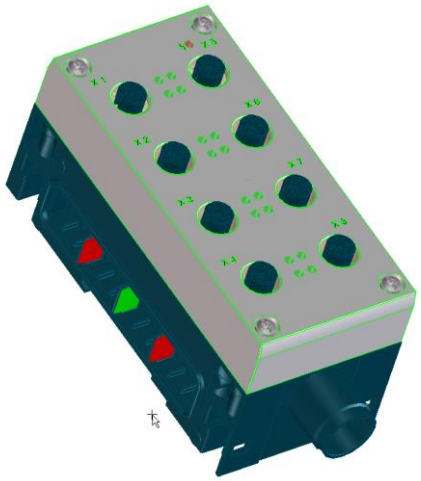
A rendszer teljesen modulárisan, a mindenkori igényeknek megfelelően építhető fel. A kiválasztott modulok egymás mellé helyezhetők, az oldalaikon található csatlakozórendszer biztosítja mind a modulok tápellátását (24V elektronika, 24V nagyobb terhelésekhez, 0V, PE), mind a modulok közötti kommunikációt. Utóbbi egy nagysebességű (1 Mbaud) soros buszon történik, melyet a cég saját protokollal látott el és speciális saját kommunikációs áramkör (CBUS ASIC) gondoskodik az adatok ciklikus átviteléről a master és a modulok között. Egy láncoló (daisy-chain) logika gondoskodik a tetszőleges módon összeállított modulkonfiguráció bekapcsolás utáni felismeréséről és a modulok azonosíthatóságáról.

Még egy-egy modul felépítése is uniformizált: a táp- és buszvezetékek (sínrendszer) összekapcsolódását biztosító alépítménybe [3] rugós érintkező pontokon csatlakoznak a modulok tényleges funkcionalitását ellátó alsó, elektronikai egységek [2], melyekre felülről többféle ipari csatlakozási szabványhoz igazodó csatlakozó-feltét (M8 kisáramú, M12 nagyáramú, rugós gyorscsatlakozós, stb.) illeszthető [1].



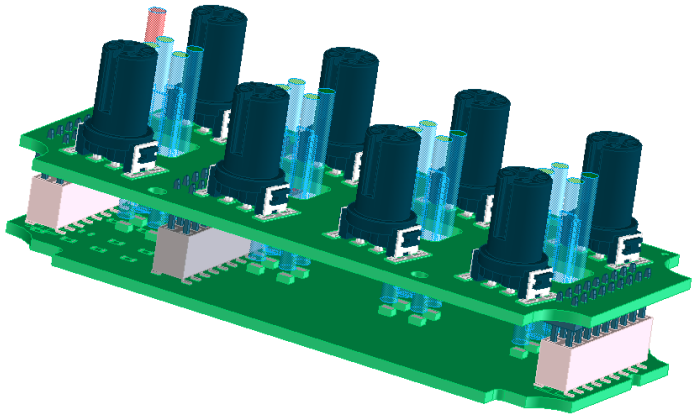
A CPX modulok moduláris felépítése (csatlakozás – elektronika – CBUS)

A fejlesztési megbízás egy olyan elektronikus bemeneti egység kifejlesztése volt, amely 8 csatlakozón összesen 16 db 24V-os kétállapotú (digitális) jelet szolgáltató szenzor túlterhelésvédett tápellátását biztosítja, fogadja a szenzoroktól érkező jeleket és azokat paraméterezhető előfeldolgozás után továbbítja a master egység felé. Felügyeli és jelzi a szenzorok tápellátásának túlterhelését is (visszakapcsolás/végleges lekapcsolás).



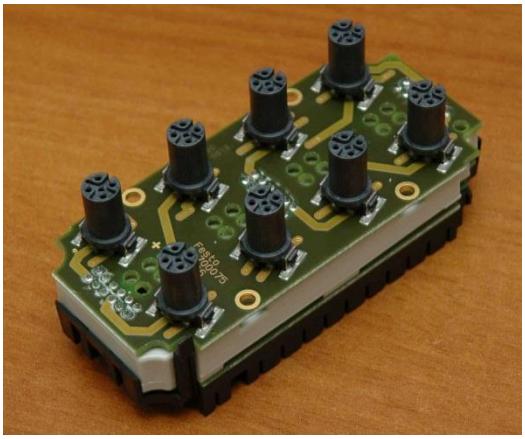
A CPX-16DE-D digitális bemeneti modul

A modul konstrukciójában igazodik a korábban vázolt ”rétegezett” felépítéshez (még a hardver tervezés beindítása előtt ilyen konstrukciós tervek készülnek egy-egy modul részletezett felépítéséről):



A 16DE-D modul konstrukciós terve

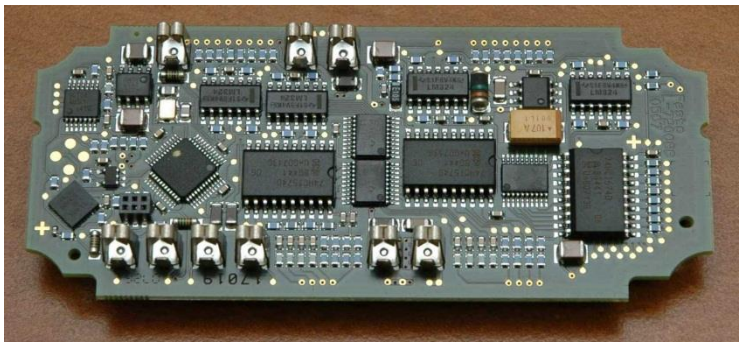
A terven jól látható, hogy a felső áramköri panel csak a csatlakoztatást szolgálja, még a kijelző elemek is az alsó, elektronikai egységen helyezkednek el, fényvezetők szolgálnak a kijelzés (zöld fény: bemenet aktív, piros fény: szenzor táp túlterhelés) „kivezetésére” a felül található előlap felé. Szokjuk meg, hogy a mechanikai elemek sok tekintetben determinálják az elektronika kialakítását és elhelyezését is (nem ritkán azért kell a jól bevált alkatrészeinket más típusokra cserélni, mert azok a mechanikai felépítés következtében nem „férnek el” megfelelő helyen az áramköri panelen).



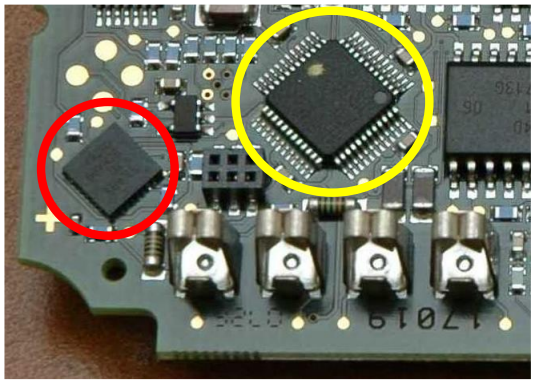
A modul fedél nélkül (felső panelen látszanak a fényvezetők helyei)

Kicsit lépjünk előre az időben, és rögtön megmutatjuk a tervek kialakításakor még csak az elképzeléseinkben létező modul elektronikai paneljének végleges kialakítását is:

A tényleges elektronika (a felső, csatlakozó panel felőli oldal)

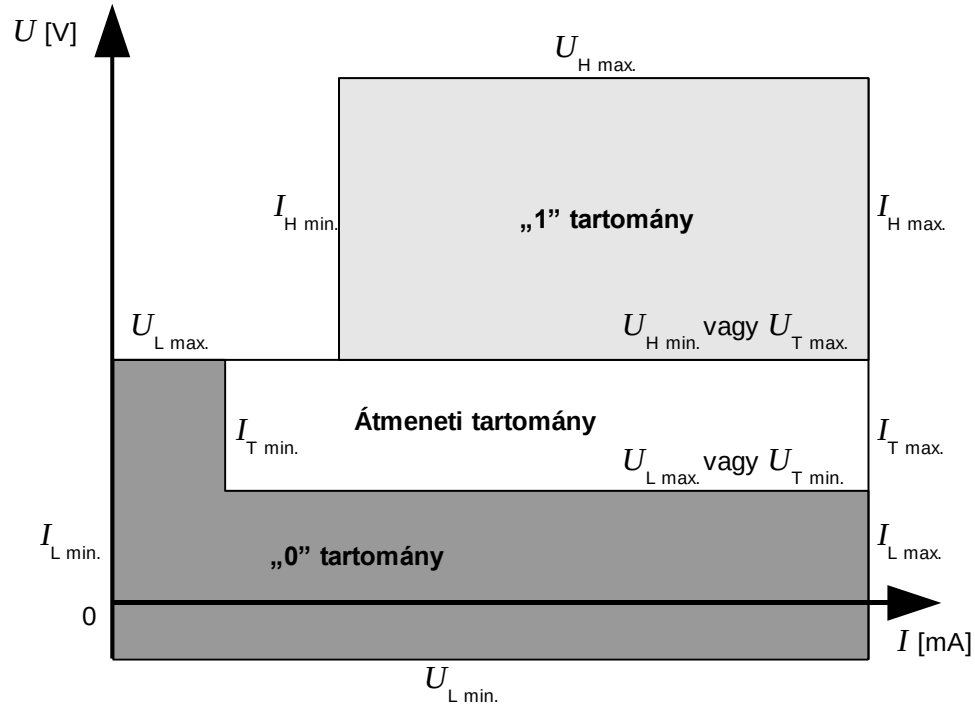


A tényleges elektronika (alsó oldal – jól látszanak a rugós buszcsatlakozók)



A mikroprocesszor (piros) és a buszkommunikációs ASIC (sárga)

De nézzük magát a megoldandó feladatot! A 16 db 24V-os, kétállapotú bemenet az IEC 61131-2 szabvány előírásainak megfelelően egy definit ”0”, egy jól definiált ”1” és a kettő között egy ún. átmeneti tartománnyal kell rendelkezzen.



Digitális bemenet U-I tartományai (IEC 61131-2)

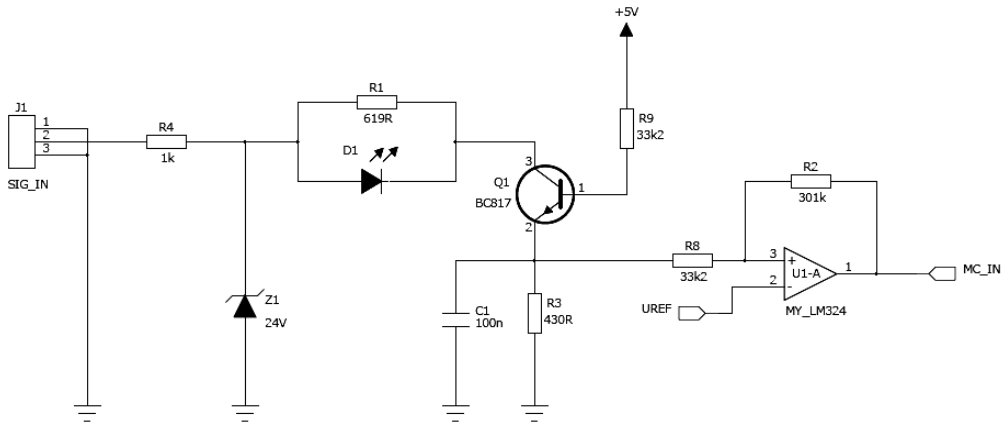
Bemenő jel	Limit	„0” állapot		Átmeneti tartomány		„1” állapot	
		U_L [V]	I_L [mA]	U_T [V]	I_T [mA]	U_H [V]	I_H [mA]
DC 24V	Max.	11/5	30	11	30	30	30
	Min.	-3	n.d.	5	2	11	6

Digitális bemenet működési jeltartományai (IEC 61131-2; n.d. = Not Defined)

A diagram alapján azonnal látható a szabványalkotónak az a határozott törekvése, hogy

- Ameddig a bemenő feszültség értéke egy határt (U_{Lmax}) nem halad meg, bármilyen terhelő áram esetén is logikai „0” szintet kell felismernie az áramkörnek.
- A bemeneteknek (a védőáramkörök nyitóirányú feszültségese következtében) adott határig negatív bemenő feszültség (U_{Lmin}) esetén is helyesen kell működjenek.
- Kis bemenő áramok tartományában a bemenő feszültség magasabb értéke is még logikai „0” jelszintet takar (statikus túlfeszültségek).
- A logikai jeltartományok határozottan váljanak el egymástól, az átmeneti tartomány elkerülhetetlen, ez a sáv a kapcsolás számára bármelyik bemeneti jelszint azonosítását is eredményezheti.
- A logikai ”1” jelszint ne lehessen véletlen műve – ehhez meghatározott terhelőáramnak kell társulnia
- a 24V mindig meghatározott toleranciasávok között értendő, az ipari alkalmazások esetében ez tipikusan ± 25% (18..30V)

A fenti követelményeket teljesítő bemeneti fokozatot a következő kapcsolással valósítottuk meg:



Egy digitális bemeneti fokozat

A működés részletes elemzését az olvasóra bízzuk, egy-két, a megértést segítő megjegyzés:

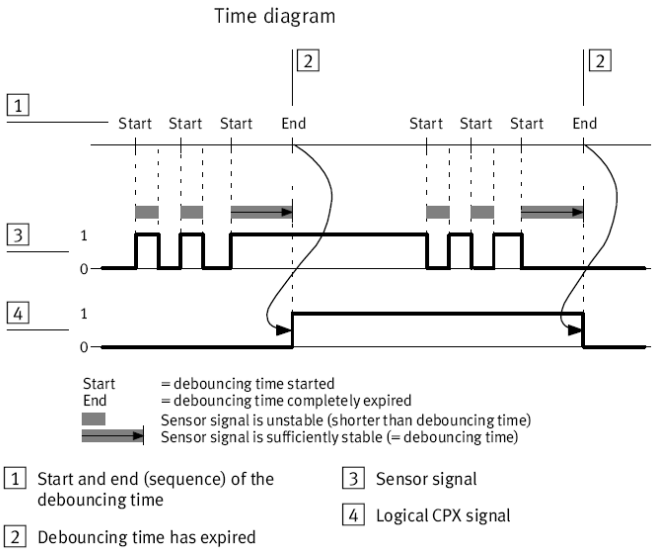
- A bemeneti védőkapcsolás (R4-Z1) a kapcsolási túlfeszültségek és a fordított polaritású feszültség ellen nyújtanak védelmet.
- A speciális tranzisztorkapcsolás (Q1, pszeudo áramgenerátor) biztosítja a megkívánt áramterhelést, valamint azt, hogy a visszajelzés (D1) fénye lehetőleg ne változzon lényegesen a logikai „1” tartományban (kb. 15-30V között).
- A kimenőjel a tranzisztor emitterkörében található munkaellenálláson keletkezik.
- A határozott billenést hiszterézises komparátor kapcsolás (U1-A) biztosítja.
- A mikrokontroller mintavételezésénél (100 μsec – ld. később) gyorsabb jelátmenetek (alapvetően környezeti zaj) hardver úton történő szűrése (Q1 emitterkörében található R3-C1 szűrő).

A digitális bemenőjelek paraméterezhető jelformálás után kerülnek továbbításra, ezek a következők lehetnek:

- ✓ Pergésmentesítés: 0.1 ms, 3 ms, 10 ms, 20 ms
- ✓ Impulzushosszabbítás: 0, 0.5 ms, 15 ms, 50 ms, 100 ms

Mire jó (avagy miért kell) mindez ?

- Pergésmentesítés: általában ismert az alkalmazott tényleges kapcsolóelem, ennek pergési tulajdonságai alapján beállítható egy biztonsági felső időkorlát ahhoz, hogy ne legyen téves kapcsolásjelzés (viszont minél nagyobb értéket állítunk be, annál kisebb működési frekvencia lehetséges!)
- Impulzushosszabbítás: a kiszolgáló egység a felügyelő (host) számítógép ciklusidejéhez igazítja a beérkező jelváltozások tulajdonságait annak érdekében, hogy „ne veszhessen el" impulzus (ez szintén a működési sebesség korlátozása!)



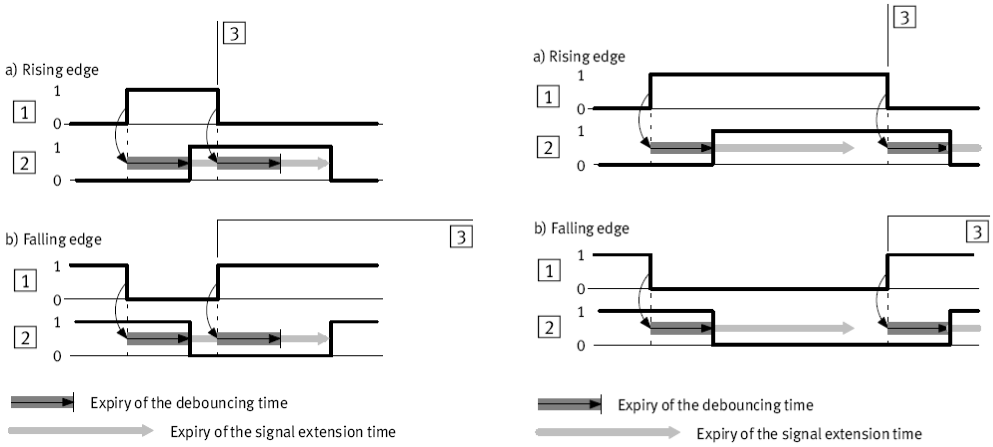
A pergésmentesítés elve

A pergésmentesítés során azt az egyszerű elvet követjük, hogy a bemenőjelet mintavételezzük, és amennyiben annak állapota előző állapotához képest nem változott, növeljük egy, a bemenethez tartozó számláló értékét. Változás esetén a számlálót nullázzuk. Amennyiben a számláló értéke elérte a paraméterben megszabott pergésmentesítési időt (pl. 100 µsec x 30 = 3 msec), abban az esetben „elhisszük" azt, és jelezzük a bemenő jel megváltozását.

Az impulzushosszabbítás során a pergésmentesített bemenőjel impulzusok hosszát (amik nem lehetnek rövidebbek, mint a beállított pergésmentesítési idő) meghosszabbítjuk a megadott értékre. Két esetet kell megkülönböztessünk:

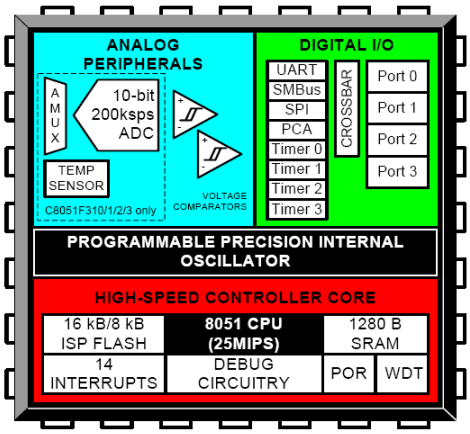
1. amennyiben a bemenőjel „visszaváltása" hamarabb következik be, mint a megadott legrövidebb impulzusidő, akkor azt meg kell hosszabbítanunk a megadott időre
2. amennyiben a bemenőjel-impulzus eleve hosszabb volt, mint a megadott legrövidebb impulzus ideje, akkor csak a pergésmentesítés leteltét kell kivárnunk.

(Vegyük észre, hogy az előállított impulzus hossza semelyik esetben sem lehet rövidebb, mint a beállított pergésmentesítési idő!)



Az impulzushosszabbítás elve (1. és 2. eset)

A feladat nem látszik túl bonyolultnak, mindenki számára megoldhatónak tűnik a jelfeldolgozás egy parányi (5x5 mm) és olcsó (nagyobb darabszámban kb. 3€/db) mikrokontrollerrel. Különösen a paraméterezett időzítés/számlálás lenne bonyolult egy esetleges analóg vagy digitális, de hardver megoldással. A kiszemelt mikrokontroller - elsősorban a viszonylag nagy művelet végrehajtási sebesség (átlagosan 15-20 MIPS) és a bitkezelést támogató belső (8051-es) architektúra miatt - a Silicon Laboratories C8051F311 típus lett.



A C8051F311 mikrokontroller (MLP-28 5x5 mm tokozásban)

Első és legfontosabb kérdésünk az volt: a lehető leggyorsabban működtetve a mikrokontrollert ”elbírjuk-e” 16 bemenet független feldolgozását annak figyelembevételével, hogy a legrövidebb előírt időkvantum 100 µsec (a legrövidebb pergésmentesítési idő), miközben a mikrokontrollert belső órajel-generátoráról a lehető legnagyobb sebességgel (f=24.5 MHz ⇒ T=40.82 nsec) működtetjük?

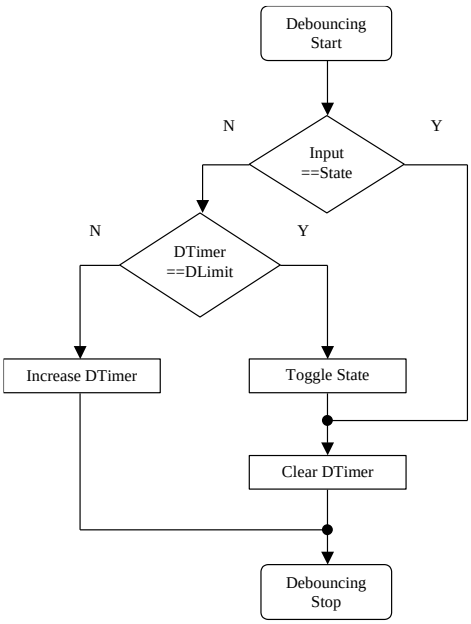
A legfontosabb alapelvek a következők:

- A 16 fizikai bemenet 2 db 8 bites portra rendezhető, ezeket 100 µsec-ként beolvassuk (a szenzortáplálás kapcsolása + diagnózisa és az SPI kommunikáció miatt a 28 láb nem elegendő arra, hogy a bemeneteket közvetlenül a portlábakra csatlakoztassuk)
- A beolvasott értékeket (2 bájt) letesszük a bitcímezhető területre ⇒ ezzel valamennyi bemenet direkt módon címezhetővé válik
- Ciklust nem szervezünk, mert
  - DJNZ időbe kerül (4T x 16 = 64T = 2.6 µsec)
  - Indirekten kellene címezni mind a biteket, mind azok számlálóit, ami egyértelmű időveszteség lenne
  - ⇒ lemásoljuk a feldolgozást 16-szor egymás mögé!

```
        CLR     bInputLEN               // Select Input0-7
        MOV     ucInputL,P1             // Read the inputs      (3)
        SETB    bInputLEN               // Deselect             (2)

        CLR     bInputHEN               // Select Input8-15     (2)
        MOV     ucInputH,P1             // Read the inputs
        SETB    bInputHEN               // Deselect
```

A bemenetek beolvasása

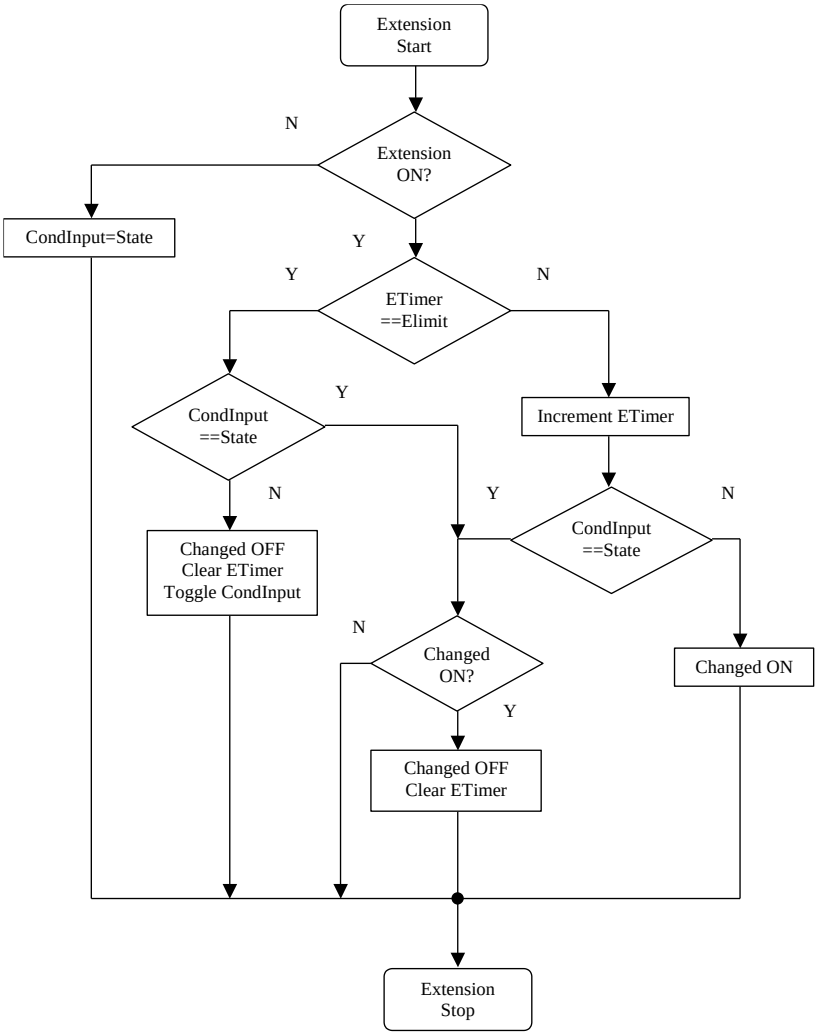
A beolvasás egyidejűségi hibája a kékkel jelzett utasítás végrehajtási periódusidők alapján 7T = 286 nsec. A bemutatott algoritmus és a hozzátartozó kódrészletek elemzését az olvasóra bízzuk.



A pergésmentesítés folyamatábrája

```
L0_Deb:
        MOV     C,bInput0                   // if (bInput==bState)
        JNB     bState0,L0001
        CPL     C
L0001:  JNC     L0002
                                            // if(bInput==bState) -> FALSE
        MOV     A,R0                        // if (ucDlimit==ucDcount)
        CJNE    A,ucDcount0,L0003
                                            // if(ucDlimit==ucDcount) -> TRUE
        CPL     bState0                     // Toggle bState

L0002:                                      // if(bInput==bState) -> TRUE
        MOV     ucDcount0,#00H              // Clear Debouncing counter
        SJMP    L0_Ext                      // jmp Signal Extension

L0003:                                      // if(ucDlimit==ucDcount) -> FALSE
        INC     ucDcount0                   // Increase Debouncing counter
L0_Ext:
```

A pergésmentesítést megvalósító programrészlet



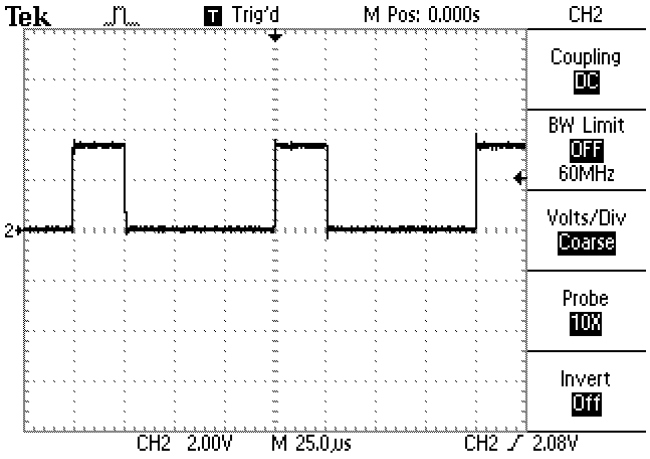
Az impulzushosszabbítás algoritmusa

```
L0_Ext: JNB     bExtension0,L0004           // if (bExtension==1)

                                            // if(bExtension==1) -> TRUE
        MOV     A,R2                        // if (usEcount==usElimit)
        CJNE    A,usEcount0+01H,L0005
        MOV     A,R1
        CJNE    A,usEcount0,L0005

                                            // if(usEcount==usElimit) -> TRUE
        MOV     C,bState0                   // if (bCondInput==bState)
        JNB     bCondInput0,L0006
        CPL     C
L0006:  JNC     L0010
                                            // if(bCondInput==bState) -> FALSE
        CLR     bChanged0                   // Clear cChanged
        CLR     A
        MOV     usEcount0,A                 // Clear usEcount
        MOV     usEcount0+01H,A
        CPL     bCondInput0                 // Toggle bCondInput
        SJMP    L0_Ext_Stop                 // Exit from Signal extension

L0004:  MOV     C,bState0                   // if(bChanged==1) ->TRUE
        MOV     bCondInput0,C
        JNC     L0_Ext_Stop

L0009:  SETB    bChanged0                   // if(bCondInput==bState) -> FALSE
        SJMP    L0_Ext_Stop

L0005:                                      // if(usEcount==usElimit) -> FALSE
        INC     usEcount0+01H               // Increment usEcount
        MOV     A,usEcount0+01H
        JNZ     L0007
        INC     usEcount0

L0007:  MOV     C,bState0                   // if (bCondInput==bState)
        JNB     bCondInput0,L0008
        CPL     C
L0008:  JC      L0009
                                            // if(bCondInput==bState) -> TRUE
L0010:  JNB     bChanged0,L0_Ext_Stop       // if (bChanged==1)

                                            // if(bChanged==1) ->TRUE
        CLR     A
        MOV     usEcount0,A                 // Clear usEcount
        MOV     usEcount0+01H,A
        CLR     bChanged0                   // Clear bChanged
L0_Ext_Stop:
```

Az impulzushosszabbítást megvalósító kódrészlet

Fenti kódrészleteken gyakorlott programozók dolgoztak kb. két munkanapon át. A kitűzött cél az volt, hogy hogyan lehet a fenti kódrészleteket a lehető legrövidebb idő alatt végrehajtatni a mikrokontrollerrel. A folyamat során folyamatosan számolták az órajel-periódusokban mért végrehajtási időt, mint a megvalósíthatóság egyik alapfeltételét.

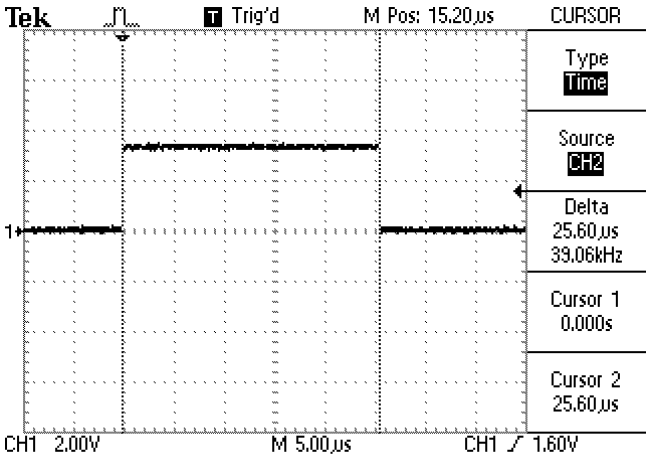
Természetesen a számított értékeket mérésekkel ellenőriztük is. A szoftver működését mérésekkel ? Igen – ez egy nagyon fontos lehetőség, ha csak tudunk, éljünk vele minden esetben ! A ténylegesen beprogramozott kódrészlet egy 100 µsec időzítéssel futtatott megszakítási rutinba került, a programrészlet elején egy kimenetet „1”-be állítottunk, majd a kódrészlet lefutása után ismét „0”-ba került a kimenet. Az oszcilloszkópon a következő kép volt megfigyelhető:



A programrészlet futásidejének ellenőrzése oszcilloszkóppal

És akkor az első tényleges ellenőrzés: a mikrokontroller 24.5 MHz órajel-frekvenciával működik, azaz egy órajel periódusideje T=40.82 nsec. Nyugalmi helyzetben (a bemeneteken nincs változás) a pergésmentesítési algoritmus 15T (612 nsec) időt igényel, az impulzushosszabbítási programrészlet 22T (898 nsec) alatt hajtható végre bemenetenként. A megszakítási rutin további 37T „hulladékidőt” igényel a regiszterek mentése/visszaállítása, a bemeneti portok beolvasása céljára. Ezzel nyugalmi helyzetben tehát a következő eredményre jutottunk:

$$(37 + 16 * (15 + 22)) * 40.82 \text{ nsec} = 25.68 \text{ µsec}$$



A feldolgozás futásideje állandó bemenetek esetén

Az oszcilloszkóp képernyőjének jobboldali középső mezőjében jól látható az igen jó egyezés a számított értékekkel.

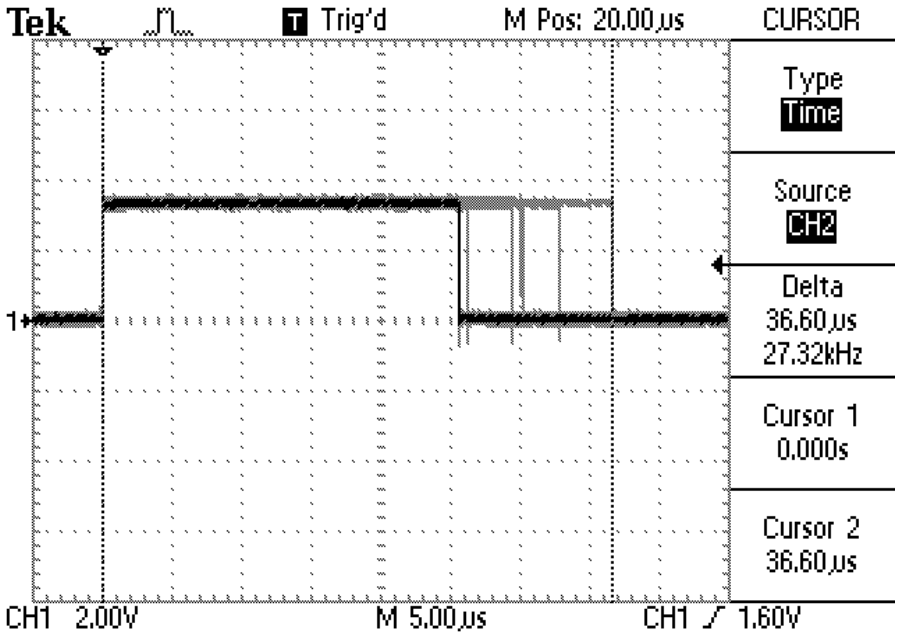
Ezt követően már bonyolultabb feladat következett: a bemenetek változásai függvényében lehetséges „futásidő-kombinációkat” számoltunk abban az esetben, ha különböző időpontokban, de valamennyi bemenet egy időben mozog. A

programrészletek független volta miatt ez a feltételezés nyugodtan megtehető, a legrosszabb eset (worst case) így is megfigyelhető kell legyen.

1 csatorna [T]	16 csatorna [T]	16 csatorna + IT-kezelés [T]	Az IT rutin számított hossza [µsec]	Megjegyzés
37	592	629	25,68	Nyugalmi helyzet
38	608	645	26,33	
43	688	725	29,59	
44	704	741	30,24	
48	768	805	32,86	
54	864	901	36,78	

Futásidők különböző állapotokból indított algoritmusok esetén

A most megfigyelt idődiagram a következőképpen alakult:

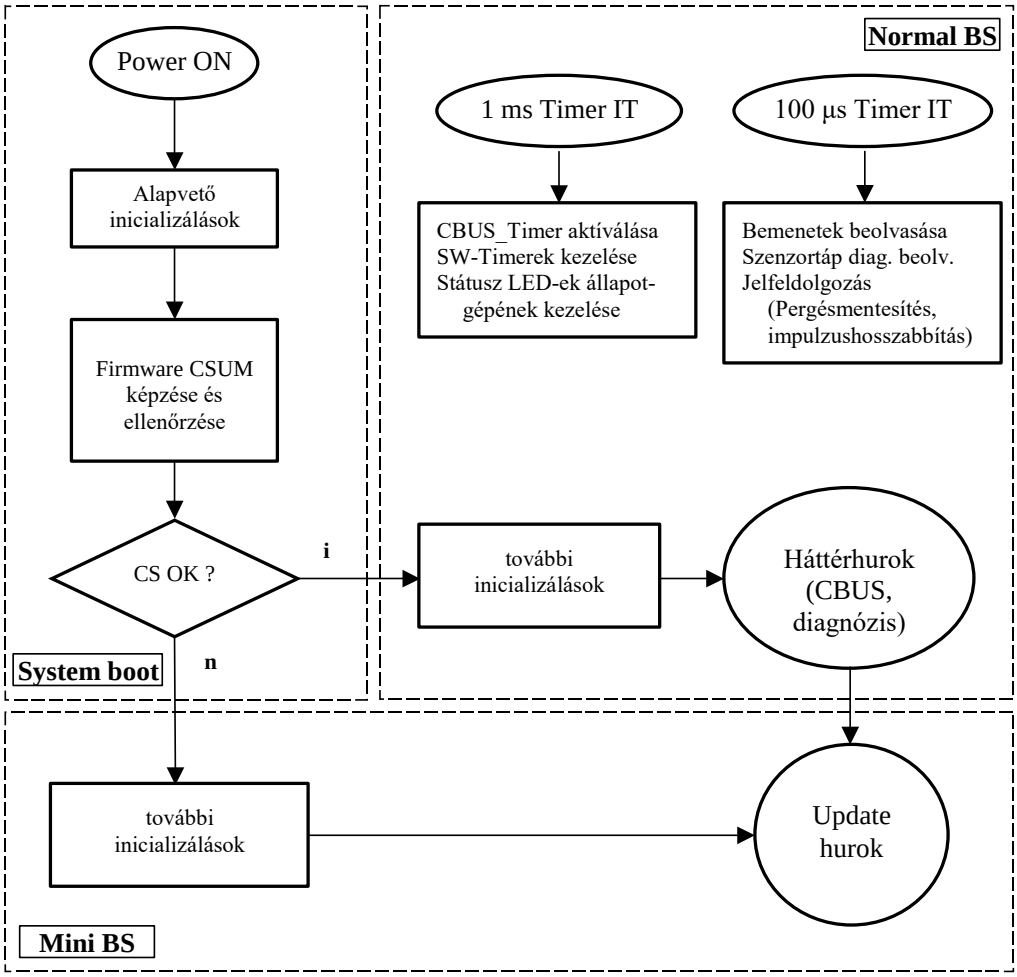


A feldolgozás futásideje változó bemenetek esetén

A számított és a mért idők ismét megnyugtató egyezést mutattak. Ezek alapján a mikrokontrollerre a jelfeldolgozás várhatóan 37 µsec / 100 µsec = 37% terhelést jelent, ami nyugodtan vállalható a környező események lényegesen lassabb volta miatt (tápellátás-figyelés kb. 50-100 msec-ként, adattovábbítás a master egység felé kb. 1 msec-ként).

Fentiek alapján már össze is állíthatjuk firmware rendszerünk működési blokkvázlatát:

1. Minden indulás (Power-ON) után a rendszerünk önellenőrzést végez. Csak az operációs rendszer hibátlan volta (külön helyre letárolt ellenőrző összeggel való összehasonlítás után) indítjuk a rendszer működését.
2. Hibás rendszer detektálása (vagy a működés alatt kapott ilyen értelmű parancs) esetén egy biztonsági „Mini-BS" kerül aktiválásra, ami csak a firmware rendszer törlését és update-elését teszi lehetővé.
3. A rendszerben a legmagasabb (senki által nem tiltható) prioritáson fut a 100 µsec-os IT rutin, ami bármit megszakíthat és max. 40 µsec-ra felfüggesztheti annak működését.
4. A „lassabb" időzítési feladatokat (mint a túlterhelés figyelő rendszer 50-100 msec-kénti működtetését, a mért adatok 1 msec-kénti továbbítását, a rendszer timeout-ok kezelését, a hibaállapotokat jelző ledek 1-2 Hz-es villogtatását) egy alacsonyabb prioritású 1 msec-os időzítő rutin végzi)
5. A nem időkritikus feladatok (további rendszerállapotok figyelése, CBUS kommunikációs tesztparancsok, stb.) a háttérhurokba kerültek.



A CPX-16DE-D modul szoftver rendszerének blokkvázlata

Foglaljuk össze, mit is csináltunk eddig a feladathoz kapcsolódóan ?

1. Megismerkedtünk annak a rendszernek a tulajdonságaival, amelyhez a fejlesztésünk során kapcsolódnunk kell
2. Részletesen megmegismertük a feladatot, amit meg kell valósítanunk (annak a „miértjeit" is, azaz semmit ne fogadjunk el csak azért, mert „azt mondták".
3. Elképzeltünk egy hardver-szoftver feladatmegosztást. Korábbi fejlesztési tapasztalataink alapján a kapcsolások „durva" részletei szinte azonnal felvázolhatók a feladathoz – a méretezés természetesen egy későbbi fejlesztési stádium feladata lesz.
4. Némi gyakorlattal azonnal érezhető, mi lesz a szoftver kritikus része, ahol a bonyolultság (memóriaszükséglet) vagy a gyorsaság (utasítások végrehajtási sebessége) esetleg szűk keresztmetszet lehet.
5. Ezekhez a programrészekhez egy valóságos (fejlesztő kitek) vagy szimulált környezetben tesztkódokat készítünk. Gondoljunk arra, hogy az itt „elfecsérelt" órák napok, hetek rossz irányba történő fejlesztési ráfordításait előzhetik meg !
6. A tesztkódokat mérésekkel ellenőriztük (méret, sebesség).
7. Fentiek alapján már össze tudtuk állítani azt a globális blokkvázlatot, aminek megfelelően – team munka esetén – akár több párhuzamos szálon is elindítható a fejlesztési munka.

Az összeállított **rendszerterv** alapján az egység hardver-szoftver elemeinek tényleges tervezése és megvalósítása elkezdődhet – a modul várhatóan valamennyi elvárásunknak meg tud majd felelni. A folyamat ily módon le is zajlott, a bemeneti egység jelenleg több tízezer példányban működik a Festo ipari automatizálási rendszereiben.

Mikrokontroller alapú rendszerek

Elektronikus jegyzet

3. fejezet

Készítette: Dr. Tevesz Gábor c. egyetemi tanár
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
Fax: 463-2871 (adm.)
Mail: tevesz@aut.bme.hu

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki alapszak (BSc)
III. évfolyam, 5. félév
Beágyazott és irányító rendszerek specializáció hallgatói

COPYRIGHT

3. MIKROKONTROLLEREK INTEGRÁLT PERIFÉRIÁI

3.1 Órajel generátorok

A mikrokontrollerek órajele a működésükhöz elengedhetetlenül szükséges, a kontroller bonyolult belső sorrendi hálózatát és a valamennyi perifériális egység működését (ütemezését, időzítését) is vezérlő digitális jel.
A mikroprocesszorok első generációira órajelként az erre a célra szolgáló „Φ", „CLK" stb. bemenetükön stabil, a logikai jeleknek megfelelő jelszintű négyszögjelet kellett kapcsolnunk. Néhány processzortípus ennél speciálisabb követelményeket is támasztott ezekkel a jelekkel szemben: az **i8080**-as mikroprocesszor számára még kétfázisú, nem TTL jelszintű órajeleket (Φ_1, Φ_2, $V_{IH} \geq 9V$!) kellett kapcsolnunk, a **Z80** esetében „csak" jóval keményebb feltételeket kellett biztosítani a a jel H-szintjével ($V_{IH} \geq 4.4V$) és terhelhetőségével kapcsolatban, mint egy normál logikai bemenet esetében.

Mivel a mikrokontrollerek működéséhez az órajel megléte elengedhetetlen feltétel, az előállításához szükséges áramköri egységek egyre inkább „integrálódtak" a kontrollerek belsejébe:

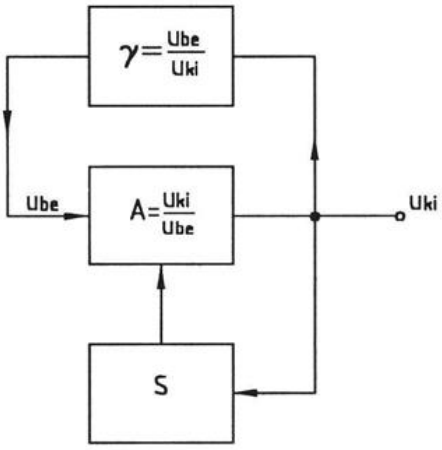
- néha a kontroller már „mindent" magában foglal (saját belső rezonátorral rendelkezik, mint a SiLabs 8051-es kontrollerek is), ezt a megoldást sokszor azért nem választhatjuk, mert a beépített rezonátor abszolút pontossága általában nem elegendő. (Amennyiben pl. a beépített óránktól elvárjuk, hogy egy hét alatt ne siessen vagy késsen 1 másodpercnél többet, az elvárt pontosság $1/(7 \cdot 24 \cdot 3600) = 1.65 \cdot 10^{-6}$!)
- gyakoribb az órajel-generátor kapcsolás olymértékű integrálása, hogy külső elemként már csak a kvarckristály (és a két blokk-kapacitás) szükséges a működtetéshez
- a nagyobb órajel-frekvenciák előállításához használatos PLL (fáziszárt hurok) áramkörök napjainkban már a kontrollerek belsejében találhatók, így egyrészt programozottan „megválaszthatjuk", vagy akár menet közben változtathatjuk is vezérlőnk működési frekvenciáját.

Milyen szempontok alapján választjuk meg egy mikrokontroller órajelét ?

1. Az **órajel-frekvencia nagyságrendjének** megválasztását többnyire az elvégzendő feladathoz szükséges műveletvégzési sebesség szabja meg
2. Minél nagyobb frekvenciát választunk, annál kritikusabbá válnak a külső illesztések (sebességek, szórt kapacitások, nyomtatott áramkör-tervezési szempontok), és annál nagyobb, a környezetünkben megjelenő kisugárzott frekvenciaspektrummal („elektroszmog") is kell számolnunk.
3. Mint ismeretes, a magasabb frekvencia a CMOS technológián alapuló áramkörök esetében lineárisan növekvő tápáramot ($\mathbf{I_{CC} = C_{pd} \cdot U_t / T = f \cdot C_{pd} \cdot U_t}$), és a frekvenciával lineárisan növekvő teljesítményfelvételt ($\mathbf{P_D = f \cdot C_{pd} \cdot U_t^2}$) jelent, ami egyrészt magasabb fogyasztásban és ennek megfelelően nagyobb veszteségteljesítményben (melegedési problémák!) fog jelentkezni.

4. Bármilyen furcsa is, az órajel pontos, egzakt megválasztását a legtöbb esetben a környezetünkkel való kommunikációhoz szükséges segédfrekvencia-szükséglet (RS232 kommunikáció baudrate-je, CAN kommunikáció sebessége, stb.) fogja megszabni.

Mivel a mikrokontrollerek órajel igényét ilyen vagy olyan módon nekünk, tervezőknek kell majd biztosítanunk, vizsgáljuk meg elsőként az oszcillátor kapcsolások legfontosabb tulajdonságait! Az általános alapkapcsolás alapelve a következő ábrán látható:



Az oszcillátorok általános alapkapcsolása

A **γ** visszacsatoló hálózat feladata, hogy kifejezetten egy frekvencián: az oszcillátor működési frekvenciáján pozitív visszacsatolást hozzon létre az **A** átvitelű erősítő kimenete és bemenete között. A működési frekvencián a visszacsatoló hálózat nem invertáló erősítő esetén 0°, invertáló erősítő alkalmazásakor 180° fázisforgatást hoz létre a kimenő és a bemenő jel között. (Általánosságban: a pozitív visszacsatolás feltétele, hogy az erősítő és a visszacsatoló hálózat együttes fázisforgatása 0°, vagy az ezzel egyenértékű 360° legyen. Ez az ún. fázis feltétel. Más frekvenciákon a fázisforgatás mértéke más, ezért ezeken a frekvenciákon nem jön létre pozitív visszacsatolás.
Az **S** jelű amplitúdó stabilizáló áramkör az **A** átviteli tényezőjű erősítő erősítését úgy szabályozza, hogy a működési frekvencián a visszacsatoló hálózat **γ** leosztásának és az erősítő A_u feszültségerősítésének szorzata 1 legyen, azaz a

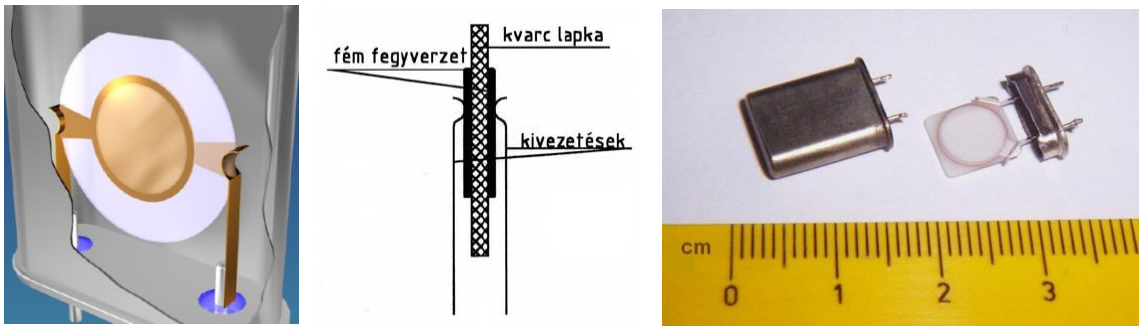
$$\gamma A_u = 1$$

feltétel teljesüljön (ez az ún. amplitúdó feltétel). Ha $\gamma A_u = 1$, akkor az erősítő kimenetén ugyanazt a jelet kaptuk vissza, amiből kiindultunk, azaz az adott frekvencián a rezgés állandó amplitúdóval fenntartja önmagát

Az elektrotechnikában sokféle oszcillátor-kapcsolás terjedt el, mi itt részletesebben csak a mikrokontrollerek számára komolyabb jelentőséggel bíró kristály- (vagy kvarc-) oszcillátorokkal (XO) foglalkozunk. Ezeknél LC rezgőkörök helyett rezgőkristályokat alkalmazunk, melynek frekvenciastabilitása sokkal jobb, a $10^{-6}..10^{-10}$ értéktartományban mozog. A kvarc- (pl. SiO_2 = kvarc) kristályok felületén mechanikai deformáció hatására

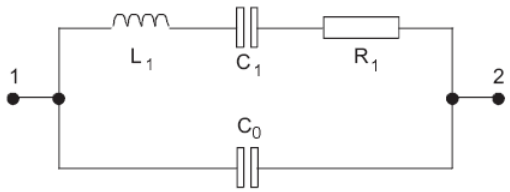
elektromos feszültség keletkezik, és fordítva: elektromos feszültség hatására a kristály rugalmas alakváltozást szenved (elektrostrikció = piezoelektromosság).

Elektronikai célra a kvarckristályból megadott módon (metszetben) meghatározott alakú lapkát vágnak ki, és azt olyan méretre csiszolják, hogy mechanikai rezonanciafrekvenciája megegyezzen a kívánt rezgési frekvenciával. A lecsiszolt kvarc lapka mindkét felületére elektrolitikus úton ezüstréteget visznek fel. A mechanikus kivezetések a kétoldali ezüstréteghez csatlakoznak



Rezgőkristály felépítése

A kivezetések közé váltakozófeszültséget vezetve, a kristály periodikusan deformálódik, rezgésbe jön. Amikor a rákapcsolt váltakozófeszültség frekvenciája megegyezik a kristály mechanikai méreteitől függő rezonanciafrekvenciájával, a kristály rezonál. A rezgőkristály úgy működik, mint egy elektromechanikai átalakító, azaz az elektromos energiát átalakítja mechanikai energiává, a mechanikai energiát pedig visszaalakítja elektromos energiává. Így a kapcsai között mérhető impedancia a működési frekvencia függvényében változik. Elektromos szempontból a rezgőkristály a következő egyszerű kapcsolással helyettesíthető:



A kvarckristály helyettesítő képe

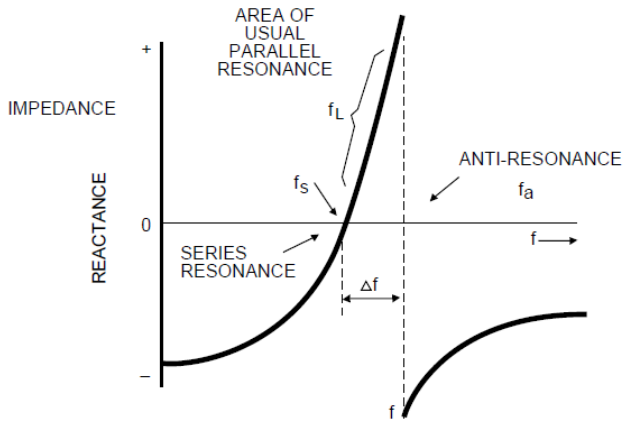
L_1, C_1 és R_1 a kristály piezoelektromos tulajdonságai által meghatározott értékek, C_0 pedig a kristály két oldalára csapatott ezüstréteg, illetve a kivezetések kapacitásából adódik ki. Az L_1 és C_1 (R_1-et hanyagoljuk el) elemekből adódó, igen nagy jóságú rezgőkör **soros rezonanciafrekvenciája** (f_s = series resonant frequency):

$$X_{L1} + X_{C1} = 0 \Rightarrow j\omega L_1 + \frac{1}{j\omega C_1} = 0$$

amiből

$$f_s = \frac{1}{2\pi\sqrt{L_1 C_1}}$$

A soros rezonanciafrekvencia fölött a soros LCR kör induktív jellegűvé válik, és C_0 kapacitással egy párhuzamos rezonancia jön létre. Ez a tartomány az impedancia maximumáig (ideális esetben végtelenig és előjelváltásáig) tart, a határpont az **antirezonáns frekvencia** (f_a = antiresonant frequency).



Rezgőkristály reaktanciája a frekvencia függvényében

$$X_{L1} + X_{C1} + X_{C0} = 0 \Rightarrow j\omega L_1 + \frac{1}{j\omega C_1} + \frac{1}{j\omega C_0} = 0 \Rightarrow j\omega L_1 + \frac{1}{j\omega(C_1 x C_0)} = 0$$

amiből

$$f_a = \frac{1}{2\pi\sqrt{L_1(C_1 \times C_0)}} = \frac{1}{2\pi\sqrt{L_1 C_1}}\sqrt{1 + \frac{C_1}{C_0}}$$

Látható, hogy a jól definiált, csak a kristály tulajdonságaitól függő f_s-től eltérően f_a értékét sajnos erősen befolyásolja a kivezetés C_0 kapacitása is, ami viszont egy nem igazán kézben tartható érték. A befolyásolás csökkentése céljából a kristállyal célszerű egy olyan C_L kondenzátort párhuzamosan kapcsolni, amelynek kapacitása sokkal nagyobb C_1-nél, így a **párhuzamos rezonanciafrekvencia** „visszacsúszik” a soros rezonanciafrekvencia közelébe:

$$X_{L1} + X_{C1} + (X_{C0} x X_{CL}) = 0 \Rightarrow j\omega L_1 + \frac{1}{j\omega C_1} + \frac{1}{j\omega(C_0 + C_L)} = 0$$

amiből

$$f_p = \frac{1}{2\pi\sqrt{L_1 C_1}}\sqrt{1 + \frac{C_1}{C_0 + C_L}} \approx \frac{1}{2\pi\sqrt{L_1 C_1}}$$

Parameter	32 kHz fundamental	200 kHz fundamental	2 MHz fundamental	30 MHz overtone
R_1	200 kΩ	2 kΩ	100 Ω	20 Ω
L_1	7000H	27H	529 mH	11 mH
C_1	0.003 pF	0.024 pF	0.012 pF	0.0026 pF
C_0	1.7 pF	9 pF	4 pF	6 pF
Q	100k	18k	54k	100k

Tipikus kvarckristály paraméterek (Fairchild)

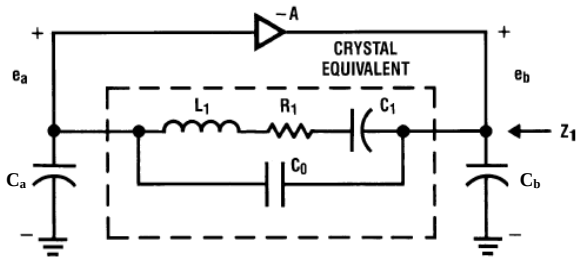
A gyártók a fenti tulajdonságokat ki is használják a rezgőkristályok paramétereinek beállításánál. Pl. a táblázatban szereplő 2 MHz-es kristály paramétereivel:

$$f_s = \frac{1}{2\pi\sqrt{L_1 C_1}} = \frac{1}{2\pi\sqrt{0.529 \cdot 0.012 \cdot 10^{-12}}} Hz = 1.997568\ MHz$$

$$f_a = \frac{1}{2\pi\sqrt{L_1 C_1}} \sqrt{1 + \frac{C_1}{C_0}} = \frac{1}{2\pi\sqrt{0.529 \cdot 0.012 \cdot 10^{-12}}} \sqrt{1 + \frac{0.012}{4}} Hz$$

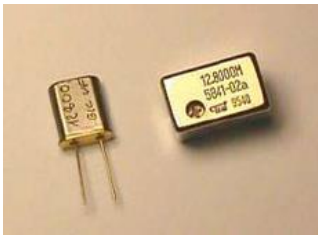
$$f_a = 2.000562\ MHz$$

Ily módon már egy igen jól használható, frekvenciastabil kapcsolást kapunk, a kvarckristályra kívül rákapcsolandó jó minőségű kerámiakapacitások értékét a gyártók általában előírják kristályaikhoz (C_a és C_b értéke tipikusan 10-30 pF, $C_L = C_a$ x C_b a két kondenzátor soros eredője). Ezekkel az elektronikában megismert ún. Pierce-oszcillátor kapcsolás:



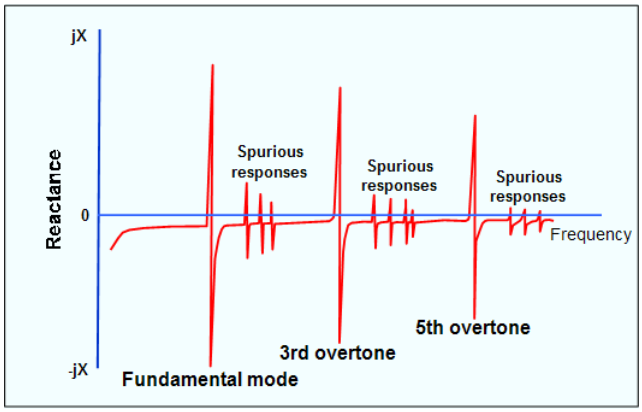
Pierce-oszcillátor kapcsolás

C_a és C_b az erősen induktív jellegű kvarckristállyal egy olyan π-szűrőt képeznek, amely közel 180 fokos fázistolást eredményez az A és a B oldal között – mindez kiegészítve az invertáló erősítővel ideálisan teljesíti a korábban bemutatott amplitúdó és fázisfeltételt.



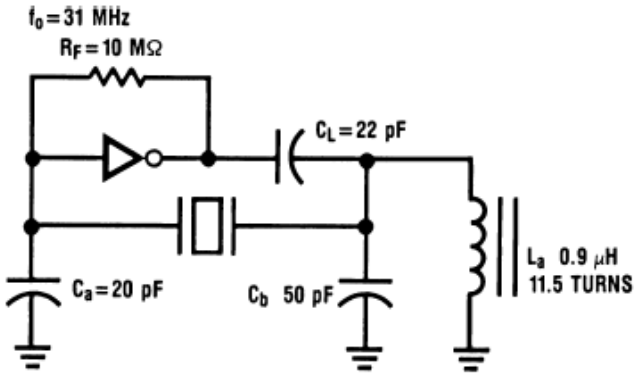
Kvarckristály és -oszcillátor

Nagyobb (kb. 25-30 MHz feletti) rezonanciafrekvenciához a kvarckristályt megoldhatatlanul kis méretűre kellene csiszolni, ezért az ilyen névleges frekvenciára szánt kvarcok általában a mechanikai rezonanciájuk valamelyik harmonikusán üzemelnek.



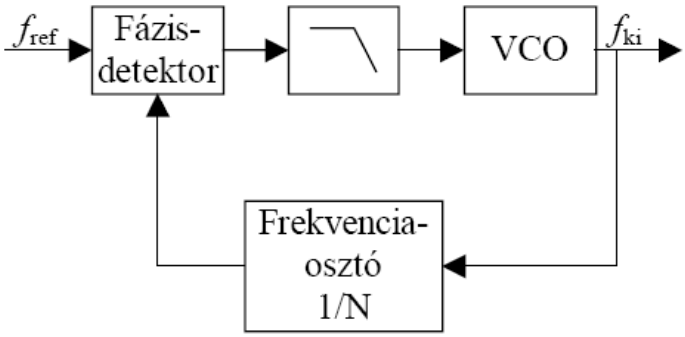
Kvarckristály felharmonikus frekvenciái

Ilyen esetben a kvarckristály hajlamos lehet más frekvenciákon (pl. az alapharmonikuson) is berezegni, ezt segédelemekkel (az alábbi ábrán C_L ill. L_a-C_b) gátoljuk meg.

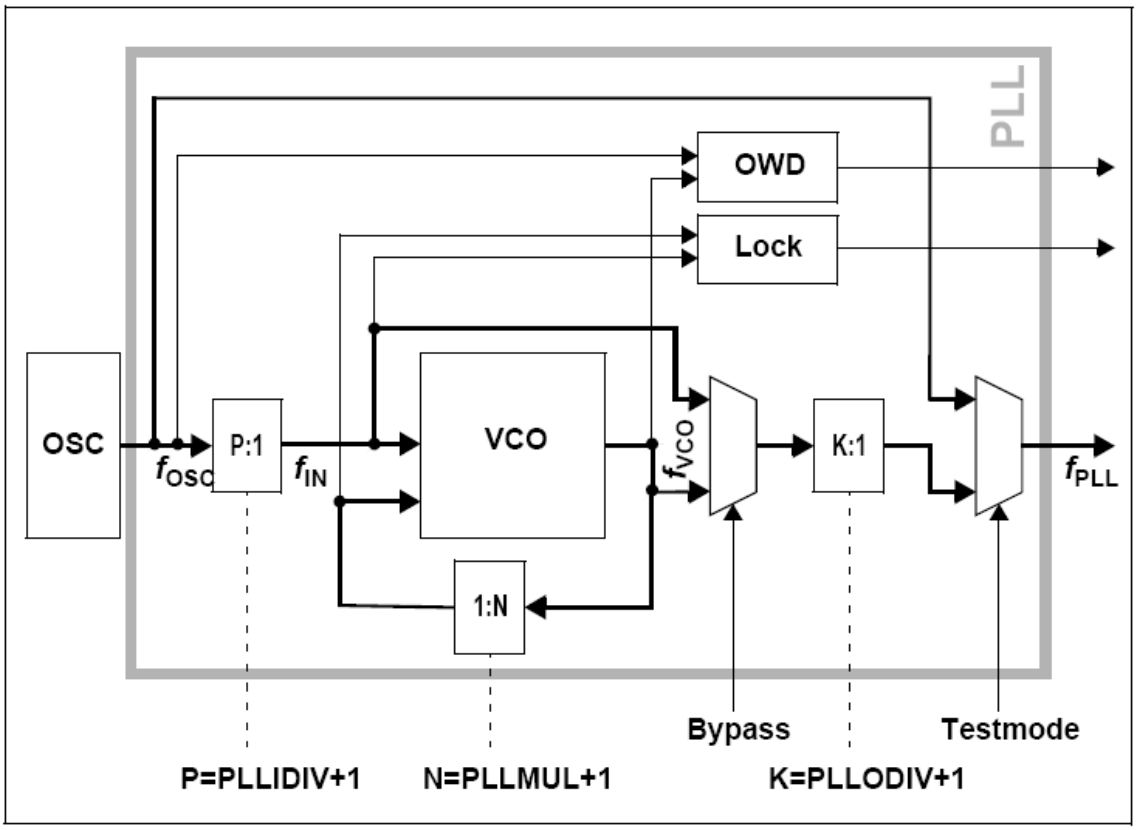


Felharmonikus kvarcoszcillátor kapcsolás

A PLL kapcsolások segítenek a működési frekvenciát tovább növelni. Magasabb órajel-frekvenciák (f > 30-50 MHz) esetében a kontrollerek már többnyire csak egy bizonyos frekvenciatartományba (pl. 1-20 MHz) illeszkedő külső kvarckristály meglétét követelik meg (ezzel megint pontosan beállíthatók a kommunikációs frekvenciák értékei), majd egy programozható belső szorzó-osztó áramkör és a jól ismert PLL alapkapcsolás segítségével ezt a frekvenciát a kontroller belsejében többszörözzük tovább. A fáziszárt hurok működési elve és legfontosabb tulajdonságai (behúzási tartomány, benntartási tartomány stb.) részletesen szerepeltek korábbi tanulmányaikban, ezekkel itt részletesen nem foglalkozunk a továbbiakban.



Fáziszárt hurok (PLL) alapkapcsolás



Fáziszárt hurok (PLL) kapcsolás mikrokontrollerben (Infineon XC164)

3.2 Reset áramkörök

A mikroprocesszorok és mikrokontrollerek működtetésének elengedhetetlen feltétele azok bekapcsolás (power-on) utáni definit alaphelyzetbe állítása.
A mikroprocesszorok bonyolult sorrendi hálózatokkal megvalósított digitális kapcsolások, a helyes működés elengedhetetlen feltétele ezen kapcsolások, állapotgépek ismert (definit) alaphelyzetbe hozása a bekapcsolás után. Hatványozottan jelentkezik ez a feltétel a mikrokontrollerek esetében, ahol az említett mikroprocesszor-maghoz kapcsolódó perifériák sokasága – hasonló felépítésükből adódóan – szintén alaphelyzetbe állítandók a rendszer indítása előtt, a bonyolult működtetési szekvenciákat is csak ismert állapotból kiindulva tudjuk programozni majd.

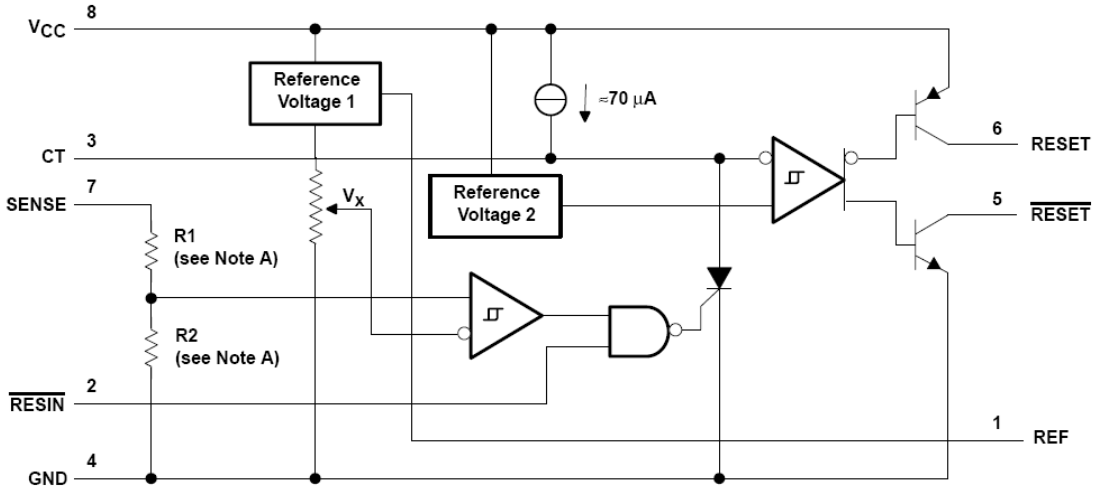
Az alaphelyzetbe állítás (a nyelvészetileg teljesen helytelen, de közkeletű szlenggel csak „resetelésnek” hívott folyamat) minden esetben szükséges lehet, mikor mikrokontrollerünk állapota a vezérlő program számára indefinit. Ennek három alapvető indoka is lehet:

1. bekapcsolás, tápkimaradás, tápbetörés stb. hatására a digitális áramkör tároló elemeinek tartalma megváltozhat és összességében a sorrendi hálózatot, az állapotgépeket számunkra indefinit helyzetbe hozhatják
2. a hardver rendszer hibás működése (pl. tervezési hazárd, nem tervezett erős külső zavarás stb.) hatására történt hibás adatátvitel (hibás utasításkód, érvénytelen, vagy hibás adat stb.) is eljuttathatja a rendszert egy általunk nem tervezett állapotba, ami okozhatja mind a hardver hálózat, mind a szoftver rendszer nem kívánatos, nem tervezett működését
3. bonyolult szoftver rendszereink minden lehetséges programágat biztonsággal le nem kezelő megoldásai is okozhatnak olyan helyzetet, mikor a teljesen helyesen működő hardveren a programrendszer „belefagy” egy nem tervezett váróhurokba vagy programállapotba. Ennek a helyzetnek jellegzetes kezelő eszköze a PC-k elején található reset nyomógomb, vagy hiányában a PC ki- és bekapcsolása.

A resetelés biztosítására céláramköröket alkalmazunk, melyek – létezésük elengedhetetlen volta miatt - a többi egységhez hasonlóan egyre inkább „vándorolnak befelé” a mikrokontroller kapcsolásokba. Feladataik a következő lényeges pontokban foglalhatók össze:

a) a tápfeszültség folyamatos figyelése. Ezzel mind a bekapcsolás, mind a kikapcsolás, mind az esetleges tápbetörések (alapvetően a tápfeszültségnek a biztonságos működés számára előírt határokon kívülre kerülése) okozta bizonytalan műveletek és működések elkerülhetők
b) az alaphelyzetbe állítás meghatározott időszükséglettel jár a mikrokontrollerek esetében – feladatuk az előírt időszükséglet biztosítása reset állapotban
c) külső reset aktiválási lehetőség (központi reset jel, reset nyomógomb) esetén ennek pergésmentes, időzített illesztése a mikrokontrollerhez

Nézzük először a tipikus külső megoldásokat! A reset áramkörök tipikusan egyszerű, kis lábszámú áramköri megoldások, melyek mégis nem elhanyagolható bonyolultságú kapcsolást rejtenek magukban:

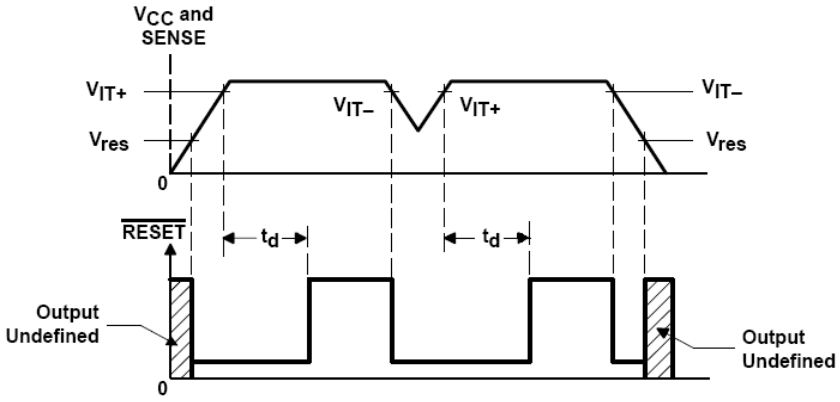


A Texas Instruments TL7705B reset áramkörének belső kapcsolása

Az áramkör főbb funkcionális egységei a következők:

- Precíziós referencia feszültség forrás a SENSE bemeneten megjelenő tápfeszültség alsó küszöbérték – Vcc-ből szükség esetén ellenállásosztóval előállított feszültségérték – figyelése (Reference Voltage 1, komparátor)
- Az így előállított reset feltétel összekapcsolása a kívülről érkező /RESIN alaphelyzetbe állító jellel (NAND kapu)
- Áramgenerátor a CT külső időzítő kondenzátor töltésére, referencia feszültség a pontos időméréshez és kisütő elem a reset folyamat definit állapotból való indításához (70 µA, Reference Voltage 2, tirisztor, komparátor)
- Kettős végfokozat a logikai „1” és a logikai „0” reset jelet követelő áramkörök mindegyikéhez való alkalmazhatóság érdekében.

A feladat már így sem olyan egyszerű, mint azt első ránézésre gondolnánk, pedig a legnehezebb feltételt nem is említettük. A tápfeszültség be- és kikapcsolásakor is a legveszélyesebb állapot ugyanis az, hogy a tápfeszültség felépülése és eltűnése is olyan, nem elhanyagolható időt igénylő folyamatok, melyek alatt – közbenső Vcc értékeken – a viszonylag tág feszültséghatárok között működőképes CMOS kapcsolások már elkezdik működésüket, viszont – éppen az előírt feszültséghatárok be nem tartása miatt – egyáltalán nem biztos, hogy az előírásoknak megfelelő módon! A gyártók úgy tervezik meg a mikrokontrollereket, hogy azok be- és kimenetei a tápfeszültség alsó határának eléréséig garantáltan definit állapotban vannak, de csak abban az esetben, ha kontrollerünk reset bemenete mindvégig aktív! Mi is olyan nehéz ebben a teljesen természetesnek tűnő feltételben? A „mindvégig” szó. Nézzük az előbb megismert áramkör adatlapján szereplő feszültség diagramot:



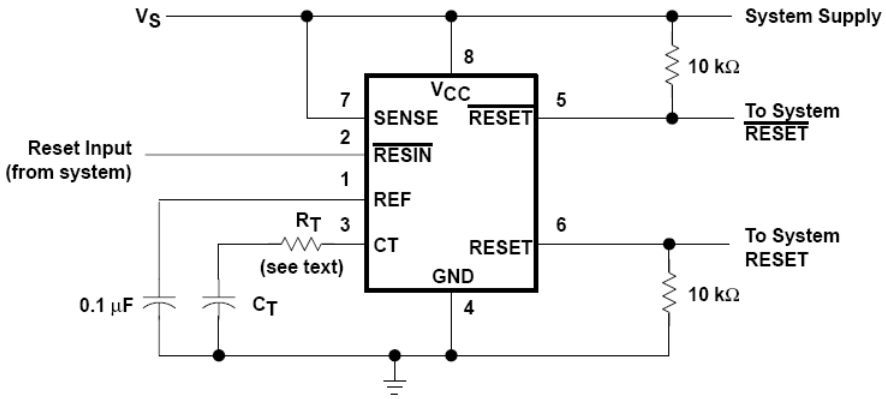
A TL7705B működési tartománya

Valószínűleg mindenki felkapja a fejét az „output undefined” jelzésre, márpedig a kérdés igen egyszerű: mi az a minimális tápfeszültség, ami felett a reset áramkör kimenetei már garantáltan definit (reset) állapotban tartják kontrollerünket? A válasz eléggé megnyugtató: az áramkörök többségében ez a szint Vres ≈ 1V. A CMOS áramkörök és kapcsolások technológiai okok miatt sem működésképesek kb. 1.2.-1.5V feszültség alatt, tehát az 1V-os alsó határ megnyugtató megoldásnak tűnik esetünkben. (Ez a határ 5V-os és 3.3V-os logikák esetén igaz. Az egyre alacsonyabb tápfeszültségek miatt az érték egyre inkább csúszik lefelé az utóbbi időben – konkrét alkalmazás esetében ellenőrzendő !)

Az áramkör működtetéséhez minimális külső áramköri alkatrészre lesz szükség. CT értéke az alkalmazástól függ, a reset állapot időtartamának beállítására szolgál

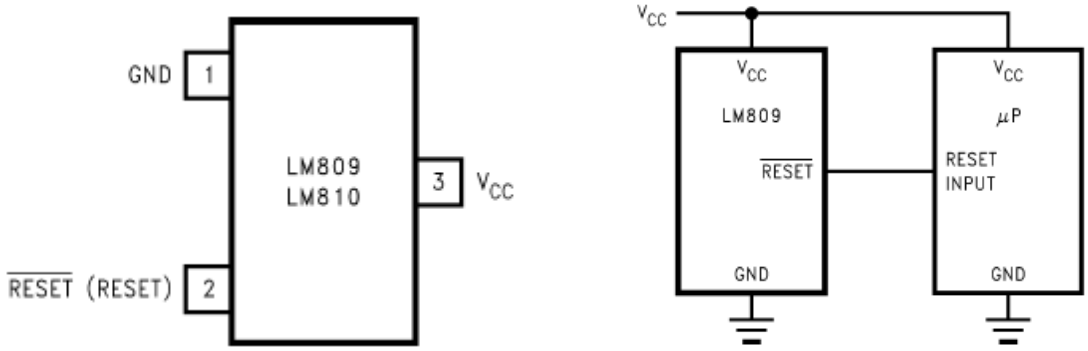
$$t_d[sec]=2.6*10^4*C_T[F],$$

R_T a kisütő áram korlátozására szolgál (az alkalmazott CT kapacitás nagyságának függvénye, hogy szükséges-e).



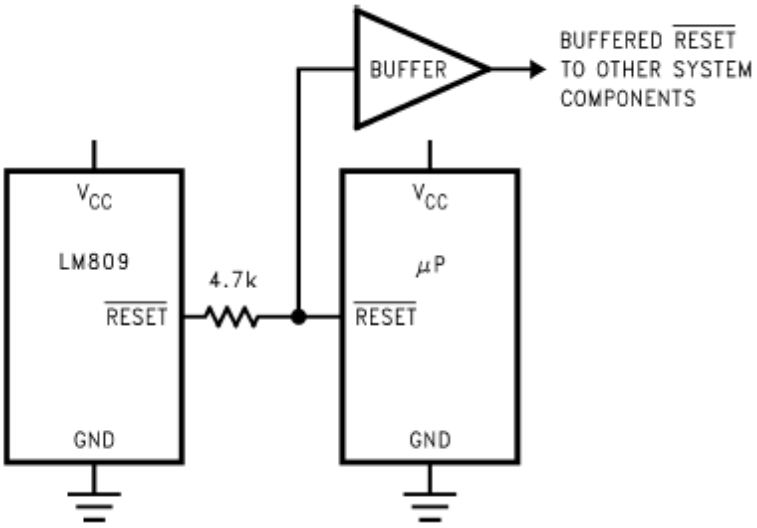
A TL7705B tipikus kapcsolása

Fenti áramkörünk létezik sokkal egyszerűbb kivitelben is (amennyiben nem kívánunk foglalkozni a különböző paraméterek beállításával):



Az LM809 (LM810) reset áramkör SOT23(„tranzisztor”) tokban

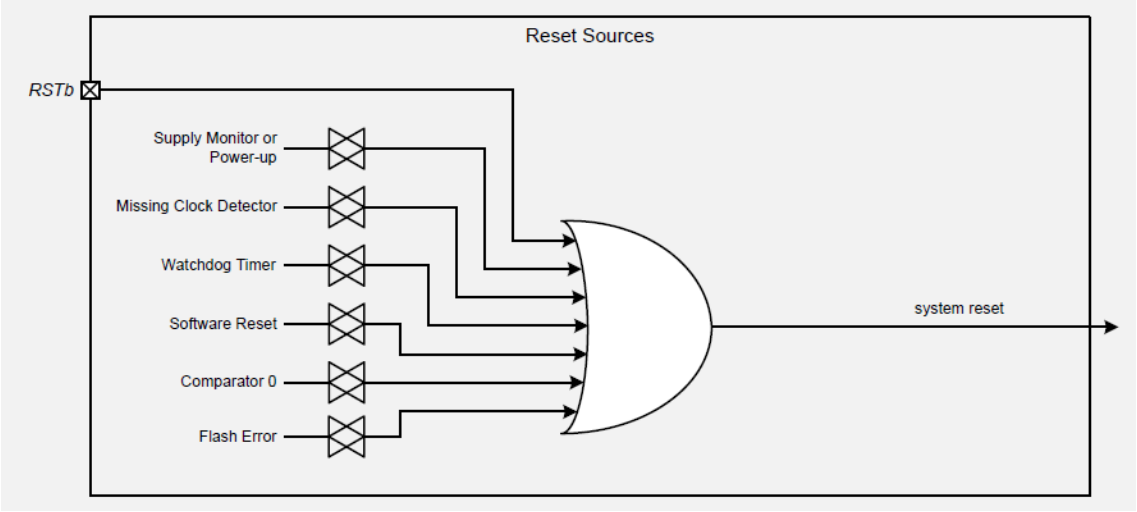
A két típus között a negált (LM809) ill. a ponált (LM810) reset kimenet tesz csak különbséget. Feszültségszintjei és időkésleltetése fixek (pl. 4.63V és kb. 250 msec), kimenete szintén ≥ 1V-tól definit.



Az LM809 reset áramkör tipikus kapcsolása

Mindkét logikai jelszinten meghajtja kimenetét, ezért olyan kapcsolásokban, ahol a mikroprocesszor RESET lába aktív is lehet (ld. később) gondoskodnunk kell az összehajtás megakadályozásáról. További külső egységek alaphelyzetbe állítása esetén buffer meghajtót alkalmazhatunk, de a tervezésnél ügyeljünk arra is, hogy ezen kapuáramkör milyen tápfeszültségig működőképes! (Tipikusan HC, AHC stb. logikai elemcsaládból származó meghajtó áramkört alkalmazzunk, melyek már Vcc ≥ 2V –tól működőképesek!)

Hogyan változik mindez a mai korszerű mikrokontrollerek világában? A Silicon Laboratories korábban már megismert 8051-es típusában (EFM8BB3) már az alábbi integrált logikát találjuk az /RST (reset) láb mögött:



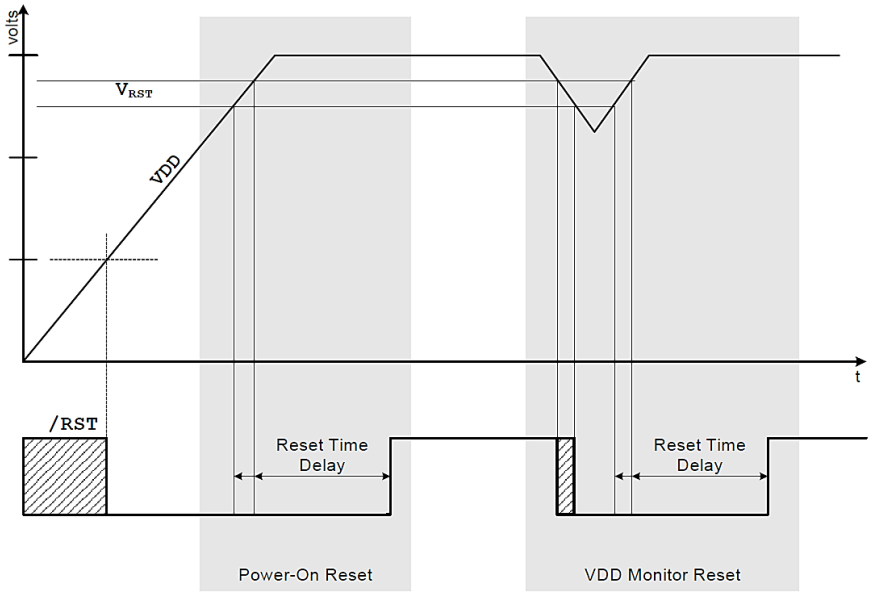
Az EFM8BB3 mikrokontroller belső reset logikája

A teljes kapcsolás kapcsolás a következő okokból válthat ki újraindítást:

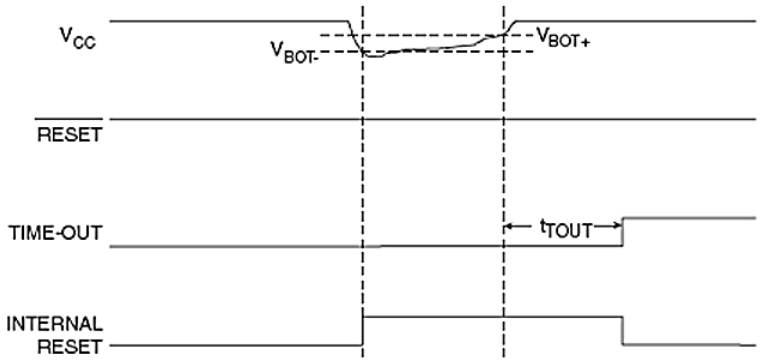
- a tápfeszültség megengedett határértékének átlépése (Supply Monitor),
- külső reset jel hatására (RSTb),
- programból történt kezdeményezésre (Software Reset),
- egy komparátor áramkörrel figyelt tetszőleges analóg jel hatására (Comparator 0),
- az órajel tartós (625 µs – 1.2 ms) kimaradása miatt (Missing Clock Detector),
- a Watchdog áramkör aktíválódása miatt (később még foglalkozunk vele),
- a programtároló memória illegális címen vagy hiányzó jogosultsággal történő hozzáférése miatt (Flash Error).

Ismerősként üdvözölhetjük az áramkör működését bemutató feszültség-idődiagramot is, melynek lényeges elemei az adatlapon található értékek alapján megegyeznek a korábban bemutatott külső reset áramkörök paramétereivel (működés alsó határa kb. 1.4V, V_{DD} alsó határértéke 1.95V – ne tévesszük szem elől, hogy a mikrokontroller névleges tápfeszültsége V_{DD} = 2.2 - 3.6V). Megjegyezzük, hogy az itt „Supply monitor”-nak hívott funkcionális egységet sok mikrokontroller típus „Brownout-reset” áramkörnek hívja az energiaellátásban megszokott terminológiából származó elnevezés alapján:

- **Blackout**: a tápellátás tartós teljes elvesztése, kiesése
- **Brownout**: a tápfeszültségnek a rendszer működéséhez szükséges minimális határa alá esése (ide tartozik pl. energiaellátó hálózatokban a fáziskimaradás)
- **Dropout**: a tápellátás rövid időre (msec – sec) történő kiesése



Az EFM8BB3 mikrokontroller V_{DD} monitorának feszültségdiagramja

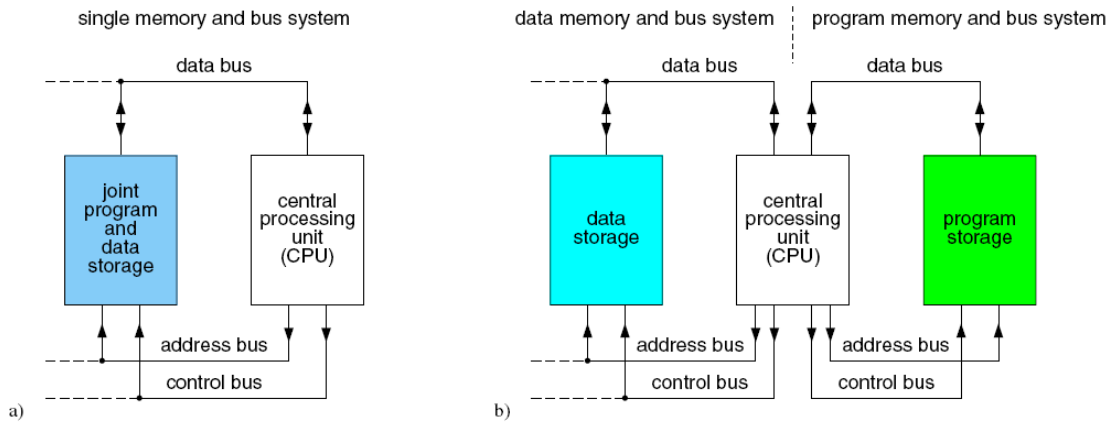


Brownout reset

3.3 Mikrokontrollerek belső memória egységei

3.3.1 Architektúrális kialakítás

A mikroszámítógépek működtetéséhez memória elemekre van szükségünk. Első megközelítésben – a működésben ellátott szerepük alapján – két fő funkcionális típusukat kell megkülönböztetnünk: a program tárolására és az adatok tárolására szolgáló egységet. Ezek illesztési módja alapján különböztetjük meg először is a legfontosabb számítógép architektúrákat:



A Neumann (a) és a Harvard (b) architektúra

A **Neumann architektúra** (Neumann János, 1903-1957) legfontosabb tulajdonsága, hogy közös buszrendszerre illeszkedő memóriákban tárolja a programot és az adatot is. Ebben nem a memória egység „közös” voltán van a hangsúly (lehet a program- és az adatmemória fizikailag két elkülönülő áramköri egység), hanem a közös buszrendszeren. Ezzel a mikroszámítógép buszrendszere felhasználói szempontból egyszerűsödik (egyetlen cím- és adatbusz kezeli a memóriákat), csökken a szükséges kivezetések száma, akár egyetlen megfelelő méretű memória is elegendő lehet számítógépünk működtetéséhez. Hátránya viszont a fentiekből következően, hogy egy program módosítani képes saját magát (ezt trükkös programozók előnyként is szokták emlegetni, de gondoljunk csak arra, hogy egy véletlenül vagy szándékosan eltévedt program mekkora pusztítást tud csinálni a memóriában…), egyértelműen szűk keresztmetszetet alakít ki a processzor és a memória között, mivel ugyanazon az adatcsatornán kell továbbítanunk utasítást, forrás- és céloperandust is. A Neumann architektúra következményeként emlegetik a cache-memóriák kialakulását (ez a kijelentés nem teljesen egyértelmű és több szempontból is kritizálható).

A **Harvard architektúra** elnevezése a Harvard egyetemhez kötődik, a mikroprocesszorok korai történetében is láttuk már, hogy az egyetem fontos szerepet játszott a kutatásokban és a fontosabb elvek kialakításában (ld. Mark I számítógép). A Harvard architektúra elkülöníti nemcsak a tároló elemeket, hanem ami sokkal fontosabb, a hozzájuk vezető adatutakat is. Fentiek egyenes következményei a Harvard architektúra legfontosabb előnyei:

- a kódmemória adathozzáférések számára nem elérhető, az architektúra nem teszi lehetővé önmagukat módosító programok írását (ún. dinamikus programozást). Ugyanez az elv természetesen gátolja az adatkonstansok programmemóriában való tárolását is.

- kevésbé jelent szűk keresztmetszetet az utasítások és az adatok kiszolgálása szempontjából
- lehetővé teszi eltérő méretű kód- és adatmemória buszok kialakítását is (az integrált belső programmemóriák miatt szokatlan – 14, 18, 20 bites utasításkódok is megjelentek, és ezek nem igazán zavarnak senkit).

A másodikként említett pont (adatok kiszolgálásának gyorsítása) leginkább az oka annak, hogy mind a mikrokontrollerek, mind a DSP-k – de ugyanígy említhetnénk a RISC gépek jelentős részét is –, elsősorban a Harvard architektúrát használják. Az első pont az általános célú számítógépek (PC-k) használhatósága tekintetében komoly hátrányt jelent, annyira, hogy a személyi számítógépek szinte kizárólag Neumann architektúrával rendelkeznek. A mikrokontrollerek világában is megjelent az „áthidaló megoldás”: ez az ún. **módosított Harvard architektúra,** ahol már elkülönített utasítások (8051: MOVC) segítségével lehetővé teszik a kódmemória adatként való olvasását is.

3.3.2 Memóriatípusok

A mikrokontrollerek alapelveikből adódóan is megpróbálják integrálni a memóriákat az áramkör belsejében, ennek legfontosabb okai a következők:

- lényegesen egyszerűsödnek az alkalmazások (nem kell memóriát illeszteni)
- gyorsabb a hozzáférés a belső memóriákhoz
- a kontroller lábkivezetéseit nem foglalják le a terjedelmes cím- és adatbuszok, ezek szabadon felhasználhatók a beépített perifériák be- és kimenetei számára

A mai mikrokontrollerek leginkább négy memóriatípust alkalmaznak integrált memóriaként:

1. Kódmemóriaként elsősorban a **flash memóriák** jönnek szóba. Mivel a XIP jelleg (eXecute In Place) elengedhetetlen követelmény a programkód végrehajtásához, a mikrokontrollerek világában a NOR típusú flash memóriák jöhetnek csak szóba (a nagyobb tárolási sűrűséget lehetővé tevő NAND flash memóriák csak nagytömegű adattárolás céljaira szolgálnak). A flash típusú memória komoly jelentőséggel bír majd az ISP (In System Programmable) jelleg kialakításánál és a firmware aktualizálhatósága (update) szempontjából is. A későbbiekben ezzel a témakörrel még részletesen foglalkozunk.
2. Szintén kódmemóriaként azonos kapacitások esetében az előzőnél olcsóbb megoldás az **OTP ROM** (On Time Programmable Read Only Memory) memóriák alkalmazása, ami a korábbi tanulmányaikban megismert PROM memóriák mai megfelelője. Nagyobb, változatlan kódtartalmú szériák esetében vitathatatlanul a legolcsóbb megoldás, viszont nehézkessé teszi a fejlesztést, és kizárja az ISP lehetőségét és a firmware aktualizálhatóságát (update) is. Ilyen memóriát tartalmazó áramkörök fejlesztése során ICE (In Circuit Emulator) egységekre lesz szükségünk.
3. Adatmemóriaként a mikrokontrollerek kizárólagosan **SRAM** memóriákat alkalmaznak. Az adatmemóriák mérete messze nincs akkora, hogy megérné a dinamikus RAM bármilyen formájának kialakítása a chip belsejében.

4. Nem felejtő (non-volatile) adatmemóriaként – elsősorban paraméterek, mért adatok, stb. tárolására – sok mikrokontrollerbe integrálnak nem túl nagy kapacitású (1-8 kbájt) **EEPROM** memóriákat. Jellegüknél fogva ezeknek a memóriáknak az írása lényegesen lassabb a szokásos memória-hozzáférési műveleteknél (msec nagyságrend), ezért itt külön gondot kell fordítanunk az adattárolás biztonságára, adatblokkok esetén az adatok konzisztens voltára. Ez csak akkor fog biztonsággal sikerülni, ha valamilyen előjelzéssel (tip. 10-50 msec) rendelkezünk a tápellátás várható összeomlásáról (erről is szó lesz még a későbbiekben).

3.3.3 Memóriák szóhosszúsága

A mikrokontrollerek igen nagy szeletet foglalnak le a beágyazott alkalmazások világában. A mai napig komoly létjogosultsággal rendelkeznek a 8 bites mikrokontrollerek, nagymértékben terjednek a 16 bites alkalmazások (a nagyobb számok kezelhetősége lényegesen gyorsítja az algoritmusok végrehajtását), a nagyobb teljesítményű alkalmazásokban – az ARM-ok és a DSP-k világában – a 32 bites kialakítás is gyakori. Gondoljunk csak bele, hogy ezzel a szóhosszúsággal válik egyáltalán „értelmezhetővé" a lebegőpontos számok világa, sőt még ez sem elegendő pontosságú terjedelmesebb műveletsorok elvégzéséhez.

Általában igaz az az alapelv, hogy a memória szervezése igazodik az alkalmazott szóhosszúsághoz. A szóhossznál rövidebb adategységek kezelése nem kizárt ugyan, azonban ezek ún. nem illesztett (non-aligned) elhelyezése – pl. bájtszervezésű memória esetén szó vagy hosszúszó páratlan címen történő elhelyezhetősége – nagyon elbonyolítja a CPU mag adatbusz-illesztő egységének felépítését (ún. byte-swapperek alkalmazása válik szükségessé, folytató ciklusokat kell bevezetnünk, stb.), ezeket az egyszerűbb felépítésű mikrokontrollerek többsége nem teszi lehetővé, a DSP-k alapelvei (maximális sebesség) pedig teljesen kizárják.

3.3.4 Memóriakapacitások

Nehéz ebből a szempontból általános érvényű számokat említeni, mivel az alkalmazások széles köre miatt a mikrokontroller gyártók tudatosan kínálnak egyes típusokból kisebb és nagyobb kapacitású variánsokat is. Hozzávetőleges számok a következők:

- kódmemória: 8-128 kbájt, nagyobb teljesítményű (16-32 bites) kontrollerek esetében ez akár 512 kbájt-1 Mbájt is lehet
- adatmemória: 1-64 kszó (a „szó" fogalma a mikrokontroller mindenkori szóhosszához illeszkedik, tehát bájt is lehet)
- non-volatile adatmemória: 0-16 kbájt

3.4 Időzítő és számláló egységek

Bármely beágyazott rendszer legalapvetőbb perifériái az időzítők (timerek). Minden, a környezetével valamilyen kapcsolatban álló rendszer esetében a feladatok ellátása során időhöz kötött részfeladatok sorozata jelenik meg. Ilyenek:

- A szoftver rendszer alapidőzítésének ellátása. A determinisztikus, kiszámítható időzítés a legtöbb beágyazott irányító rendszerben követelmény. Egy szabályozási feladatot ellátó irányító hurok tulajdonságai nehezen befolyásolhatók, ha azok paraméterei nem determinisztikus, időhöz kötődő fogalmak (időállandó, integrálási és differenciálási idő, stb.). A „csináljunk mindent olyan gyorsan, ahogyan csak lehet” bizonyos vezérlési feladatoknál megengedhető, de leginkább csak a háttérhurokban megvalósítható elv.
- Környezetünkhöz való kapcsolataink is általában időhöz kötődnek. Egyenletesen („ekvidisztáns időpontokban”) szeretnénk mintavételezni bemeneteinket, egy kijelző villogása, egy timeout figyelése időhöz kötődő műveletek. Az impulzus szélesség modulációval (ld. később) vezérelt kimenetek periódusideje, impulzusszélessége determinisztikus időfogalom használatát követelik meg.
- Egységeink közötti kommunikációt (pl. a szinkron vagy aszinkron soros átvitel) meghatározott időzítéssel bonyolítjuk
- Beágyazott operációs rendszereinkben a párhuzamosan futó feladatok az eseményvezérlésen túl időkorlát szerint ütemeződnek,
- stb.

Időzítő nélkül gyakorlatilag csak programutasítások sokaságával tudunk meghatározott időkésleltetést előállítani. A szoftver időzítés használata egy olyan programozói csapda, amiről mindenkit a leghatározottabban csak lebeszélni lehet. Gondoljuk végig a következő szempontokat:

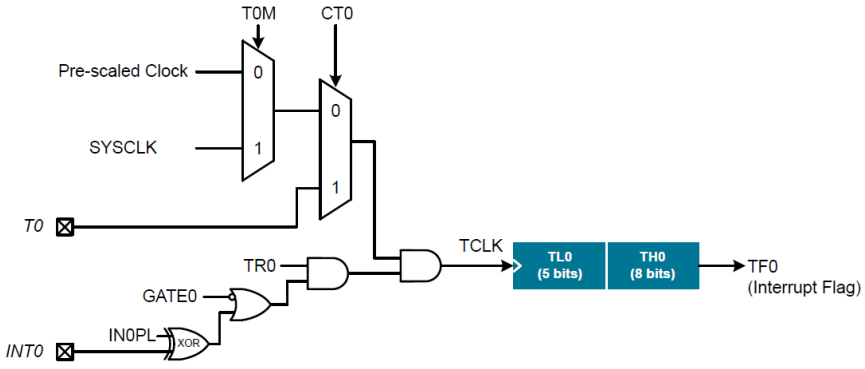
- egy programutasításokból álló késleltető hurok teljes mértékben lefoglalja a központi egységet a „semmittevéssel”. Eltekintve a gyors próbaprogramok esetétől egyszerűen belátható, hogy egy rendszer a legritkább esetben engedheti meg magának azt, hogy pl. egy timeout időzítése alatt ne figyeljen oda egyéb bekövetkező eseményekre, vagy ne lásson el további feladatokat.
- egy rendszerben általában több időzítés is zajlik egyszerre (alapidő, villogó kimenet, timeout figyelés stb.), a legtöbb esetben nem megvalósítható feladat ezen független időzítések egyetlen időzítő hurokkal történő meghatározása.
- aszinkron események kezelésére a legkézenfekvőbb megoldás a megszakítások használata. Az időzítő hurkunkat az előre nem kiszámítható időpontban, változó időtartamra megszakító rutinok teljesen kiszámíthatatlanná teszik, hiszen szoftver időzítés esetén időzítésünk alapja nem a tényleges idő, hanem az előre adott számú utasítás lefuttatása („tart, ameddig tart”)
- rendszerünk bővítése, átalakítása esetén (pl. új PC-t vásároltunk, nagyobb teljesítményű, gyorsabb, más architektúrájú processzorral) az utasítások végrehajtási idejére alapozott időzítés alapvetően megváltozhat. Akkora

programnagyhatalmak, mind a Borland cég is csődöt mondtak a rendszerükben szoftver időzítésen alapuló funkció (Delay függvény) működése tekintetében (a 286-os, 386-os processzorokra méretezett önhangoló időzítés számlálója túlcsordult a 486-os processzorokba integrált cache megjelenésével, ami a helyi kis programhurkokat akár százszoros sebességre is felgyorsíthatta).

Alapszabályként tehát fogadjuk el a hardver időzítőkön alapuló időmérés, és időmeghatározás alapelvét, és jegyezzük meg azt az alapszabályt, hogy szoftver időzítés használata egy időzített bomba programrendszerünkben, ami bármikor robbanhat és a rendszer összeomlásához vezethet!

3.4.1 Az alapvető időzítő (timer) üzemmódok

a) egyszeri lefutás (8051 Mode 0 és 1)



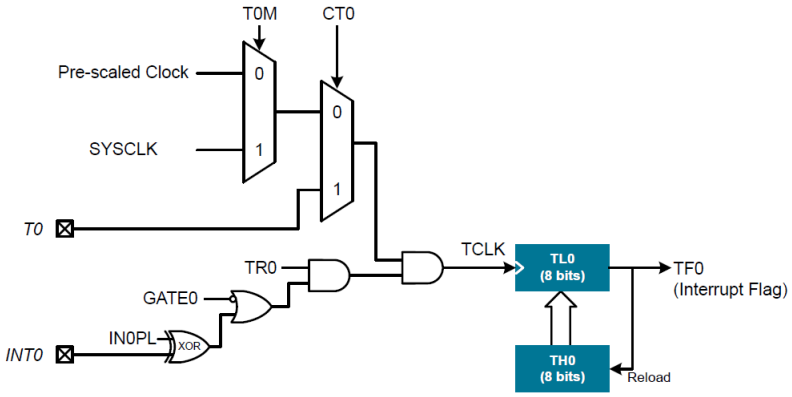
8051 Timer 0 (Mode 0)

Az időzítő vagy a rendszer órajelet (SYSCLK), vagy annak előosztott értékét (az ún. prescaler értéke 4, 8, 12 vagy 48 lehet) számolja, vagy a T0 bemenetre kapcsolt impulzusokat (TR0 és GATE0 engedélyező bitek a vezérlő regiszterekben). Mode 0 és Mode 1 között a különbséget csak az jelenti, hogy 13 vagy 16 bites a számláló. Ne keressünk logikát ebben: az „ős” 8051-es a 13 bites számlálóval a 8048-as kontrollerrel őrizte a kompatibilitást, az utód 51-esek pedig az „eredeti” 8051-gyel – ilyen következményekkel jár a teljes kompatibilitás megőrzése egy sok-sok évvel ezelőtti típussal. A számláló a beállított kezdőértéktől kezdődően felfelé számol (!), amennyiben eléri az 1FFFh (13 bit) vagy FFFFh (16 bit) értéket, akkor túlcsordul 0-ra és – amennyiben ez engedélyezve van – megszakítást kér.
Az időzítés pontossága mindig a – leosztott – órajel periódusideje, a maximális beállítható időzítés egyszerűen számítható a számláló hossza alapján.

CLK = 50 MHz	Időzítés pontossága	Max. késleltetés (Mode 0)	Max. késleltetés (Mode 1)
SYSCLK	20 nsec	163.84 µsec	1.3107 msec
SYSCLK/4	80 nsec	655.36 msec	5.2429 msec
SYSCLK/8	160 nsec	1.3107 msec	10.4858 msec
SYSCLK/12	240 nsec	1.9661 msec	15.7286 msec
SYSCLK/48	960 nsec	7.8643 msec	62.9146 msec

Az egyszeri lefutású időzítés viszonylag ritkán használatos, inkább a periodikus időzítés üzemmódjait szoktuk előnyben részesíteni. Ne használjuk ezt az üzemmódot soha arra, hogy a túlcsordulás hatására bekövetkező megszakításban újratöltjük, és ezzel újraindítjuk a számlálót! A megszakítás érvényre jutása bizonytalan lehet, (utasítást nem szakíthat meg, magasabb prioritású megszakítást nem tud megszakítani, amennyiben rövid időre tiltjuk a megszakításkérést, az nem tud érvényre jutni), az így létrehozott periodikus megszakításkérés periódusideje csak adott bizonytalansággal kalkulálható és a programrendszer pillanatnyi állapotától függ!

b) periodikus lefutás (8051 Mode 2)



8051 Timer 0 (Mode 2)

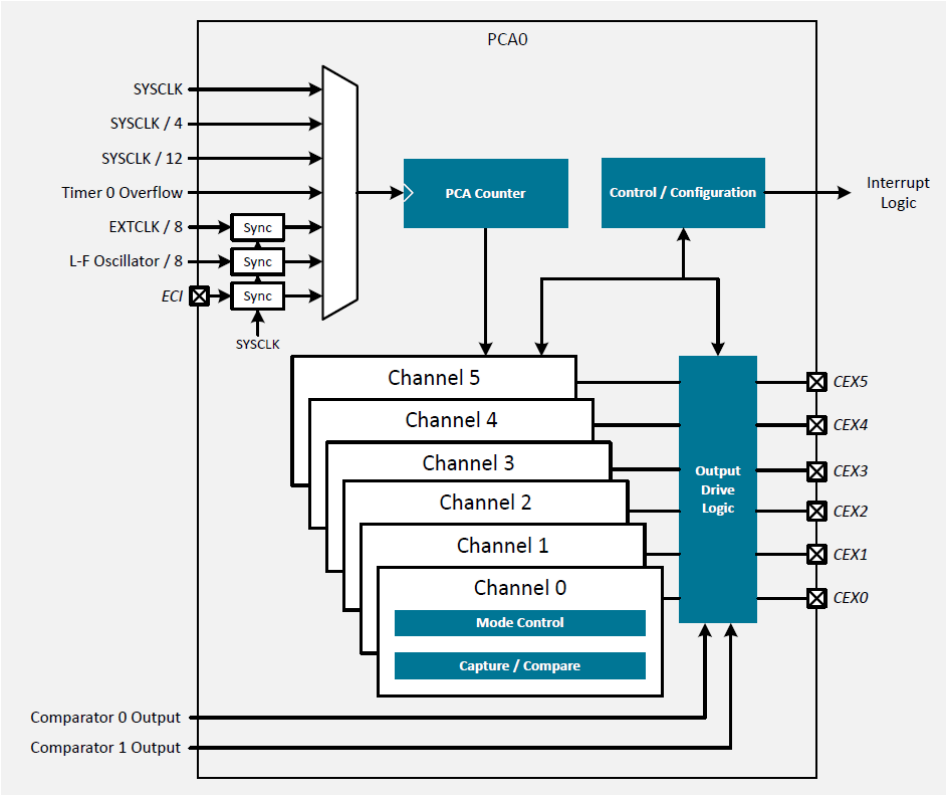
Ebben az üzemmódban a timer csak 8 bites számláló lehet, amely a megadott kezdőértékről (TL0) számol felfelé, majd az FFh ⇒ 00h túlcsordulás hatására a TL0 regiszter tartalma automatikusan felülíródik a kezdőértékkel (TH0), melynek értéke ilyenkor mindvégig változatlan. A beállítható időzítési értékek a 8 bites számlálóból adódóan sajnos elég rövidek (16 bites számláló esetében ld. a korábbi Mode 1 táblázatoszlop számértékeit).

CLK = 50 MHz	Időzítés pontossága	Max. késleltetés (Mode 2)
SYSCLK	20 nsec	5.120 µsec
SYSCLK/4	80 nsec	20.480 µsec
SYSCLK/8	160 nsec	40.960 µsec
SYSCLK/12	240 nsec	61.440 µsec
SYSCLK/48	960 nsec	245.760 µsec

Ezt az üzemmódot használjuk a leggyakrabban periodikus rendszeridőzítés beállítására. Egy rendszer alapidő (pl. 1 msec, vagy 10 msec) szoftver számláló segítségével könnyedén továbbosztható nagyobb időzítések elérése érdekében. Ez a módszer nem tévesztendő össze az utasításokból összeállított késleltető ciklusok (ún. szoftver időzítés) módszerével! Ilyenkor az időzítésünk alapját hardver idő képezi, annak értékét a megkívánt legkisebb időkvantum, vagy időfelbontás értékére célszerű beállítani. Az így beállított időzítések nem foglalják a CPU idejét (ettől nem szoftver időzítés!), az

ekvidisztáns időközöket a számláló hardver úton történő újratöltése biztosítja. A megszakításkérések időpontjai „ingadozhatnak” ugyan a tényleges érvényre jutás függvényében, ez azonban nem befolyásolja a következő késleltetés idejét – és ezzel az átlagos periódusidőt – mindaddig, ameddig a tiltott állapot miatti késleltetett érvényre jutás nem éri el a következő megszakítás időpontját. Ha ez bekövetkezett, akkor elvesztettünk megszakításkérést, a rendszeridő „felborult”, ez csak hibásan tervezett és méretezett rendszerben fordulhat elő.

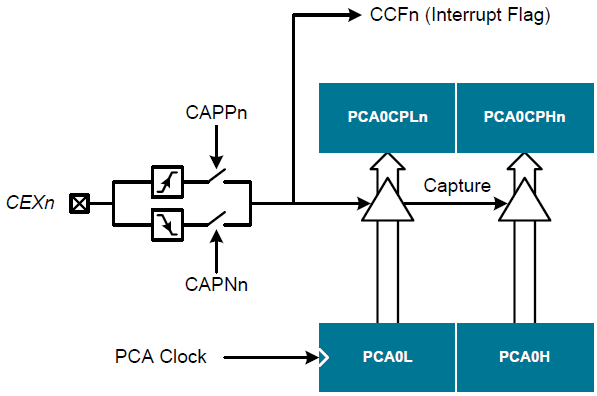
Korszerűbb időzítő/számláló modulok (EFM8BB3 PCA0)



Az EFM8BB3 PCA0 modul felépítése

A korszerű mikrokontrollerek időzítő/számláló egységei fentieknél általában többre is képesek. A tanult EFM8BB3 család mikrokontrollerei tartalmaznak egy olyan PCA0 (Programmable Counter Array) egységet, amely egy közös időzítő órajel (SYSCLK és néhány leosztott értéke, vagy Timer 0 mint programozott előosztó, vagy külső számláló bemenet/órajel) időzítésével 6 függetlenül programozható modul (Channel 0..5) segítségével sokkal „nehezebb” feladatok ellátásával is képes hardver úton támogatni szoftver rendszerünket. A teljesség igénye nélkül (részletesen ld. a hivatkozott mikrokontroller leírásában) az ilyen modulok legtipikusabb funkcióit emeljük csak ki.

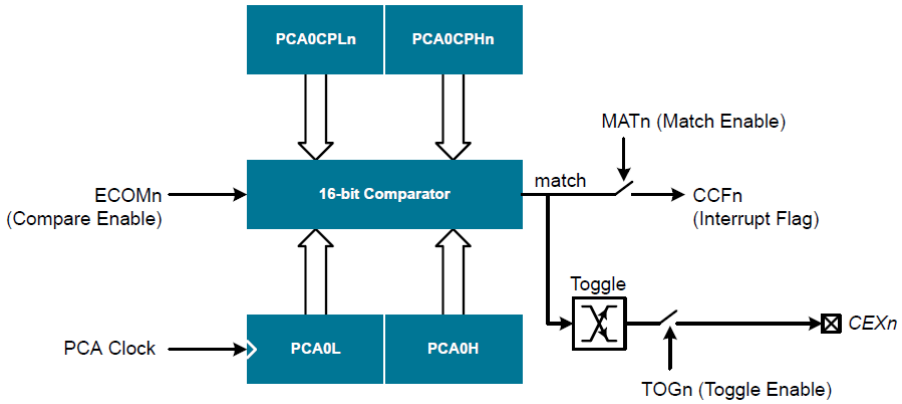
d) capture funkció



EFM8BB3 PCA0 modul capture mód

A capture (magyarul: „elcsíp, elkap, elfog”) modulok feladata egy adott (indítási) időponthoz képest bekövetkezett hardver jelváltozás időpontjának pontos megmérése (időtartammérés). Szabadon futó számlálónk (PCA0H, PCA0L) pillanatnyi állapotát hardver sebességgel tárolhatjuk el egy átmeneti tárolóban (PCA0CPH, PCA0CPL), a tárolás ténye megszakítást vált ki, és a valamilyen késleltetéssel érvényre jutó megszakításkérés rutinjában a hardver pontossággal eltárolt időpontot olvashatjuk ki az átmeneti tárolóból. Az időmérés pontossága elvben az alkalmazott (PCA) órajel periódusideje (SYSCLK=50 MHz esetében ez akár 20 nsec is lehet), az abszolút pontosságot csak az időmérés indításának esetleges bizonytalansága kérdőjelezi meg valamilyen mértékben. A jelváltozás fel- vagy lefutó élének időpontja is mérhető.

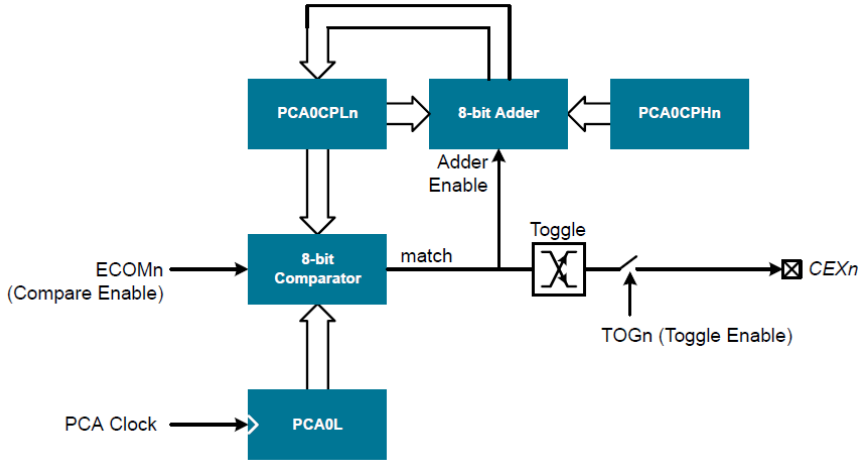
e) időzítő nagysebességű kimenettel



EFM8BB3 PCA0 modul időzítő + nagysebességű kimenet

Szoftver időzítő módban a szabadon futó 16 bites számláló (PCA0H és PCA0L) értéke folyamatosan összehasonlításra kerül a PCA0CPH, PCA0CPL 16 bites regiszter értékével. Egyezés esetén megszakítás generálódik és a nagysebességű kimenet ellenkező logikai szintre vált.

f) frekvencia generátor mód



EFM8BB3 PCA0 modul frekvencia generátor mód

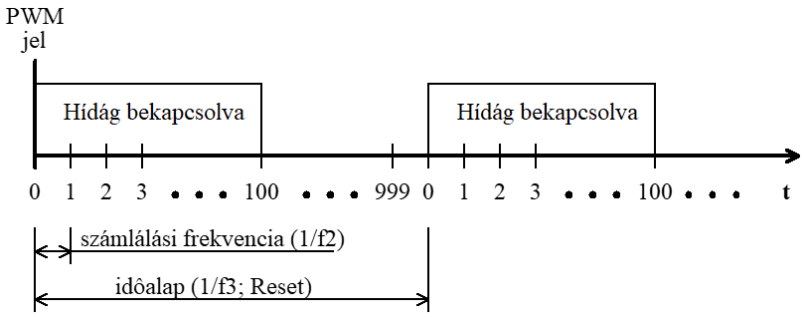
Ebben az üzemmódban

$$f_{SQR} = \frac{f_{PCA}}{2 \times PCA0CPH}$$

frekvenciájú négyszögjelet tudunk generálni az adott modulhoz rendelt kimeneten. A modul félperiódusidő-regisztere (PCA0CPH) minden egyezéskor hozzáadásra kerül a következő időpontot meghatározó PCA0CPL regiszterhez, amelyet a komparátor folyamatosan összehasonlít a PCA egység szabadon futó számlálójával (PCA0L). Egyezés esetén a kimenet ellenkezőjére vált, és a vizsgált regiszter értéke „megnő” a következő váltási időpontra (PCA0CPL + PCA0CPH). A folyamat minden 255-ös érték után túlcsordul, mivel azonban ez a figyelt következő összehasonlítási értékre is igaz, a generált jel szempontjából a túlcsordulás teljesen érdektelen.

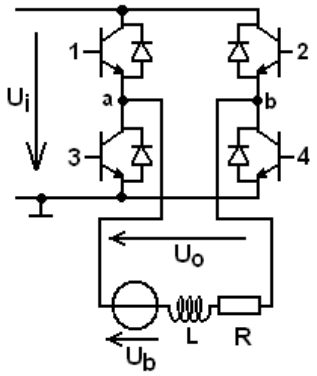
g) impulzusszélesség-modulátor (PWM) funkció

A korszerű digitális irányítások legegyszerűbb – analóg kimenetet és erősítőfokozatot helyettesítő – vezérlési elve az impulzusszélesség moduláció. Az impulzusszélesség-modulátorral megtakarítható a bináris érték előbb analóg jellé alakítása, majd pl. egy motor vezérlése során a teljesítményfokozatban a hídágak vezérléséhez szükséges digitális jellé történő visszaalakítása. Az átalakítás elve a következő (részletesen ld. az Elektronika 2 tárgyban): legyen pl. egy vezérlőregiszterbe tölthető maximális érték 1000 (= 1 ezrelékes felbontás), az aktuálisan betöltött érték pedig 100 (10%), ekkor a kimenet az első 100 órajel alatt aktív lesz, a többi 900 alatt pedig inaktív. Az így kapott négyszögjel átlagértéke a maximális kimenőfeszültség 10%-a (feltételezve, hogy egy szűrő áramkör a kimenetet a PWM jel periódusidejénél lényegesen nagyobb időállandóval átlagolja. Az átlagolás történhet a végrehajtó vagy a beavatkozó szervben, de alapulhat más fizikai törvényszerűségen is (pl. az emberi szem lassúságán vagy a hőtechnikai folyamatok lényegesen nagyobb időállandóin).



Impulzusszélesség-modulált jel

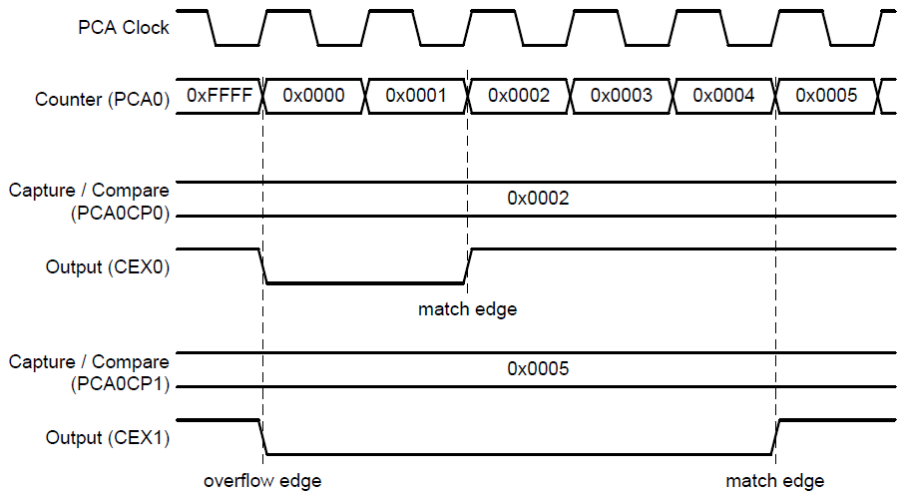
Egy ilyen vezérlőjellel a megkívánt forgásiránytól függően vezéreljük a motort meghajtó hídkapcsolás 1-4 (pozitív kapocsfeszültség) ill. 2-3 (negatív kapocsfeszültség) kapcsoló elemeit, a nagy sebességgel vezérelt ki- és bekapcsolások pillanatértékét a lényegesen lassabb, tehetetlenebb motor „átlagolja” és az átlagértékeknek megfelelő kapocsfeszültség szerinti fordulatszámmal forog.



Az impulzusszélesség-modulált jellel vezérelt hídkapcsolás elve

Fenti elv nem csak villamos hajtásokban használatos. A bármilyen időbeli átlagoló hatás megléte esetén jól használható módszert tipikusan használják fényerő szabályozásra (az átlagolást a szemünk végzi), galvanikusan leválasztott analóg kimenetek digitális kimenettel történő előállítására (a digitális „igen-nem” jel egyszerűen leválasztható akár egy optocsatolóval, az ezt követő szűrőkapcsolás átlagképzése állítja elő a leválasztott analóg jelet), de hasonló elv alapján kapcsolgatják a fűtőelemeket korszerű hőfokszabályozó berendezéseink is.

A módszer támogatása céljából valamennyi PCA0 csatorna képes a hozzá rendelt digitális kimeneten PWM jel előállítására (N = 8, 9, 10, 11 vagy 16 bites üzemmódban). A bitszám és a választott bemenő frekvencia (PCA Clock) együtt határozzák meg a PWM jel frekvenciáját/periódusidejét, a bitszámból ezen kívül kiadódik a PWM jel felbontása. A kimeneti impulzus kezdődhet a számláló túlcsordulásánál (ún. edge-aligned PWM), vagy elhelyezkedhet szimmetrikusan ehhez a ponthoz képest (ún. center-aligned PWM).

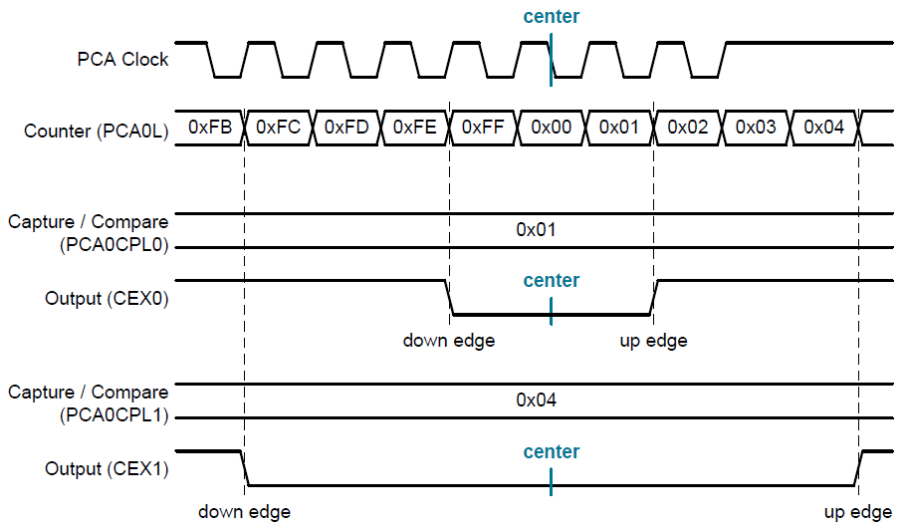


Élre illeszkedő PWM generátor (edge-aligned PWM)

A szabadon futó PCA0 számláló (az induló példában 1000-ként szereplő időalap esetünkben a PCA órajel periódusidejének 256, 512, 1024, 2048 vagy 65536 szorosa) folyamatosan összehasonlításra kerül a PCA0CP regiszterrel: PCA0 ≥ PCA0CP esetén a kimenet a paraméterezésnek megfelelően 0-ba vagy 1-be állítódik, a számláló túlcsordulásakor (az időalap leteltekor) pedig automatikusan ellenkezőjére vált (élre illeszkedő impulzus esetén). Ily módon a generált négyszögjel kitöltési tényezője a választott polaritástól függően a

$$k = \frac{PCA0CP}{2^N} \text{ vagy } k = \frac{2^N - PCA0CP}{2^N}$$

összefüggés alapján számítható.



Szimmetrikusan illeszkedő PWM generátor (center-aligned PWM)

Szimmetrikusan elhelyezkedő PWM impulzus esetén kicsit bonyolultabb a helyzet: a beállított N = 8, 9, 10, 11 vagy 16 bitszám alapján (N+1) biten történik a számlálás, a nullátmenet előtt és után is a beállított értéknek megfelelő ideig generál ponált vagy negált impulzust a kimenetén a generátor (az impulzus *2·PCA0CP+1* órajel széles lesz és szimmetrikusan helyezkedik el a számláló túlcsordulását követő 0 érték közepén elhelyezkedő órajel lefutó élhez). Ennek megfelelően a négyszögjel kitöltési tényezője ilyenkor a választott polaritástól függően a

$$k = \frac{PCA0CP+1/2}{2^N} \text{ vagy } k = \frac{2^N-(PCA0CP+1/2)}{2^N}$$

képletekből adódik.

3.4.2 A watch-dog timer

Korszerű mikrokontrollerek elengedhetetlen biztonsági eszköze az ún. watch-dog timer. Ez az időzítő kapcsolás legegyszerűbben egy **újraindítható monostabil multivibrátor** egységgel szemléltethető, amelyet újra és újra aktiválnunk kell egy meghatározott utasítással ill. periféria írási művelettel. Ezt a műveletet a watch-dog „felhúzásának” (újraindításának) nevezzük. Elmulasztása esetén időzítőnk „lejár”, ez pedig igen súlyos következménnyel, tipikusan a mikrokontroller teljes újraindításával (reset) jár együtt.

A dolog célja annak megakadályozása, hogy teljes rendszerünk egy hardver- vagy szoftvertervezési hiba folytán végtelen váróciklusba (ún. dead-lock) kerüljön. Ennek érdekében az alapvetően ciklikus működésű szoftver rendszerünk meghatározott pontjaira „életjel adó” watch-dog újraindításokat helyezünk el, mutatva, hogy rendszerünk az általunk előre tervezett helyes irányvonal szerint működik. Egy előre nem tervezett váróhurokba, „rendszereltévedésbe” való kerülés esetén a jelzések kimaradnak, és a beállított idő letelte után a watch-dog kontrollerünket újra fogja indítani (tipikusan egyértelmű státuszflag-jelzés mellett, azaz szoftver rendszerünk tudatában lehet annak, hogy nem bekapcsolás utáni reset, hanem watch-dog reset miatt lett újraindítva.

A watch-dog lefutási idejének beállítása programozható és rendszerfüggő. Olyan értéket kell találjunk, ami még nem hátráltatja a rendszer programrendszerének működését (ne legyen minden tizedik utasításunk a WD felhúzása, így nem lehet rendszert és ciklusokat tervezni), de nem szabad, hogy hosszabb legyen, mint az a fizikai időkorlát, mikor nem működő rendszerünk már veszélyt jelenthet a környezete számára. Egy bekapcsolva felejtett fűtőtest pl. egy másodperc alatt valószínűleg nem képes túl nagy kárt tenni környezetében, de ha egy nukleáris szabályozórendszerben csak egy másodperc késéssel jövünk rá, hogy beindult a láncreakció, az könnyen belátható módon nem lenne szerencsés. Fentiek miatt a WD időzítők tipikusan a mikroszekundum és a néhány másodperc közötti időkre állíthatók be, időnként a felső határ (egy 32 bites számlálóval) akár több száz másodperc is lehet.

A programozók sajnos többnyire nem szeretik ezt az egységet. Kellemetlen feladat mindazt végiggondolni, hogy egy józan ésszel pl. 10 msec-ra beállított WD időzítő

programunk mely részén (pl. egy hosszabb lebegőpontos műveletsor, egy memóriaterület másolás, stb.) „járhat le" és okozhat olyan galibát, amit valójában programozási hiba és nem a rendszer helytelen működése okoz. Szeretik a programozók a WD időzítőt ilyen-olyan indokokkal kiiktatni, kikapcsolni – **ne tegyék ezt, a legfontosabb felügyeleti eszközt kapcsolják ki a rendszerben!**

Milyen alapelvek mondhatók el a WD helyes működtetéséhez?

- Gondoljuk végig, hogy a rendszer működése szempontjából mi egy még „elviselhető" kiesési idő (a legrosszabb vezérlési kombinációban is), és ennél rövidebb időre állítsuk a WD időzítőt!
- Soha ne tegyük időzítő megszakításba a WD felhúzását, így ugyanis az képtelen védeni a háttérhurok váratlan befagyása ellen!
- A WD felhúzás helye a rendszer háttérciklusában van. Végig kell gondolnunk ennek a huroknak a worst case időzítését, és megfelelő helyekre be kell tennünk a WD felhúzását.
- Biztos, hogy valamennyi háttér váróhurok belsejében kell szerepeljen WD aktiválás
- Túl hosszú műveletsor, függvény, ciklus, stb. esetén „fel kell törnünk" azt részekre és a részek közé be kell iktatnunk a WD kezelését.
- Fejlesztés alatt a WD kezelés nagyon utunkban lesz (töréspontokon való leállás újraindulást jelent, stb.) ezért tegyük feltételesen kikapcsolhatóvá a WD timert és csak a fejlesztés idejére kapcsoljuk ki (pl. egyetlen definíció segítségével)!
- Gondoljuk végig, hogy mi mindent kell „máshogyan csinálnunk", ha programrendszerünk a WD miatt indult újra és nem power-on reset miatt! (Lehet, hogy a vezérelt motorunk teljes fordulatszámmal pörög!)

A WD kezelést a gyártók igyekeznek biztonságossá tenni, hogy egy eltévedt program ne tudja véletlenül sem becsapni felügyeleti rendszerünket. Ehhez általában a következő módszereket használják:

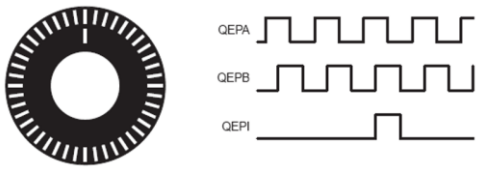
- Sok mikrokontrollerben az aktivált watch-dog időzítőt nem lehet többet kikapcsolni, néha az időzítést sem lehet utólag változtatni.
- Amennyiben az időzítés változtatható, válasszunk az inicializálás alatt (a hosszabb másolási és kapcsolat-felvételi műveletek idejére) hosszabb időzítést, az üzemszerű működtetés során azonban vegyük vissza ezt az időt a biztonságosabb tartományba!
- A WD „felhúzás" általában több utasításból álló szekvencia, bonyolultabb adatkombinációk írásával összekapcsolva (ami véletlenszerűen „nehezen jön össze")
- A WD felhúzás műveletsorának elvégzése sokszor időhöz van kötve, tehát pl. 2 bonyolultabb írási műveletet max. 4 CPU órajel-ciklus alatt kell elvégeznünk, különben a művelet érvénytelen. Ezzel megint az eltévedt programok véletlen módon összeálló művelet-szekvenciáinak a kiszűrése a cél. Vigyáznunk kell, hogy az optimalizáló C fordító ne szóljon bele az elképzeléseinkbe, és ne rendezze át az elképzelt műveletsorokat (ld. fordítási és optimalizálási opciók lokális vezérlése).

3.4.3 Az alapvető számláló (counter) üzemmódok

Az időzítő üzemmódok mindegyike használható külső jelváltások számlálására is, ennek részletezésére a dolog magától értetődő jellege miatt ki sem térünk. Egy olyan lényeges számlálási funkciót említenénk csak meg, amely a nagyobb mikrokontrollerek és a DSP-k funkciói között találhatók csak meg: a pontos pozíció- és fordulatszám érzékelés legfontosabb szenzorainak, az inkrementális szöghelyzet adók fogadására alkalmas ún. kvadratúra enkóder impulzus (QEP) számlálókat.

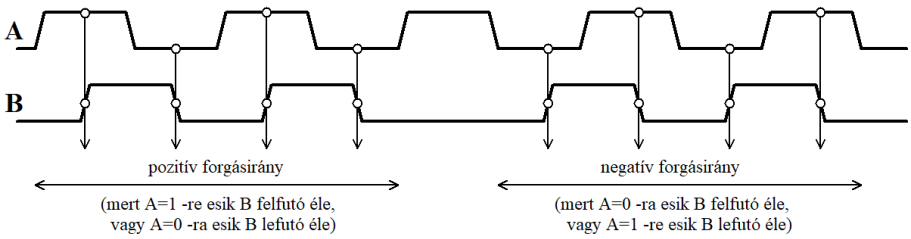
Az inkrementális adók nevüket onnan kapták, hogy szemben az abszolút pozíció információt szolgáltató érzékelőkkel (pl. kódadók vagy potenciométerek) ezek az érzékelők csak egy-egy "továbbléptető impulzust" szolgáltatnak egy, az érzékelőket követő integrátor (digitális integrátor = számláló) számára. Ezek az impulzusok végzik a számláló inkrementálását/dekrementálását.

Az inkrementális adó 2, egymáshoz képest fázisban eltolt jelsorozatot szolgáltat (A és B). Az információ hordozója a jelek frekvenciája és a 2 jel egymáshoz mért fázisszöge, ami csak ± 90°-os érték lehet (ill. csak ez az információ kerül kiértékelésre). A frekvencia az adó tengelyének szögsebességével arányos, míg a fázisszög a tengely forgásirányára vonatkozó információt hordozza.



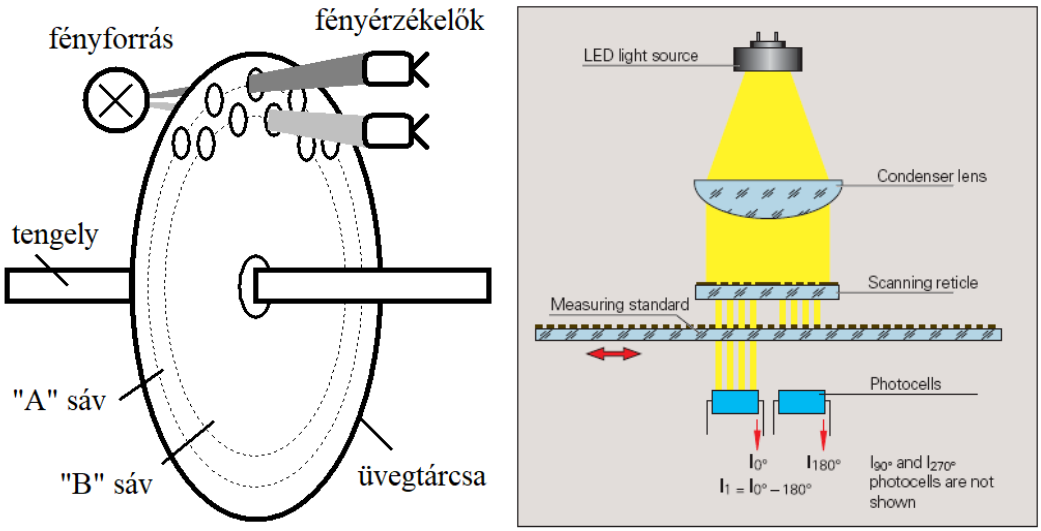
Inkrementális jeladó jelei

Az egyszerűség kedvéért tételezzük fel, hogy a jelsorozat négyszögjel (belső jelformálás révén), ekkor a jelek által hordozott információ a következő:

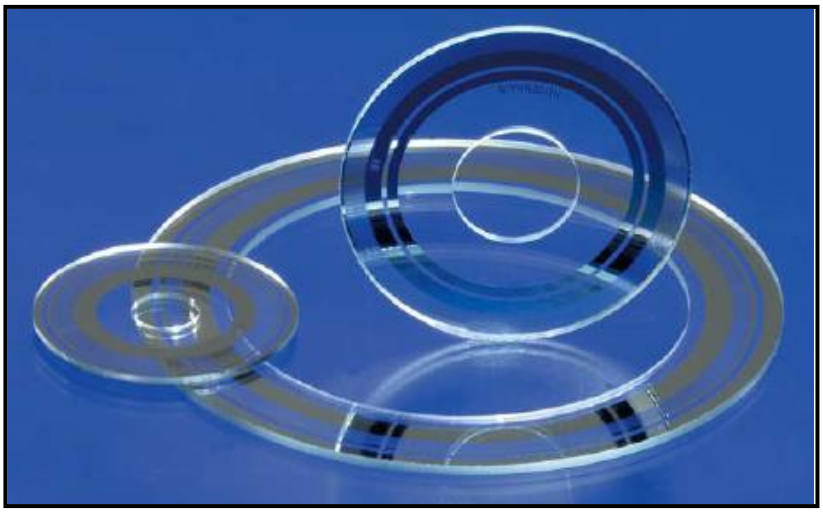


Inkrementális jeladó jeleinek értelmezése

Természetesen egy impulzus ideje alatt sem állandó a szögsebesség, ily módon B fáziseltolásában a 90°-ot nincs mihez mérni, tehát csak az az információ áll rendelkezésünkre, hogy B váltása A milyen szintje mellett következett be. A sebességinformációt hordozó impulzussorozat irányhelyes integrálásával (számlálásával) nyerhetjük a pozíció-információt.



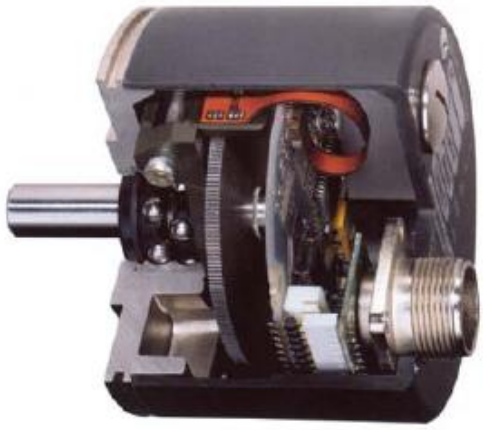
Inkrementális adók vázlatos belső felépítése



Rácsszerkezet az inkrementális adók üvegtárcsáin

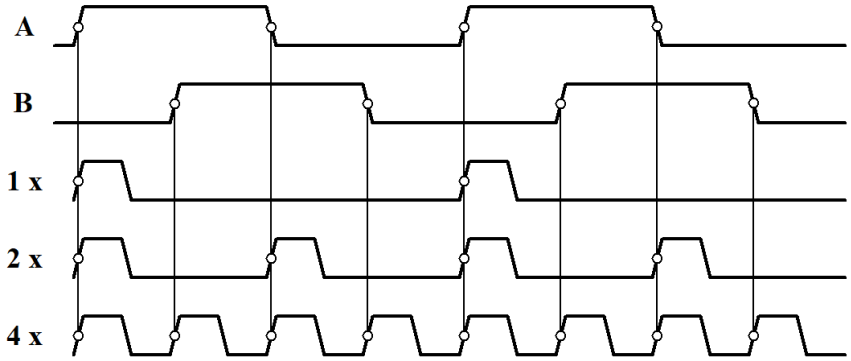
Fizikailag az inkrementális adó egy zárt házba épített néhány cm átmérőjű "lyuktárcsából" és a hozzá kapcsolódó elektronikából áll. A lyuktárcsa valójában egy üvegtárcsa, amire vákuumgőzöléssel krómot hordanak fel és így egy egyenletes rácsszerkezetet alakítanak ki, amely optikailag váltakozva átlátszó, ill. átláthatatlan. Erre a megoldásra azért van szükség, mivel a megkívánt "lyukszám" (több ezer / kerület) egyenletes mechanikai kifúrása technológiailag nehezebb feladat lenne.

Az optikai megoldáson kívül használatos még egy másik, olcsóbb megoldás, amelynél a fém forgótárcsa kerülete vagy fogazott, vagy a kerületét sűrűn váltakozó mágneses polaritással látják el, és magnetorezisztív (elektromos tulajdonságait a mágneses tér hatására változtató) érzékelővel veszik le az előzőekhez hasonló információt.



Mágneses elven működő inkrementális adó

A jelekben található információtartalom több annál, semhogy a számlálót csak impulzus-periódusonként növelnénk vagy csökkentenénk 1-gyel. A fogadó/kiértékelő áramkörök többsége megkülönböztet ún. 1-, 2- vagy 4-szeres kiértékelést a következők szerint:

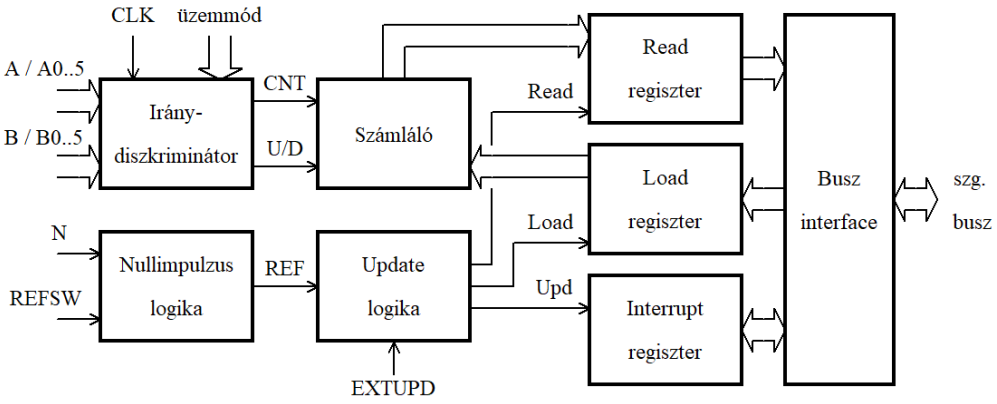


Inkrementális jeladó jeleinek 1-, 2- és 4-szeres kiértékelése

Az 1-szeres kiértékelés esetén egy impulzus („lyuk”) egyetlen éle kerül kiértékelésre, a teljes körülfordulást a lyukak számával egyező részre tudjuk felbontani. A beérkező jelekből bizonyos feltételekkel azonban ennél több információ is kinyerhető: amennyiben biztosítjuk, hogy az impulzusok 50%-os kitöltésűek (a lyukak ugyanolyan szélesek, mint amekkora a közöttük lévő távolság – beleértve ebbe a feldolgozó elektronika impulzustorzító tulajdonságait is), akkor egyetlen impulzus fel- és lefutó élét is feldolgozva már 2-szeres kiértékelésről beszélünk, felbontásunk a lyukszám duplájára nőtt. Ha emellett biztosítani tudjuk a másik (eltolt) impulzussorozat pontosan 90°-os eltolását, akkor abból nem csak a forgásirány információját nyerhetjük ki, hanem még egyszer duplázhatjuk a felbontásunkat – ilyenkor beszélünk 4-szeres kiértékelésről. Fontos tehát megkülönböztetnünk az impulzusok számát (ami a fizikai kialakítás jellemzője, megegyezik a „lyukak” számával) az inkrementek számától, ami az előzőnek többszöröse is lehet a kiértékelés függvényében.

Az inkrementális adók legnagyobb előnye a nagy felbontásból adódó pontosságuk, hátrányuk viszont az abszolút pozíció információ hiánya. A léptető inkrementek csak relatív pozíció információt hordoznak, az integrálást megvalósító számláló kezdőértékének meghatározásáról induláskor gondoskodnunk kell (továbbá azt is könnyű belátni, hogy a menet közbeni esetleges impulzusvesztés is mindenképpen kerülendő, hiszen a következő impulzus nem tudja „korrigálni" egy elvesztett előző hiányát). Fentiek miatt az inkrementális adókon alapuló pozícióérzékelés egyik jellegzetes feladata a kalibrálás, a számláló kezdőértékének meghatározása. Ehhez a környezetünkből mindenképpen egy abszolút pozíció információ beérkezésére lesz szükségünk, melynek pontossága ugyanakkora kell legyen, mint a felbontásunk pontossága, az integrális információgyűjtés végső pontossága nem lesz jobb a kiindulási adatok pontosságánál. Hogyan teljesíthető ez a feltétel ?

Az inkrementális adók szolgáltatnak egy további jelet, az ún. nullimpulzust (ez a jelimpulzusok periódusidejének 1/4-ig (90°) aktív). Ez felhasználható pl. a fordulatok számlálására is, de igazi jelentősége az abszolút pozíció kezdőértékének beállításánál van. A nullimpulzus mellett szükségünk lesz még egy információs jelre a mechanika egy (vagy több) pontjáról (ez vagy egy, a rendszerrel együtt forgó pontatlanabb potenciométer, vagy egy mechanikus kapcsoló lehet a mozgási pálya egy pontján), de ennek pontossága (mechanikus beállítása és reprodukálhatósága) elegendő, ha egy teljes körülfordulás mértékéig pontos, ezen belül már az inkrementális adó nullimpulzusa biztosítja az adó pontosságával történő pozíció-definiálást. (Az információs jel megkívánt pontosságára ily módon akár a teljes pontosság tízezred része is elegendő lehet.)



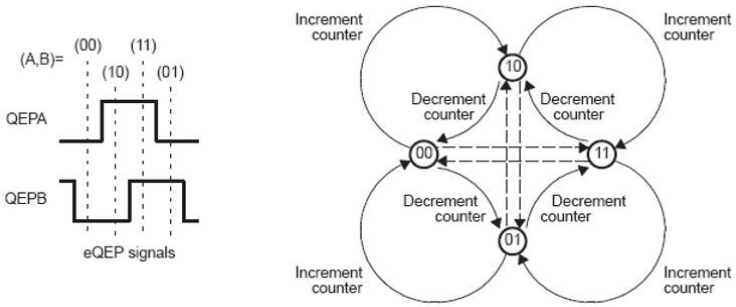
Inkrementális adó fogadó egységének blokkvázlata

Az elmondottak alapján már érthető, hogy az inkrementális adók fogadására egy olyan speciális fogadóegységet kell kialakítanunk, melynek feladatai:

- az impulzussorozatok fogadása, ebből – a beállított kiértékelési mód szerint – a számláló órajelének és irányvezérlésének előállítása (ún. iránydiszkriminátor)
- az inkrementeket (≠ impulzusok) számláló előre-hátra számláló számlálóegység. Ennek hossza (bitszáma) szabja meg a teljes mozgástartományt, az ábrázolható pozíció értékeket (ez a méret szoftver úton is bővíthető – ennek részleteire itt nem térünk ki).

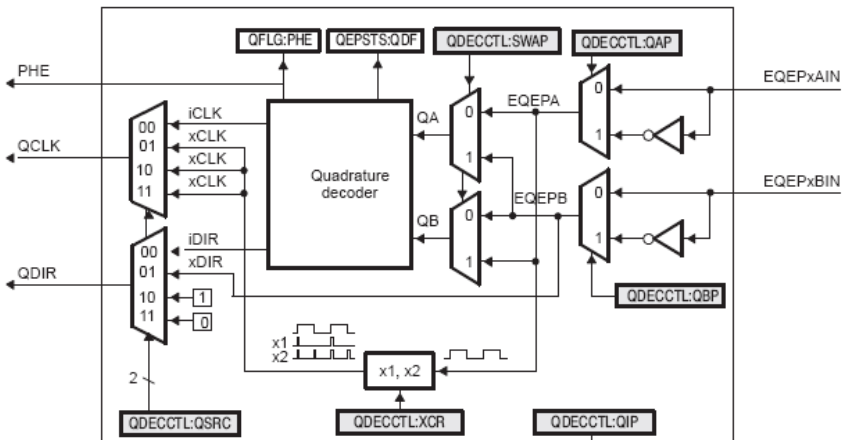
- a számlálóegység írására (kalibráláskor az nullimpulzus hatására hardver úton történő beállítására) és olvasására szolgáló update logika az író és olvasó regiszterek segítségével biztosítja a pozícióadatok konzisztens kezelését abban az esetben is, ha a pozíció bitszám-mérete eltér a kezelő mikrokontroller adatbusz méretétől.
- a nullimpulzus logika a körülfordulásonként érkező nullimpulzus és az azokat megkülönböztető – pontatlanabb – külső információs jel (REFSW) összehangolt kezelését biztosítja.

A kvadratúra enkóder modulok jellegzetes alapkapcsolása tehát az ún. iránydiszkriminátor egység, melynek feladata a beérkező A és B jelekből az irányvezérlő jel (DIR) és a léptető impulzusok (CLK) előállítása az aktuális kiértékelési mód (x1, x2, x4) szerint. Ennek állapotdiagramját mutatja az alábbi ábra.



Az iránydiszkriminátor állapotdiagramja

Amint azt a bevezetőben említettük, sok mikrokontroller (elsősorban a pozíciószabályozásokban nagy előszeretettel alkalmazott jelfeldolgozó processzorok) a fenti kapcsolásokat készen tartalmazza – akár több példányban is. Működési elvük alapján ezek a modulok többnyire az időzítő/számláló egységek részei, a tulajdonságok összefoglalásánál azonban mindig kiemelésre kerül, hogy az egység tud-e kvadratúra enkóder modulként (QEP =Quadrature Encoder Pulse) működni.



QEP modul iránydiszkriminátor kapcsolása (TMS320F2808)

A pozíció információt igénylő alkalmazások (pl. szabályozások) többnyire igénylik az objektum sebességének ismeretét is. Ezt a sebességet könnyen meghatározhatjuk a rendelkezésre álló információk alapján: számításához emlékezzünk vissza a fizikában tanult következő alapösszefüggésre:

$$v = \frac{ds}{dt} \cong \frac{\Delta s}{\Delta t} = \Delta N \cdot \frac{q}{T_v}$$

Első ötletünk az összefüggés alapján: csupán képeznünk kell a pozícióértékek előjeles különbségét (Δs), azt elosztjuk a két mérési időpont különbségével ($\Delta t = T_v$) és így máris megkaptuk a kívánt sebességet. Amennyiben a pozícióértékek különbségét ekvidisztáns időpontokban képezzük, még az osztási művelet is elhagyható, hiszen a Δs mennyiség arányos a sebességgel, az arányossági tényező megfelelő normálás révén vagy 1-nek tekinthető, vagy a számítási képletek más konstansaiba beépíthető.
A kérdés csupán az, mekkora legyen az említett időköz ? A képlet azt sugallná, hogy minél kisebb, hiszen Δt növelése a pillanatnyi sebességet egyre inkább valami átlagsebességgé torzítja. Ne feledjük azonban el két lényeges szempontot:

1. A digitális irányításokban Δt nem csökkenthető tetszőlegesen. Ennek egyszerűen az szab korlátot, hogy a sűrű megszakításkérések túlterhelik a mikroprocesszort, ill. a további megszakításkérések szükségszerű megléte miatt az időzítő megszakításkérésének átmeneti tiltottsága az időnk ekvidisztáns voltát egyre jobban megkérdőjelezi.
2. Pozícióinformációnk szintén nem folytonos, hanem kvantált mennyiség (egész szám), melynek pontossága ±1. Amennyiben tehát a választott időintervallum alatt a számlálóállás ΔN értékkel változott meg, sebességünk pontossága (felbontása) nem jobb, mint 1 / ΔN (pl. max. 100 számlálóimpulzus / Δt esetén pontosságunk nem jobb, mint 1%).

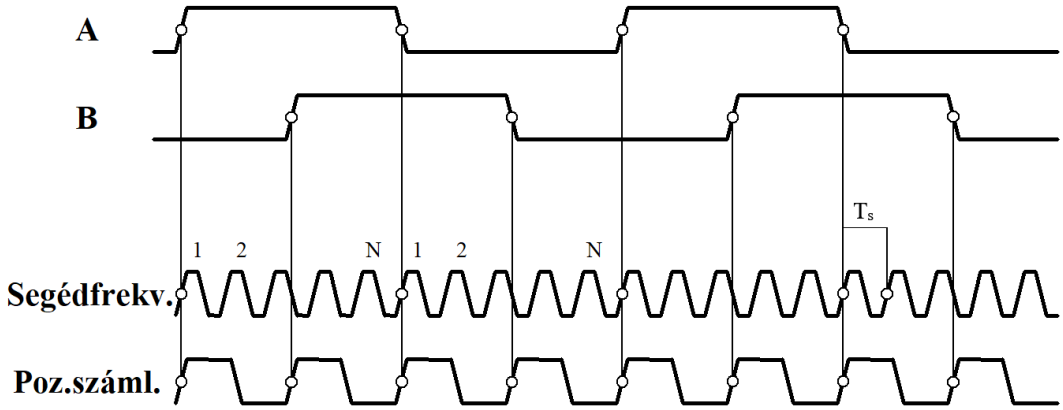
Különösen az utóbb említett szempont szomorú következménye az, hogy amennyiben az időintervallumot úgy választjuk meg, hogy pl. a maximális sebesség esetén 1% pontossággal tudjuk számítani a sebességet (ez már egy elég nagy Δt-nek felel meg, hiszen két sebességszámítási pont között a számlálóértékeknek legalább 100-zal el kell térniük !), számított sebességünk 10%-os névleges értéknél már csak 10% pontosságú lesz, a megállás közelében pedig a végtelenségig nő számításunk pontatlansága !

Mi ennek az oka ? Ne tévesszük szem elől azt a méréstechnikában tanult alapszabályt, hogy a különbségképzéssel számított mennyiségek végtelen nagy hibát hordozhatnak, amennyiben a két kiindulási mennyiség nagysága, melyek különbségét képeztük, megközelítőleg azonos !

A gyakorlatban a vázolt problémát úgy szokták áthidalni, hogy a kisebb sebességek tartományában a sebességet már nem az ismertetett különbségképzéssel számítják, hanem (többnyire hardver támogatással) a beérkező impulzusok élei közötti időket mérik. Az így kapott érték a sebességgel fordítottan arányos:

$$v = \frac{ds}{dt} \cong \frac{\Delta s}{\Delta t} = \frac{2 \cdot q}{N} \cdot \frac{1}{T_s} = 2 \cdot f_s \cdot \frac{q}{N}$$

ahol *v [m/s]* a keresett sebesség, *q [m]* a pozíciómérés felbontása négyszeres kiértékelés esetén, $f_s=1/T_s$ *[1/s]* az alkalmazott segédfrekvencia, *N* a sebességszámláló pillanatnyi állása.

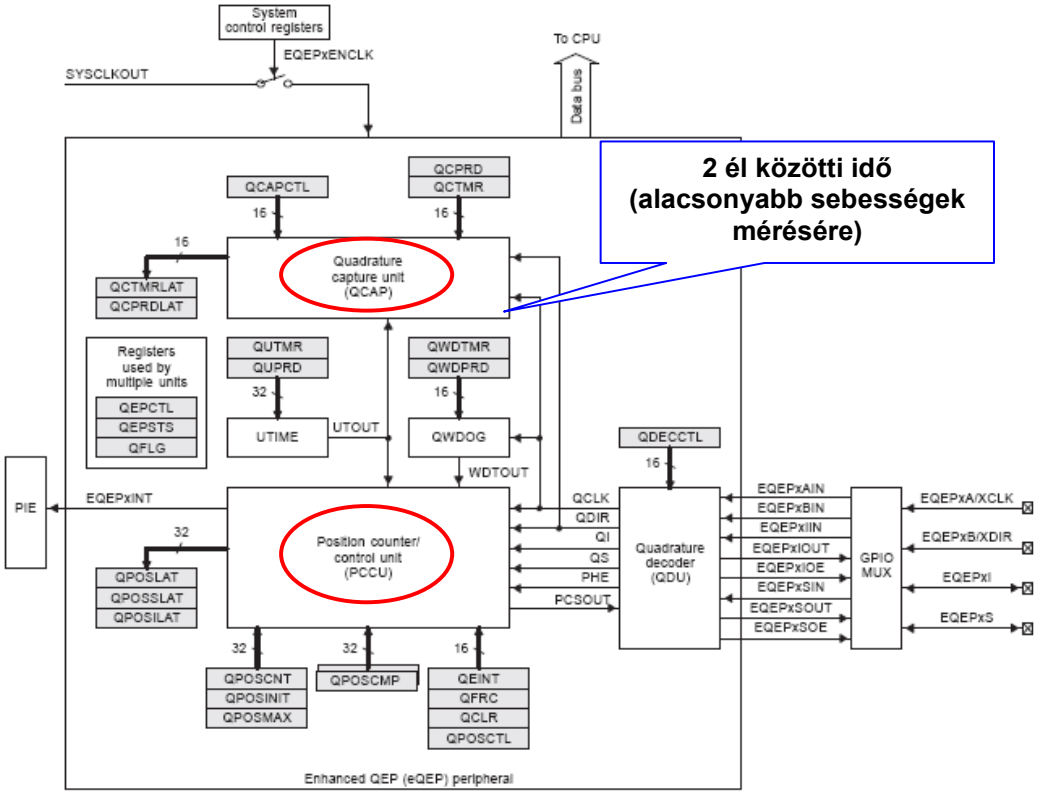


A sebesség segédfrekvenciával történő mérésének elve

A módszer alkalmazásakor a következő megoldandó feladatokkal kell számolnunk:

- a sebesség függvényében „mérési módot kell váltanunk” (nagyobb sebességek esetében ez a mérési módszer válik pontatlanná, hiszen itt *N* hordoz magában különbségi információt),
- hardver támogatás szükséges a segédfrekvenciával történő számláláshoz és a számlálóértékek automatikus tárolásához, különben minden impulzus-élnél megszakítással kell számolnunk, ami a processzort túlterhelné és a mérést pontatlanná tenné (N-et minden impulzus élnél „ki kell olvasni”),
- garantálnunk kell, hogy a számlálóérték a tengely álló helyzetében vagy kis sebességeknél nem csordulhat túl (a számláló a maximális értéken megáll, és nem fordul át).

A felsorolt problémák ellenére igen gyakran alkalmazzák a felsorolt mérési/számítási módszer kombinációját, mivel igen jól kiegészítik egymást: nagy sebességnél a különbségképzéses módszer biztosít nagyobb pontosságot, kisebb sebességeknél pedig az időméréses eljárás pontossága lesz megfelelő.



QEP modul (TMS320F2808)

A QEP modul hiánya a mikrokontrollerben egy néhány kaput igénylő CPLD/FPGA kapcsolással egyszerűen pótolható. Pozíció- és sebesség-meghatározási feladatokban általában a 16 bites számláló nem elégséges, vagy 32 bites számlálót alakítsunk ki, vagy gondoskodjunk a hardver egységben található rövidebb számláló szoftver úton történő bővítéséről (ún. szoftver kaszkádosítás).

3.5 Digitális be- és kimenetek

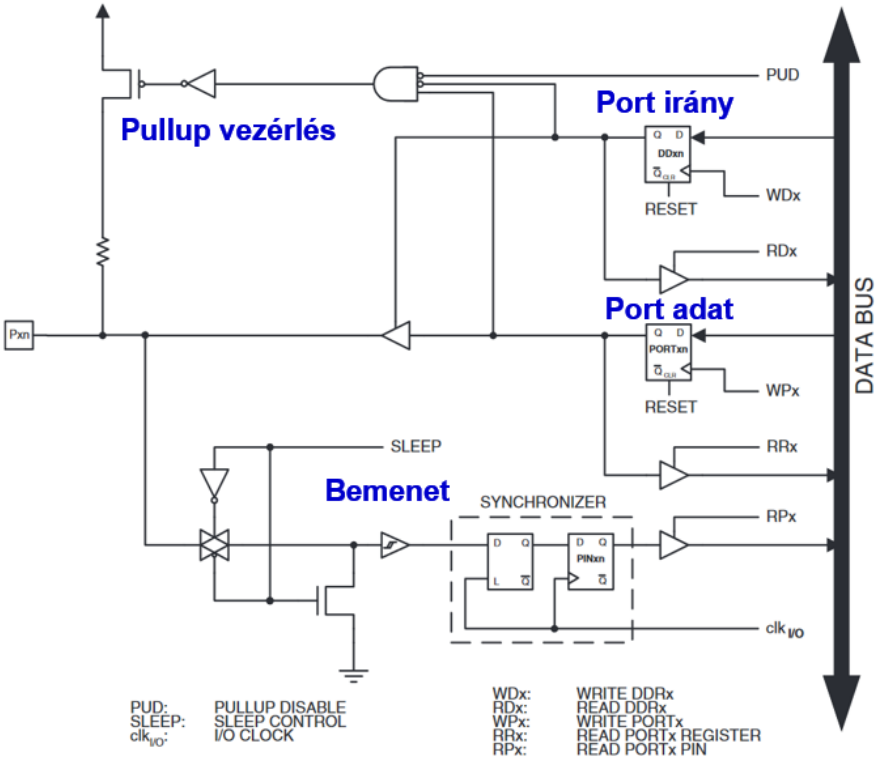
A digitális be- és kimenetek általában nagy számban találhatók meg a mikrokontrollerek perifériái között. A legtöbb gyártó úgy alakítja ki a kontrollereket, hogy azok valamennyi lehetséges lábára portfunkciót telepít, amely funkciójában osztozik a különböző perifériák egyéb be- és kimeneteivel. Azt a konfigurációs kérdést, hogy végül melyik lábon milyen funkció jelenjen meg, általában konfigurációs regiszterekkel oldják meg, melyekben az egyes funkciókat engedélyezni és tiltani tudjuk.

Maguk a digitális portok nem képviselnének külön említésre méltó érdekességet, hat fontos dolgot azonban meg kell említenünk velük kapcsolatban:

- a be- és kimenet közötti választás lehetőségét (irányválasztás),
- a digitális bemenet felhúzó ellenállással való segítésének lehetőségét (pullup),
- a kimeneti meghajtó típusának megválasztását (típusválasztás),

- a kétirányú adatforgalomra való használhatóság kérdését (adatvezetékek irányvezérlése)
- nagyobb sebességű mikrokontrollerek esetén az egység sebességének megválasztását (jelváltozások max. sebessége),
- a 3.3V-os kontrollerek elterjedése óta a bemenetek feszültségtartománya (5V tolerancia – ezzel a kérdéssel a következő fejezetben még részletesebben foglalkozunk).

a) Atmel Atmega128



Egy digitális portbit kialakítása (ATmega128)

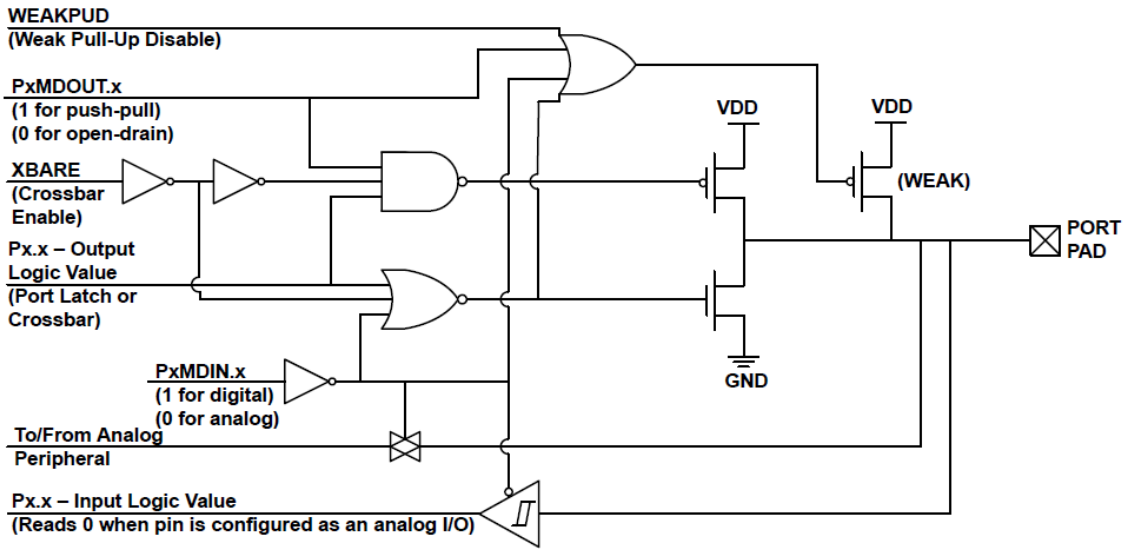
- A mikrokontroller 8 bites portjai bitenként konfigurálhatók a DD (Data Direction) regiszter segítségével be és kimenetnek.
- Kimeneti módban (DDn = 1) mindkét logikai szintet aktívan meghajtja (push-pull kimenet), ilyen módon tehát OC (OD) kimenetként nem használható. A kapcsolás mégis lehetővé teszi a kimenetnek csak egyik oldalról történő aktív meghajtását, ehhez azonban a PORT (adat) regiszterbe az aktívan meghajtani kívánt logikai szintet kell írnunk (0 vagy 1), míg a kimenet invertált értékét a DD regiszterben kell beállítanunk. A megoldás kezelése kicsit nehézkes, mivel az adott bit „elválik” a többi kimenti bittől, hiszen a push-pull kimeneti bitek állapotát a PORT, míg az ily módon kialakított OC bitek állapotát a DD regiszteren keresztül kell kezelnünk. Még a felhúzó ellenállást is aktiválhatjuk logikai 1 állapot esetén, ehhez azonban mindkét regiszter szinkron vezérlése szükséges (logikai 0 állapot: PORT=0, DD=1, logikai 1

állapot: PORT=1, DD=0). A PORT regiszterbe írt adat akár bemeneti, akár kimeneti üzemmódban is visszaolvasható, de bemeneti módban nem a bemenet állapotát, hanem a kimeneti tároló állapotát adja vissza.

- Bemenetként konfigurálva a bitet a PIN regisztert kell olvasnunk a bemenet állapotának lekérdezéséhez.
- A portokra központilag vezérelhető (engedélyezhető/tiltható) egy-egy felhúzó ellenállás megléte (20..50 kΩ). Ez kimeneti módban tiltódik, bemeneti módban pedig akkor aktív, ha azt a központi PUD bit engedélyezi, és a portbithez tartozó kimenet „1” állapotban van. Ily módon a felhúzó ellenállások portbitenként külön-külön vezérelhetők (ld. a kimenetek OC meghatóként történő előállítását).

b) Silabs EFM8BB3

Egyszerűbb felépítéssel több lehetőséget kínálnak a korábban megismert Silabs 8051-es család kontrollerei.

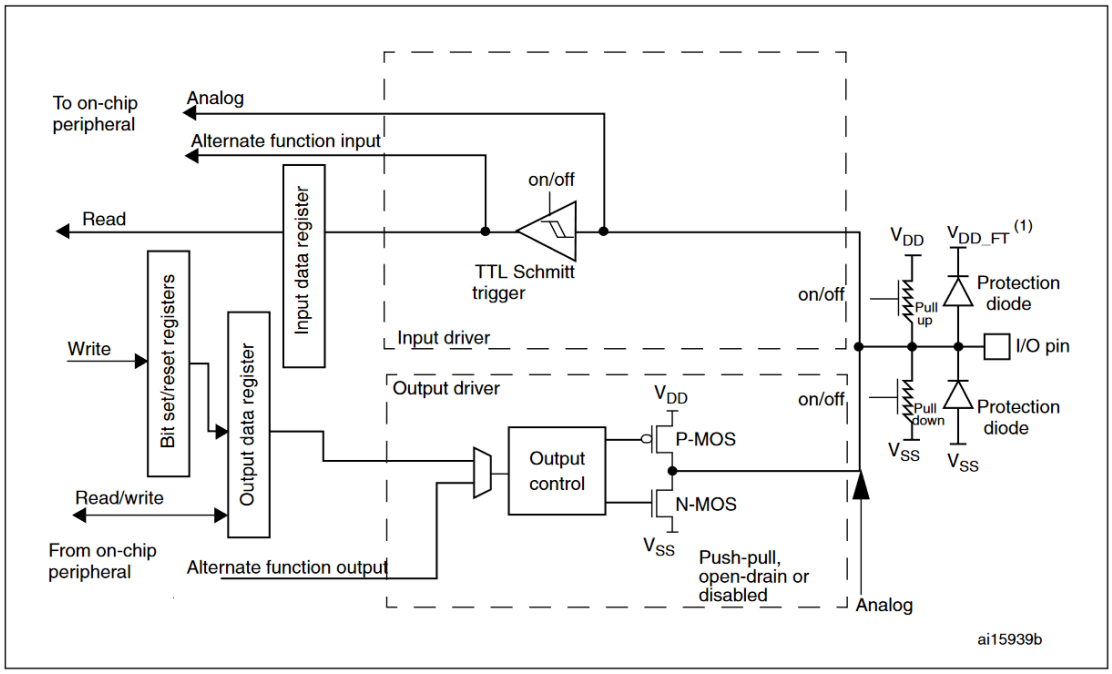


Egy digitális portbit kialakítása (EFM8BB3)

- A portbitek kimenetként a Px regiszteren keresztül vezérelhetők (Px.x). A kimenetek lehetnek mindkét irányban aktívan meghajtottak, amennyiben a Port Output Mode regiszterének megfelelő bitjét „1”-be állítjuk (PxMDOUT.x=1). Ezt a bitet „0”-ba állítva viszont letiltódik a felső meghajtó tranzisztor vezérlése, és ily módon open drain (OD) kimenetet kapunk.
- Konfigurálásnál a portbiteknek nincsenek elkülönített irányvezérlő bitjeik. Bemenetet úgy tudunk csinálni belőlük, hogy a PxMDOUT regiszterben a kimenetet OD módba, valamint a kimenetet értékét logikai „1”-be állítva „inaktíváljuk” a kimenet mindkét meghajtó tranzisztorát. Ekkor a blokkvázlaton látható alsó kapun keresztül a portláb állapota bemenetként beolvasható.

- Itt is létezik a központilag engedélyezhető felhúzó ellenállás (100 kΩ). Külön-külön tiltani nem lehet (és nem kell), mivel a kimeneti port logikai „0” állapota automatikusan tiltja.

c) Az ARM Cortex-M4 mikrokontrollerek megoldása

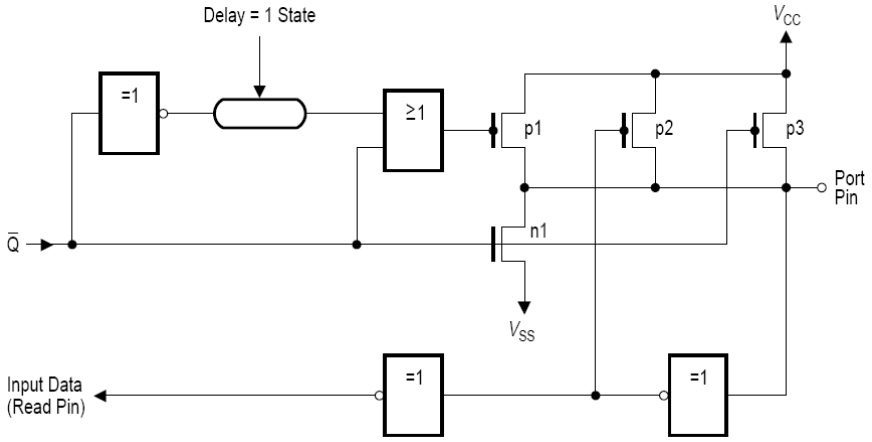


A portbitek kialakítása az STM32F446xx mikrokontrollerekben

Hasonló a kialakítása a portbiteknek a nagyobb teljesítményű (pl. ARM Cortex-M4) mikrokontrollerekben is. A beállítható kombinációk itt még bővülnek az egység sebességtartományának állíthatóságával (1-150 MHz), ami értelemszerűen a maximális jelváltozási sebességeket is megszabja. A vezérlőregiszterek száma tovább bővül (a nagyobb címtartományból adódóan ez semmilyen problémát sem okoz), lábanként kapcsolgató a portbit iránya, típusa, fel- és lehúzó ellenállással való ellátása. Fontos a nagyszámú (a kontroller lábszámától függően akár 100 feletti) be- és kimenet bitenkénti vezérelhetősége, ugyanis a 32 bites szóhosszúság miatt az egyes portok csak szavankénti kezelhetősége nagyon lelassítaná a portbitek kezelését.

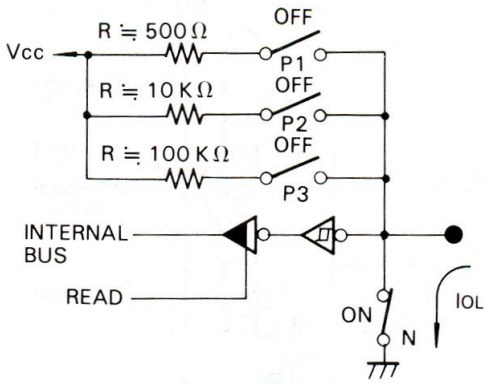
d) Egy érdekes megoldás: az „ős” 8051-es

Még egyszerűbb kapcsolást találunk az eredeti Intel 8051-es mikrokontrollerben és sok utódjában (Phillips, Infineon, Atmel). Az érdekes kapcsolás megértéséhez haladjunk lépésről lépésre a következő ábra alapján.



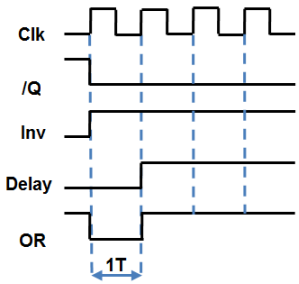
Egy digitális portbit kialakítása (i8051)

- A kapcsolás <u>se irányvezérlő, se felhúzó ellenállás vezérlő</u> biteket nem tartalmaz, mindent automatikusan tesz. Aktív kimeneti tranzisztora az N csatornás **n1** FET, a felette található 3 db P csatornás FET nagy csatorna ellenállású vezérelhető ellenállások (**p1**= 500Ω, **p2**=10kΩ , **p3**=100kΩ).
- <u>Digitális kimenet:</u> A /Q kimenet „1”-be állítása (kimenet = 0) aktiválja az **n1** tranzisztort és kikapcsolja valamennyi felhúzó ellenállást.

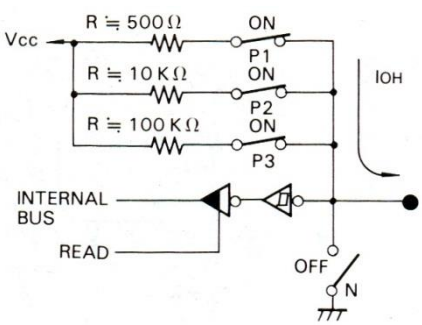


8051 port kimenetként „0” logikai szinten

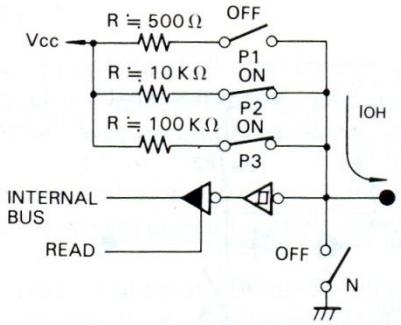
/Q = „0” (kimenet=1) aktiválja a **p3** felhúzó ellenállást (100kΩ) és kikapcsolja az **n1** tranzisztort. Az érdekes inverter – késleltetés – VAGY kapcsolat egy 1 órajel szélességű impulzus generátort képez negatív jelátmenetnél – hatására /Q = 1→0 jelátmenetnél egy „rásegítő 0 impulzus” keletkezik a **p1** tranzisztor vezérlő lábán, ami egy órajel-perióduson keresztül kisimpedanciás generátorként (500Ω) segíti a jelváltást a kimeneten (0→1), majd ezt követően megszünteti a rásegítést. Ezen túlmenően a kimeneten kialakuló „1” szint az első inverteren visszajutva bekapcsolja **p2**-t (10kΩ) is. A megoldás egy push-pull kimenethez hasonló felgyorsított OD kimenet kb. 9kΩ-os felhúzó ellenállással.



8051 port időzítése „0”→ „1” átmenetnél



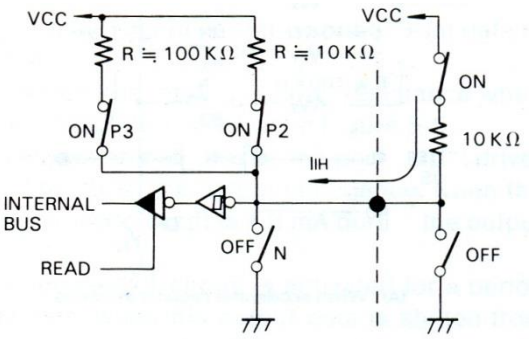
a jelváltás gyorsítása (t=1T)



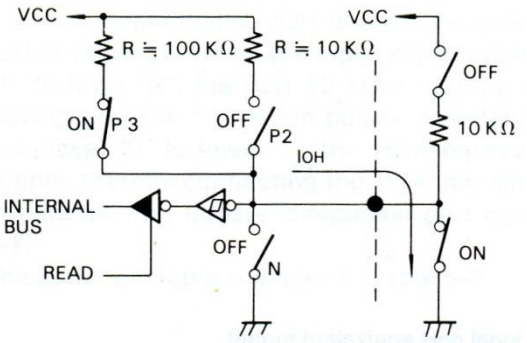
állandósult állapotban

8051 port kimenetként „1” logikai szinten

- Digitális bemenet: Logikai „1” szintre állítva a kimenetet az előbbiek alapján kb. 9kΩ-os felhúzó ellenállás húzza a bemenetet „1” szintre. Amennyiben ennek ellenében a bemenetet „0” szintre húzzuk, **p2** inaktiválódik és bemenetünk már csak a 100kΩ-os felhúzó ellenállás áramával terheli a bejövő jelet. Amennyiben a bejövő jel ismét elindul az „1” szint felé, **p2** (10kΩ) ismét aktiválódik, mintegy segítve ezzel a jelváltást a kívülről jövő jelnek.



Logikai”1” jelszint esetén



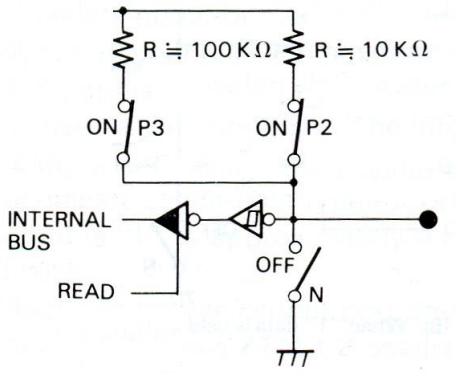
Logikai „0” jelszint esetén

8051 port bemenetként

Az ötletes megoldás ötvözi a push-pull és az O/D kimenetek előnyeit és segít elkerülni a környezeti zajok miatt bekövetkező véletlenszerű jelváltásokat a bemeneten. A

megoldásnak azonban nem csak előnyei, hanem hátrányai is vannak, melyeket nem szabad szem elől tévesztenünk:

- a kimeneti „1” szintet megvalósító pullup (**p2**‖**p3** ≈ 9kΩ) azért nem képvisel annyira alacsony impedanciát, mit egy aktív meghajtó. A jelszint terhelhetősége „1” szinten lényegesen kisebb, mint „0” kimenő jelszint esetén.

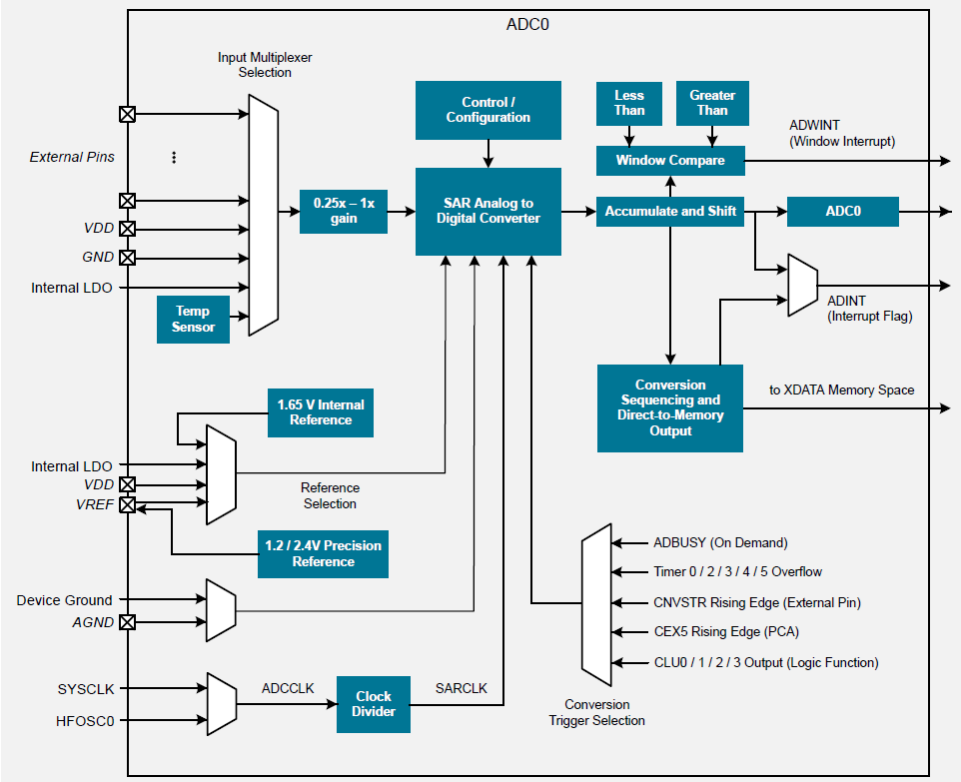


8051 port helyettesítő képe logikai „1” szint mellett

- Az előző pontból következik, hogy bemenet esetén is az „1” szintet ez a kb. 9kΩ-os felhúzó ellenállás biztosítja, így a bemenet nem tekinthető nagyimpedanciásnak ebben az állapotban. Szokásos tervezői módszer, hogy az üzemszerűen kimenetként használt portlábat – ami a processzor reset állapotában még bemenetnek van konfigurálva –egy pullup/pulldown ellenállással szoktuk határozott állapotba hozni a kimeneti meghajtó áramkör aktiválásáig (a rá kapcsolódó áramkör „definiálatlan” állapotának elkerülésére). A 8051-es port kimeneteknél nem tudunk pulldown ellenállást alkalmazni, mivel az **p2**‖**p3** ellenállásokkal egy feszültségosztót képezne, a kapcsolódó áramkört úgy kell kialakítanunk, hogy annak alapállapota logikai „1” szint legyen.

Időben váltakozó kétirányú adatátvitelre („kétirányú buszként”) csak azok a digitális portok használhatók, melyek irányvezérlése nem szoftver irányvezérléssel történik. Programutasításokkal nem tudunk elég gyorsan reagálni a külső eszköz irányváltására, és eszközeink kimenetei „összehajthatnak”. Tervezéskor irányvezérlés nélküli OC/OD kimenetekkel kerülhetjük el a kimenetek összehajtását, a vonal állapotának bármely időpillanatban beolvashatónak is kell lennie. Ezt a módszert követi az 51-es portok kialakítása, amely a váltásokat segítő, jelszinttel vezérelt pullup ellenállásokkal gyorsítja az OD kimenetek áramköri kialakításból adódó lassúságát (a kétirányú meghajtással szemben itt lényegesen magasabb a generátor-impedancia, amely sajnos RC-szűrőt képez a külső – mindenképpen meglévő – áramköri és szórt kapacitásokkal). Elsőként bemutatott Atmega128 mikrokontrollerünk ilyen kapcsolások kialakítására közvetlenül (segédkapcsolás nélkül) nem alkalmazható.

3.6 Analóg be- és kimenetek



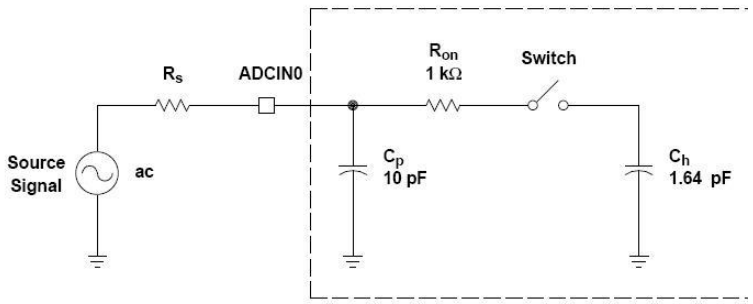
10/12 bites A/D átalakító az EFM8BB3 mikrokontrollerben

A mikrokontrollerekbe többnyire a legáltalánosabb, tetszőleges célokra felhasználható szukcesszív approximációs A/D konvertereket építenek be. Ennek megfelelően sebességük a 100 ksps (SPS: Sample pro Second) – 1 Msps tartományban várható, felbontásuk általában 8-10-12 bit. Pontossági paramétereik legtöbbször nem érik el az önálló, nagypontosságú A/D átalakítók pontossági értékeit, de általában nem is precíziós célokra használják őket. Többnyire egy analóg multiplexeren keresztül kapcsolódnak a mikrokontroller lábaihoz, igényesebb alkalmazásokban programozható belső erősítő (PGA) is a rendelkezésünkre áll a kívülről érkező jelnek az átalakító jeltartományához való illesztésére.

Megoldandó problémák a használatuk során:

- Az analóg-digitális átalakítók működéséhez nagypontosságú, ún. referenciafeszültségre lesz szükségünk. Ezt az átalakítók vagy külső forrásból veszik (VREF bemenet), vagy saját referencia – esetleg a tápfeszültség – segítségével állítják elő. Egy (tipikus) 10 bites, 2.5V jeltartományú A/D átalakító elvben is csak akkor lehet 1 LSB pontosságú, ha referenciafeszültségének (abszolút) toleranciája nem haladja meg a 2.5 V / 1024 = 2.44 mV-ot. Ezt a pontosságot általában sem az integrált referencia forrás (hőmérséklet!), sem a tápfeszültségből levezetett referenciafeszültség nem tudja teljesíteni.

- Az A/D átalakítók egy része tartalmaz mintavevő-tartó (S/H) egységet, másik részük nem. Az egység hiánya esetén feltétlenül ügyelnünk kell arra, hogy a mintavétel ideje alatt számottevő jelváltozás ne következzen be.
- A multiplexer egységek gyors váltást tesznek lehetővé az egyes csatornák között, itt azonban ügyelnünk kell a bejövő jel váltás utáni stabilizálódására.



Analóg bemenet helyettesítő kapcsolása

A külső ellenállás (R_s), a mindenképpen jelenlévő parazita kapacitások (C_p), a multiplexer kapcsoló csatorna ellenállása és a belső tároló kapacitás együttesen szabják meg a kívülről jövő jel beállási idejét. Az ismert exponenciális beállásnak megfelelően:

$$\frac{U-u(\tau)}{U}=\frac{U-U(1-e^{-\tau/T})}{U}=\frac{SA}{2^n}$$

ahol U a mérni kívánt bemeneti feszültség, $\tau = RC$ a kör időállandója, SA (sample accuracy) a megkívánt beállási pontosság (felbontásban, azaz „LSB-ben” kifejezve), míg n az alkalmazott átalakító bitszáma. A fenti egyenletet átrendezve

$$t=\ln(\frac{2^n}{SA})\cdot R_{tot}\cdot C_{sample}$$

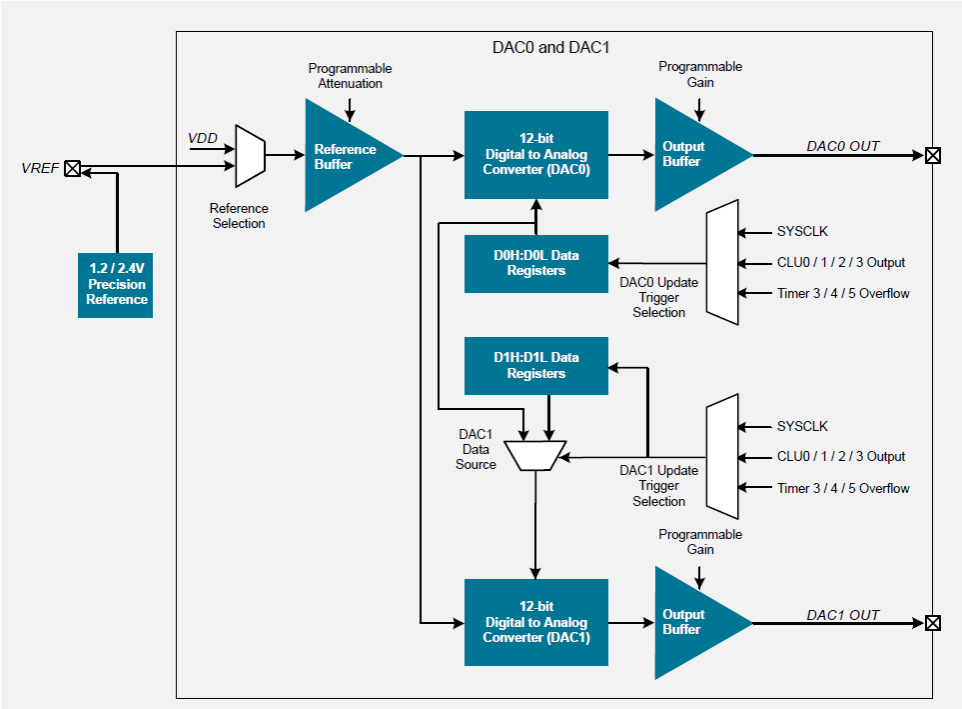
Ebben az összefüggésben Rs = 5 kΩ esetén (SA = 0.5 LSB pontosság választásával) a minimális beállási idő

$$t\approx\ln(\frac{2^{10}}{0.5})\cdot 6\cdot 10^3\cdot 12\cdot 10^{-12}\approx 550ns$$

Nagyobb sebességű alkalmazásoknál fenti összefüggést okvetlenül vegyük figyelembe!

- Némely mikrokontrollerekben az A/D átalakító bemeneti ellenállása nem tekinthető végtelennek (a tipikus > 1 MΩ, de ritkán 10-20 kΩ bemenő ellenállás is előfordul.) Ezekben az esetekben ügyelnünk kell a fogadó áramkör kialakítására, hogy ennél az értéknél a generátorkör kimenő ellenállása min. 3 nagyságrenddel legyen alacsonyabb (= néhány Ohm)!

A digitál-analog átalakítók is egyre inkább megjelennek a mikrokontrollerekben, dacára annak, hogy a 10-12 bites felbontáshoz szükséges precíziós ellenálláslétra előállítása technológiailag nem egyszerű feladat. Referenciafeszültség-érzékenységük hasonló az analóg-digitál átalakítókéhoz, ennek előállítása történhet szintén belső vagy külső forrásból. Kritikus a pontosság szempontjából a kimenetükön található meghajtó erősítő, amely az alacsony generátorimpedancia biztosítása miatt elengedhetetlen.



10/12 bites D/A átalakítók az EFM8BB3 mikrokontrollerben

3.7 Soros kommunikációs egységek

Különböző funkcionális egységek összekapcsolására soros vagy párhuzamos adatkapcsolatot alakítunk ki. A soros kapcsolat kevesebb összekötő vezetéket igényel, viszont az adatbitek soros átvitele miatt általában lassabb kapcsolatot tesz lehetővé, mint a párhuzamos illesztés. Korábban a mikroprocesszorok és a közelben (néhány cm) elhelyezkedő perifériáikat általában párhuzamosan illesztettük, a nagyobb távolságban (1m – több száz méter) elhelyezkedő egységeket soros adatátviteli kapcsolattal kötöttük össze.

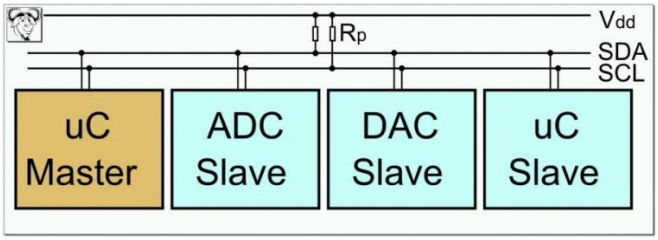
A memóriáknak és a gyors perifériáknak a mikrokontrollerekbe történő integrálása, a mikrokontroller lábainak különböző be- és kimeneti funkciókra való felhasználása miatt a párhuzamos illesztések manapság már az integrált áramkörök közötti kapcsolatok kialakításánál is egyre inkább háttérbe szorulnak. A kontrollerekbe és az önálló perifériákba is napjainkban olyan nagysebességű, tipizált protokollal rendelkező soros kommunikációs egységeket integrálnak, melyek néhány (tipikusan 3-5) vezetéken kapcsolódva a mikrokontrollerhez akár busz jelleggel is képesek 100kbit/s – 100 Mbit/s

sebességű kommunikációt folytatni egymással. Egy-egy vezérlő- vagy adatbájt [néhány tíz nsec] – [néhány μsec] alatt történő továbbítása az esetek többségében elegendő lesz számunkra, figyelembe véve a tervezési, huzalozási, és a szükséges lábszámok csökkenésével együtt járó előnyöket.

A nagysebességű kommunikáció többnyire **szinkron adatátvitellel** történik (az időzítő órajel is továbbításra kerül), ami ugyan egy többletvezetékkel jár, viszont nem követeli meg valamennyi egységben az alapidőzítés meglétét, azok pontos egyezését és szinkronizációs módszerek használatát. Az egységek közötti nagyobb távolságú kommunikáció viszont többnyire **aszinkron**, mivel az egzakt időzítést biztosító órajel maga is torzulna az átvitel során (időkésés, élek torzulása), ami – lévén nem garantálható az adatinformáció időzítésének hasonló torzulása – megszüntetné az órajel és az adatvezeték szoros időbeli kötöttségét egymáshoz.

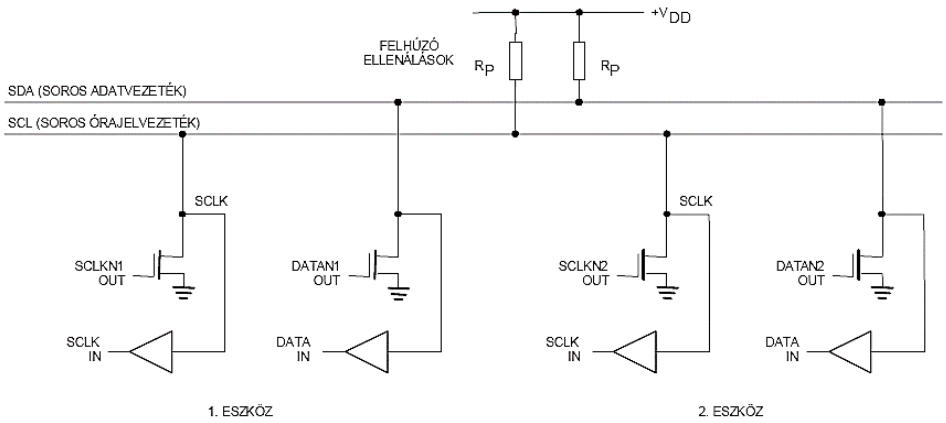
3.7.1 Az I^{2}C adatátviteli interfész

Az I^{2}C (vagy IIC) adatátviteli kapcsolat az **Inter-Integrated Circuit** szavak kezdőbetűiből kapta nevét, a Philips cég fejlesztette ki. Már az elején fontos megemlítenünk, hogy a mai mikrokontrollerek egy részébe már utódját, az ún. SMBus (System Management Bus, Intel 1996) interfészt integrálják, amely egy szigorúbb elektromos és időzítési paraméterekkel definiált változata az I^{2}C busznak. I^{2}C interfésszel rendelkező eszközeink (általában lazább) specifikációját mindig ellenőrizzük, amennyiben azt SMBus interfészre kapcsoljuk, ugyanis a lazább előírások miatt kommunikációs hibák léphetnek fel az egységek között.



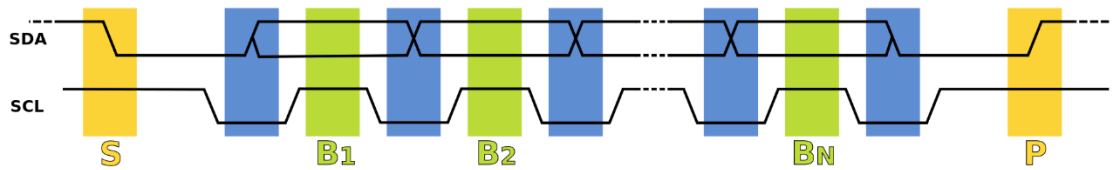
Az I^{2}C kapcsolat felépítése

Az I^{2}C interfész **kétvezetékes, szinkron kétirányú kapcsolatot** tesz lehetővé, a kapcsolat azonban ”csak” félduplex, azaz valójában váltakozva egyirányú. Az interfészt kisebb sebességű (néhány 10 – néhány 100 kbit/sec), kis mennyiségű adatátvitelre fejlesztették ki. A megoldás lehetővé teszi a busz topológia (több egység azonos vezetékeken) kialakítását, mind a kommunikációs vonal (SDA), mind az órajel (SCL) nyitott kollektoros meghajtásúak kell legyenek. Mint ismeretes, a nyitott kollektoros kapcsolatban a ”0” szint domináns, szemben az ”1” szinttel, ami a meghajtó fokozat kikapcsolt állapotát jelenti, és a jelszintet csak a vonalat felhúzó ellenállás(ok) biztosítja. Ilyen módon bármelyik meghajtó aktív (”0”) szintet tesz a kimenetre, az fog érvényre jutni (logikai VAGY kapcsolat – a ”0” szint szemszögéből nézve) a lezárt meghajtó tranzisztorok és a felhúzó ellenállás(ok) ellenében.



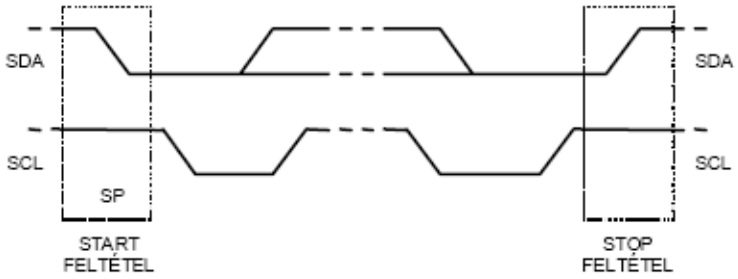
Az I²C kommunikációs hálózat topológiája

Az I²C átvitel során megkülönböztetünk mester (master) és szolga (slave) egységeket. A master vezérli a kommunikáció keretrendszerét (kezdet, vég), és meghajtja az órajelet. A mi rendszereinkben szinte mindig a mikrokontroller a mester, a perifériák a szolga egységek. Egy rendszerben több mester egység jelenléte is megengedett (multimaster jelleg), szigorú előírások vonatkoznak arra, hogy ki mikor kezdeményezhet adást, illetve minden master egységnek az órajel- és adatvonalat nem csak meghajtania, hanem olvasnia is kell. Eltérő logikai jelszint esetén a huzalozott vagy kapcsolat révén mind az órajel-, mind az adatvezeték állapotát a domináns (0) szint határozza meg, az ellenkező (1) állapotot meghajtó egységnek az adását meg kell szüntetnie (arbitráció). Az ilyen alkalmazások viszonylag ritkák, így az arbitráció előírásaival itt nem foglalkozunk részletesebben. Az egyes adatátvitelek során az aktuális működési fázis szerint akár a mester, akár a szolga egységek lehetnek küldő vagy fogadó egységek. Az SCL órajel vezetéket mindig a mester hajtja meg, a szolga egység az átvitel végén – ha nem kész a következő adat fogadására – aktív állapotban (”lent”) tarthatja az órajelet, késleltetésre kényszerítve ezzel a mestert. Az SDA adatvezetéket az adatbitek átviteli ideje alatt mindig az aktuális küldő egység (adatok írásánál a mester, adatok olvasásánál a szolga) hajtja meg, a nyugtázóbit (ACK) átviteli ideje alatt pedig a fogadó egység vezérli. Ennek megfelelően, pl. olvasásnál a címzési fázisban a mester, az adatküldési fázisban a szolga a küldő fél.

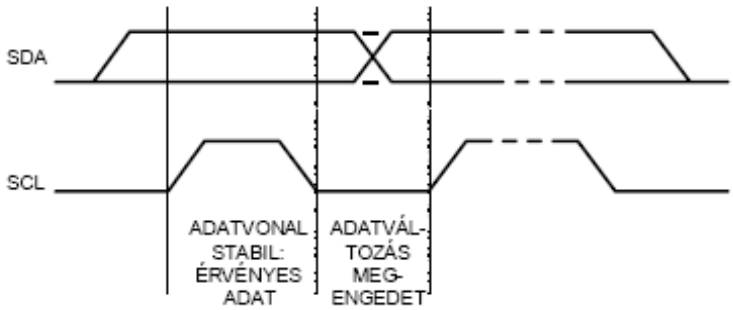


Adatkeret (frame) átvitele az I²C buszon

Az I²C buszon az adatátvitel keretek (frame) formájában történik. A kereteket minden esetben a START (S) és STOP (P) fázisok határolják. A START és STOP fázisok specialitása, hogy ekkor az SDA adatvonal az SCL órajel magas értéke alatt vált állapotot. Minden más esetben az SDA adatvonal az órajel magas értéke alatt stabil, új értékét az SCL alacsony állapota alatt állítja be. Egy átvitel során előfordulhat ún. ismételt START fázis is, tehát a START – START – STOP fázis sorozat is jelölhet érvényes kereteket.

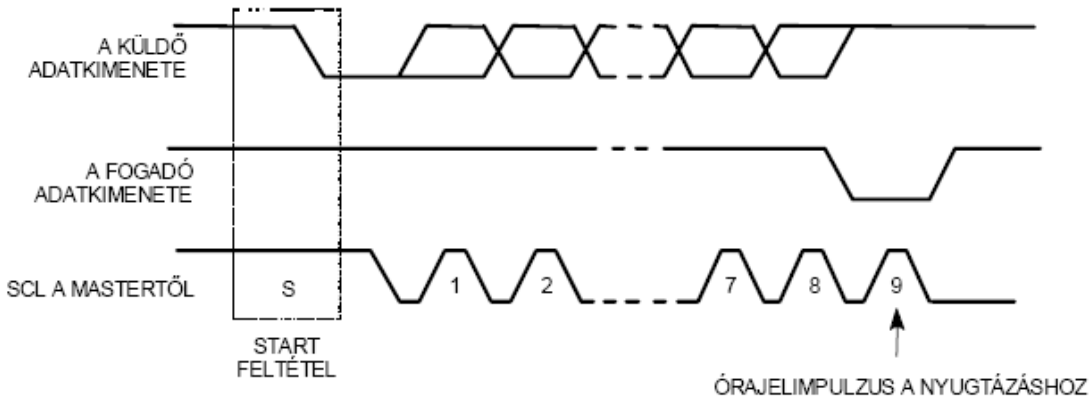


START és STOP fázis az I²C buszon



Adatbit átvitele az I²C buszon

Az I²C buszon egy-egy bájt átvitele 9 órajel ütemet vesz igénybe. Az első 8 ütem alatt az aktuális fázis küldő egysége a karakter bitjeit sorban (a legnagyobb helyiértékű bittel, az MSB-vel kezdődően) a buszra helyezi, majd a 9. ütemben a fogadó egység az átvitelt nyugtázza (aktív „0” szintű /ACK bit).



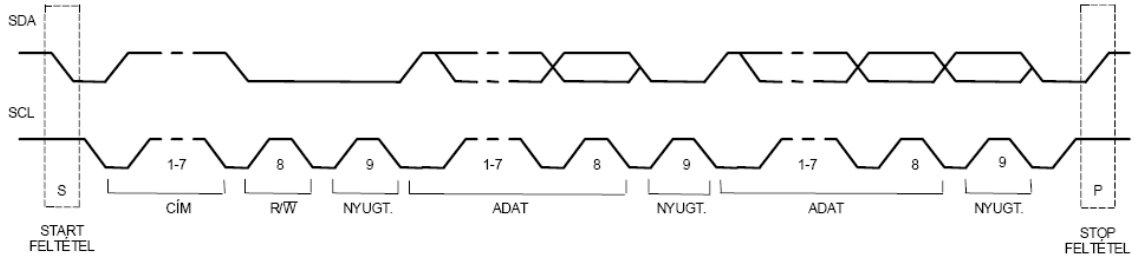
Egy adatbájt átvitele az I²C buszon

A nyugtázás elmaradása (NACK) a küldő egység számára az adatátvitel sikertelenségét jelzi. Ennek okai a következők lehetnek:

- a megszólított slave egység nem létezik
- a vevő elfoglaltsága miatt nem volt képes venni az adatot
- az adatátvitel értelmezhetetlen parancsot vagy adatot tartalmazott

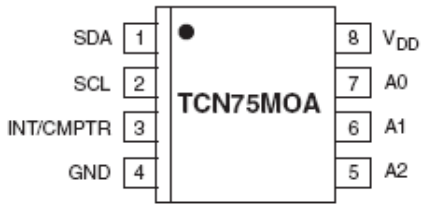
- adatbájtok egymás utáni (blokkos) átvitele esetén a vevő egység nem képes több adatot fogadni
- master vevő megteheti, hogy így jelzi a több bájtos adatátvitel végét, mivel a slave adó úgysem kezdeményezhet ismétlést.

Az I²C buszon több potenciális slave egység is jelen lehet, pedig nincsenek közvetlen chip-select jelek a megszólított résztvevő kiválasztására. Ezért a master által továbbított első bájt mindig tartalmazza a megfelelő egység címét (7 bit) és az adatátvitel irányát (R/W) a master szemszögéből nézve.



Egy összetett adatframe átvitele az I²C buszon (a példában 1 cím- és 2 adatbájt átvitele)

A 7 címbit felső 4 bitje általában egy adott gyártóra fix érték, az alsó 3 bit hardver úton kódolható az alkatrész lábain. Léteznek olyan áramkörök is, melyeknél valamennyi bitet fixen bekódolta a gyártó, ezekből kettő nem kapcsolható ugyanarra a buszrendszerre.



Hőmérő áramkör I²C interfésszel

Az ismertetett címzési eljárást később kiterjesztették 10 bitre (két 8 bites címbájt segítségével történik a címzés), ennek részleteivel itt nem foglalkozunk.

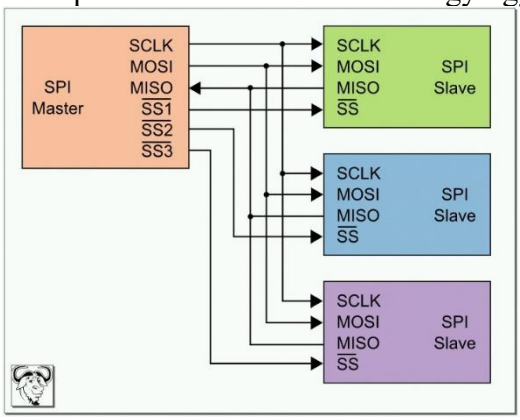
Az I²C protokoll adatátviteli sebességére a gyártók 4 kategóriát ajánlanak:

- Standard mód (SM): 0-100 kbit/sec
- Gyors mód (FM): 0-400 kbit/sec (a legelterjedtebben használt)
- Gyors mód plusz (FM+): 0-1 Mbit/sec
- Nagysebességű mód (HS): 0-3.4 Mbit/sec

A gyors plusz és a nagysebességű módnál már speciális áramköröket alkalmaznak a vonalak meghajtására az O/C kapcsolat aszimmetrikus meghajtási tulajdonságainak kompenzálása érdekében (ún. kapcsolt felhúzó áramkörök).

3.7.2 Az SPI adatátviteli interfész

Az SPI (Serial Peripheral Interface) is egy szinkron periféria adatátviteli interfész, amely az I^2C busznál nagyobb sebességű (1-25 Mbit/sec) full duplex adatátvitel lehetőségét nyújtja. Az interfész elemei 3 + (1..n) vezetéken keresztül kapcsolódnak egymáshoz, ahol n a slave egységek száma. A tipikusan használt adatátviteli sebesség 2-20 Mbit/sec. Az egységek közötti összeköttetések 3 vonalon busz rendszerűek, viszont az n darab kiválasztó vonal pont-pont kapcsolatot létesít a master és egy-egy slave egység között.

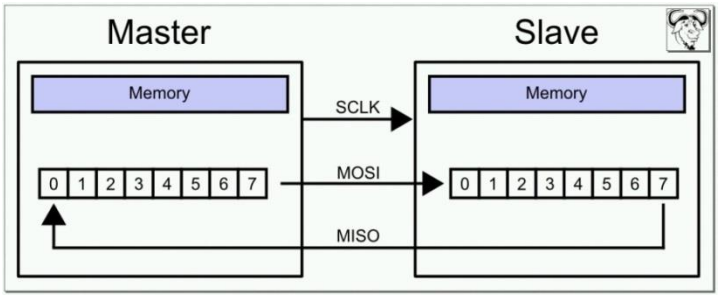


SPI buszkapcsolat 1 master és 3 slave egység között

Az egységek számát leginkább a kiválasztó vonalak száma korlátozza. A vonalak elnevezései:

- SCLK: Serial Clock
- MOSI (Master Output, Slave Input)
- MISO (Master Input, Slave Output)
- /SS (Slave Select)

Az SPI interfész belső kommunikációs egységeinek hardver felépítése lényegesen egyszerűbb, mint az I^2C interfész megfelelő egységeié. A slave egységeket közvetlenül a /SS kiválasztó jelek azonosítják, az átvitelben nincs sebességet csökkentő címzési fázis. A busz topológia követelményének megfelelően csak a kiválasztott szolga MISO kimenete engedélyeződik (ez háromállapotú kimenet kell legyen).

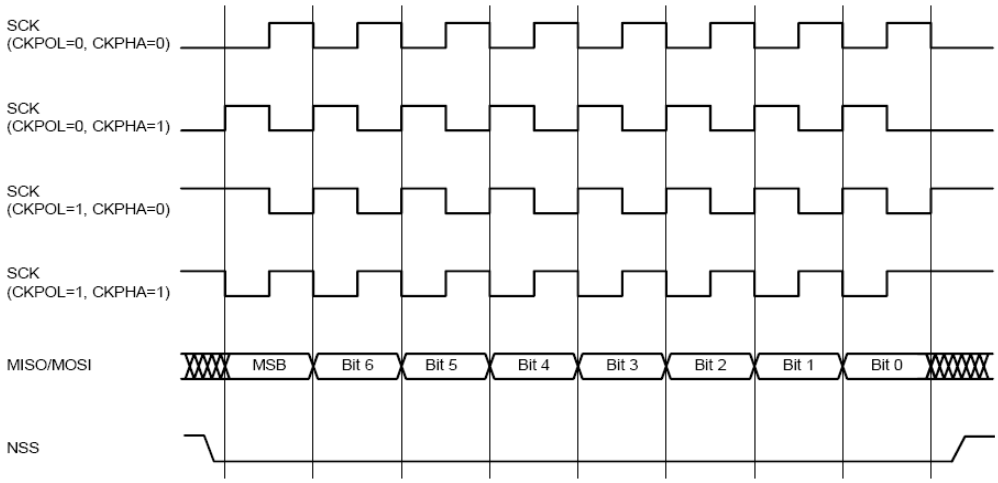


Az SPI adatkapcsolat modellje két egység között

Az adatbiteket a két egység ki- és bemeneti shift regiszterében egyszerre léptetjük, közös órajel hatására. Az órajel generálása az /SS kiválasztójel aktiválásakor indul és a 8 adatbit ki-/beléptetése után leáll. Az adatok kiléptetése és a beérkezett adatok mintavételezése az órajel egymással ellentétes éleire történik. Ennek megfelelően az órajel két lehetséges fázishelyzete (ponált, negált) és kétféle lehetséges nyugalmi/kiindulási állapotának megfelelően négyféle SPI átviteli ciklus (ún. SPI-mód) definiálható.

SPI-mód	CKPOL	CKPHA	SCK nyugalmi helyzete	SCK felfutó élére	SCK lefutó élére
0	0	0	alacsony (0)	Mintavétel	Adatbit váltása
1	0	1	alacsony (0)	Adatbit váltása	Mintavétel
2	1	0	magas (1)	Adatbit váltása	Mintavétel
3	1	1	magas (1)	Mintavétel	Adatbit váltása

Az SPI üzemmódok



Az SPI üzemmódok értelmezése

Az adatátvitelhez semmiféle előre definiált protokoll nem tartozik, az adatbájtok felépítése minden esetben a master és a slave egység definícióinak kérdése. Egyes áramkörök megengedik még a bitek léptetési sorrendjének megválasztását is, de alapvetően az MSB-vel kezdődő átvitel az általános.

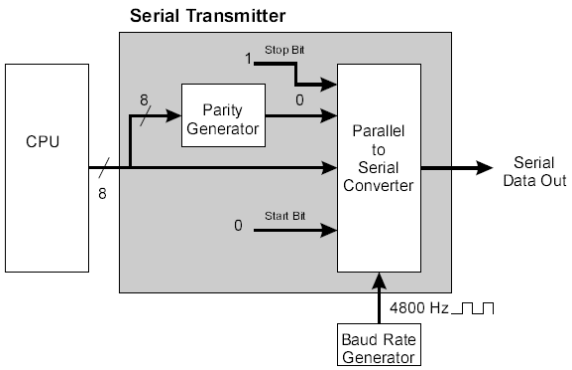
Az /SS kiválasztó jel lefutó élét több slave-egység gyártó is a kommunikációs interfész alaphelyzetbe hozására (is) használja a tápfeszültség megjelenése után. Ilyen egységek esetében a „mindig aktív” bekötés még egyetlen slave egység esetében sem megengedhető.

3.7.3 A UART adatátviteli interfész

Eddig egy számítógépen belüli alkatrészek kommunikációjával foglalkoztunk, a megismert adatátviteli interfészek mindegyike közös órajellel időzített, un. szinkron adatátvitelt valósított meg. Ennek oka elsősorban abban keresendő, hogy az egymáshoz

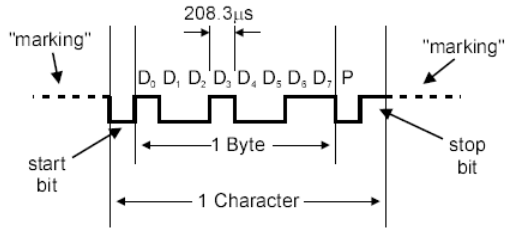
fizikailag közel (néhány cm) elhelyezkedő eszközök között nagyobb sebességű adatforgalom zajlik, itt a közös órajellel történő pontos időzítés létfontosságú követelmény.
A különböző számítógépek és fizikailag máshol található perifériáik között a távolság már lényegesen nagyobb (1 m – több 100 m). A közöttük zajló, kommersz vezetékeken zajló adatátvitel ennél csak kisebb adatátviteli sebességeket fog megengedni (max. néhány 100 kHz), viszont fontos ügyelnünk arra, hogy az egységek összekötéséhez minél kevesebb vezetékre legyen szükség. Ne gondoljunk most a speciális, legalább CAT6 típusú kábelt igénylő Ethernet kapcsolatra, ott a kábel speciális tulajdonságai (alacsony kapacitás, sodrott érpárok), a differenciális jelátvitel és lényegesen bonyolultabb jelkódolás teszik lehetővé az említettnél jóval nagyobb átviteli sebességet (pl. 1-10 Gbit/sec). A 0 – néhány 100 kbit/sec adatsebesség számára még megfelelő lesz a lényegesen egyszerűbb kialakítású, közös órajel meglétét nem igénylő aszinkron soros adatátviteli technika is.
Az aszinkron soros átvitel pont-pont típusú, tehát egyszerre két eszköz közötti kommunikációt tudunk megvalósítani. Amennyiben az átvitel egy adattovábbító csatornán (vezetéken) alapul, azon csak egyirányú (szimplex), vagy váltakozó irányú (félduplex) kapcsolatot tudunk kialakítani, két adatcsatorna esetén ez egy időben kétirányú (duplex) is lehet. (Szándékosan csatornáról és nem vezetékről beszéltünk, hogy ne zavarjon bennünket az áramköri földvezeték, vagy egy esetleges differenciális jelcsatorna esetén a vezetékek duplázott száma). Tipikus alkalmazása a berendezések közötti „hagyományos” adatkapcsolat megvalósítása (pl. számítógép és modem, számítógép és terminál között), beágyazott rendszerekben is gyakran használják. Ne keverjük azonban össze a fogalmakat: az aszinkron soros kapcsolat fizikai jeltovábbításához a berendezések között szabványos jelszintek alkalmazására lesz szükség. Ez utóbbiakat és a jelek további tulajdonságait írják elő aszimmetrikus jelvezetés esetére az RS232C, szimmetrikus duplex jelvezetésre RS422B, míg szimmetrikus félduplex jelvezetésre az RS485 szabványok. (A közismert RS232 elnevezés a Revised Standard 232 angol kifejezés rövidítése). Ezek a szabványok nem magát az aszinkron soros adatátviteli elvét definiálják, hanem annak fizikai rétegét, így ezekkel majd később, a környezeti kapcsolatok kiépítése témakör kapcsán foglalkozunk.

A soros aszinkron adatátvitel alapelve már szerepelt a Digitális technika 2 alaptárgyban ezért ezt csak emlékeztető jelleggel ismételjük át.



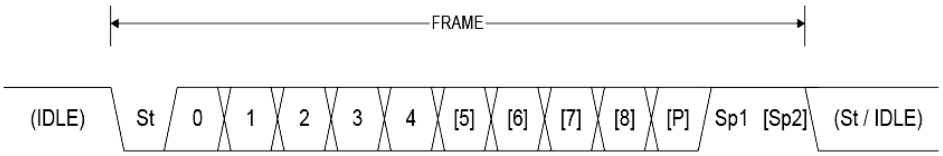
A soros adó egység működésének elve

A soros adó egység kimenete alaphelyzetben logikai „1” szinten (ún. „mark” jelszint) található. Az átvitel úgy zajlik majd, hogy minden egyes információs bit (ezek nem csak az adatbitek lesznek!) azonos ideig jelenik meg egymás után a jelvezetéken az átvitel során. Az átvitel sebességét ez a „bitidő” szabja meg, ezt nevezzük bitsebességnek vagy az angol elnevezése alapján bit rate-nek. Az elterjedten használt baudrate = bit/sec összefüggés esetünkben ugyan igaz (a két számérték megegyezik), de a különböző modulációs technikák esetén ez a két mennyiség egymástól el is térhet, ne szinonimaként jegyezzük meg ezt a két fogalmat !
Az adatátvitel egy bitidőnyi mindenképpen „0” szintű (ún. „space” jelszint) startbit átvitelével kezdődik. Ezt követik a léptető regiszterrel sorosított adatbitek a legkisebb helyértékű bittel kezdve, és egy esetleges – nem kötelező – paritásbit (ennek értelmezésére és képzésére később visszatérünk). Az adategység átvitelét a mindenképpen „1” szintű (mark) stopbit vagy bitek követi(k), és az adatvonal szintje ezen a szinten is marad a következő átviteli adatkeret (frame) kezdetéig. Vegyük észre, hogy a start bit „0” szintje mindenképpen jelszint-váltást jelent az előző karakter átvitelének befejezését jelző szinthez képest !



Soros aszinkron adatátviteli frame felépítése 4800 baud sebesség mellett

Az adatátvitel tehát keretekben (sokszor karaktereknek is nevezik őket) történik. A kapcsolat – mint már említettük – aszinkron volta miatt nem igényli külön órajel vezeték meglétét az eszközök között. Az egyes egységek időzítése saját lokális rendszeridőzítésükön alapul. Ebből kifolyólag az berendezések közötti adatátvitelben használható sebességértékek egy véges listából választhatók, valamint ezen sebességértékek forrásának frekvenciahibája nem lehet nagyobb kb. 1-2%-nál (indoklást ld. később). Az adatkeretek általános felépítése tehát a következő:



Soros aszinkron adatkeret általános felépítése

Összefoglalva eddigi ismereteinket minden adatkeret egy olyan egység, amely az adatbiteken kívül szinkronizációs biteket, továbbá hibaellenőrzés céljából paritás bitet tartalmazhat. Az adatkeret tulajdonságai paraméterekkel adhatók meg. A felépítés tehát a következő:

- 1 startbit (logikai „0” vagy „space” jelszint)
- Választható számú – 5, 6, 7, 8, esetleg 9 – adatbit
- 1 paritásbit, amely – ha van –, lehet páros vagy páratlan ill. fixen „1” vagy „0”
- Stopbit(ek), amelyekből 1 db megléte elengedhetetlen (a stop – start váltás felismeréséhez), de a korábban kialakult előírásoknak megfelelően számuk lehet 1, 1.5 vagy 2 is

Fenti tulajdonságokat a soros kapcsolat jellemzésére gyakran tömörített formában adják meg – a „8N1” leírás értelmezése: 8 adatbit, nincs paritás, 1 stopbit

Az egységek közötti adatkapcsolat vezérlésére az említett szabványok definiálják a hardver átvitelvezérlő jelek (DTR, DSR, RTS, CTS, DCD, stb.) használatának lehetőségét is, ezek azonban a most ismertetendő átvitel lényegét nem befolyásolják.
Az adatátviteli sebesség a két egység között a szabványok által rögzített értékek közül választható. A teljes tartomány a 110 bit/s-tól tipikusan 115200 bit/s–ig terjed (néhány egység tud még ennél nagyobb sebességet is).

Bitsebesség	18.432 MHz / bitsebesség	Megjegyzés
110	16 * 10473	Régi, nagyon lassú perifériák esetén (csak kompatibilitás)
300	16 * 3840	
600	16 * 1920	
1 200	16 * 960	Alacsony, napjainkban is használt sebességek (nagyobb távolságok)
2 400	16 * 480	
4 800	16 * 240	
9 600	16 * 120	
19 200	16 * 60	Napjainkban gyakran használt sebességek
38 400	16 * 30	
14 400	16 * 80	Modemek által használt sebességek
28 800	16 * 40	
57 600	16 * 20	Napjaink gyors kommunikációs sebessége
115 200	16 * 10	
230 400	16 * 5	Nem minden egység tudja

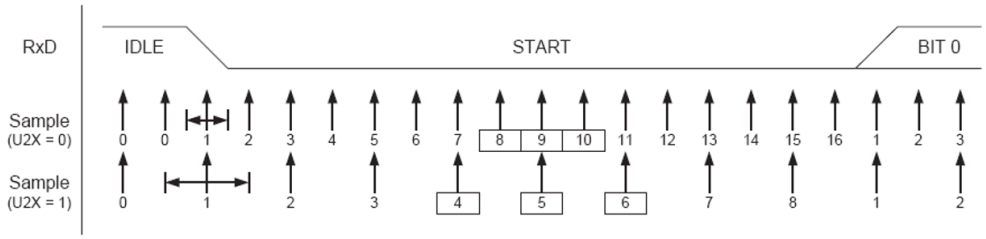
Soros aszinkron átvitelhez használatos baud rate-ek

Az aszinkron soros átvitel vételi oldalán a vevő áramkör jelentős, a bitsebességhez képest általában 16-szoros, (ritkábban 8-szoros) túlmintavételezéssel (oversampling) figyeli a bejövő vonalat. A tetszőleges időpontban beérkező adatkeret start bitjének lefutó éle szolgáltatja a szinkronizációs időpontot annak kijelölésére, hogy az adó és vevő egységek (nem pontosan megegyező) frekvenciáinak fázishelyzetét összehangoljuk. Ehhez gondoljuk végig a következőket:

- a beérkező biteket az órajelek lehetséges eltérése és a jelváltások átvitel során elszenvedett torzulásai miatt is egy-egy bitidő középpontjában próbáljuk meg mintavételezni
- az előbbi megállapítás miatt két tökéletes összhangban (szinkronban) lévő adó és vevő egység esetében is az órajelek abszolút értékének eltérése nem okozhatja a mintavétel ½ bitidőnél nagyobb elcsúszását. Ez 8 adatbit esetében (1 start-, 1 stop- valamint 1 paritásbitet feltételezve) az órajelek max. 0.5/11 = **4.55%**-os eltérését

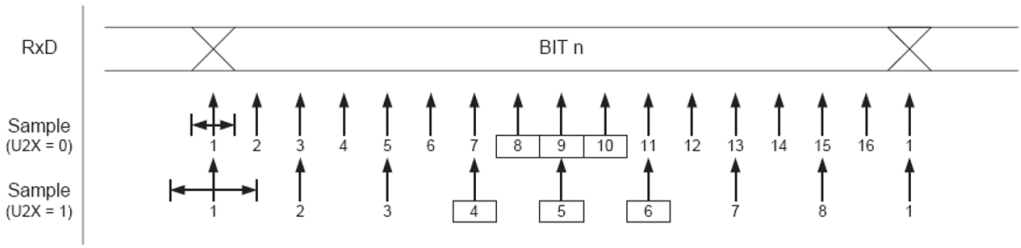
teszi lehetővé. Amennyiben a startbit lefutó élének felismerése 1/16 vagy 1/8 bitidővel „elcsúszhat”, ebből a fenti arány (1/2-1/16)/11 = **3.98%**-ra ill. (1/2-1/8)/11 = **3.41%**-ra változik.

- a bitidők határán a fizikai jelek nem képesek pillanatszerűen megváltozni. Amennyiben feltételezzük, hogy a jelek változásánál kb. 10-10% bitidő szükséges stabilizálódásukhoz, a fenti összefüggések számértékei (0.45-1/16)/11 = **3.52%** ill. (0.45-1/8)/11=**2.95%** lesznek.
- az előző pontban kapott toleranciahatárokat az adó és vevő egységek között „egyenletesen” elosztva érthető, hogy a frekvenciaalapok megkívánt pontossága kb. **±1.75%** ill. **±1.5%** lesz.



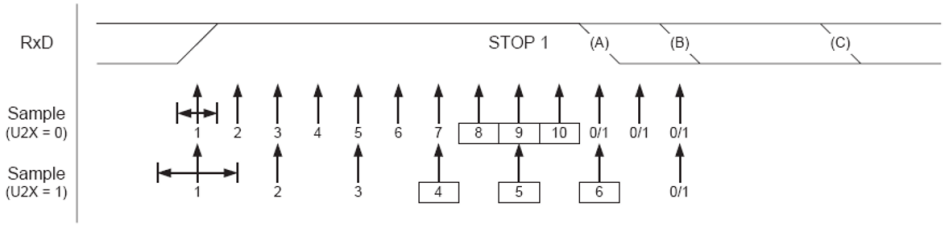
Az aszinkron soros átvitel start bitjének detektálása

A start bit jelváltozásakor alaphelyzetbe állítjuk a mintavételezést időzítő számlálót (az ábrán 1 -es állapot, majd a bitidő közepén (9 -es ill. 5 -ös állapot) történik meg a bitek mintavételezése. A zavarok kiszűrésére a logikák ún. többségi mintavételezést alkalmaznak, azaz 3 egymást követő mintából (pl. 8 , 9 , 10 állapotok) azt a jelszintet „hisszük el”, amelyik kétszer, vagy háromszor fordult elő a minták között.



Az aszinkron soros átvitel adatbitjeinek mintavételezése

Az adatbitek után következő stop bit mintavételezése jelenti a vétel befejezését



Az aszinkron soros átvitel stop bitjének mintavételezése

A vevő áramkör a stop bit mintavételezése és kiértékelése után azonnal képes a következő karakter vételére. Az éppen fogadott adatkeret stop bitjének vége elvileg ugyan a (C) időpontban várható, de mivel a vevő már az (A) időpontban befejezte az adatkeret kiértékelését, ezt követően újra képes a következő keret stop–start (1⇒0) váltásának az időpontját figyelni.
A paritásbit generálása és ellenőrzése is a következő szabályok szerint történik:

- letiltott paritás esetén az adó nem generálja, a vevő nem ellenőrzi a paritásbitet, annak helyét sem feltételezi egyikük sem.
- fix (0 vagy 1) érték esetén az adott érték kerül generálásra és ellenőrzésre
- a páros ill. a páratlan paritás a következőképpen értelmezendő:

$$P_{even} = d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$
$$P_{odd} = d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

Fenti összefüggéseket egyszerűbben megfogalmazva a paritásbit az adatbitekben található egyesek számát **párosra** (páros paritás) vagy **páratlanra** (páratlan paritás) egészíti ki.

Meg kell emlékeznünk még néhány olyan további fontos működési elvről, melyre az aszinkron soros adatátvitel használata során szükségünk lesz:

1. Amennyiben a vevő a vett paritásbitet más logikai jelszinten érzékeli, mint az adatbitek által általa számított érték, a státuszában **paritáshibát** jelez (ez általában megszakításkérés forrása is lehet).
2. Amennyiben a vevő a start-/stopbit(ek) mintavételezése során helytelen jelszintet érzékel, **kerethibát** (frame error) jelez (a vevők többségében ez is megszakításkérést válthat ki).
3. Az egységek fizikai kapcsolatát úgy szokták kialakítani, hogy a megszakadt jelkapcsolat (szakadt jelvezeték) folyamatos „0” (space) szintet okozzon a vevő bemenetén. A megismert konvenciók alapján az 1⇒0 átmenetet a vevő startbitként értelmezi, a beinduló vétel során mindvégig „0” adatbiteket érzékel, majd a helytelen („0”) stopbit(ek) miatt kerethibát jelez. Fenti feltételek együttes meglétét a korszerű vevők vonalszakadási (un. **break**-) hibaként érzékelik és jelzik vissza. Ennek a feltételnek az értelmezése sajnos nem egységes, vannak vevők, melyek csak több karakternyi „0” szint esetén jeleznek break feltételt.

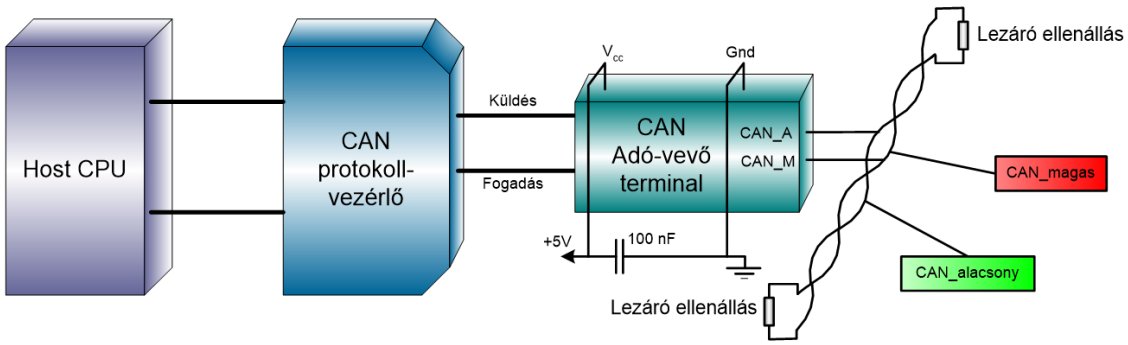
Az aszinkron adatátvitel sebességének felső határát a fizikai jelváltások korlátos volta és az 1⇒0 / 0⇒1 váltások aszimmetrikus torzulása okozza. Ezek az idők ugyanis abszolút időegységben mérendők és nem arányosak a frekvenciával – ily módon az adatátviteli sebesség növekedésével a korábban említett ±5% bitidőnek megfelelő jelváltási idő aránya egyre nagyobb mértékben növekedni fog, irreális mértékben lecsökkentve az adó és a vevő órajel-generátorainak frekvencia-toleranciáját.

3.7.4 A CAN adatátviteli interfész

A CAN betűszó a Controller Area Network angol kifejezés kezdőbetűiből tevődik össze. Az elsősorban gépjárművekben történő felhasználásra szánt és szabványosított (ISO 11898, 1991) CAN rendszer kialakítási és fejlesztési munkáit a **Robert Bosch GmbH** mérnökei kezdték meg még a 80-as évek elején. Célkitűzésük az volt, hogy a járművekben uralkodó, az elektronika számára sokszor meglehetősen nehéz működési feltételek mellett is biztonságosan működő, egyszerű és olcsó kommunikációs rendszert hozzanak létre. A feladat tömören a következőképpen volt megfogalmazva: kidolgozandó egy olyan soros aszinkron adatkommunikációs rendszer, mely egyetlen buszvonal (tehát két vezeték) segítségével képes a gépjárművekben található mikrokontrollerek és perifériáik között megbízható, valós idejű vagy időosztásos adatátvitelt biztosítani. A rendszer lehetőleg több master egység fizikai jelentétét is tegye lehetővé (multimaster jelleg). Ez a feltétel azért teszi bonyolultabbá a kialakítást, mivel a több master egység számára egyértelműen definiálandó, hogy közöttük a buszért való versengés (arbitráció) milyen feltételek között kell történjen – mikor, milyen feltételek mellett lehet adatátvitelt kezdeményezni és információt a buszra helyezni. Szemben az Ethernet kapcsolatnál megismert CSMA-CD protokollal cél volt a lehetőleg roncsolás mentes arbitráció, ami nem teszi szükségessé az információcsomagok ismétlését ütközés esetén.

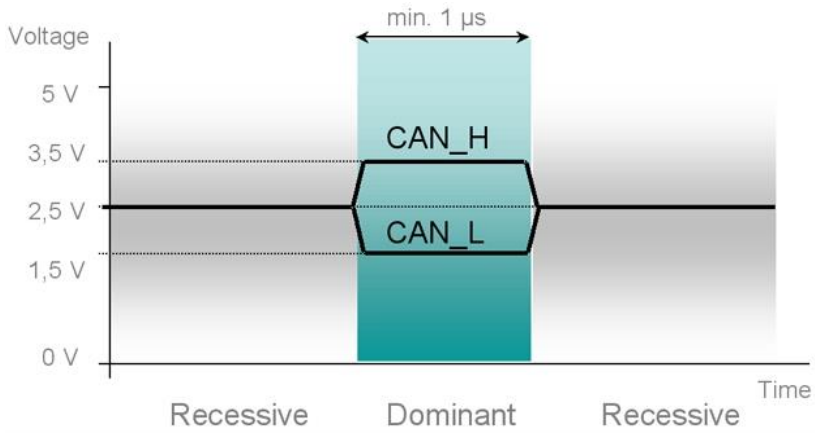
A megoldás oly mértékben sikeres lett, hogy az autóipar nagy feladatán túlmenően a különböző egyéb elektronikai vezérlő egységek (ECU: Electronic Control Unit) összekapcsolására is elterjedten használják napjainkban (az összes CAN alkalmazások mintegy 60-70%-a nem az autóiparban történik). Komoly nagy egységes kommunikációs protokollrendszert alakítottak ki az egységek kommunikációjának összehangolására (CAN-Open, ill. a járműiparban elterjedt SAE-J1939).

A rendszer felépítése igen egyszerű: a funkciójában teljes mértékben specifikált CAN vezérlő áramkört önálló egységként illesztik a mikroprocesszorokhoz, de a mikrokontrollerek és a DSP-k egy jelentős része rendelkezik integrált CAN-kontrollerrel. A vezérlő még minden esetben logikai szintekkel dolgozik, ehhez egy egyszerű meghajtó áramkört illesztve már csatlakoztathatjuk is egységünket a CAN buszra.



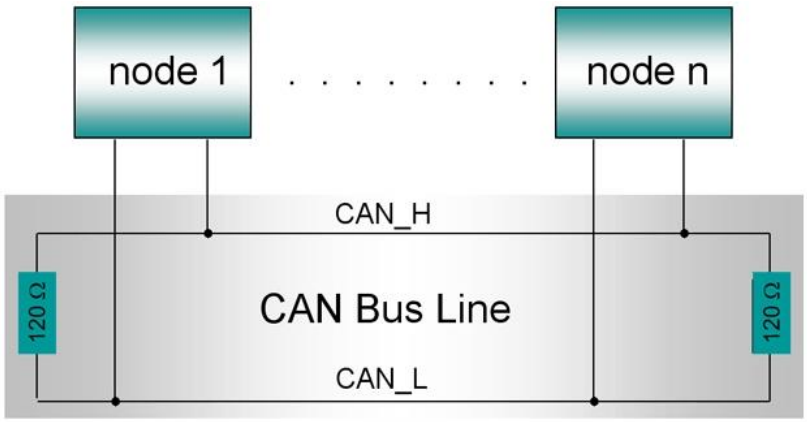
CAN illesztés tipikus kialakítása

A buszon differenciális jelátvitel történik, a két jelvezetéken (CAN_H és CAN_L) nem a megszokott logikai jelszinteken továbbítjuk az információt.

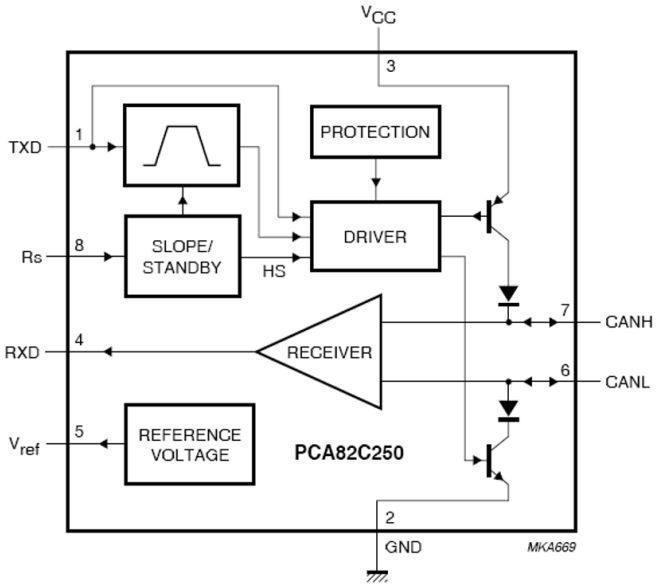


Jelszintek a CAN buszon

Lényeges, hogy részleteiben is megértsük a fenti jelszintek keletkezését. A logikai „1” jelszintet a továbbiakban recesszív (visszahúzódó, engedékeny), míg a logikai „0” jelszintet domináns (uralkodó) jelszintnek nevezzük. A buszillesztő áramkör domináns jelszintre „széthúzza” a fizikai jelvezetékeket az azokat összekötő 60Ω (2 x 120Ω párhuzamosan kapcsolva) ún. lezáró ellenállás ellenében, recesszív jelszintre kikapcsolja a meghajtó tranzisztorokat, így a busz két vezetéke potenciálisan „összezáródik” a lezáró ellenállások hatására. Könnyen beláthatjuk, hogy az alkatrészek károsodása nélkül megengedhetjük több meghajtó esetében a kimenetek összehajtását is, azonos (domináns-domináns ill. recesszív-recesszív) meghajtás esetén a kívánt jelszint fog kialakulni a buszvezetékeken, különböző (domináns-recesszív) meghajtók esetében az uralkodó „széthúzó” áramkör fog győzni a passzív „engedő” áramkörrel (igazából a buszt lezáró ellenállásokkal) szemben.

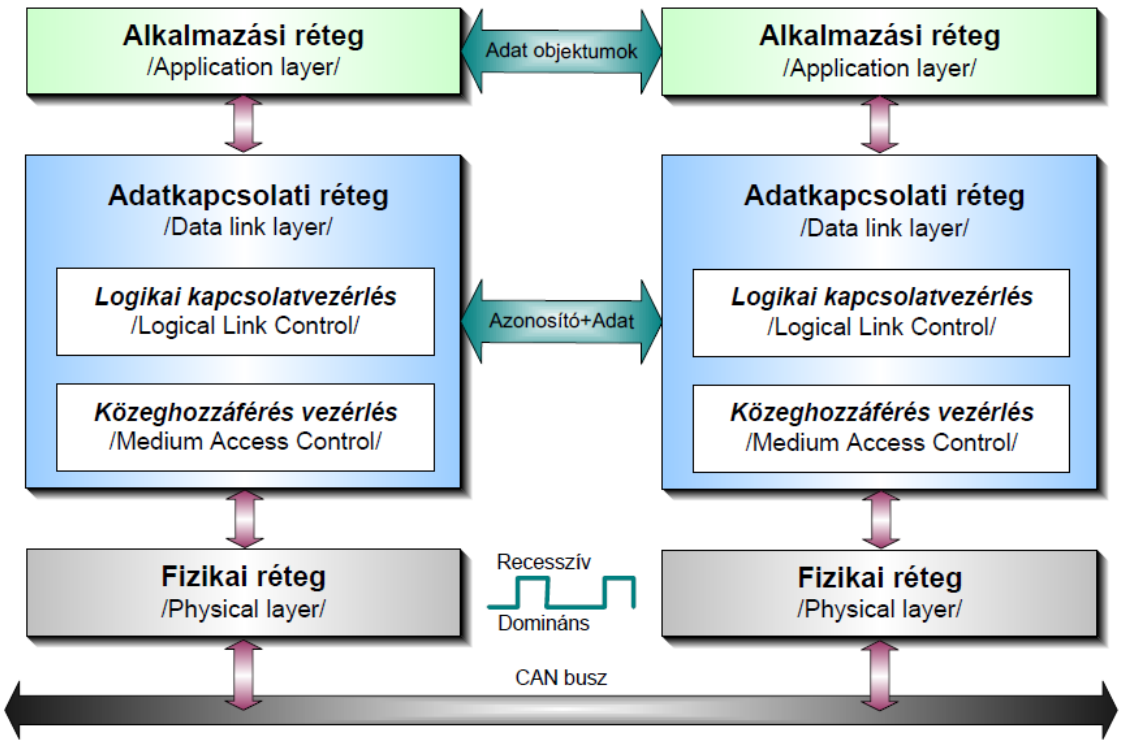


CAN busz topológia



CAN fizikai interfész (PCA82C250) blokkvázlata

A fizikai interfész blokkvázlatán jól látható, hogy a meghajtó csak a domináns („széthúzott”) állapotot hajtja meg aktívan, a jelvezetékek „összehúzására” nincs aktív eszköze. A buszvezetékekre azonnal egy differenciális vevő is kapcsolódik, amely a busz mindenkori fizikai állapotát látja (tehát a saját maga által kiváltott adást is).

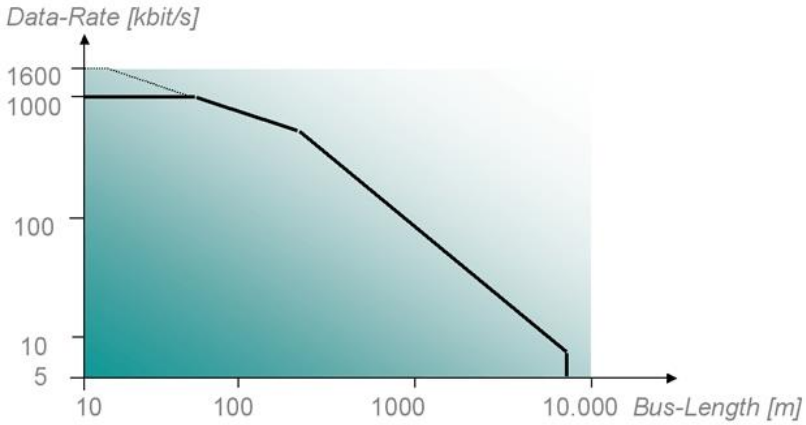


ISO/OSI rétegek a CAN busz esetében

A CAN busz legfontosabb jellemzői a következők:

- Busz topológiát valósít meg (ún. szórásos típusú hálózat, a hálózatra adott valamennyi jelet minden résztvevő veszi)
- A vázolt tulajdonságok miatt tetszőleges topográfia alakítható ki (szokásos a busz, pont-pont, vagy csillag jellegű elrendezés használata)
- A huzalozott vagy-kapcsolat (bárki domináns jelszintet hajt a buszra, az érvényre fog jutni) megengedi a többszörös hozzáférést és a nem destruktív ütközéskezelést
- Többféle kábeltípussal is megvalósítható, tipikus a sodrott érpár használata
- Felépítése nem teljesen igazodik az OSI modellhez, az abban definiált rétegek nem válnak el élesen egymástól.

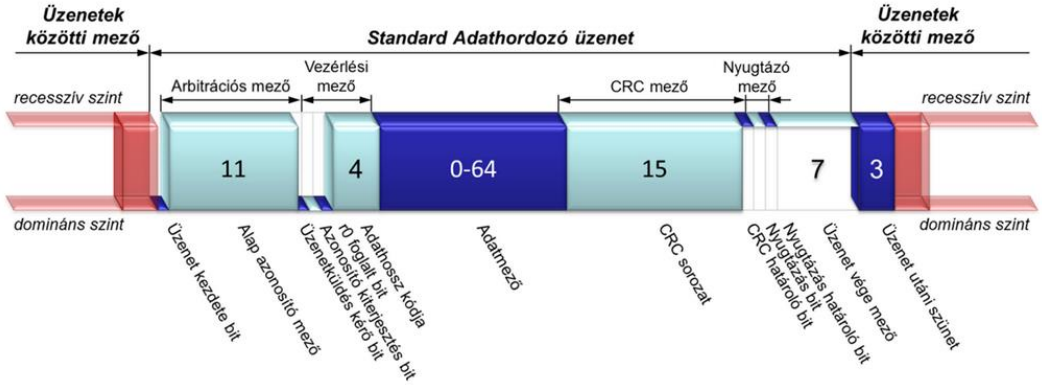
Az elmondottak alapján teljesen logikus, hogy az adatvezeték tulajdonságai (kapacitás, induktivitás) miatt a jelvezetékeken torzuló jelalak és a jelterjedési sebesség miatt is a buszon alkalmazható maximális adatátviteli sebesség függ a busz teljes hosszától. Az illesztő áramköröket és a jelszinteket úgy alakították ki, hogy a maximális adatátviteli sebesség (1 Mbit/sec) csak max. 30 m buszhossz mellett garantálható – alacsonyabb bitsebességek esetén a busz hossza nőhet. Az elmondottak miatt az esetleges galvanikus leválasztás (pl. optocsatolók) miatt keletkező időkésésekre a rendszer elég érzékeny: maximális bitsebesség (1 Mbit/sec) esetén tipikusan 40 MHz (!) átvitelére alkalmas optocsatolókat szoktunk alkalmazni a számottevő jelcsúszások elkerülése érdekében.



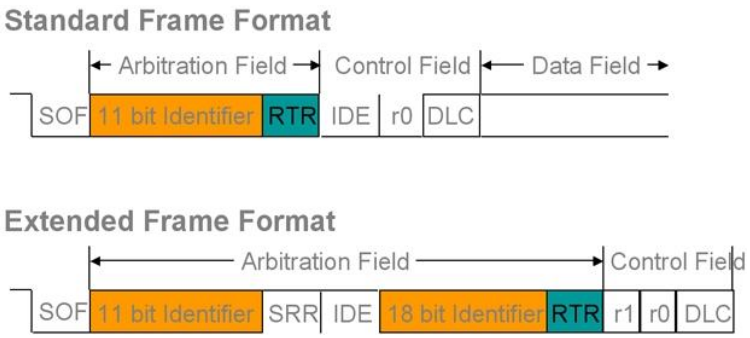
Bit Rate	Bus Length	Nominal Bit-Time
1 Mbit/s	30 m	1 µs
800 kbit/s	50 m	1,25 µs
500 kbit/s	100 m	2 µs
250 kbit/s	250 m	4 µs
125 kbit/s	500 m	8 µs
62,5 kbit/s	1000 m	20 µs
20 kbit/s	2500 m	50 µs
10 kbit/s	5000 m	100 µs

Maximális adatátviteli sebesség a CAN busz hosszának függvényében

A továbbítandó információt a CAN buszon is un. keretekbe (frame) foglaljuk. A keretek általános felépítése a következő:



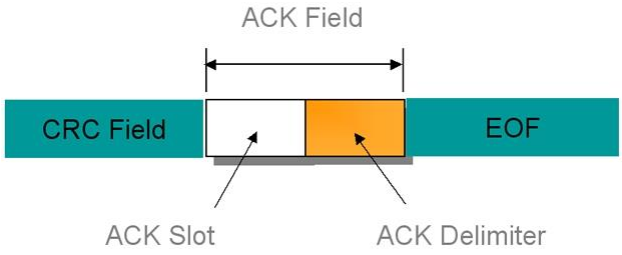
A (standard) *CAN keretformátumok felépítése*



A CAN keretformátumok adatok előtti részének felépítése

- Minden keret egy startbittel (SOF: Start of Frame) indul
- Ezt egy 11 bites üzenetazonosító (ID) követi, ennek fontos szerepe lesz az üzenetek (adók) közötti arbitrálásban
- Bármely egység adást kérhet egy másik egységtől RTR=1 jelzéssel (RTR: Remote Transfer Request). Az ún. remote frame nem tartalmaz adatokat. Az RTR bit olyan értelemben az ID-hez tartozik, hogy részt vesz az arbitrálás folyamatában.
- A vezérlő mezőt a mindig domináns IDE (ID Extension) és r0 (reserved) bitek, valamint az adathosszt jelző, DLC (Data Length Code) képviseli, ami 0..8 értéket vehet fel (ez lesz az adatbájtok száma).
- A CAN2.0B szabvány kiterjesztette az azonosítót 29 bitre úgy, hogy az RTR helyén levő SRR (Substitute Remote Request) és az IDE biteknél fixen recesszív („1”) szint kerül továbbításra – IDE=1 jelzi a vevőnek, hogy az azonosító további 18 bittel folytatódik –, majd ezt követik a már megismert RTR, két foglalt (r1, r0) bitek és az adathossz.

A tényleges adatátvitel az adatmezőben történik, melyben a DLC mező által megadott számú (0-8) adatbájt kerül továbbításra. Megengedett a 0 bájt adattartalmú üzenet továbbítása is.



A CAN keret záró részének felépítése

- Az adatmező után 15 bites CRC (Cyclic Redundancy Check) ellenőrző összeg, és egy ezt kiegészítő recesszív („1”) bit következik. A CRC-t a SOF, az arbitrációs, a control és az adat mezők alapján kell képezni.
- Az ACK és az ACK határoló mezőt az adó recesszíven hajtja meg, minden résztvevő, aki (CRC-) helyesen vette az üzenetet, köteles dominánsan nyugtázni azt (még akkor is, ha nem neki szólt az üzenet – ld. később). Ebből tudja az adó, hogy az üzenet sikeresen továbbításra került a buszon, hiszen „valaki” megértette azt.
- A keretet egy 7 bites EOF (End of Frame) recesszív bitekből álló mező zárja.

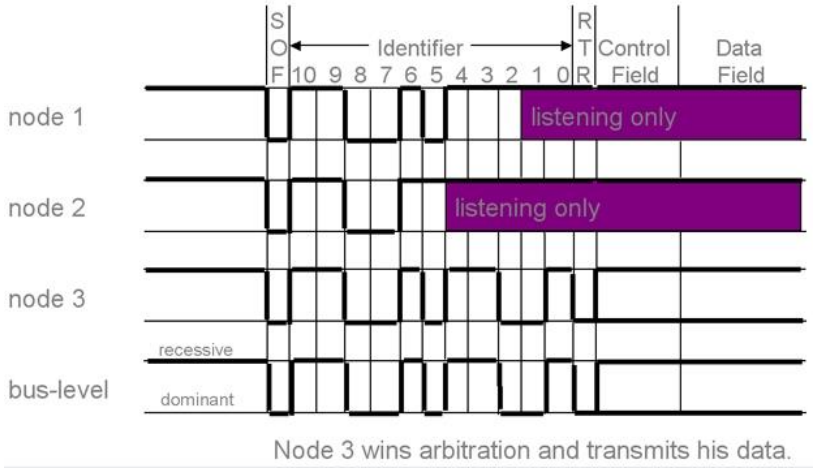
Az elmondottak alapján egyszerűen kiszámítható, hogy egy

- standard CAN üzenet mérete 44-108 bit
- kiterjesztett CAN üzenet mérete 64-128 bit

lehet (mint később látni fogjuk, ezek a számok még az ún. automatikusan beszúrt bitek számával bővülhetnek). Meg kell jegyeznünk, hogy a CAN szabvány minden keret után egy min. 3 bites recesszív szintet (Intermission) ír elő a következő üzenet megkezdéséig – ez a 3+7=10 bites mező jelenti a belépési feltételt a következő üzenet megkezdésére várakozó adók számára.

Nézzük ezek után a nem destruktív arbitrálás folyamatát! (CSMA/CA: **C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision **A**voidance)

Minden CAN üzenet egy azonosító (ID) mezőt tartalmaz. Az adásra várakozó master egységek adásukat csak akkor kezdhetik meg, ha senki sem forgalmaz a buszon. Ha valaki „hamarabb” kezdte el adását a buszon, abba a többiek már nem avatkozhatnak bele. Ha két vagy több adó egy nagyon szűk intervallumon belül mégis egyszerre kezdene el adni, akkor következik be a bitütközéses arbitráció. Minden adó egység bitről bitre ellenőrizni köteles (vevő egysége segítségével) az általa meghajtott bit helyességét a buszon. Ez a startbit (SOF) esetében nem fog eltérést mutatni, de az arbitrációs mezőben az üzenetek azonosítói már el fognak térni egymástól. Ütközés esetén a domináns szint fog nyerni, a recesszív szintet hajtó adó késedelem nélkül abba kell hagyja adását és vevővé kell változnia. Az elmondottakból következik, hogy az az üzenet fog győzni, amelyben az MSB felől nézve a leghamarabb lesz „0” bit, azaz számértékre nézve a kisebb szám prioritása lesz a magasabb.



Arbitráció 3 egység között – node 3 nyeri el az adás jogát

Vegyük észre, hogy az arbitráció az egységek között anélkül zajlott le, hogy a nyertes adó üzenete megsérült volna – ő mit sem tudva a versengésről folytatja az üzenet továbbítását. Két azonos ID-jú üzenet versengése esetén csak különböző adattartalom okozhat problémát – amennyiben egy adó hibás bitet (recesszív helyett dominánst) észlel, azonnal erőszakosan beleavatkozik az adásba és ún. hibakeretet generál.

A CAN szabvány részletesen foglalkozik a hibaesetek lehetőségeivel és az egységek erre vonatkozó reakcióival. Ismerkedjünk meg először a lehetséges hibaállapotokkal.

Minden CAN egység rendelkezik két belső hibaszámlálóval (TEC – Transmitter Error Counter ill. REC – Receiver Error Counter), melyek az észlelt hibajelenségek szerint automatikusan számol felfelé/lefelé. A hibaszámlálók 8 bitesek és a következő esetek szerint történik a számlálás:

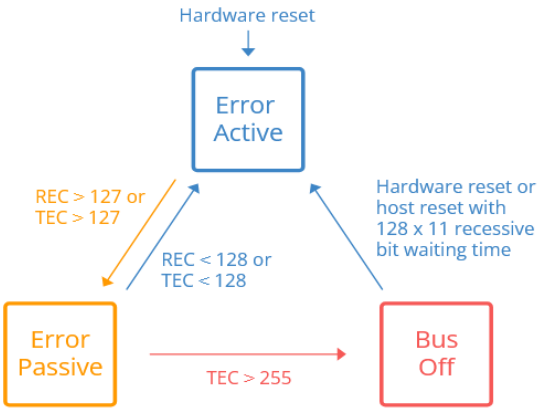
- amennyiben az adó hibátlanul továbbított egy üzenetet, a számláló értéke csökken (de természetesen az értéke nem csökkenhet 0 alá);
- amennyiben a vevő hibátlanul vett egy üzenetet, a számláló értéke csökken (de természetesen az értéke nem csökkenhet 0 alá);
- az adó és a vevő egységek vevő hibákat észlelhetnek, melyek a következők lehetnek:
    - **Bit hiba**: az egység domináns/recesszív bitet továbbít, de ezzel ellentétes szintű bitet olvas vissza a saját vevője (adó érzékeli);
    - **Bitbeszúrás hiba**: a vevő egység 6 azonos szintű bitet érzékel egymás után a SOF és a CRC bitek között (vevő érzékeli – a bitbeszúrás fogalmával később foglalkozunk);
    - **Form hiba**: az egység hibás logikai szintet érzékel a SOF/EOF mezőkben vagy az ACK/ACK határoló bitek között (vevő érzékeli);
    - **ACK hiba**: az egység kiküldött üzenete hatására nem érkezik meg a nyugtázás (adó érzékeli);
    - **CRC hiba**: a kiküldött és a vevő által számított CRC értéke nem egyezik (vevő érzékeli).

A fenti felsorolásban előforduló hibákat tipikusan az eseményt elsődlegesen érzékelő egység fogja észlelni és jelezni (ezeket aláhúzással jelöltük), de egyes hibákat – mint pl. a bithiba, a bitbeszúrás hiba – más egységek is észlelhetnek, ezeket másodlagos észleléseknek nevezzük. A számlálók reakciója a fenti eseményekre a következő:

TEC +8	ha az adó elsődleges hibát észlel
REC +8	ha a vevő elsődleges hibát észlel
REC +1	ha a vevő másodlagos hibát észlel
REC -1	ha a vevő sikeresen vett egy üzenetet
TEC -1	ha az adó sikeresen továbbított egy üzenetet

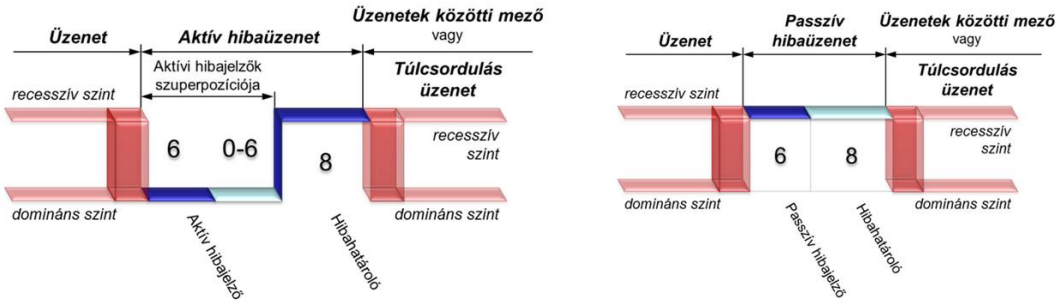
A hibaszámlálók logikája a CAN buszon

A 8 bites számlálók a félérték (128) és a maximális érték (255) határán a következő hibaállapotot váltják ki az egységekben:



A CAN egységek lehetséges hibaállapotai

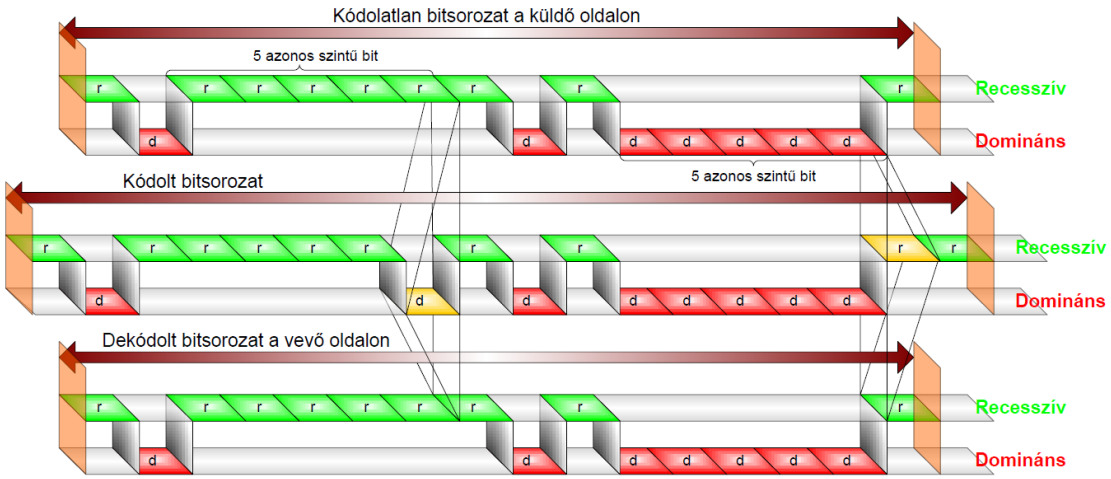
Egy csomópont ”hiba aktív” (Error Active) állapotban van, ha a saját hibaszámlálója félérték (128) alatt van. Hibát észlelve az egység ilyenkor ún. **aktív hibaüzenetet** fog küldeni a buszra. Ilyenkor a csomópont biztos benne, hogy hiba történt, és azt nem ő okozta. Egy csomópont ”hiba passzív” (Error Passive) állapotban van, ha átlépte a hibaszámláló félértékét (nagy valószínűséggel helyi meghibásodása van), de még nem olyan súlyos a helyzet, hogy le kelljen válnia a buszról. A csomópont ilyenkor hiba észlelése esetén csak ún. **passzív hibaüzenet** küldésére képes. A maximális számlálóérték elérése esetén az egység le kell váljon a buszról (Bus Off) és csak a vezérlő egység nyugtázó resete után térhet vissza újra aktív állapotba.



Hibakeret generálása üzenethiba észlelése esetén (aktív és passzív)

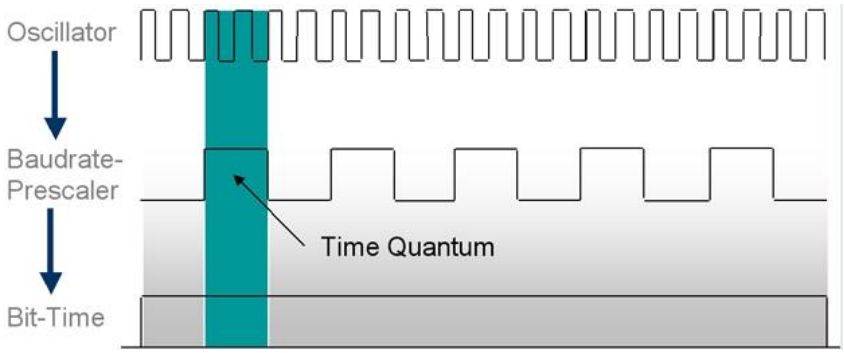
A hibaüzenet 6 azonos állapot továbbítását jelenti, lévén a többi egység lehet, hogy később észleli a hibaállapotot és kezdi meg a hibaüzenet továbbítását, ily módon a buszon 6-12 azonos jelszint továbbítása is megjelenhet a hiba jelzésére.

Jogos lehetne a kérdés: miért egyértelmű hibajelzés ez, mi van, ha az éppen aktív adó üzenetében is 6 db domináns bit következett volna? Nos, ez nem fordulhat elő, mivel bármelyik adó kötelessége, hogy amennyiben 5 domináns bit került egymás után továbbításra, automatikusan egy recesszív bitet kell beszúrnia az üzenetbe (ez az ún. bitbeszúrás, amely meghosszabbítja az üzenetet), amit a vevő szintén automatikusan el fog távolítani (bit beszúrás és eltávolítás, ez csak a CRC végéig érvényes, a frame végét jelző bitekre nem).



A bitbeszúrás (bit stuffing) elve a CAN buszon

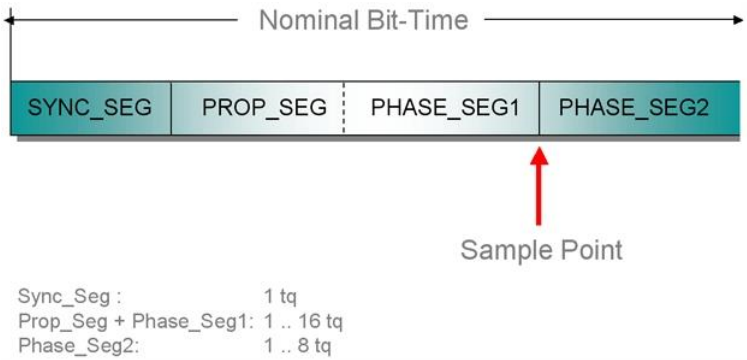
Fentiek után egy fontos kérdés vetődhet fel bennünk: hogyan zajlik a leírt aszinkron kommunikáció időzítése és szinkronizációja, hiszen itt legkevesebb 44, de lehet, hogy 128 bitet kell átvinnünk egy üzenetben! A CAN szabvány pontosan előírja a mintavételezés és a bitidőzítés eljárásait, amik érthetően lényegesen bonyolultabbak, mint a UART-nál látott feltételek. Haladjunk lépésről lépésre az eljárás megértéséhez!



Az időkvantum értelmezése a CAN buszon

A kontroller órajelét itt is előosztóval redukáljuk, azonban nem közvetlenül a bitidőre, hanem egy annál kisebb egységre, az ún. időkvantumra (tq: Time Quantum). Minden bit legalább 8, maximum 25 időkvantum terjedelmű lehet. Természetesen minél magasabb ez az érték, annál jobb felbontással tudunk igazodni a vétel során a vett bit időzítéséhez. Ennek részleteibe azért kell belemennünk, mert ez részben a felhasználó programozásának kérdése.

Minden bitidőt időszegmensekre bontunk a következők szerint:



Bitidő szegmensekre bontása

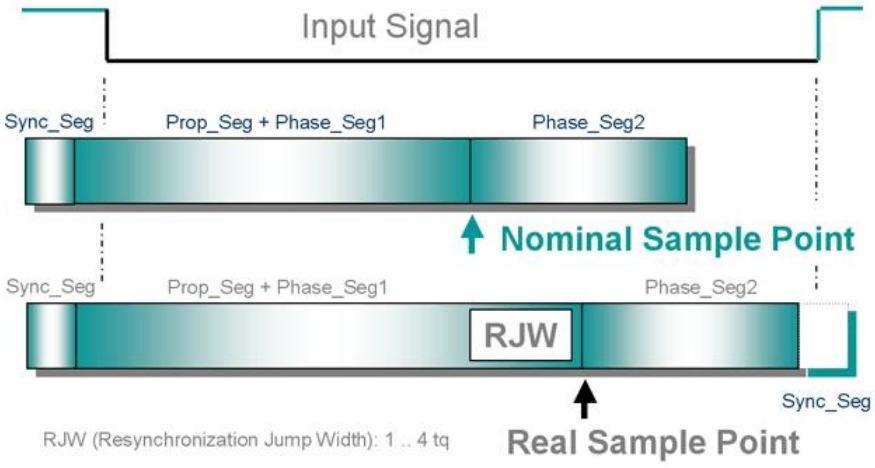
- Az ún. szinkronizációs szegmens (1 tq) alatt történik a jelváltás. Amennyiben történik jelváltás, de nem ebben a szegmensben, az újraszinkronizálást okoz. Figyelembe véve a bitbeszúrásnál elmondottakat, az átvitt keret legalább 5 bitenként jelváltást kell tartalmazzon.
- Az ún. jelterjedési (propagation) szegmens a fizikai jelterjedés miatti késést hivatott képviselni.
- Fentieket követően a maradék bitidőt két részre osztva (Phase_Seg 1 és 2) ezek határán történik meg a mintavételezés.

Ugyan a szegmensek értéke külön-külön min. 1 időkvantum lehet, a szabvány a szegmensek között további összefüggéseket is definiál, melyek 8 tq alatti értékeket meg sem engednek (néhány kontroller a 4 tq beállítását is megengedi, de ezt semmilyen

körülmények között sem ajánljuk). Nem is célszerű ilyet választani, inkább a magasabb tq értékek irányába „tereljük" a beállítást – magasabb alap-órajelek alkalmazásával. Pl: 1 Mbit/sec bitsebességnél

- 16-25 MHz között PS =1, a szegmensidőkkel állítsuk be a pontos osztásviszonyt
- 30-50 MHz-ig kénytelenek vagyunk már PS=2 –t állítani, és ismét 15-25 tq között állíthatjuk a szegmensek idejét.

Nem beszéltünk még a jelváltásonként megtörténő ún. újraszinkronizálás folyamatáról. Amennyiben a vevő úgy érzékeli, hogy a jelváltás a szinkronizációs szegmensen kívül történt, saját hatáskörben intézkedik a fázis szegmensek hosszának változtatásáról.



A bitidő növelése túl későn érkezett jelváltás esetén

Az RJW (Resynchronisation Jump Width, néha SJW, ami ugyanazt takarja a „Re-" előtag nélkül) paraméterrel azt adjuk meg programozáskor, hogy a vevő egyetlen újraszinkronizálás során mennyivel nyújthatja meg a bitidőt (túl későn érkezett él) ill. rövidítheti azt (túl korán érkezett él esetén).

A szabvány előírásainak megfelelően valamennyi CAN kontroller paraméterezhetővé kell tegye a felhasználó számára a fenti értékeket. Ezeket az alkalmazott sebesség és jelkésleltetést befolyásoló paraméterek (kapacitás és hosszviszonyok, topológia, esetleges leválasztás miatti jelkésés stb.) ismeretében tudjuk beállítani.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	TSeg2			TSeg1				SJW		BRP					
r	rw			rw				rw		rw					

Az időzítő regiszter felépítése a Bosch CAN User's Guide szerint

Az üzenetek adása-vétele a korszerű CAN kontrollerekben komoly regiszter- (nevezzük inkább memória-) igénnyel jár. A különböző üzenetek adása-vétele un. Message Object egységek segítségével történik, melyek mérete min. 10, általában 12 bájt, néha 16

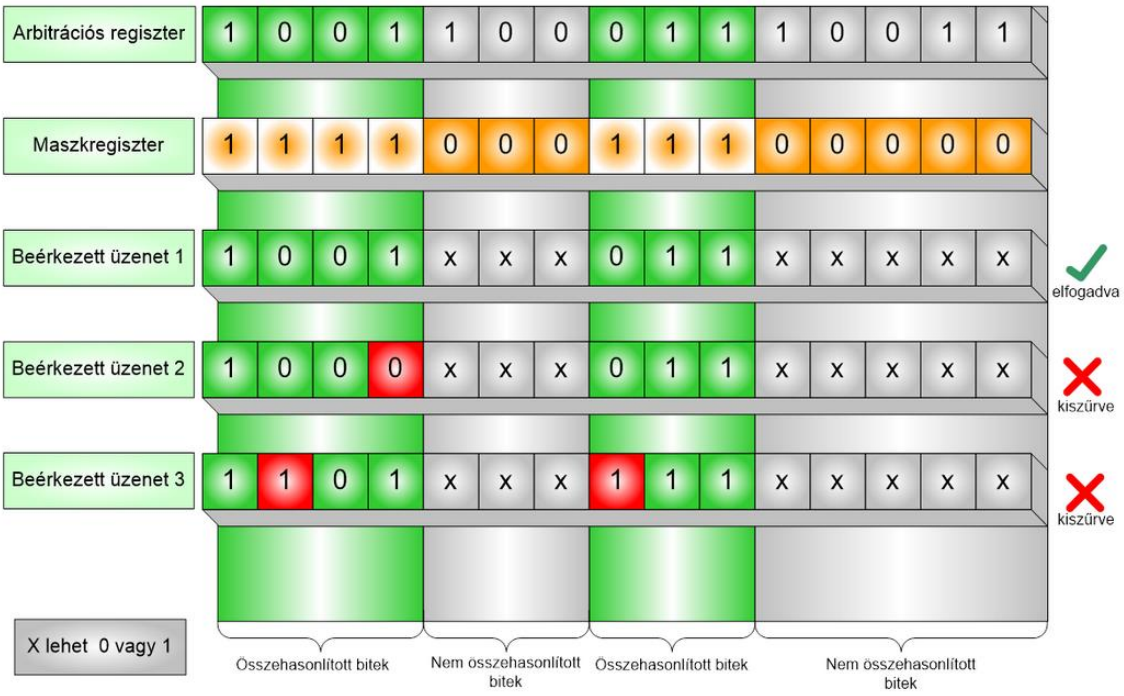
üzenetobjektumonként. Minden objektumban megadható kell legyen egy ID (2-4 bájt), egy adatterület (0-8 bájt) és még az objektum vezérléséhez szükséges vezérlőszavak is. Az objektumok általában konfigurálhatók, hogy az adót vagy a vevőt szolgálják ki. Ezekből az objektumokból az egyszerűbb CAN kontrollerek min. 3 darabot (1 adó, 2 vevő) a közepesek 16 darabot, a legigényesebbek 32/64 darabot is kínálnak.

	Offset
Message Control Reg. Low	+0
Message Control Reg. High	+1
Upper Arbitration Reg. Low	+2
Upper Arbitration Reg. High	+3
Lower Arbitration Reg. Low	+4
Lower Arbitration Reg. High	+5
Message Configuration Reg.	+6
Data Byte 0	+7
Data Byte 1	+8
Data Byte 2	+9
Data Byte 3	+10
Data Byte 4	+11
Data Byte 5	+12
Data Byte 6	+13
Data Byte 7	+14
Reserved	+15

Egy Message Object felépítése (Infineon C505)

Az adó objektumokban az ID mezőben azt adhatjuk meg üzenetenként, hogy az milyen ID mezővel kerüljön továbbításra. Ugyanez a mező a vevők esetében azt a célt szolgálja, hogy megadhassuk, milyen azonosítójú buszüzenetet kell elfogadnia a vevő objektumnak (ún. acceptance filter). Mivel a buszon sokféle azonosítójú üzenet közlekedhet, nagy hiba lenne mindegyik vétele és egyenkénti azonosítása. A korszerű CAN kontrollerek két lépcsős szűrőrendszert alkalmaznak az üzenetek vétele során:

- egy – általában valamennyi üzenetre közös – un. Acceptance Mask Registerben adhatjuk meg, hogy a beérkező üzenetek ID mezőjében mely bitek kerüljenek ellenőrzésre vétel szempontjából (=1), mely bitek közömbösek (=0). Egy csupa „1”-t tartalmazó maszk regiszter hatására egyetlen egy üzenettípus (ID) vételére lesznek alkalmasak objektumaink, csupa „0” esetén valamennyi, a buszon található üzenetet venni fogja objektumunk (némely kontrollereknél ez pont fordítva van).
- sok kontroller egy kitüntetett objektumra külön Maszk regisztert is megad, amiben az általánostól eltérő feltételt adhatunk meg az adott objektumra.



Üzenetek szűrése (acceptance filter & mask)

Mikrokontroller alapú rendszerek

Elektronikus jegyzet

4. fejezet

Készítette: Dr. Tevesz Gábor c. egyetemi tanár
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
Fax: 463-2871 (adm.)
Mail: tevesz@aut.bme.hu

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki alapszak (BSc)
III. évfolyam, 5. félév
Beágyazott és irányító rendszerek specializáció hallgatói

COPYRIGHT

TARTALOMJEGYZÉK

4. MIKROKONTROLLEREK KÖRNYEZETE, ILLESZTÉSEK

Ebben a fejezetben már elhagyjuk a mikrokontrollerek belsejét és a környezetükkel való kapcsolatukat vizsgáljuk – mikrokontrolleres rendszereket kezdünk építeni. A feladat rendkívül szerteágazó, a tárgy keretei között csak a legfontosabb, legtipikusabb problémákat és megoldásaikat tudjuk megvizsgálni. Viszont nyugodtan állíthatjuk, hogy azok a hallgatók, akik a vázolt témaköröket, áramköri megoldásokat megismerték és elsajátították, képesek lesznek a gyakorlati kapcsolásokban felmerülő további problémák megoldására is.

4.1 Tápfeszültségek

A digitális kapcsolások tápfeszültsége hosszú időn keresztül szinte kizárólag +5V volt. Ez a kijelentés az TTL logikai elemek megjelenése (kb. 1964) óta igaz. A bipoláris elemeken alapuló logikai kapuáramkörök (Texas 74xx sorozat) oly mértékben elterjedtek, hogy a szélesebb (3-15V) tápfeszültség tartományban is működőképes – de kezdetben lényegesen lassabb – CMOS elemcsaládok (CD4000-es sorozat, 1968), dacára az alacsonyabb teljesítményfelvételüknek háttérbe szorultak mindaddig, míg a két rendszer sebességben meg nem közelítette egymást. Jóval később, a 80-as évek közepén jelentek meg a mai napig oly népszerű 74HC, 74HCT sorozat elemei, melyekkel a logikai jelszintek már kompatibilissé tehetők voltak a családok között. A korábban mintegy egy nagyságrenddel lassabb CMOS kapuáramkörök működési sebességükkel kezdték utolérni bipoláris elődeiket, és lényegesen alacsonyabb fogyasztásuk révén lassan ki is szorították őket (ne feledjük: $I_{CC} = C_{pd} \cdot f \cdot V_{CC}$).

A működési frekvencia emelkedésével és az integráltsági fok rohamos növekedésével az áramkörökben keletkező veszteségteljesítmények exponenciálisan növekedtek. Megjelentek a 3,3V-os áramkörök és elemcsaládok, de a folyamat ezzel nem állt meg: jöttek a 2,5V, 1,8V, 1,5V és 1,2V-os tápfeszültséggel működő logikák. Utóbbiak a csökkenő jelszintek miatt lecsökkent jel-zaj viszony miatt nem igazán terjedtek el diszkrét logikai kapcsolásokban, inkább az egy egységbe integrált nagysűrűségű (VLSI) áramkörök belső egységeinek második-harmadik tápfeszültségeként (mikroprocesszorok magfeszültsége) használatosak.

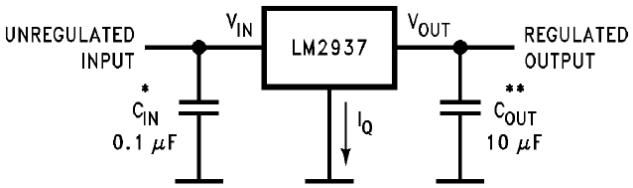
Logikai áramköreink napjainkban meghatározó mértékben 5V-os és 3,3V-os tápfeszültséget használnak – az alkalmazott elemek tápfeszültségei egyre inkább keverednek. Az ennél alacsonyabb tápfeszültségek többnyire a nagyintegráltságú, nagysebességű mikroprocesszorok (pl. DSP-k) segédtápfeszültségei.

A logikai áramköri kapcsolások stabil tápfeszültségeket igényelnek, ezeket az alkalmazási környezetből érkező többnyire stabilizálatlan, számottevő zajjal terhelt szabványos ipari tápfeszültségekből lokálisan állítjuk elő kapcsolásaink számára. Tipikus tápfeszültségek: 9V, 12V (autóipar), 24V (a legelterjedtebb ipari tápfeszültség), 48V. A törpefeszültség tartományba tartozó 24V-os tápfeszültség növelése már komolyabb

életvédelmi előírásokkal jár együtt, ezért a magasabb tápfeszültségek használatát többnyire csak a nagyobb teljesítmények igénye indokolja az ipari alkalmazásokban.

A tápegységek elméletével és kapcsolásaival önálló tárgyak keretében ismerkednek meg képzésünkben, ennek részleteire itt nem térünk ki. Csak áttekintő jelleggel mutatjuk be a legtipikusabb kapcsolásokat.

a) Soros áteresztő tranzisztoros (lineáris) tápegységek

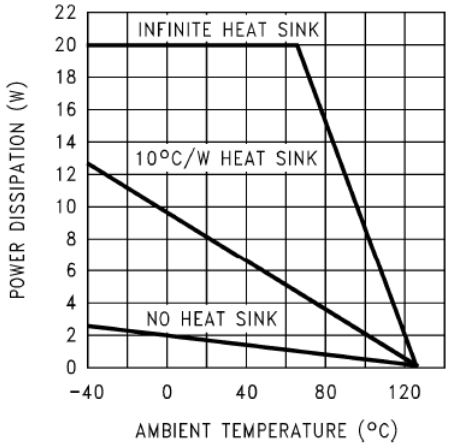


Soros áteresztő stabilizátor kapcsolás

Ez a tápegység típus felépítését tekintve igen egyszerű, a stabilizátor áramkörön túl alig igényel külső elemet. Lineáris jellegéből adódóan pontos, kis zajszintű tápfeszültséget szolgáltat, többnyire túlterhelés (rövidzár-) védett. Alkalmazásakor a legnagyobb figyelmet a rajta keletkező veszteségteljesítményre kell fordítanunk, ami a nyugalmi áram (I_Q) elhanyagolásával

$$\boldsymbol{P_D = (V_{IN} - V_{OUT}) \cdot I_T}$$

Ez a teljesítmény pl. $\boldsymbol{V_{IN} = 24V}$ bemenő és $\boldsymbol{V_{OUT} = 5V}$ kimenő feszültség és $\boldsymbol{I_T = 200mA}$ terhelő áram esetén $\boldsymbol{P_D = 3,8W}$, ami figyelembe véve az áramkör (tokozástól függő) veszteségteljesítmény-karakterisztikáját csak valamilyen hűtőfelület biztosításával viselhető el számára.

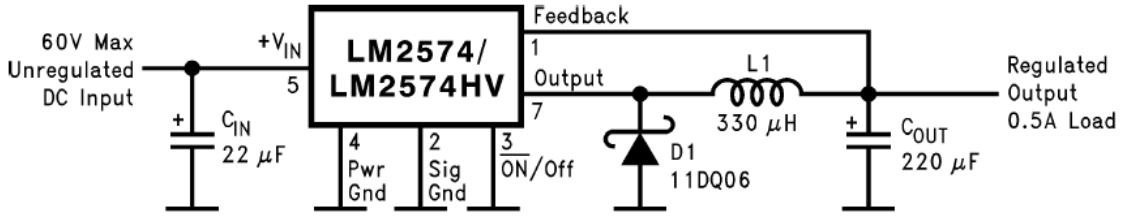


Veszteségteljesítmény a hőmérséklet függvényében (TO-220 tokozás)

A megoldás hatásfoka: $\frac{\boldsymbol{P_{hasznos}}}{\boldsymbol{P_{összes}}} = \boldsymbol{1W/4,8W = 20,8\%}$

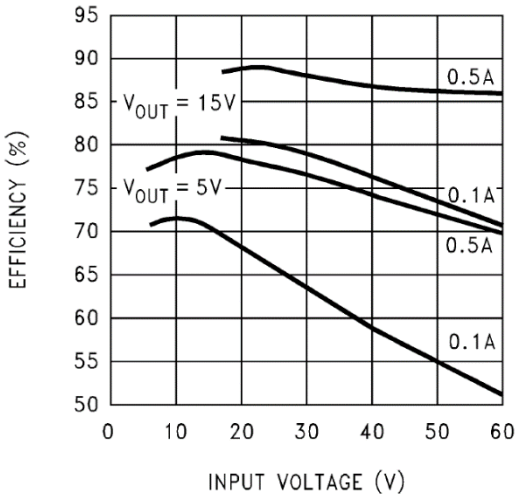
b) Kapcsolóüzemű tápegységek

Lényegesen jobb hatásfokot érhetünk el – némileg több alkatrész alkalmazásával – a kapcsolóüzemű tápegységekkel. A 20-1000 kHz frekvenciatartományban valamilyen fix kapcsolási frekvenciával működő kapcsolások ma már egyetlen áramkörbe integrálva állnak rendelkezésünkre, használatuk esetén a fő problémát a kapcsolás jobboldalán látható 3 elem: a Schottky dióda, a ferritmagos induktivitás és a jó nagyfrekvenciás tulajdonságokkal rendelkező puffer-kapacitás kiválasztása szokta jelenteni.



Kapcsolóüzemű feszültségcsökkentő tápegység

Kisebb kapcsolások tápegységeként többnyire extra hűtőfelület nélkül is működőképesek, mivel hatásfokuk lényegesen kedvezőbb az előbb említett lineáris társaiknál.

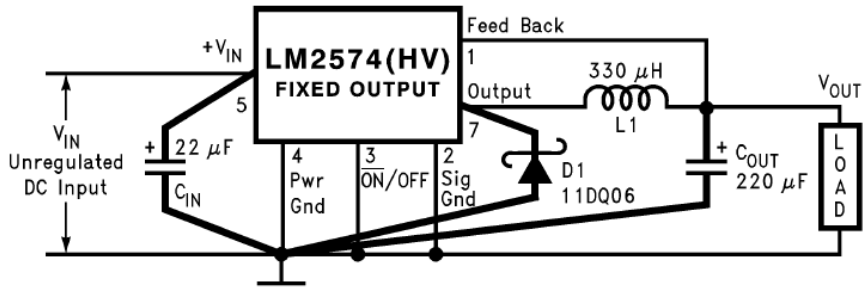


Kapcsolóüzemű tápegység hatásfoka

Előző példánkat erre az esetre alkalmazva kb. 70-80% hatásfokot várhatunk a kapcsolástól, ami ebben az esetben

$$P_D = P_{in} - P_{out} = P_{in}(1-\eta) = P_{out}\frac{1-\eta}{\eta} = 5V \cdot 0{,}2A \cdot \frac{1-0{,}75}{0{,}75} = 0{,}33W$$

Ez hozzávetőlegesen az egytizede az előző kapcsolás veszteségének.

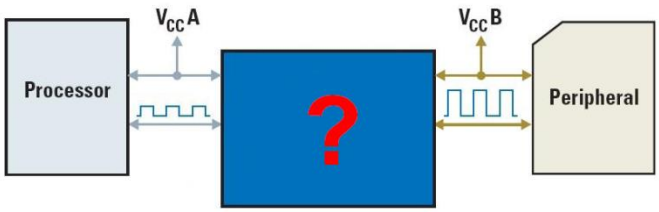


Fizikai kialakítás: a főhurok impedanciája a lehető legkisebb legyen

Ezeknek a kapcsolásoknak az áramköri kialakításánál nagyon kell ügyelnünk arra, hogy a nagy frekvenciával átterelődő áram útjában a huzalozásból adódó impedanciák értékét minimalizálnunk kell. A csillagpontszerűen kialakított közös földponthoz a három fő áramköri elem (a bemeneti, a kimeneti puffer-kondenzátor és a nulldióda) vastag és lehetőleg rövid vezetékekkel csatlakozzon.

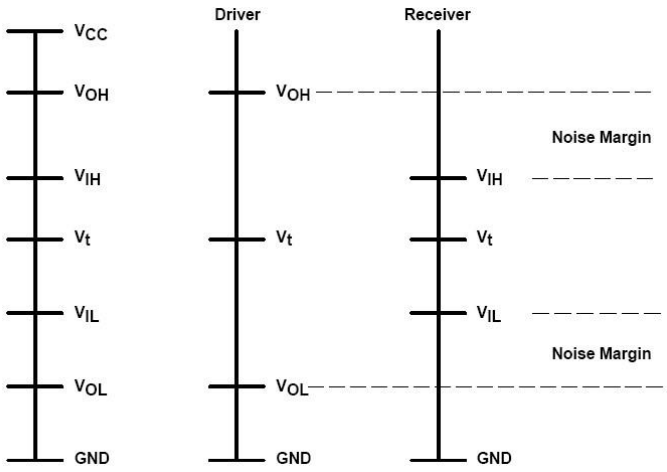
4.2 Logikai jelszintek

A logikai elemcsaládok különbözősége már erősen feszegette, de a különböző tápfeszültségek egy rendszerben való előfordulása végképp szükségessé teszi a logikai jelszintek összehangolását az egyes elemek között. Ebben a fejezetben ezt a témakört tekintjük át.



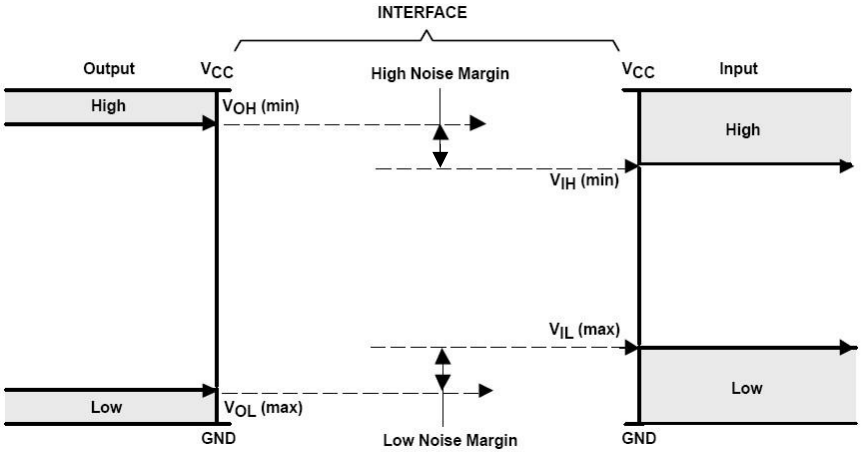
Tipikus probléma: VccA ≠ VccB

Napjainkban a leggyakrabban megoldandó feladat az 5V és a 3,3V tápfeszültségű logikai áramkörök (processzorok, memóriák, perifériák, kapuáramkörök) egymáshoz történő illesztése. A még Digitális technikában tanult elméletre alapozva emlékezzünk arra, hogy bármely illesztés során a meghajtó áramkör V_{OH} és V_{OL} kimeneti jelszintje magasabb ill. alacsonyabb kell legyen, mint a fogadó áramkör V_{IH} és V_{IL} küszöbértékei. A megfelelő értékek közötti távolságot nevezzük zajtávolságnak – a kimenő jelre kerülő ekkora zajfeszültség még nem okozhat hibás jelszint-felismerést a fogadó áramkörben.



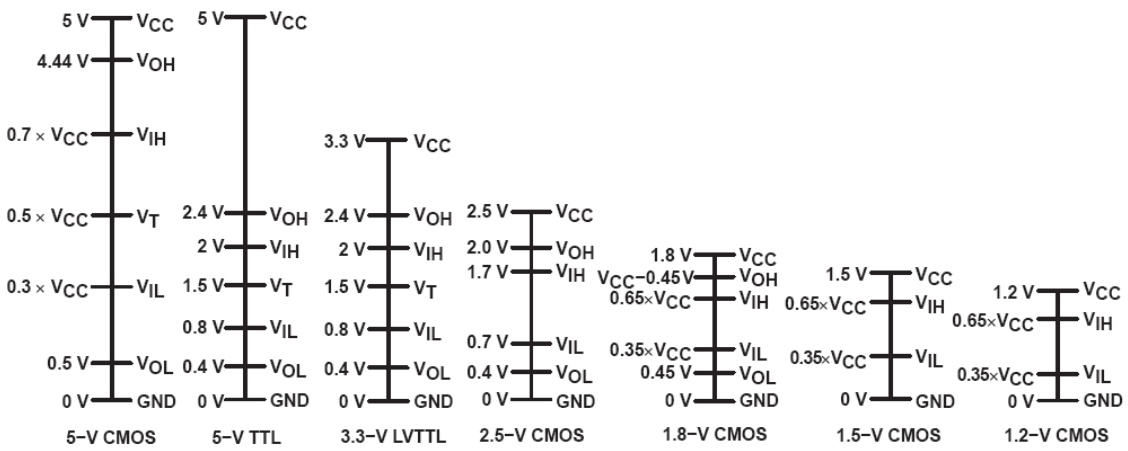
Meghajtó és fogadó jelszintjeinek értelmezése

Mind a kimeneti, mind a bemeneti küszöbszintek egy-egy tartományt határolnak. Kimenet esetében a kimenő jel szintje V_{OHmin} és a tápfeszültség (V_{CC}) közé ill. V_{OLmax} és a földpotenciál (GND) közé fog esni. Mindkét küszöbérték a terhelőáram (I_{OH}, I_{OL}) függvénye, ezt a kapcsolás tervezésénél figyelembe kell vegyük. Hasonlóan alakulnak a sávok a bemeneteknél – itt viszont a V_{IHmin}-V_{CC} ill. a V_{ILmax}-GND sávokban biztonságos a jelszintek helyes felismerése. A kimeneti sávoknak teljes terjedelmükben „bele kell találniuk" a bemeneti sávokba, a határok közötti tartalék jelenti ismét a zajtávolságot.



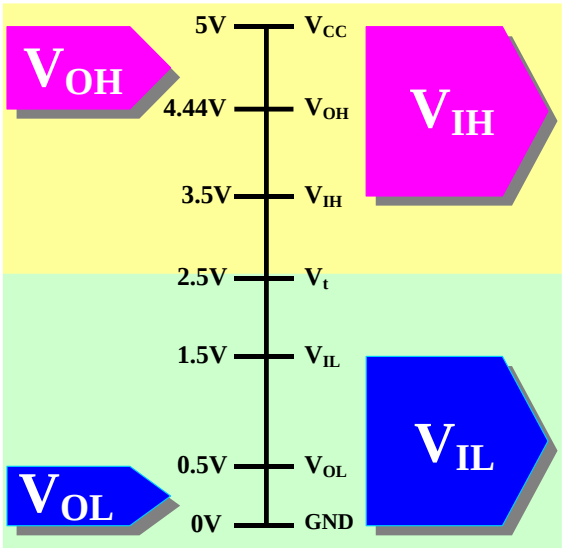
Meghajtó és fogadó jelszint sávok valóságos áramköröknél

Az eddig elmondottak általában automatikusan teljesülnek azonos elemcsaládon belül, nem feltétlenül a különböző elemcsaládok között és szinte biztosan nem a különböző tápfeszültségről működő elemek között. Elborzasztásul tekintsük a különböző tápfeszültséghez és elemcsaládokhoz tartozó jelszint-összehasonlító diagramot.



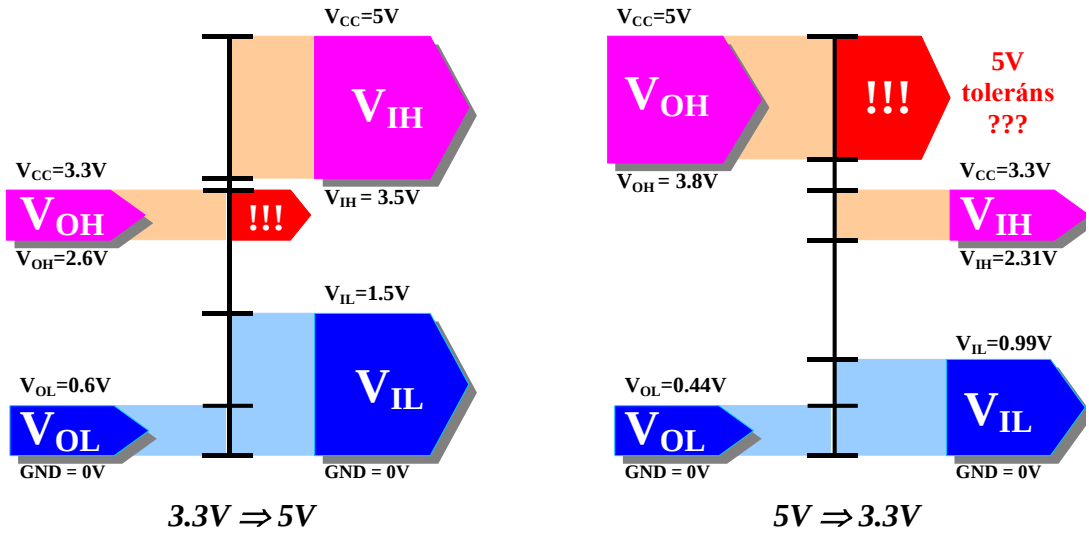
A ma használatos tápfeszültségekhez tartozó logikai jelszintek

Nézzük a konkrét értékeket! Az 5V tápfeszültségről működő CMOS logikai kapcsolások (processzorok, perifériák, HC típusú kapuk) áramköri felépítésükből adódóan általában a tápsínekhez elég közeli kimeneti jelszintet állítanak elő, billenési feszültségük (V_t) a tápfeszültség félértékénél, logikai jelszintek küszöbértékei tipikusan a V_{IL} = 0,3·V_{CC} (=1,5V) és V_{IH} = 0,7·V_{CC} (=3,5V).



5V-os CMOS logikák tipikus jelszintjei

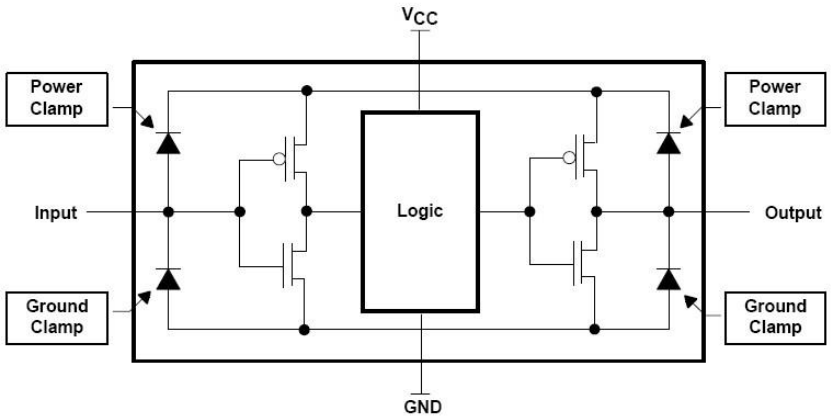
Amennyiben ezt a szabályt 3,3V-os CMOS logikai elemek esetére alkalmazzuk, V_{IL}=0,99V és V_{IH}=2,31V küszöbértékeket kapunk. Ha közvetlenül összekapcsolunk 5V és 3,3V tápfeszültségű elemeket, két helyen is kritikus probléma vetődik fel.



3,3V ⇔ 5V (CMOS) jelszintek keveredése

Az első problémát az 5V-os bemenet 3,3V-os kimenetről történő meghajtása jelenti. Az ábrán jól látható, hogy a kimeneti jeltartomány feszültségszintje alacsonyabb, mint a fogadó áramkör bemeneti feszültségszintjei, az áramkörök közvetlen összekapcsolása esetén a logikai „1” szint hibátlan felismerése nem garantálható. A logikai „0” jelszintek tökéletesen illeszkednek egymáshoz, itt nincs további teendő.

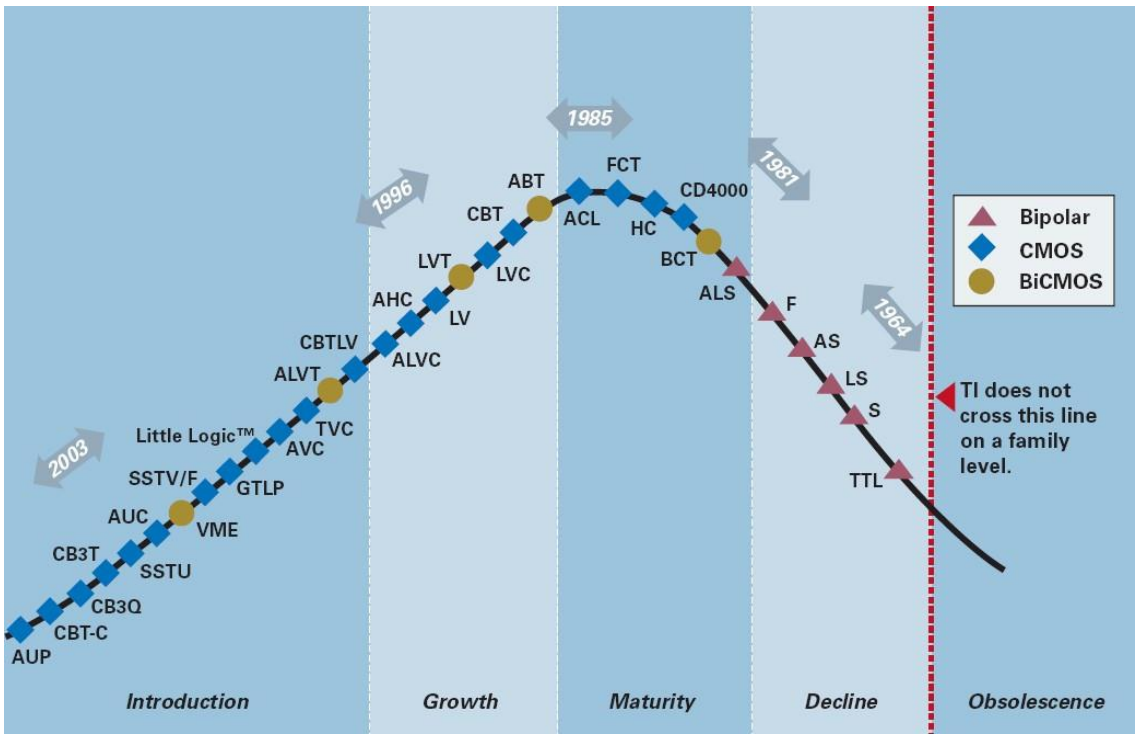
A második probléma a 3,3V-os logika 5V-os jelszintekkel való meghajtásakor vetődik fel. Első ránézésre a kimenő jelszintünk „1” szintű feszültségértékei nem lehetnek alacsonyabbak, mint a bemenő tartomány, sőt – erősen felette találhatók. Pontosan ez okozza a problémát! CMOS áramkörök védelmére a gyártók gyakran alkalmaznak a ki- és bemeneteken ún. vágódiódákat (clamp diode), megvédendő az áramkör belsejét az akár sztatikus (energia nélküli) túlfeszültségektől is.



Vágódiódák CMOS logikai áramkörökben

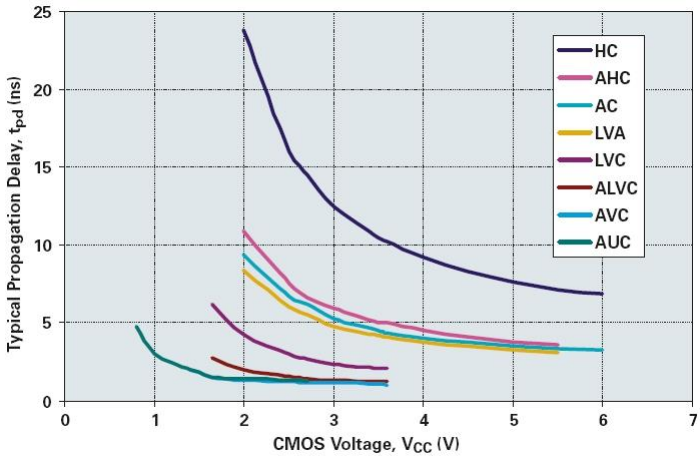
Nem nehéz belátnunk, hogy a bemenetre kapcsolt V_{CC} feletti jelszintre a felső bemeneti vágódióda nyit és vezérlő kimenő jelünk árama „tápegységként” viselkedve megemelheti az alacsony fogyasztású 3,3V-os áramköri elemek tápfeszültségét – szélsőséges esetben akár 5V közelébe is! A 3,3V-os rész tápegységének szabályozója nyilván teljesen lezár, de lévén csak „termelni” tud, „fogyasztani” nem – tehetetlenül nézi a tápfeszültség emelkedését, ami akár az áramköri elemek tönkremenetelét is okozhatja.

Hogyan keressünk megoldást a fenti problémákra?
A logikai elemcsaládoknak rendkívül széles választéka áll rendelkezésünkre. Érdekességként nézzük meg a Texas Instruments által gyártott különböző logikai elemcsaládokat bemutató ábrát, ami egyben az időben is segít elhelyezni ezeket a családokat.



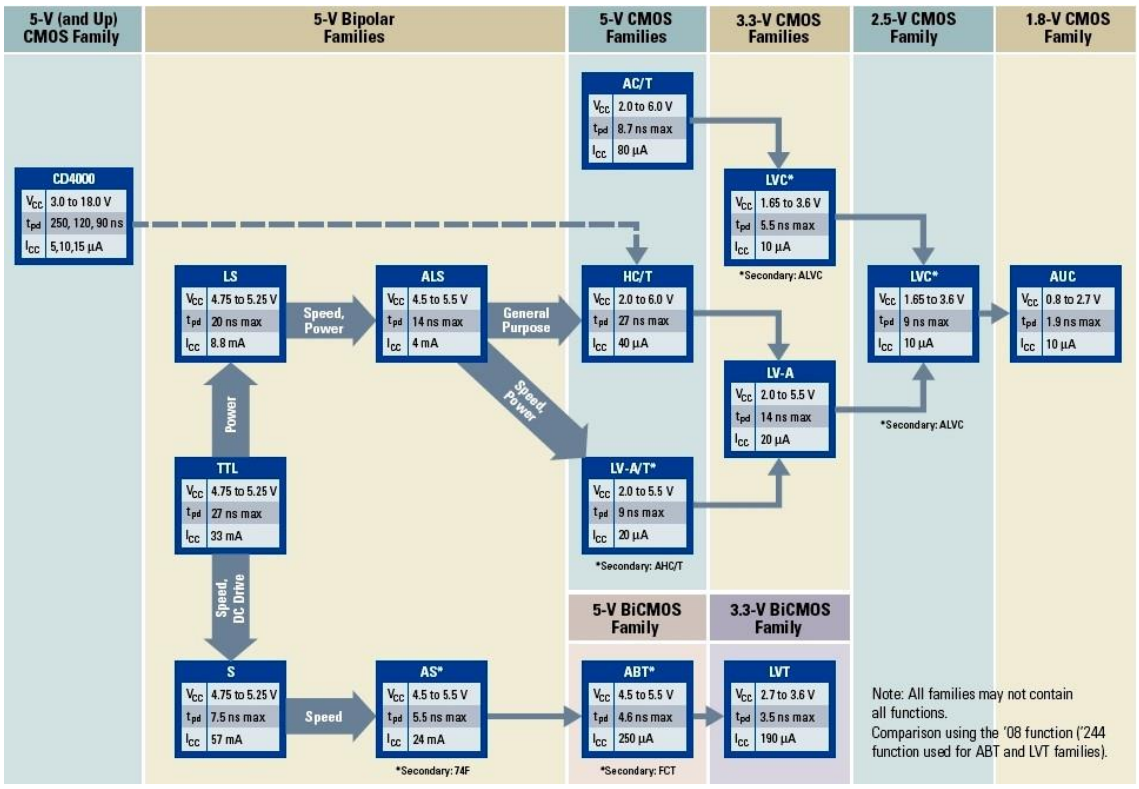
Logikai áramkör-elemcsaládok (TI)

Az ábrán jól látható, hogy a bipoláris elemcsaládok gyakorlatilag eltűnnek a piacról, helyüket átveszik a CMOS logikai elemek. Az elszórtan megjelenő BiCMOS elemek a bipoláris és a MOS technológia integrálásából születtek, ötvözendő a kétféle technológia különböző előnyeit: MOS bemenetek és logika, bipoláris kimenetek. (Bipoláris: nagyobb sebesség, nagyobb erősítés, kis kimenő ellenállás – MOS: alacsony fogyasztás, nagy bemenő impedancia).



Az elemcsaládok sebességviszonyai (TI)

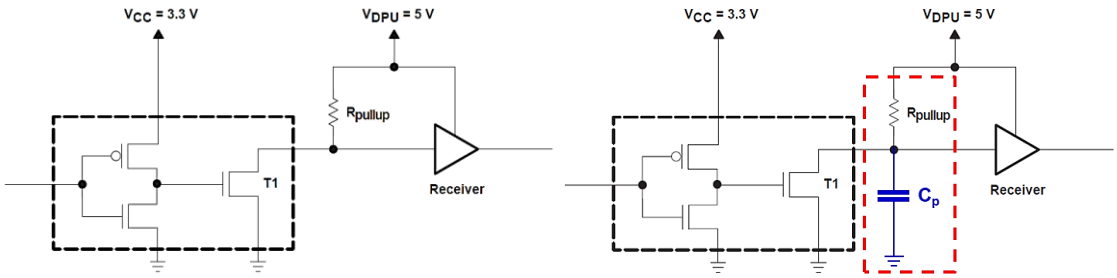
A következő ábrán jól követhetők a főbb elemcsaládok tápfeszültség tartományai, melyektől nem független a működésük sebessége sem. A családok közötti eligazodást segíti a családok kapcsolódási gráfját bemutató ábra.



Az elemcsaládok kapcsolódási gráfja (TI)

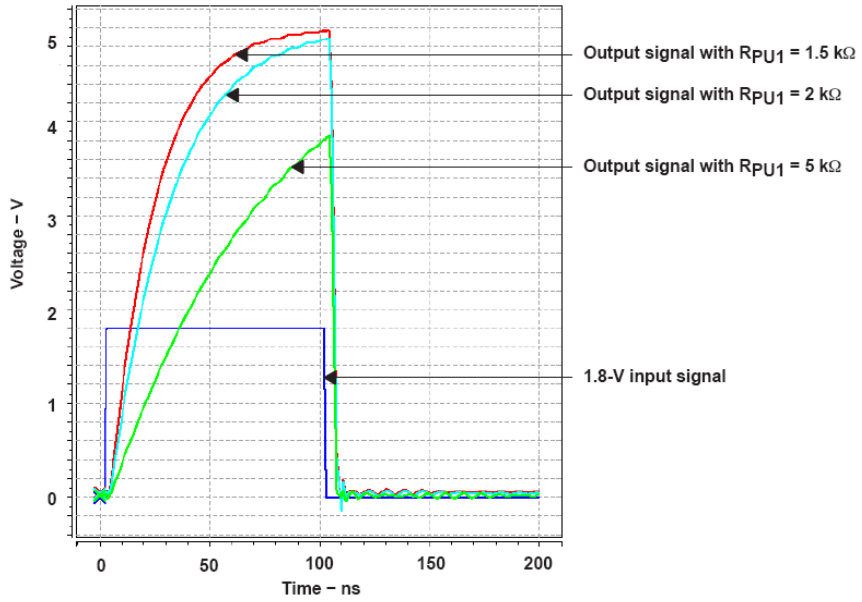
De térjünk vissza a bemutatott problémákhoz! Az első probléma (3,3V $\Rightarrow$ 5V) egyik lehetséges megoldását az OC/OD kimenetű áramkörök kínálhatják. A következő ábra alapján egyszerűen megérthető, hogy a V_{CCA} tápfeszültségű áramkör T1 kimeneti

tranzisztorának kollektorát (drainjét) V_{CCB} feszültségre felhúzva jelszint váltást hozunk létre a logikai „1” szintű jelek esetében.



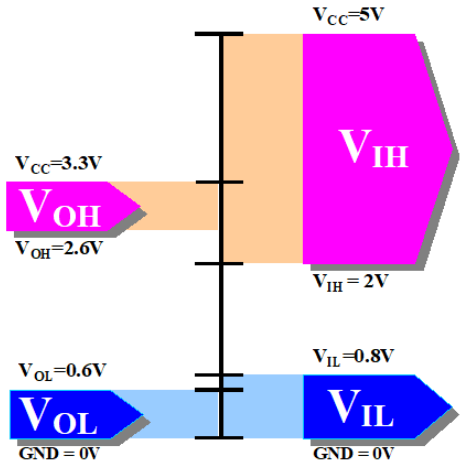
OC/OD jelszint-transzformáció – a mindig jelenlévő szórt kapacitással

A megoldással áthidalhatjuk az első problémaként említett „1” szint feltranszformálását 3,3V ⇒ 5V meghajtás esetén. A megoldás azonban csak bizonyos – nem túl nagy sebességeket igénylő – illesztések esetében alkalmazható. Az OC/OD kimeneteknél már korábban is említettük, hogy az „1” szint generátor-impedanciája lényegesen nagyobb, mint a „0” szint esetén, R_{pu} a környezet szórt és a vevő bemeneti parazita kapacitásaival (C_P) egy RC-szűrőt alkot, lényegesen lelassítva a 0→1 jelváltások sebességét az aktívan meghajtott 1→0 jelváltásokhoz képest.



OD kimenet jelalakjai nagyobb frekvenciákon

Az idődiagramon jól látható, hogy a felhúzó ellenállás növelésével az $(1 - e^{t/RC})$ időfüggvény szerinti felfutás $R_{PU} > 2\ k\Omega$ felett már számottevő időkéséssel jár. Viszont $R_{PU} = 1\ k\Omega$ értékű felhúzó ellenállás logikai „0” kimenő jelszint esetén $I = 5V/1\ k\Omega = 5mA$ árammal terheli a kimeneti tranzisztort, ami sok esetben nem megengedhető.



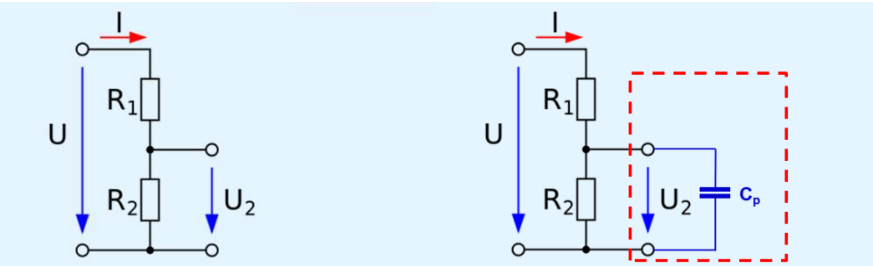
3.3V ⇒ 5V (HCT, AHCT) jelszintek alakulása

Megoldást jelenthetnek a problémára a CMOS elemcsaládok TTL-kompatibilis variánsai (74HCT, 74AHCT stb.). Ezeknél az elemeknél egy speciális bemeneti fogadóáramkörrel visszaállították a V_{IH}/V_{IL} jelszint-korlátok értékeit a TTL családnál megszokott 2V / 0,8V feszültségekre. Amint az az ábrából jól kivehető, a 3,3V-os logikák 5V-os jelszintre transzformálására az említett HCT/AHCT elemcsaládok kiválóan megfelelnek.

A másik probléma (5V ⇒ 3,3V, azaz az alacsonyabb tápfeszültségű áramkör V_{CC} feletti jelszinten történő meghajtása) megoldására szintén többféle megoldás kínálkozik:

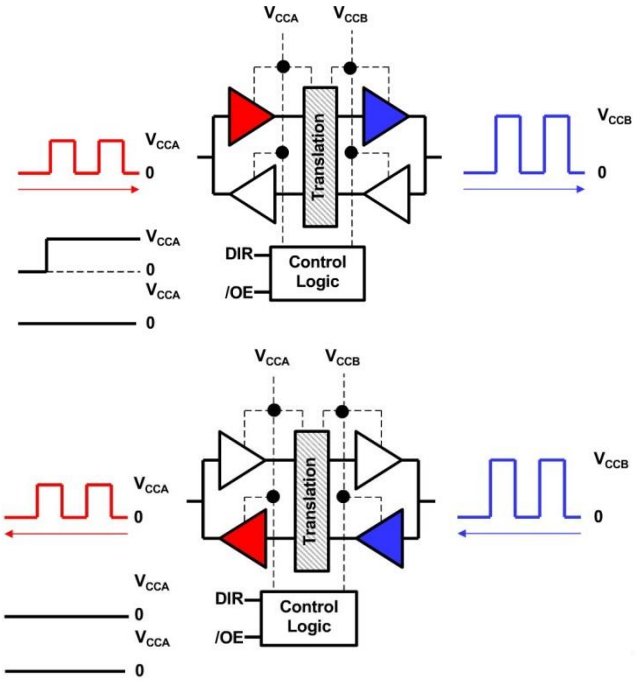
a) A legkézenfekvőbb megoldása a gyártók által szerencsére gyakran felkínált „**5V toleráns**” 3,3V-os áramkörök használata. Fenti (gyakori) problémát felismerve a gyártók ezen áramkörök bemeneteit a V_{CC} tápfeszültség felé nem vágódiódákkal védik, hanem bonyolultabb ESD kapcsolásokkal, melyek ennek köszönhetően üzemszerűen, károsodás nélkül elviselik az 5V-os jelszintet is.

b) Egyszerű ellenállásosztó alkalmazása a jel felszültségének csökkentésére – azonban az osztó ellenállásainak kb. néhány kΩ nagyságrendű értékei és az itt is meglévő szórt kapacitások miatt ebben az esetben is számolnunk kell az OC/OD kimeneteknél bemutatott RC szűrő lassító hatásával.



Jelszint csökkentése ellenállásosztóval

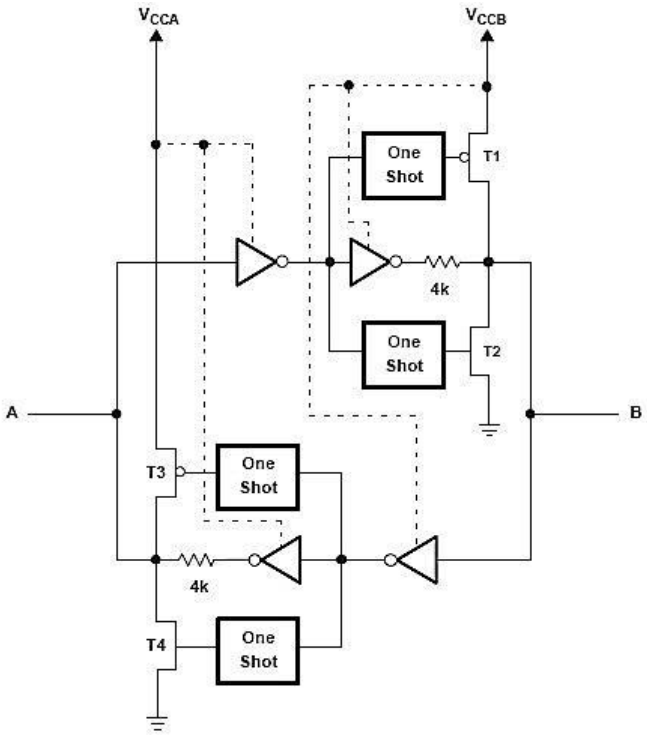
c) Speciális jelszint-illesztő áramkörök alkalmazása.



Kétirányú adattovábbítás jelszint váltással – DIR elengedhetetlen?

A jelszint-váltás során nagyobb problémát jelentenek a kétirányú jelvezetékek. A gyártók kínálatában megtaláljuk azokat a két különböző tápfeszültséggel működő kétirányú illesztő elemeket, melyek a jelszint-váltást oda-vissza megoldják, gondot jelenthet viszont az, hogy az adatvezeték irányát az adó és a fogadó elemben egy bonyolult logika dönti el, amely döntésnek az „eredménye” az irányvezérlő jel (DIR) az eszközön kívül nem áll rendelkezésünkre. Az ilyen kétirányú kapcsolatok irányának automatikus vezérlésére fejlesztette ki a Texas Instruments az irányvezérlés nélküli kétirányú meghajtó elemcsaládot.

Hogyan lehetséges ez ? Kombináljuk össze a kétirányú reset jelnél és a 8051-es portoknál tanultakat és máris kész a megoldás! A minimális fogyasztású CMOS bemeneteket akár egy soros impedancián keresztül is összeköthetjük a kimenettel, statikusan (állandósult állapotban) a jelszint tartására a kimenet egy soros ellenálláson keresztül is képes lesz. Az egyetlen gond ezzel a megoldással a jelszint-váltások dinamikus tulajdonságaival lenne: az OD kimeneteknél látottakhoz hasonlóan a jelszintek beállása a parazita kapacitások miatt túl lassú lenne. Jöhet az ötlethez a 8051-nél megismert átkapcsolási „gyorsimpulzus”: amennyiben a meghajtó kimenetén érzékelt jelváltozás hatására impulzusszerűen „meglökjük” az ellenállás túloldalán található bemenetet – gyorsan feltöltve/kisütve az említett parazita kapacitásokat –, az gyorsan beáll a kívánt szintre, a jelszint statikus tartását pedig az ellenállás is képes ellátni már.



Kétirányú meghajtó irányvezérlő jel (DIR) nélkül

Az impulzusgenerátorok (one shot) az A vagy a B port fel- vagy lefutó jelének változására reagálnak. Felfutó él (0→1) hatására a PMOS tranzisztorok (T1, T3) kapcsolódnak be egy rövid időre, felgyorsítva ezzel a jelszint alacsonyról magas szintre váltását. Hasonló módon, egy lefutó él hatására az NMOS tranzisztorok (T2, T4) kerülnek aktiválásra egy rövid időre – segítve a magasról alacsony jelszintre való váltást. A gyorsító áramkör tipikus kimeneti impedanciája 1,2..1,8V tápfeszültségen 70 Ω, 1,8..3,3V táplálás mellett 50 Ω, míg 3,3..5V-on 40 Ω.

A következő táblázat összefoglalja a tárgyalt lehetőségeket az eltérő tápfeszültségről üzemelő egységek jeleinek illesztésére.

Szintváltás	Megoldás		Nehézség
5V → 3.3V	5/3-a	5 V toleráns bemenet	Ellenőrzendő
	5/3-b	Ellenállásosztó	Lassú
	5/3-c	Aktív szintáttevő	Plusz eszköz, késleltetés
3.3V → 5V	3/5-a	HCT, AHCT, …	Ha lehetséges
	3/5-b	O/C, O/D	Lassú
	3/5-c	Aktív szintáttevő	Plusz eszköz, késleltetés

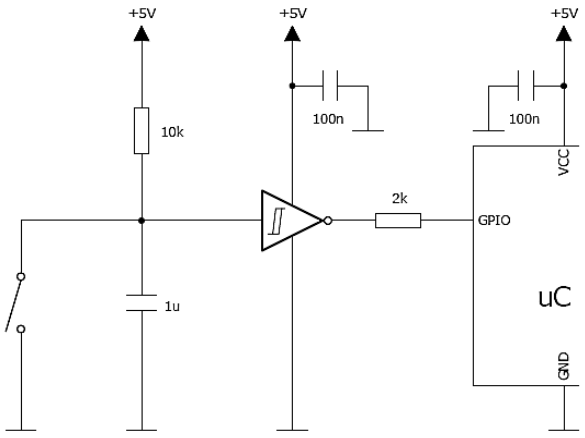
Megoldási lehetőségek jelszintváltásra

4.3 Digitális be- és kimenetek

Beágyazott rendszerek kezelőszervei igen gyakran nyomógombok, kapcsolók, melyek állapotát egyszerűen **digitális bemenetek** segítségével olvashatjuk be. Az ilyen egyszerű illesztésekkel foglalkoznunk sem nagyon kellene, az egyetlen dolog, amire figyelnünk kell az áramkör kialakításánál – mint az a Digitális technika tárgyból már ismeretes –, az a mechanikus kapcsolóelemek pergésmentesítése.

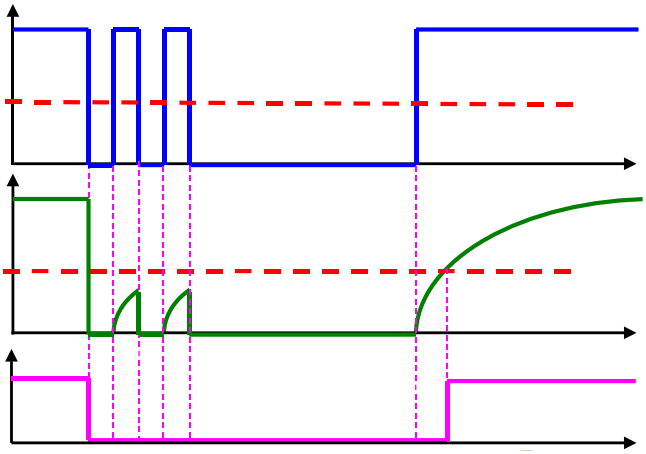
A kapcsolóelem érintkezőit általában valamilyen mechanikusan előfeszített állapotból kényszerítjük át egy másik hasonló állapotba, felgyorsítva ezzel az átkapcsolás folyamatát a kézi beavatkozás sebességéhez képest, valamint kikényszerítve egy, az érintkezőket összeszorító erőt a biztonságos elektromos kapcsolat kialakulásához (ún. ”pattogó” vagy ”békázó” nyomógombok). A véges tömegű egymásnak ütköző érintkezők azonban a fizika törvényei szerint az őket összekényszerítő rugóerő ellenére visszapattannak egymásról és ily módon több érintkezés-elválás periódus alakulhat ki a nyugalmi helyzet beálltáig. Ezt a jelenséget a kapcsolók pergésének nevezzük, a hatás kiküszöbölését pedig **pergésmentesítésnek**. Frekvenciája és a pergés időtartama függ a kapcsoló érintkezőinek mechanikai méreteitől és tömegétől, kisebb kapcsolók esetében a néhány 10- 100 Hz tartományba esik.

A pergés nagyon zavaró lehet olyan esetekben, mikor egy nyomógombbal pl. több lehetséges választás sorozatán lépegetünk végig (egy menürendszer elemein, de gondolhatunk pl. egy óra beállítására is). A pergés hatására „elszaladó” választás ellehetetleníti a kívánt elem pontos kiválasztását, hatását tehát ki kell küszöbölnünk.



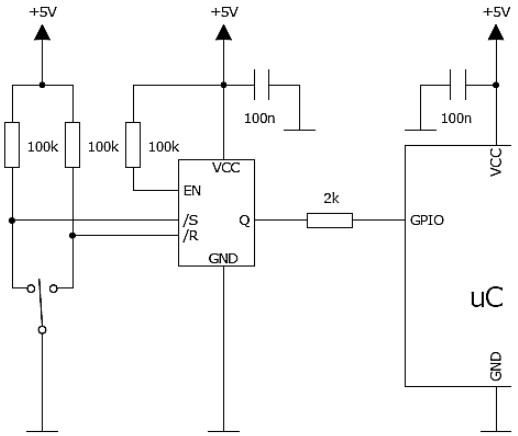
Egyszerű pergésmentesítő kapcsolás RC tárolóval

A legegyszerűbb megoldás egy aluláteresztő RC szűrőelem alkalmazása a kapcsoló jelére. A kapcsoló bármelyik érintkezése azonnal „0” jelszintet állít elő, a kondenzátor feladata az, hogy az érintkezők elválásának ideje alatt az exponenciálisan felfutó jelszint ne érje el az őt követő fogadó áramkör logikai „1” jelszintjét.



Az RC szűrővel végzett pergésmentesítés idődiagramjai

A megoldás egyszerű és különösebb magyarázatot nem igényel. Hátránya, hogy a pergés frekvenciája csak nehezen becsülhető, ezért általában a szükségesnél nagyobb energiatárolót szoktunk a kapcsolásba tervezni. Ez egyrészt felesleges költség, a kondenzátor kisülésekor kialakuló áramcsúcs károsítja az érintkezőt és késlelteti az „1” szint időbeli felismerését. A lassú jelfelfutás miatt mindenképpen hiszterézises áramkörrel fogadjuk a szűrt jelet (Schmitt-trigger), nehogy a szűrt jelen meglévő zajfeszültség okozzon hibás impulzusokat a jelváltás alatt.

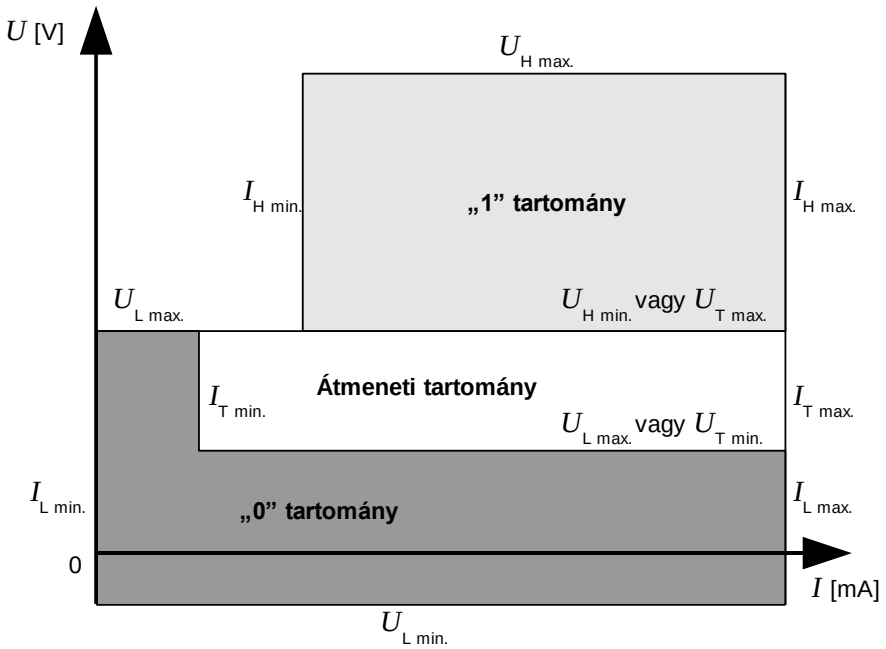


/R	/S	Q
0	0	? (tiltott)
0	1	0
1	0	1
1	1	Q (tartás)

Pergésmentesítés R-S tároló alkalmazásával

A pergésmentesítés másik bevált módszere a nyomógomb R-S tárolóval történő fogadása. Alkalmazásához mindkét szélső helyzetében záró kontaktust adó kapcsolóra lesz szükségünk (váltó érintkező). Az ötlet azt a tényt használja ki, hogy a mozgó érintkező bármelyik szélső érintkezőtől elpattan ugyan, de ez a mozgás nincs akkora, hogy „átpattanjon” az ellenkező helyzethez tartozó érintkezőhöz. Az akár 2 db NAND kapuból is felépíthető /R-/S tároló pedig csak akkor változtatja állapotát, ha valamelyik bemenetére határozott „0” jelszint érkezik.

A beágyazott kapcsolások „belsejében” (ide értve a lokális kezelőszerveket is) keletkező digitális bemenetek kezelése lényegesen egyszerűbb, mint a külvilágból származó tetszőleges kétállapotú jeleké. Ezek a jelek általában már nem logikai jelszintekkel rendelkeznek (pl. 0-24V), számottevő zajjal lehetnek terheltek. Kezelésükre a programozható vezérlőberendezések tulajdonságaival foglalkozó IEC-61131 nemzetközi szabvány 2. fejezete tartalmaz kötelező érvényű előírásokat.



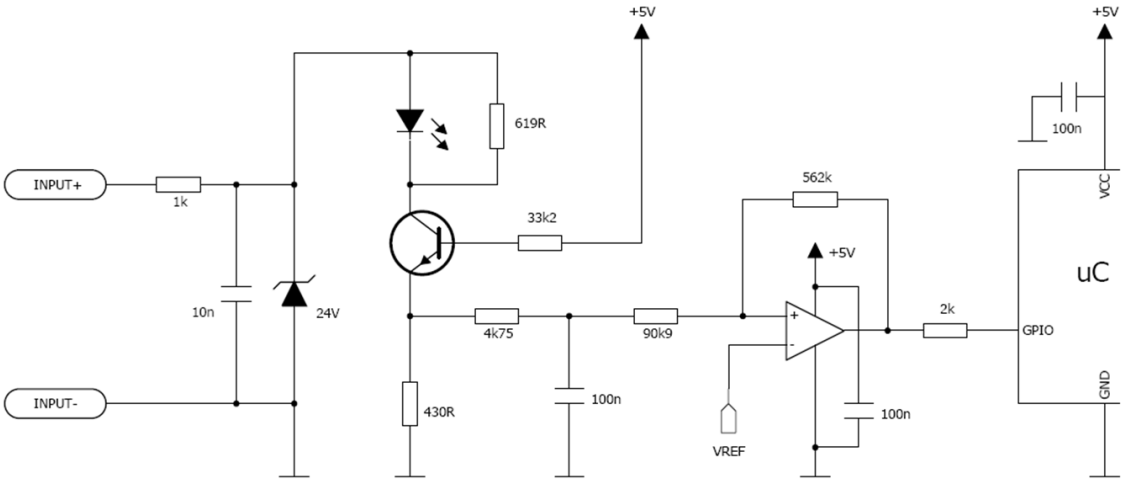
Bemenő jel	Limit	„0” állapot		Átmeneti tartomány		„1” állapot	
		U_L [V]	I_L [mA]	U_T [V]	I_T [mA]	U_H [V]	I_H [mA]
DC 24V	Max.	11/5	30	11	30	30	30
	Min.	-3	n.d.	5	2	11	6

Digitális bemenet U-I tartományai (IEC 61131-2)

Fenti ábrával a korábbi esettanulmányunk kapcsán már megismerkedtünk, bár ott elemzésével nem foglalkoztunk. Lényege két igen fontos tulajdonságban foglalható össze:

- egyrészt definiálja azokat a bemeneti feszültségtartományokat, melyek a logikai jelszintekhez hasonlóan (V_{ILmax}, V_{IHmin}) küszöbértékeket ír elő a jelszintek biztonságos felismerésére,
- másrészről minimális és maximális terhelő áramértékeket ír elő a fogadó áramkör számára, elkerülendő az energia nélküli statikus feszültségcsúcsok hibás felismerését (minimális érték) ill. a jel túlterhelését (maximális érték).

Az említett esettanulmányban már megismerkedtünk egy, az előírásokat kielégítő kapcsolással, itt most egy másik hasonlót mutatunk be. A kapcsolás tulajdonságait a bemenet felől elindulva elemezzük lépésről lépésre.



24V –os digitális bemenet kialakítása az IEC 61131-2 előírásai szerint

- A soros ellenállás, és a (több amperes csúcsáramot elviselő) Zener dióda a túlfeszültségek és a negatív jelszintek ellen nyújt védelmet. A kerámia kondenzátor a nagymeredekségű túlfeszültségek esetén korlátozza a kialakuló feszültséget a dióda nyitásáig (< μsec).
- Megfelelő nagyságú bemenő feszültség esetén a tranzisztor annyira nyit ki, hogy emittere kb. 3V-ra emelkedjen. Ehhez mintegy 7-8 mA kollektoráram szükséges a beérkező jel feszültségszintjétől függetlenül. Az érték hozzávetőlegesen az

$$5V - \frac{I_E}{B_N} \cdot 33{,}2k - U_{BE} = I_E \cdot 0{,}43k$$

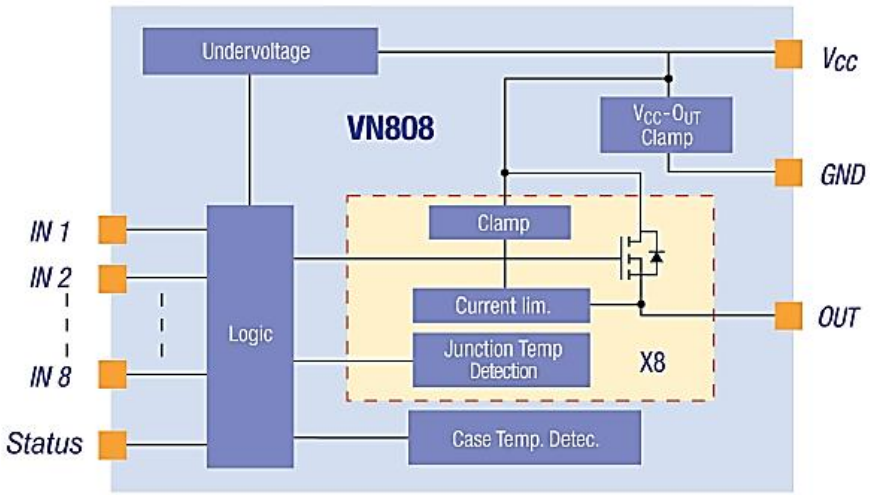
  összefüggésből számítható ($I_E \approx I_C$, B_N a tranzisztor nagyjelű áramerősítése, U_{BE} a bázis-emitter dióda nyitóirányú feszültségesése).
- A kollektorkörben található LED a bemenet állapotát jelzi, a 11-30V tartományban a közel állandó kollektoráramnak köszönhetően a fényereje gyakorlatilag nem változik.
- Az emitteren keletkező feszültséget egy 500 μsec időállandójú kapcsolás szűri (ebben a kapcsolásban kb. ekkora a bemenetre vonatkozó mintavételi idő)
- A szűrőt egy hiszterézises komparátor kapcsolás követi, amely a 2V középérték (V_{REF}) körül kb. ±0,5V hiszterézissel biztosítja a biztonságos billenést.

A **digitális kimenetek** esetében (itt is megfelelő áramterhelhetőséggel rendelkező, tipikusan 0-24V-os kimenetekre gondolunk) manapság már igen kevés a tennivalónk: az integrált áramkörgyártók komplex védelmi rendszerrel ellátott kész megoldásokkal kényeztetnek el bennünket napjainkban. Elnevezésük nem teljesen egységes, az **IPS** (Intelligent Power Switch) összefoglaló elnevezés mellett gyakori a **HSS** (High-Side Switch) ill. **LSS** (Low Side Switch) aszerint, hogy kimenetünk a tápfeszültség, vagy a földpotenciál felé rendelkezik-e kapcsoló elemmel. A **SSR** (Solid State Relay) több gyártó esetében a mechanikus relékhez hasonló galvanikus elválasztást is takar, de pl. az

egyik legnagyobb gyártó, az STMicroelectronics ugyanazon kimeneti áramkörét (VN340SP) ISP, HSS, SSR néven is illeti…

Ezek az áramkörök általában a következő legfontosabb tulajdonságokkal rendelkeznek:

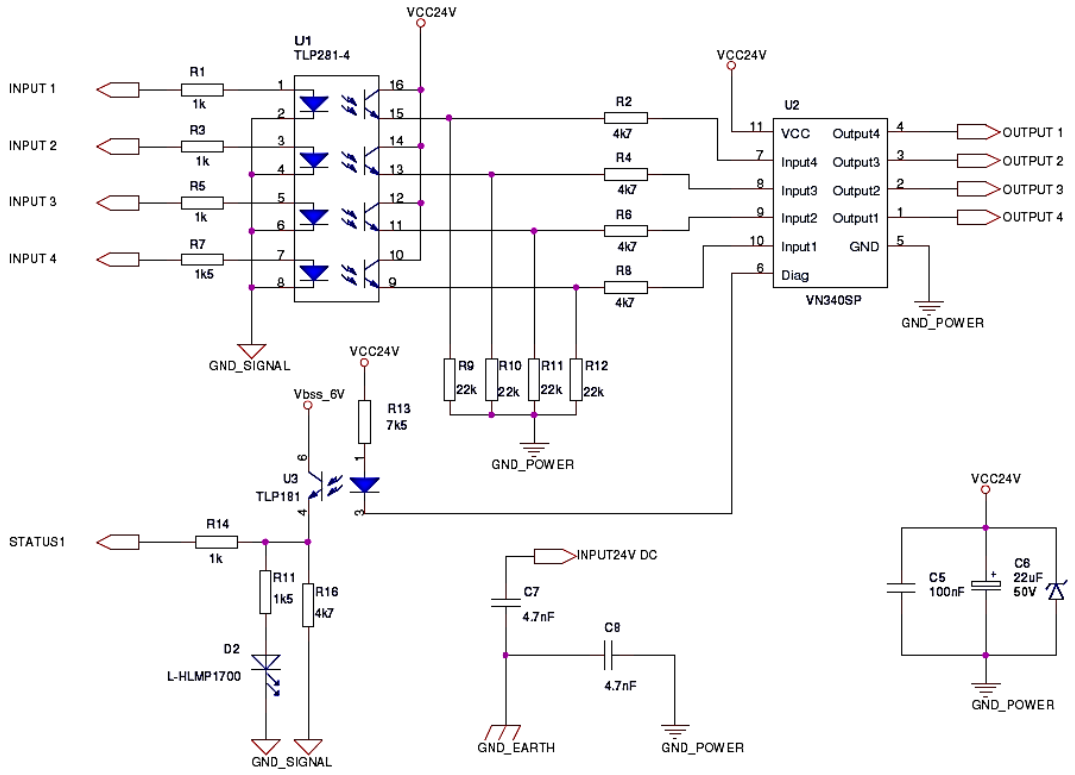
- Megfelelő erősítő kapcsolás logikai jelszint 0-24V/1-2A kimenő jelszintre erősítéséhez
- A kimenetek termikus túlterhelés védelme (lekapcsolással)
- Diagnosztikai visszajelzés a logikai oldalon a kimenetek túlterheléséről.
- A kimenetek EMC védelme (min 1-2 kV)
- Olyan áramköri tokozás (pl. PowerSO-10), ahol az IC „hasa alatti” fémnek a nyomtatott áramkör rézfelületéhez való termikus kontaktusa biztosítja a megfelelő hőelvezetést és -tehetetlenséget (ld. következő példák).



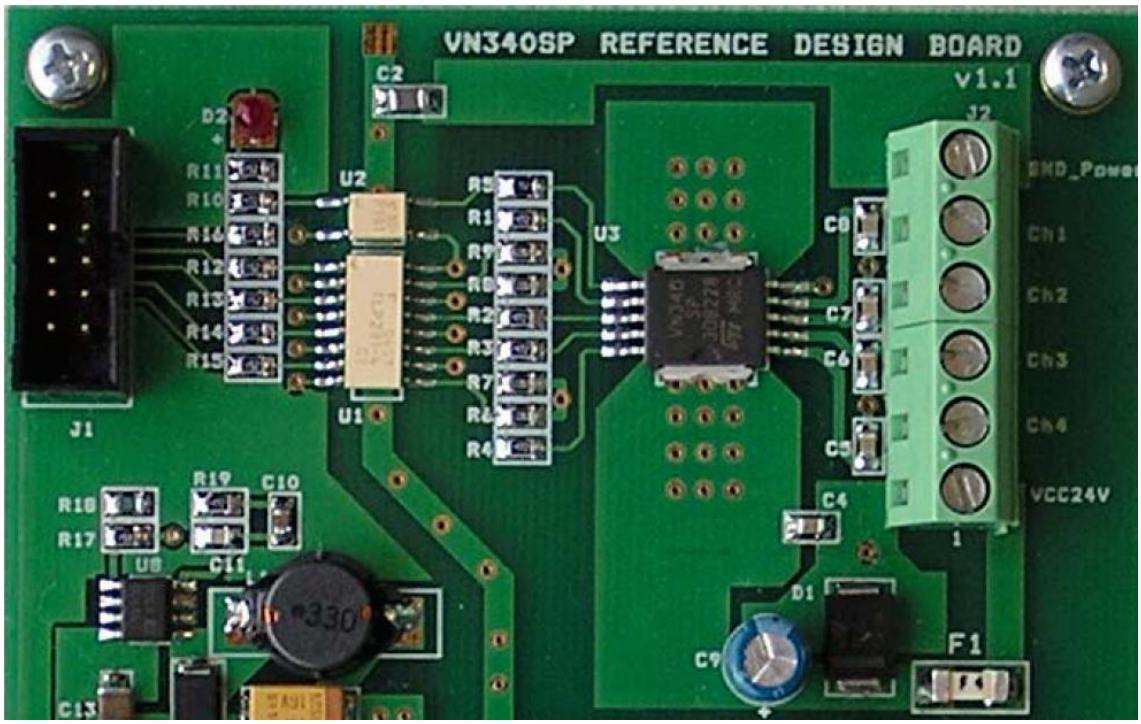
High Side Switch (STMicroelectronics)

Fenti tulajdonságokat ötvözi az ábrán látható VN808 áramkör, amely 8 db 24V/0,5A digitális kimenetet állít elő a logikai vezérlőjeleknek megfelelően, digitális státusz visszajelzéssel az esetleges túlterhelés és lekapcsolás tényéről.

Kicsit nagyobb testvére a VN340SP névre keresztelt 4 x 24V/0,7A kimenetet biztosító áramkör. Alkalmazását mutatja a következő kapcsolási vázlat (a galvanikus leválasztásról a következő fejezetben még részletesen beszélünk), és az ennek megfelelő fizikai kialakítás. Jól megfigyelhető az ábrán a már említett PowerSO-10 tokozás, amely az IC alatt meghúzódó rézfelületnek adja át a keletkező hőt. Az áramkör mellett látható nagyszámú áramköri átvezető furat („via”) nem a tervező hóbortos kedve miatt került oda, még az áramerősség sem indokolná ennyi furat alkalmazását. Az ok az alkatrészoldali rézfelületnek az áramkör túloldalán található még nagyobb rézfelülettel való termikus összekapcsolása és ezzel még nagyobb hőleadó felület biztosítása.



A VN340SP High Side Switch alkalmazása

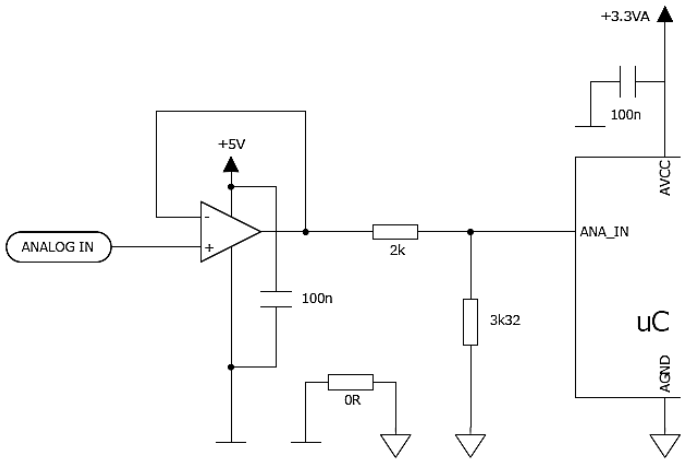


A fenti kapcsolás fizikai kialakítása

4.4 Analóg be- és kimenetek

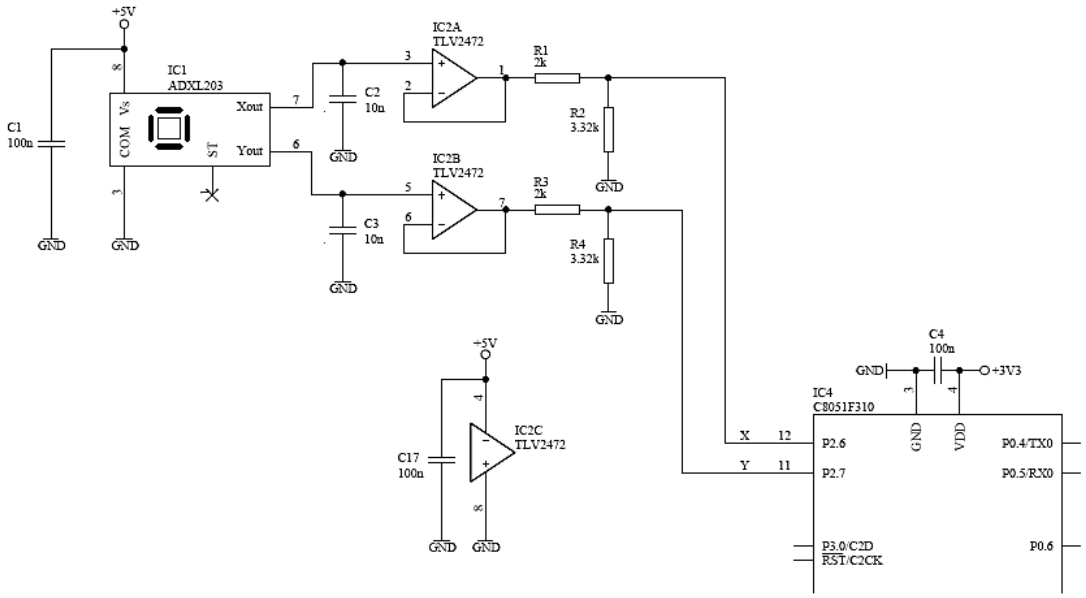
Beágyazott rendszerek gyakran igényelik analóg be- és kimenetek használatát. Ehhez a mikrokontrollerek többnyire tartalmazzák a szükséges átalakító (A/D ill. D/A) egységeket. Alkalmazásukkor többnyire a jelszint- és impedancia illesztés említhető megoldandó feladatként (az EMC szempontokra a későbbiekben még visszatérünk).

A mikrokontrollerek analóg bemeneteinek jelszintje általában nem haladhatja meg az alkalmazott tápfeszültség értékét (pl. 0-3,3V). Ez a jeltartomány a külvilágból érkező jelek számára általában nem megfelelő, mivel a szinte védhetetlenül jelenlévő néhány 10 mV zajfeszültséghez képest a jeltartomány túl kicsi (pl. 50 mV/3,3V = 1,5%). Ezért az analóg technikában megszokott 0..5V, 0..10V, esetleg ±10V jeltartományt át kell transzformálnunk a mikrokontroller bemenő jeltartományára.



Analóg bemenet (0..5V) fogadása követő erősítővel és feszültségosztóval

- A fogadó erősítő első és legfontosabb szerepe mindenekelőtt a mikrokontroller analóg bemenetének a külvilágtól való **elválasztása** (ez nem azonos a galvanikus leválasztással, ld. később!). A viszonylag érzékeny CMOS bemenetet bipoláris vagy egyéb műkapcsolással „megvédjük" a környezeti túlfeszültségektől egy fogadó áramkörrel, amely az említett feszültségeket elnyeli, nem engedi tovább.
- A **jelszintek illesztésére** sokszor ellenállásosztót alkalmazunk (vigyázat: az ellenállások pontossága meghatározza a teljes lánc pontosságát!)
- Fogadó áramkörünk az előző pontban alkalmazott osztó miatt messze nem lenne nagyimpedanciás bemenetnek tekinthető (a példa kb. 5 kΩ-os bemenő ellenállása az esetek többségében túl kicsi lenne). Emiatt egy követőerősítő (A_u = 1) kapcsolással biztosítjuk az $R_{be} \approx \infty$, $R_{ki} \approx 0$ **impedancia transzformációt**.
- Alkalmazott műveleti erősítőnk a tápfeszültségeket nagyon szorosan megközelíteni képes ún. **rail-to-rail** („síntől sínig") típus kell legyen, a jeltartomány szélein (kb. 0.1V) azonban még így is ellenőrzendő a jelkonverzió pontossága.



Nagy generátor-ellenállású szenzor (gyorsulásérzékelő) illesztése

Az eddig ismertetett elveket követhetjük végig az egy gyorsulásérzékelő analóg kimeneteinek 3,3V-os mikrokontrollerhez történő illesztését bemutató kapcsolásban. A kétdimenziós gyorsulásérzékelő (Analog Devices ADXL 203) 5V-os tápfeszültség mellett alakítja át a gyorsulást a legpontosabban analóg jelekké (X_{out}, Y_{out}), a jeltartomány Vcc/2 ± 1,7[g] = 2.5V ± 1,7V, melyet 0…3,3V tartományba kell leképeznünk úgy, hogy a szenzor kimenetek soros ellenállása tip. 30-40 kΩ. A szűrt (10 nF) analóg kimeneteket $R_{be} \approx \infty$, $R_{ki} \approx 0$ paraméterekkel rendelkező követő erősítő transzformálja kisimpedanciásra, majd az ellenállásosztó képezi le a 2,5V ± 1,7V = 0.8…4,2V jeltartományt a 0.5…2,621V feszültségtartományba (0,1% pontosságú ellenállások alkalmazásával).

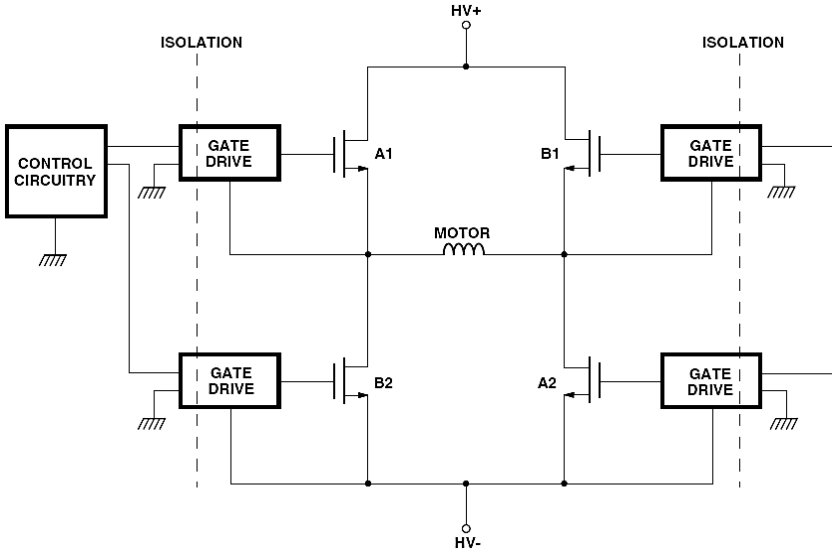
4.5 Galvanikus leválasztás

Az Elektronika tárgyban (is) megismert galvanikus leválasztás az elektromos rendszerek részeinek egymástól való elektromos elszigetelése úgy, hogy ennek határán az információ kizárólag nem elektromos úton (fény, mágneses tér, hanghullám, mechanikai mennyiség stb.) juthat át, elektromos töltéshordozó átvitelére azonban nem kerülhet sor.

Jelentőségét két alapvető alkalmazási szempontja alapján emeljük ki:

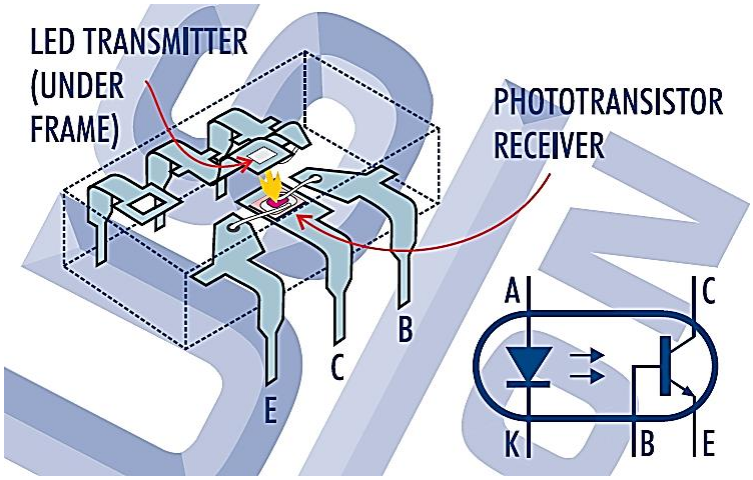
- sok esetben az átviendő információt elektromos szempontból **különböző potenciálszinten** lévő egységek között kell továbbítanunk,
- rendszerünk fizikailag is elkülönülő részegységei (pl. két asztali számítógép) esetleg (közel) azonos potenciálon is vannak, azonban az őket összekötő jelvezetékekbe bejutó zavarok, zajfeszültségek miatt meginduló **kiegyenlítő (hurok-) áramok** az egységek rövid idejű potenciális elemelkedését okozhatják.

Ez a potenciálkülönbség okozhatja a továbbított jelek információtartalmának sérülését, de akár az áramkörök tönkremenetelét is. Ezek a hurkok többnyire a berendezésnek/részegységnek az életvédelmi védőföld felé való lezárásain keresztül alakulnak ki, potenciálisan akár több ezer voltra is elemelve rövid időre a kérdéses egységeket egymástól.



Különböző potenciálszinten lévő egységek vezérlése azonos helyről

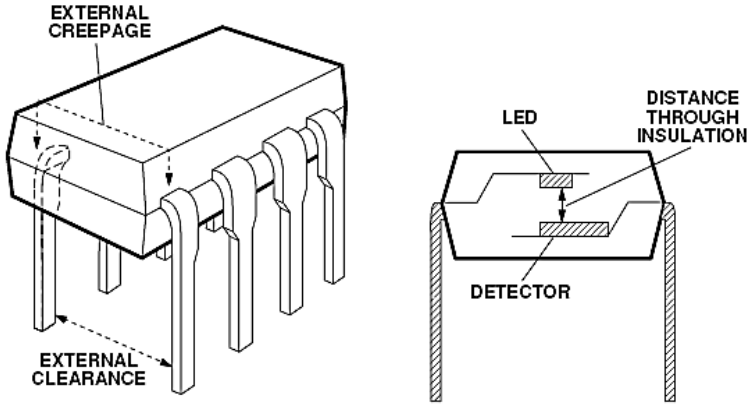
A galvanikus leválasztás talán leggyakrabban használt eszközei az **optikai csatolók (optocsatolók).** Ezekben az eszközökben a potenciálisan független két oldal között fény (fényemittáló dióda + fényérzékeny elem) segítségével történik az információ átvitele.



Optocsatoló belső felépítése

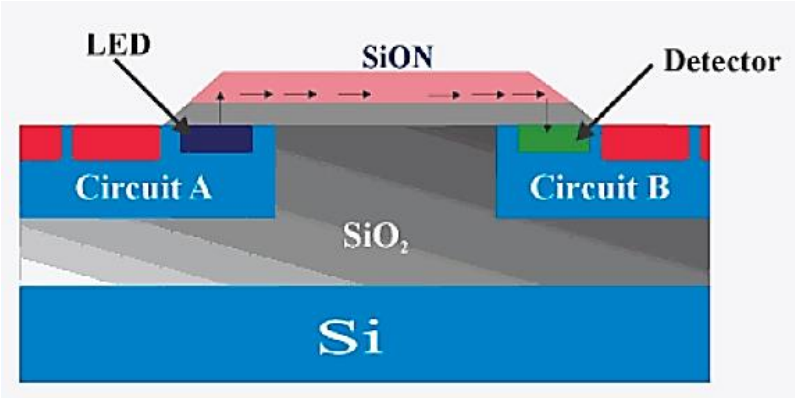
Ezek az eszközök egy-egy integrált áramkörben kerülnek kialakításra, egy tokban sokszor több (2-4 db) optocsatoló is található. Az áramkör két oldala általában fizikailag is a két

elválasztandó oldalnak van megfeleltetve, hiszen nem ritkán itt 2,5-4 kV szigetelési távolságot kell áthidaljunk ezekkel az elemekkel! Számít tehát az áramkör kivezetései közötti fizikai távolság (clearance), a tokozás felületén kialakuló kúszóáram (creepage) és az előzőeknél lényegesen kisebb távolság az optikai adó és vevő között. (Ne feledjük: zárt, szigetelőanyaggal kitöltött térben az anyagjellemzőknek megfelelően lényegesen kisebb távolság is elegendő lehet ugyanazon feszültség elválasztására, mint párás, esetleg szennyezett levegőben!)



Az optocsatoló szigetelési paraméterei

Az optocsatolók felépítésénél bemutatott megoldás (két különböző hordozólap a két oldalnak megfelelően) megdrágítja ezeknek az áramköröknek az előállítását. Az utóbbi időben kísérleteznek olyan kialakításon is, ahol azonos lapkán helyeznék el az adó és a vevő áramköröket, a kettő között pedig egy jó fényvezetési tulajdonságú csatornát alakítanának ki az információ továbbítására.

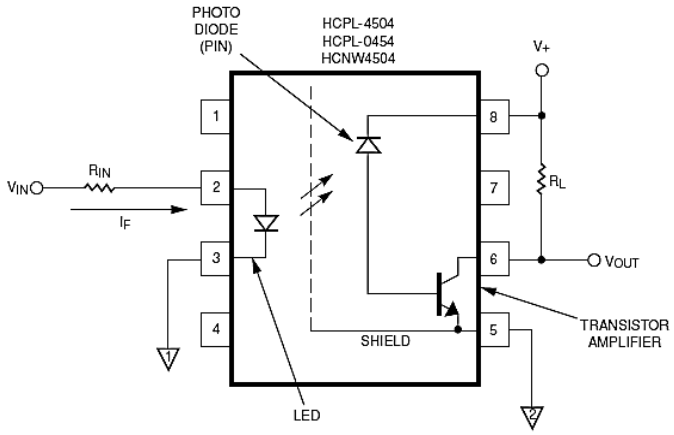


Adó és vevő közös lapkán (STMicroelectronics)

Kialakításukat tekintve az optocsatolók a következő főbb csoportokba oszthatók:

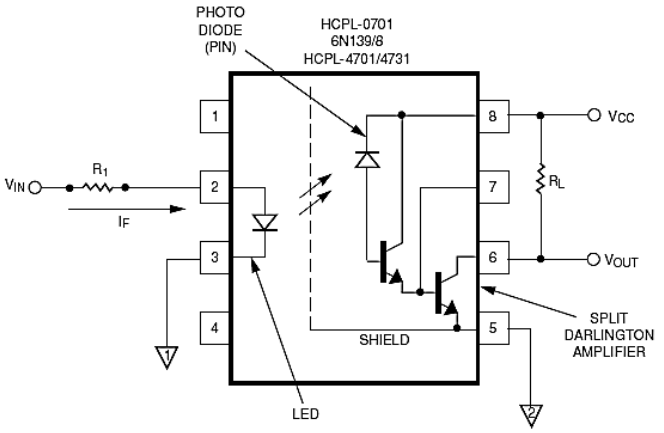
- aktív erősítő nélküli és
- aktív erősítős típusok.

Az aktív erősítő nélküli típusok nem igényelnek külön tápellátást. Primer oldalon egy fényemittáló dióda (LED) meghajtásáról kell gondoskodnunk, szekunder oldalon egy fotodióda vagy fototranzisztor kivezetései állnak rendelkezésünkre a vett jelek érzékelésére.



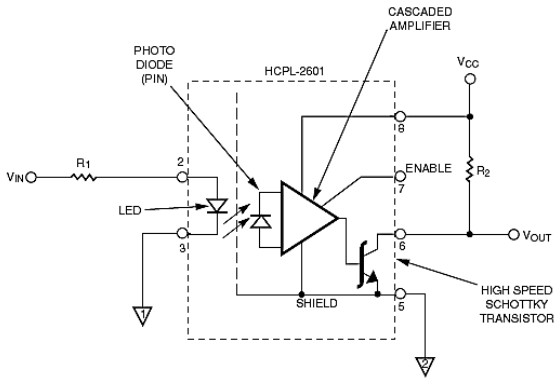
LED – tranzisztor kialakítású optocsatoló

Az optocsatoló egyik legfontosabb jellemzője az áramátviteli tényező (CTR: Current Transfer Ratio) lesz, amely azt mutatja, hogy a primer oldalon alkalmazott diódaáram hányszorosa nyerhető ki kollektoráram formájában a szekunder oldalon (I_C/I_F). Az alapkiviteleknél szokásos értékek a 10-200% között szórnak. Ezen az arányon lényegesen javít egy Darlington kapcsolású tranzisztorpár a szekunder oldalon.

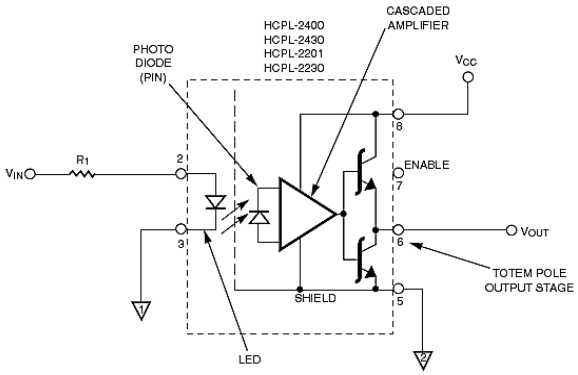


LED –darlington tranzisztor kialakítású optocsatoló

Ezek az optocsatolók általában a 10 kHz – 1 MHz sebességtartományban használatosak, egyszerű kialakításuk révén áruk sem túl magas. Nagyobb sebességű alkalmazásokhoz (MHz feletti soros adatátviteli csatornák: SPI, RS485, CAN stb.) nagyobb CTR értékekre lesz szükségünk, ezeket aktív erősítők alkalmazásával érhetjük el.

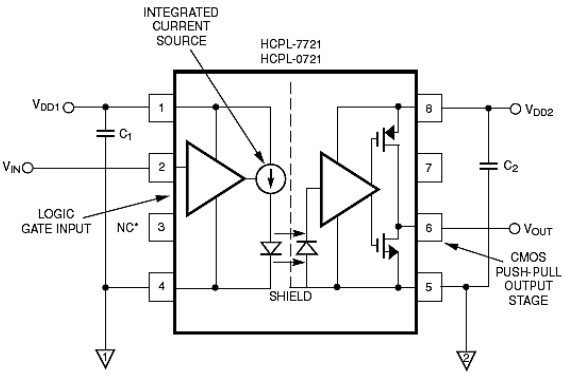


Aktív erősítős OC kimenetű optocsatoló



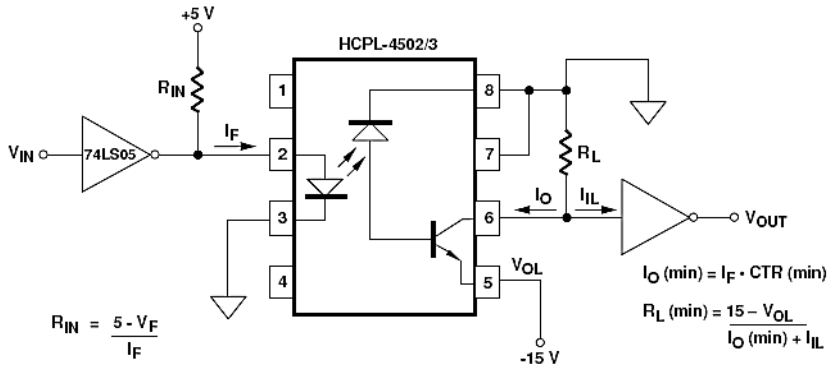
Aktív erősítős push-pull kimenetű optocsatoló

A legnagyobb sebességű (20-50 MHz) leválasztások esetében már a meghajtás árama és a fogadó oldalon meglévő szórt kapacitások is gondot jelenthetnek, ezért az optocsatoló „becsomagolódik” egy-egy meghajtó és fogadó logikai áramkör közé – itt már mindkét oldalon tápellátásról kell gondoskodnunk, viszont az optocsatoló az áramkör tervezője számára már úgy jelenik meg, mint egy különösebb illesztést nem igénylő logikai kapuáramkör.



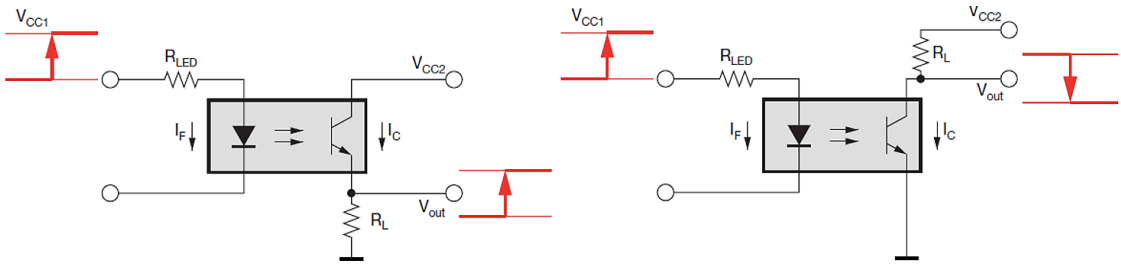
Nagysebességű CMOS optocsatoló

Alkalmazástechnikájukat tekintve az optocsatolóknál primer oldalon is egyszerűen kialakítható invertáló és nem invertáló meghajtás is (az alábbi példánál V_{IN}=0 esetén a LED nem világít, ha ugyanezt a meghajtót áthelyezzük a LED katódjára, fordított működést kapunk).



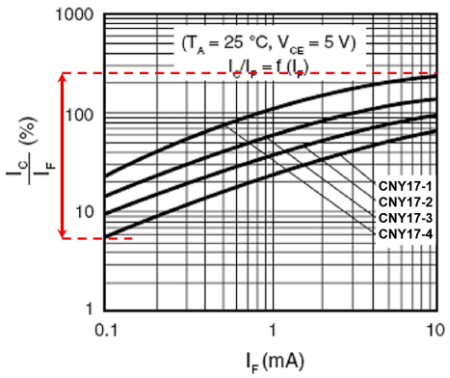
Alkalmazási példa 15V szinttolással

Az invertáló/nem invertáló jelleg a kimeneti oldalon is kialakítható:

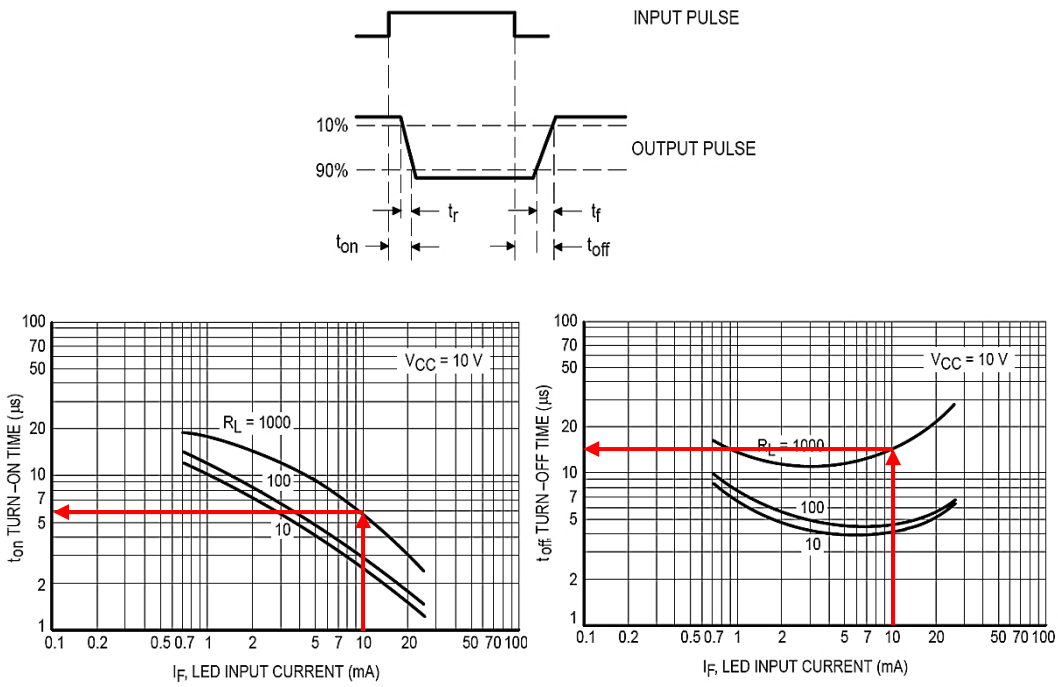


Nem invertáló és invertáló csatolás a kimeneten

Az optocsatolók két igen fontos jellemzője a már említett CTR faktor (amely ráadásul többnyire nem is állandó, függvénye a dióda I_F áramának, a tranzisztor V_{CE} feszültségének és a hőmérsékletnek), valamint a működési sebességet meghatározó be- és kikapcsolási idő (t_{on}, t_{off}).



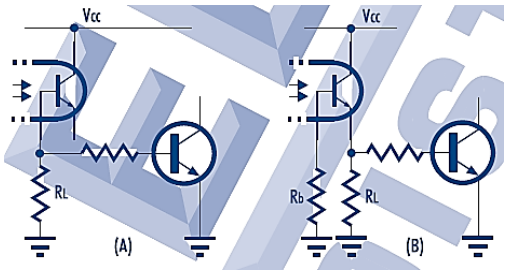
CTR változása a bemenő áram függvényében (CNY17)



A be- és kikapcsolási idők alakulása (4N25)

A passzív optocsatolóknál minden esetben a be- és a kikapcsolási idők lényeges eltérésével kell számolnunk (fenti példában I_F=10mA és R_L=1kΩ kollektor ellenállás esetén t_{on}=6µsec, t_{off}=14µsec). Ez nem csak a működési sebesség szempontjából jelent korlátot, hanem torzítja az átvitt digitális impulzusok fázishelyzetét, ami egy inkrementális adó bemenet vagy egy PWM kimenet leválasztása esetében komoly problémák forrása is lehet.

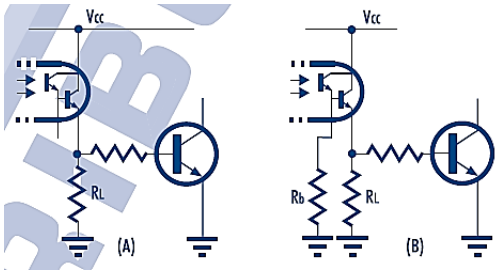
Az ok nagyon egyszerű: a tranzisztorok közismert bázis-kollektor kapacitása bekapcsoláskor a fényelektromos hatás révén gyorsan („aktívan”) feltöltődik, viszont kikapcsoláskor a megszűnő fény nem tudja „kisütni” ezt a kapacitást, a töltések lassabb távozása késlelteti (és meredekségében is lassítja) a kikapcsolás folyamatát.



Optocsatoló gyorsítása

Fentieken egyszerűen segíthetünk, ha kivezették a tranzisztor bázisát (amire egyébként az alapkapcsolás szerint nem lenne szükség). Az „A” példában a fototranzisztort fotodiódaként használjuk, szimmetrizálva ezzel mind a be- mind a kikapcsolás folyamatát. A „B” kapcsolásban egy bázisellenállással (R_B) a földre kötjük a bázist,

amivel ugyan elvezetjük a „hasznos” bázisáram egy részét is (CTR csökken), viszont segítünk a báziskapacitás kisütésében, ez pedig gyorsítja a kikapcsolás folyamatát.

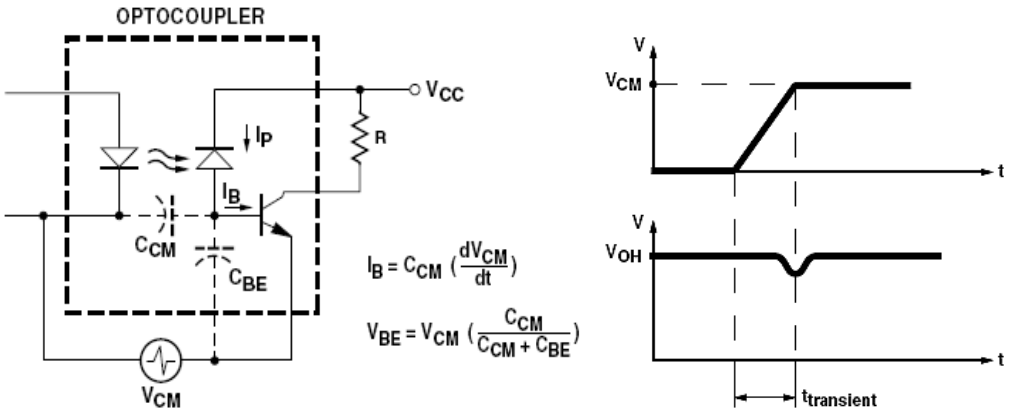


Darlington kimenetű optocsatoló gyorsítása

Ugyanez a „trükk” alkalmazható darlington kapcsolású tranzisztorok esetében is („A” a hagyományos, „B” a felgyorsított kapcsolás).

Az optocsatolók elsődleges feladata tehát a jelátvitel szempontjából **közös modusú jelnek** számító két oldal közötti potenciálkülönbség **statikus és dinamikus** elnyomása. A két oldal között fennálló statikus feszültség az egység specifikációs határain belül nem okoz gondot, nehezebb problémát jelentenek annak nagysebességű változásai. Ilyet okoz üzemszerűen pl. a fejezet elején bemutatott motorvezérlő hídkapcsolás FET-jeinek a vezérlése: amint nyitottuk pl. a bal felső elemet (A1), annak vezérlőelektródája a felső tápsín (HV+) közelébe kerül – ellenkező irányú vezérlés (B2 elem nyitása) esetén viszont az alsó tápsín (HV-) potenciáljához kerül közel. Az egymástól leválasztott számítógép egységeket összekötő kommunikációs adatvezetékeket ért külső nagyfeszültségű zavarás is ilyen nagysebességű, közös modusú feszültségváltozásokat okoz az optocsatoló két oldala között.

A probléma kezeléséhez nézzük a közös modusú zavarás tranziens modelljét!

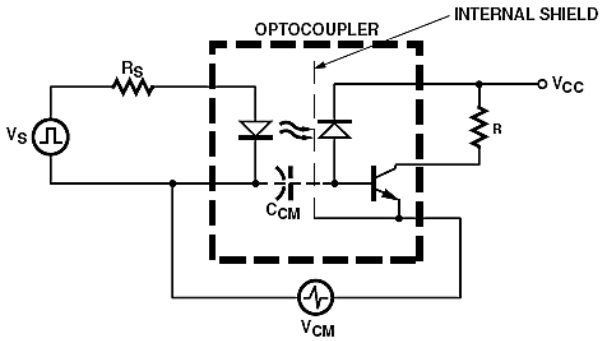


Közös modusú zavarás tranziens modellje

A két oldal között mindenképpen meglévő C_{CM} szórt kapacitás sorban a vezérelt tranzisztor bázis-emitter kapacitásával teljes egészében felveszi a V_{CM} közös modusú

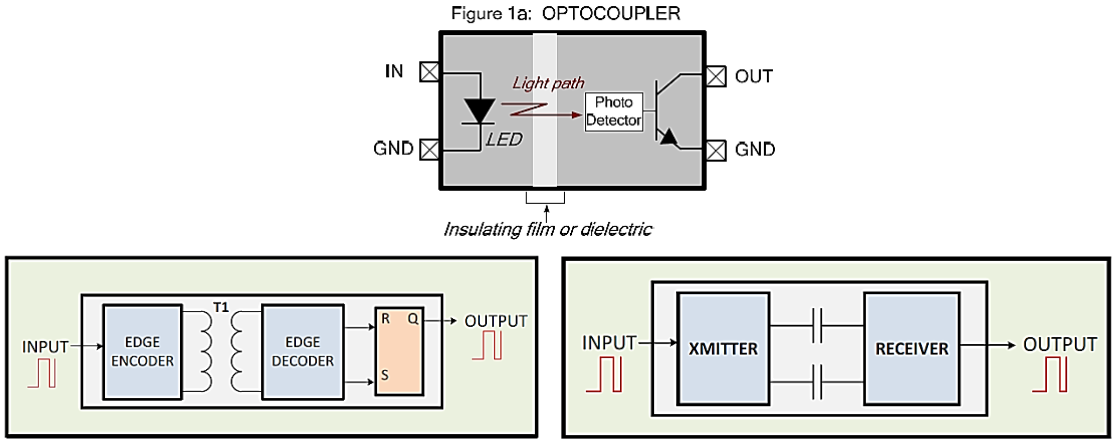
feszültség minden változását. Legyen optocsatolónk teljesen lezárva (I_P=0, V_{OH}=V_{CC}). Egy pozitív ugrás a V_{CM} közös modusú feszültségben az egyenleteknek megfelelő pozitív I_B bázisáramot indít meg, ami elkezdi nyitni a tranzisztort – ennek megfelelően a kimeneten egy rövid jelbetörés várható.

Az egyenletekből is látszik, hogy minden problémánk okozója a két oldal közötti C_{CM} kapacitás megléte, ennek csökkentésével a hatás teljesen kiküszöbölhető lenne. C_{CM} csökkentésére a gyártók a jobb minőségű optocsatolók adó és vevő áramköreit elválasztó térrészbe a vevő tranzisztor emitterével közösített árnyékolást építenek be. Ezzel a megoldással a mindenképpen meglévő C_{CM} kapacitás a primer oldal és ezen árnyékolás között alakul ki, a rajta folyó dinamikus áramlökés nem befolyásolja kimeneti tranzisztorunk bázisának vezérlését.



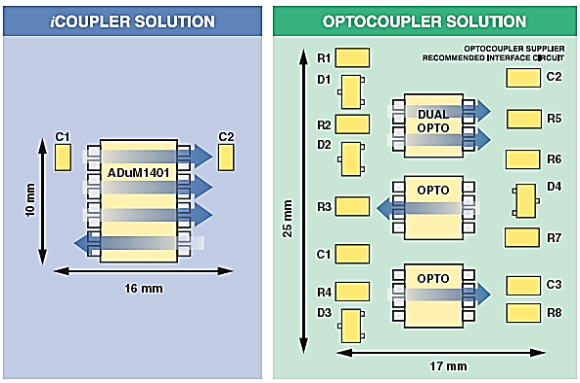
Közös modusú elnyomás javítása belső árnyékolással

Az optocsatolók vezető szerepet játszanak a galvanikus leválasztásban egyszerűségük, olcsóságuk, megbízhatóságuk és zavarérzéketlenségük miatt. Megjelentek azonban más konkurens megoldások is, melyek az információ átvitelére induktív vagy kapacitív csatolást alkalmaznak.



Leválasztás - optikai, induktív vagy kapacitív úton ?

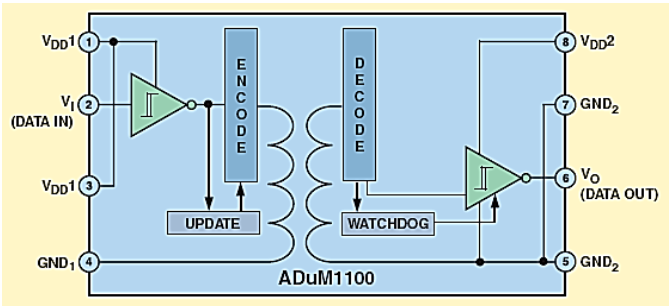
Nézzük először az Analog Devices cég konkurens termékét, az induktív csatolást alkalmazó iCoupler névre hallgató eszközcsaládot, mely figyelemreméltó alternatívát kínál az optocsatolós megoldásokkal szemben.



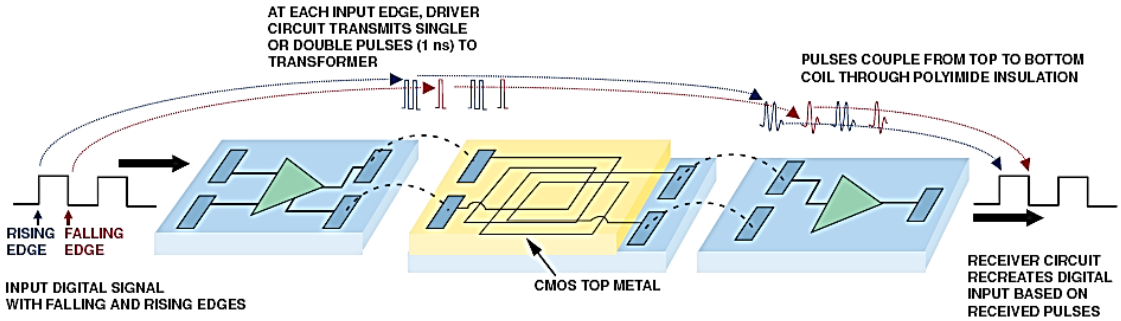
Hely- és alkatrész takarékosabb megoldás

A javaslat mindenekelőtt egy nem drágább, terjedelmében, alkatrészigényében lényegesen kedvezőbb megoldásnak tűnik, ami a miniatürizált termékek legdrágább kategóriájával, a helyigénnyel bánik a legtakarékosabban.

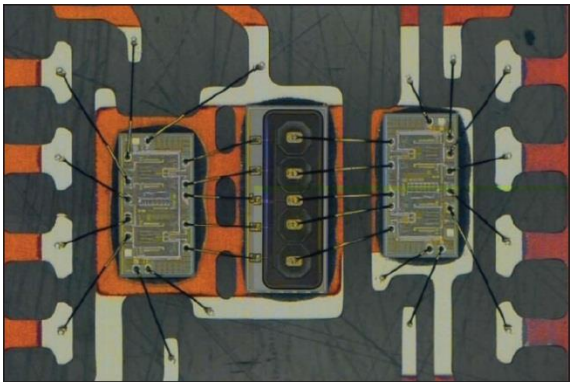
Szabadalmaztatott működési elve a következő: a bejövő digitális jel minden élét a belső kódoló logika igen rövid (1 nsec időtartamú) impulzusokká alakítja. A különböző (le- vagy felfutó) élek 1 vagy 2 impulzus formájában kerülnek átvitelre a túloldali dekódolóhoz, miniatűr légmagos induktív csatolásban lévő tekercsek segítségével. Fontos tehát hangsúlyoznunk, hogy egy élvezérelt, a két él között a szekunder oldalon tárolt információátvitelről van szó. Annak érdekében, hogy egy hosszú ideig (pl. több másodpercig) fennálló statikus jelszint hibás működés folytán ne lehessen változás hiányában hamis szinten a túloldalon, egy frissítő áramkör adott időnként megismétli a jelszintet (ezt megteheti, hiszen a különböző élváltások jelzésére különböző számú impulzust használ), aminek periodikus megérkezését watchdog áramkör ellenőrzi a túloldalon. Amennyiben max. 5 µsec időtartamig nem érkezik jelzés, a szekunder kör a primer kört hibásnak vagy tápellátás nélkülinek minősíti és a kimenetet alapállapotba állítja.



Az iCoupler blokkvázlata (Analog Devices)

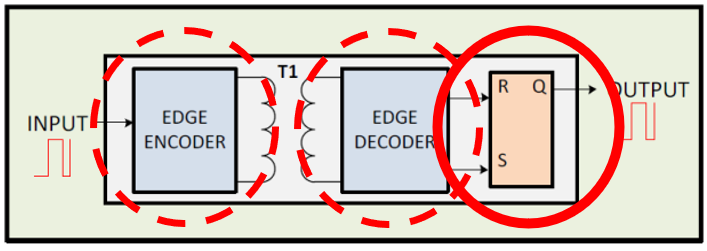


Az iCoupler működési elve



Az iCoupler belső felépítése

Mielőtt a fenti előnyök alapján végképp lemondanánk valamennyi optoelektronikus és egyéb elven működő galvanikus leválasztó egységről, meg kell említsük az iCoupler eszközök egy jelentős hátrányát is. Ehhez ismételten emlékeztetni szeretnénk arra, hogy az eszköz az élek változásán alapuló információátviteli módot használ, két impulzus között a leválasztott oldalon tárolt információ szolgáltatja a jelszintet. Érezhető a megfogalmazásból, hogy a problémát az elektromágneses zavarásból adódó téves impulzusok, és az ezek alapján bekövetkező téves jelszint-váltások fogják jelenteni. Amennyiben ugyanis a szekunder oldali állapot hibás impulzusérzékelés folytán megváltozik, ezt az állapotot vagy a következő jelszint-váltás, vagy az áramkörbe beépített update-logika fogja tudni helyreállítani. Mint említettük, utóbbi 5 µsec-onként következik be, azaz a jelszint ennyi idáig hibás állapotot is tükrözhet.

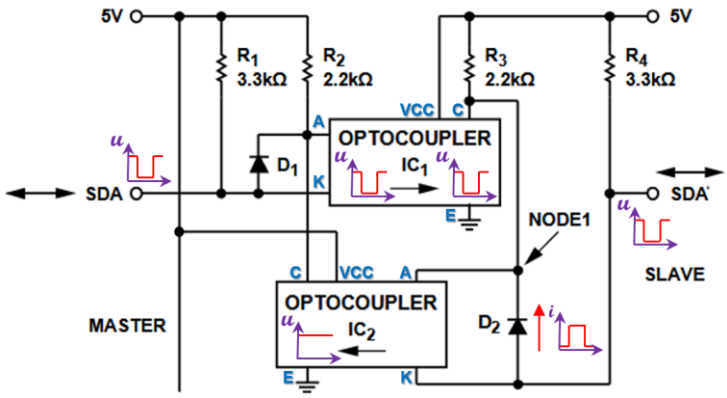


iCoupler = élvezérlés + állapottárolás

Ez a rövidnek tűnő idő is nagy bajt tud okozni nagyobb sebességű adatátvitelek esetén. Gyakran alkalmazott módszer, hogy potenciálisan elválasztott egységek logikai összekapcsolására galvanikusan leválasztott SPI interfészt alkalmazunk, melynek kommunikációs adatátviteli sebessége akár több tíz MHz is lehet (ld. pl. a következő fejezet esettanulmányában a buszkommunikációs egységet a mikrokontrollerrel összekapcsoló adatátvitelt). Ilyen adatátvitel esetében akár néhány tized µsec ideig fennálló téves jelszint is bittévesztést válthat ki a szekunder oldalon – ami visszafordíthatatlanul hibás adatátvitelhez vezet.

Mi is a bemutatott hibajelenség oka ? Az, hogy a leválasztó egység tényleges adatátviteli sebességéhez (akár 50-100 MHz) képest relatíve „sokáig" maradhat fenn hibás jelszint elektromágneses zavarás következtében a szekunder oldalon. Természetesen a hiba felismerése, az azt követő pl. ismételt adatátvitellel javítható, vagy az információ helyreállításhoz szükséges időnél (5 µsec) lényegesen lassabb adatfolyamok esetében a fenti jelenség nem is érzékelhető a fogadó oldali eszköz számára.

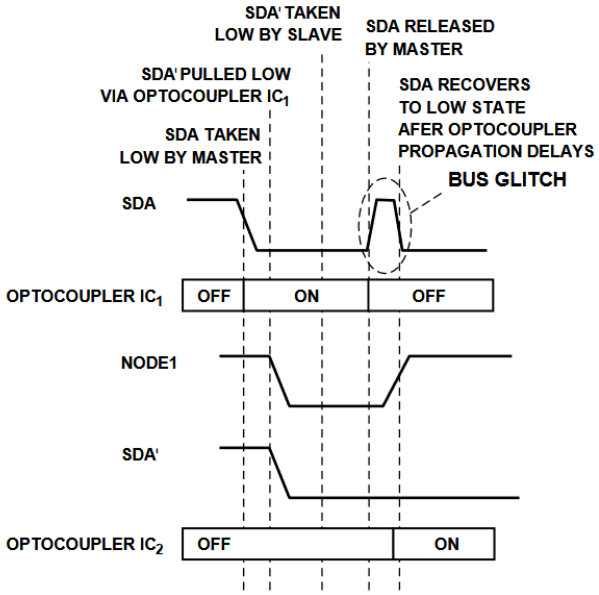
A galvanikus elválasztásnál is felvetődik ugyanaz a probléma, amivel a különböző tápfeszültségről működő egységek illesztésénél már találkoztunk: a kétirányú adatátviteli vonalak elválasztása azokban az esetekben, mikor az adatáramlás aktuális iránya digitális vezérlőjel formájában nem áll rendelkezésünkre. Ennek tipikus esete pl. az I^2C busz adat- és órajel vezetékeinek galvanikus leválasztása. Természetesen erre is léteznek már megoldások, de ezeknél fokozottan ügyelnünk kell a következőkre:



Kétirányú vezeték galvanikus elválasztása optocsatolókkal

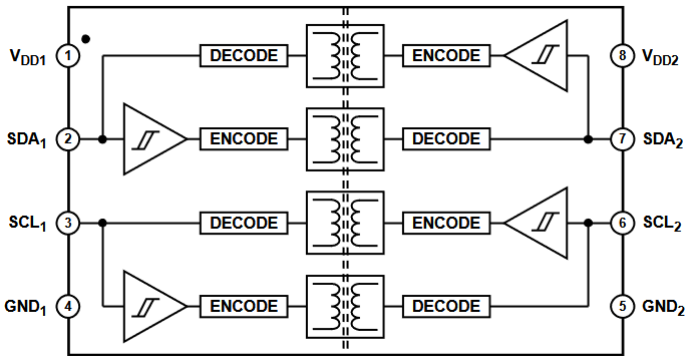
A leválasztás mindkét irányba egy-egy optocsatoló segítségével történik. Az optocsatolók kimenetei tipikusan O/C jellegűek, így pl. ha az ábrán látható master egység a vonalat logikai 0 állapotba húzza, IC_1 optocsatoló LED diódája világítani fog, azaz kimenete (az R_3 ellenállásnál) logikai 0 állapotba kerül. IC_2 optocsatoló inaktív lesz, D_2 diódán keresztül a slave egység SDA' vonala 0 potenciálra kerül (pontosabban a dióda nyitóirányú feszültségesésének megfelelő potenciálra). Ha SDA logikai 1 állapotba tér vissza, IC_1 optocsatoló LED diódája kialszik (mindkét optocsatoló inaktív). Ha ebben az állapotban a slave egység aktíválja az SDA' vonalat logikai 0 szintre, az előbb elmondott folyamat fordítottja játszódik le IC_2 és D_1 segítségével.

A működés így első ránézésre egyszerűnek látszik, mégis rejt némi kellemetlenséget, ha részletesebben megnézzük egy-egy jelváltás időbeli lefolyását.



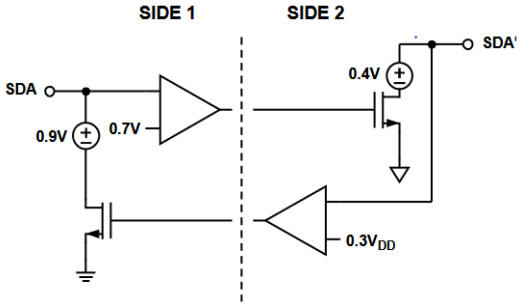
Az optocsatolós galvanikus kétirányú leválasztás időbeli lefolyása

SDA logikai 1→0 jelváltása után IC_1 kimenete (az ábrán NODE1 pont) némi időkéséssel vált logikai 0 szintre. Ezt természetesen követi az SDA' vonal is (D_2 nyitóirányú feszültségesésének szinteltolásával – de még ez is bőven benne van a logikai 0 jeltartományban). Most a slave egység is aktív állapotba kapcsolja az SDA' vonalat (megteheti akár arbitrációs, akár hibajelzési céllal), aminek hatására semmi nem változik, de az SDA' vonalat már nem D_2 tartja lehúzott állapotban. Amikor a master visszaveszi logikai 1 jelszintre saját SDA vonalát, IC_1 LED-je kikapcsol, amit kimeneti tranzisztora is követ a szokásos t_{off} időkéséssel. Eddig az időpontig IC_2 LED diódája nem tud aktiválódni, IC_2 kimenete inaktív, SDA visszatér logikai 1 állapotba. Viszont amint IC_1 kimenete kikapcsolt, IC_2 aktiválódik és lehúzza az SDA vonalat logikai 0 állapotba. Ily módon a váltáskor rövid idejű zavaró tüske keletkezhet, ami nagyfrekvenciás zavarás forrása lehet.



Az AduM1250 áramkör belső felépítése

De térjünk vissza az Analog Devices induktív megoldásához az adott kérdésre. Az AduM1250-es áramkör mind az SDA, mind az SCL buszvezetékek leválasztását megoldja egyetlen kis 8 kivezetéses integrált áramköri tokban.

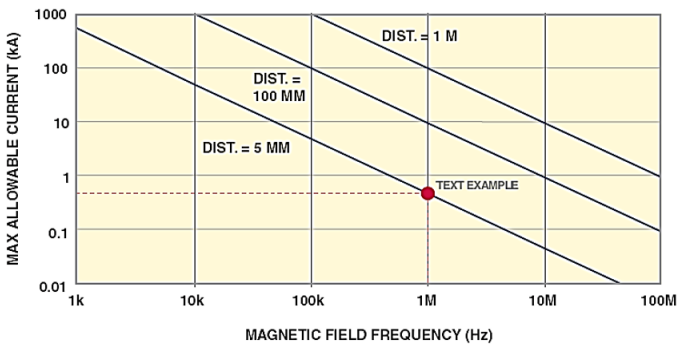


Vázlat az AduM1250 áramkör működésének megértéséhez

Az áramkörben az SDA és az SDA' vonalakat mindkét oldalon egy O/D kapcsolás húzza aktív 0 állapotba, a vonalakat egy-egy vevőerősítő figyeli, amely a már megismert induktív csatolással kapcsolódik ezekhez a FET tranzisztorokhoz. A kapcsolás első ránézésre „önzárónak" tűnhet, mivel ha SDA logikai 0 jelszintre vált, a jobboldali FET 0 szintre húzza az SDA' vonalat, minek hatására az alsó erősítő is bekapcsolja a baloldali FET tranzisztort, ezzel pedig a kapcsolás reteszelődne logikai 0 állapotban.

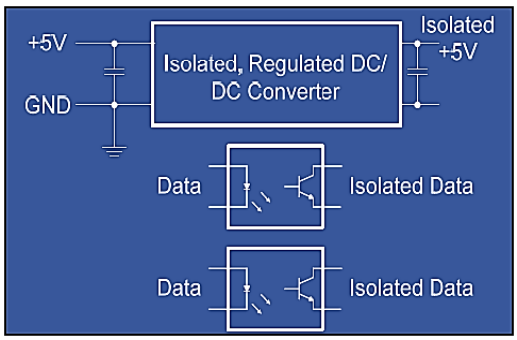
A problémát szinteltolásokkal oldják meg, ezzel szüntetik meg a pozitív visszacsatolást. Az SDA' oldali meghajtásról működő baloldali FET kimenete 0,9V szinttolással húzza logikai 0 szintre az SDA vonalat (bőven benne vagyunk a logikai 0 tartományban), az ugyanezen az oldalon található vevőerősítő viszont csak 0,7V alatt érzékeli logikai 0 állapotnak a vonalat. Emiatt az SDA' vonal logikai 0 szintű meghajtása SDA-t logikai 0 szintre viszi (0,9V-ra), de a felső erősítő és az SDA' oldali FET inaktív marad, ezáltal a hurok felnyílt és a pozitív visszacsatolás nem jön létre. A hurkot elegendő egy ponton felnyitni, a SIDE 2 oldalon hasonló szinttolásra már nincs szükség.

Mágneses elven működő érzékelők esetében azonnal adódik az egyik legfontosabb kérdés: a mágneses mezőt használó berendezések és a nagyáramú vezetékek által létrehozott mágneses tér által történő zavarás. Emiatt az Analog Devices valamennyi iCoupler eszköz adatlapján publikálja az alábbi diagramot:

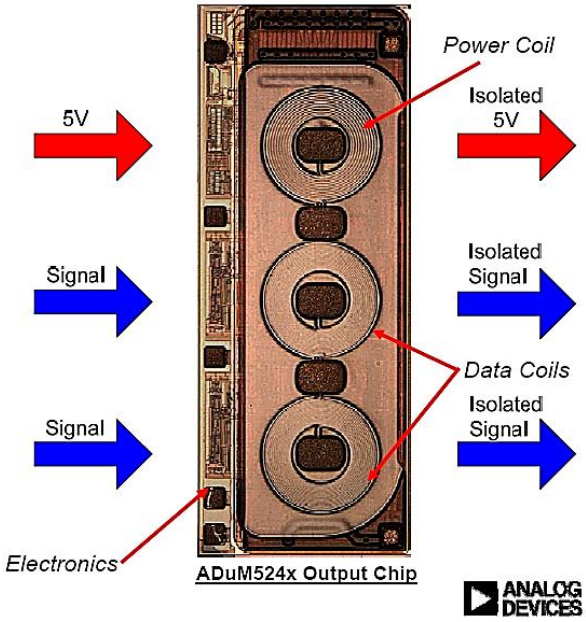


Mágneses immunitás a környezettel szemben

Az ábrán bejelölt pont azt mutatja, hogy a kb. 500 A-es áramerősségű kábel 5 mm-es távolságból akkor zavarja meg az adatátvitelt, ha a váltakozó áram > 1 MHz frekvenciájú komponenst tartalmaz.
Figyelembe véve az alkalmazásokat, a galvanikus leválasztások többsége galvanikusan leválasztott tápfeszültséget is igényel. A leválasztás túloldalán található fizikai jelszint illesztő meghajtó és fogadó áramkörök többnyire igen kis tápenergiát igényelnek, viszont annak előállításához vagy központi tápegységünkben egy további, leválasztott tápfeszültségre, vagy egy lokális DC/DC konverterre lesz szükségünk.



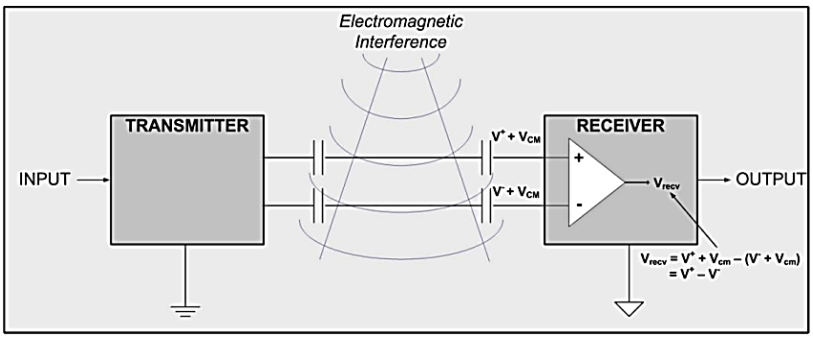
Galvanikus leválasztás tipikus problémája: leválasztott +5V



A leválasztott +5V igényének megoldása

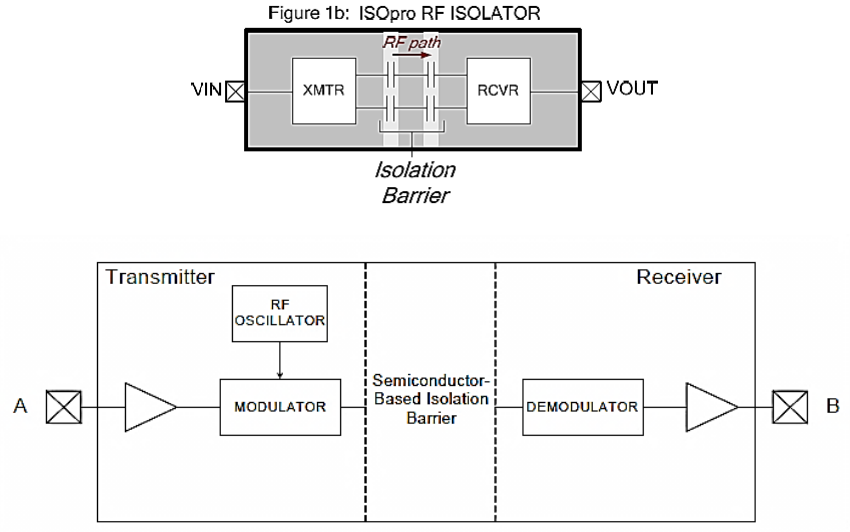
Az iCoupler eszközcsalád 524x sorozata ezt a problémát is megoldja: az integrált áramkör a két jelvezeték leválasztásán túlmenően magában foglalja a tápellátás leválasztott átvitelét is a túloldalra (max. 5 mA terhelőáram erejéig, ami az alkalmazások jelentős részéhez elegendő).

Az elmondott tulajdonságokból tanulva a Silicon Laboratories már egy zavarokra érzéketlenebb, kapacitív elven működő eszközcsaláddal (ISOpro Digital Isolator) jelent meg a piacon. (A cég analóg üzletágát 2021-ben felvásárolta a Skyworks Solutions Inc., így jelenleg már ők gyártják és forgalmazzák az eszközt.) Az elv viszonylag egyszerű: az információátvitel a maximális jelsebességnél (150 MHz) lényegesen nagyobb sebességű modulált RF jel segítségével történik, ily módon az eszközben nincs szükség állapottárolóra. Természetesen a jelátvitel elektromágneses térrel történő megzavarása itt sem zárható ki, ezt azonban a differenciális (kapacitívan elválasztott) jelátvitel lényegesen megnehezíti, egy esetleges hibaállapot pedig a zavaró impulzus megszűnése utána automatikusan helyreáll. A differenciális jelátvitel nagymértékben javítja az eszköz zavarérzéketlenségét, hiszen az átviteli jelutak megzavarása önmagában még nem vált ki biztosan információtorzulást, mivel a vett jel két jel különbségéből kerül előállításra. Az elvet ma már több nagy áramkörgyártó is alkalmazza (pl. a Texas Instruments ISO776x elemcsaládja, 5000V_{RMS} állandó és 12,8 kV lökőhullám szigetelési paraméterekkel).



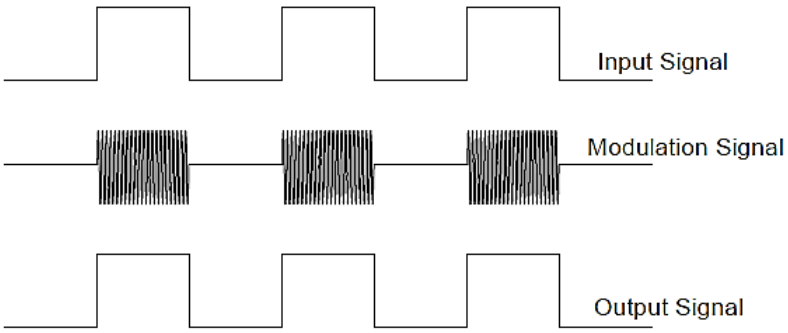
Differenciális jelátvitel + keskenysávú vétel
elnyomja a szigetelőrétegben keletkező elektromágneses zavarásokat

A digitális leválasztók belső blokkvázlatát mutatja a következő ábra:



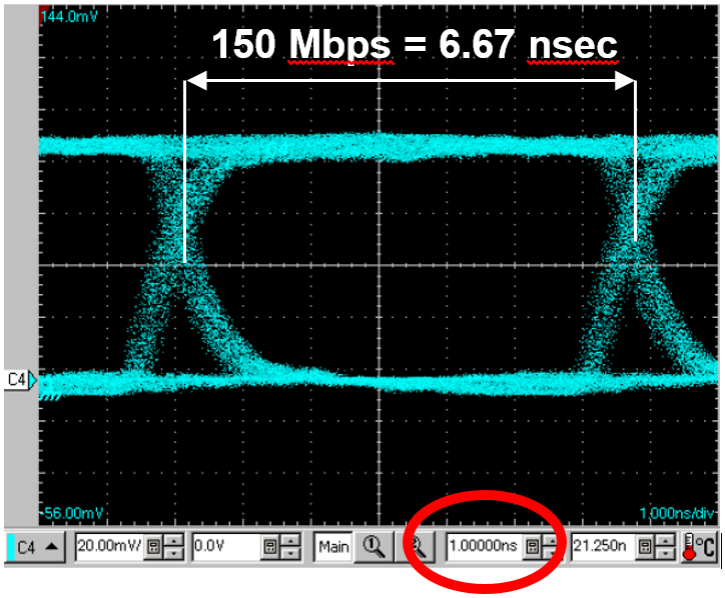
A digitális leválasztók belső blokkvázlata

Az információ átvitelére a modulátor egység rendkívül egyszerű módszert használ: az ampitúdóeltolás-billentyűzés (ASK) legegyszerűbb fajtája került beépítésre, amikor az átvinni kívánt információ szintje „kapcsolgatja” a vivőjelet (On-Off-Keying – OOK).



On-off billentyűzés (OOK)

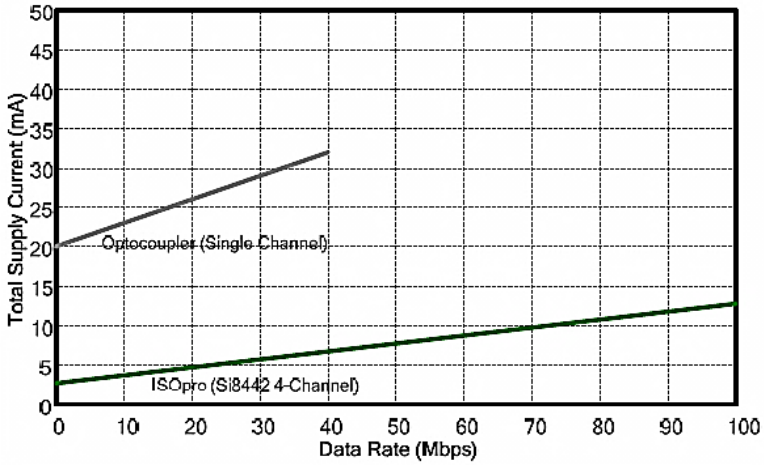
Bár a cég az átvitelhez használt vivőfrekvencia értékét nem adja meg, a digitális jelek átvitele során elterjedten használt szem-ábra – amelyet a jel „elkenődésének”, torzulásának vizsgálatára használunk – jól mutatja a tényleges jelátvitel jóságát. (Emlékeztetőül: a szem-ábra felvételénél a vizsgált jelkombinációkhoz tartozó jelek sorozatát a bemenő adat éleihez szinkronizálva egymásra rajzolják, és ezt jelenítik meg a képernyőn. A szem-ábrán valamennyi zaj: a csatorna véges sávszélessége, a frekvenciától függő jelfutási idő által kiváltott alaktorzulás, valamint a csatorna saját zajának hatásai is megjelennek.) Felhívjuk az olvasó figyelmét a gyártó által publikált mérésben alkalmazott oszcilloszkóp felbontására: 1 nsec/osztás !



Szem-ábra maximális bitsebesség mellett (jitter ≈ 350 psec)

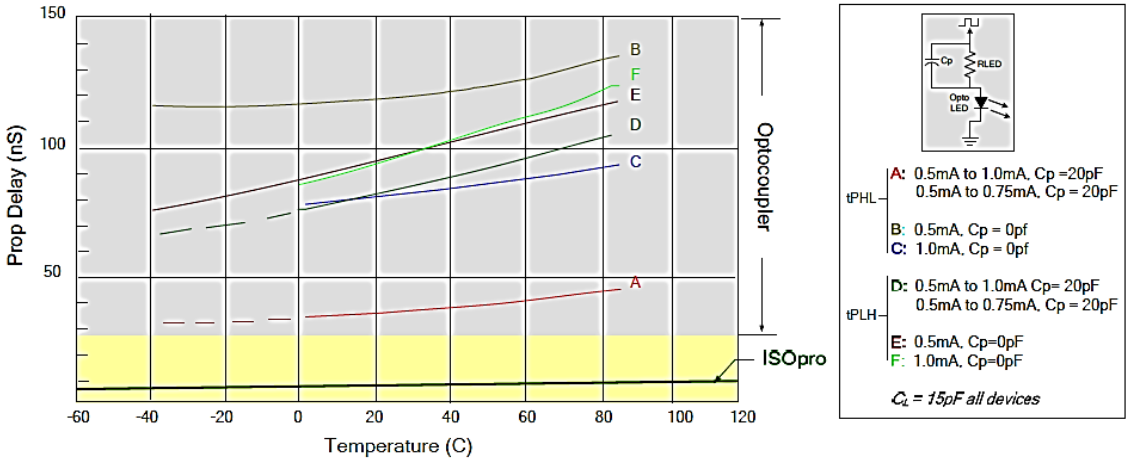
A működési elv megismerése után nézzük a számunkra legfontosabb eszköztulajdonságokat: tápenergia-szükségletet, a jelkésleltetést/maximális adatátviteli sebességet, a zavarérzéketlenséget (a közös modusú zajelnyomást és külső tér iránti érzéketlenséget) és végül a nem elhanyagolható helyszükségletet.

Az alkalmazott CMOS technológia még a nagyobb frekvenciák ellenére is lényegesen jobb értékeket mutat, mint az optocsatolók – sajnos nem túl jó – energiafogyasztási tulajdonságai (20-30mA vs. 3-12mA), nem is beszélve a sebességről (max. 150 MHz) és a jelkésleltetési időkről (impulzusszélesség-torzítás max. 4,5 nsec). Természetesen vegyük figyelembe, hogy az optocsatolók esetében is a legnagyobb sebességű elemeket hasonlítottuk az ISOpro család elemeihez.



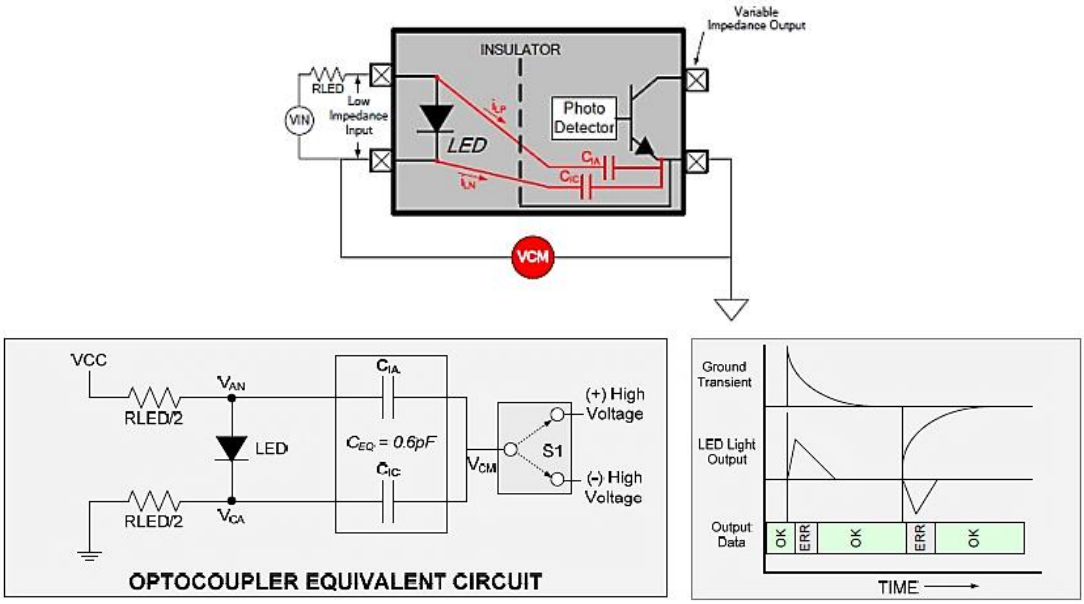
Tápáram-felvétel vs adatátviteli frekvencia

Az optocsatolók esetében bevett technika, hogy a bekapcsolási folyamatot „áramlökéssel” forszírozzuk, erre a fotodióda munkaellenállásával párhuzamosan kapcsolt kapacitás szolgál, ami viszont kellemetlen áramcsúcsokat vált ki a meghajtó eszközben. Ezzel együtt az ISOpro elemek jelkésleltetési idejei mintegy 3-5-ször kisebbek az optocsatolóknál megszokott értékeknél.

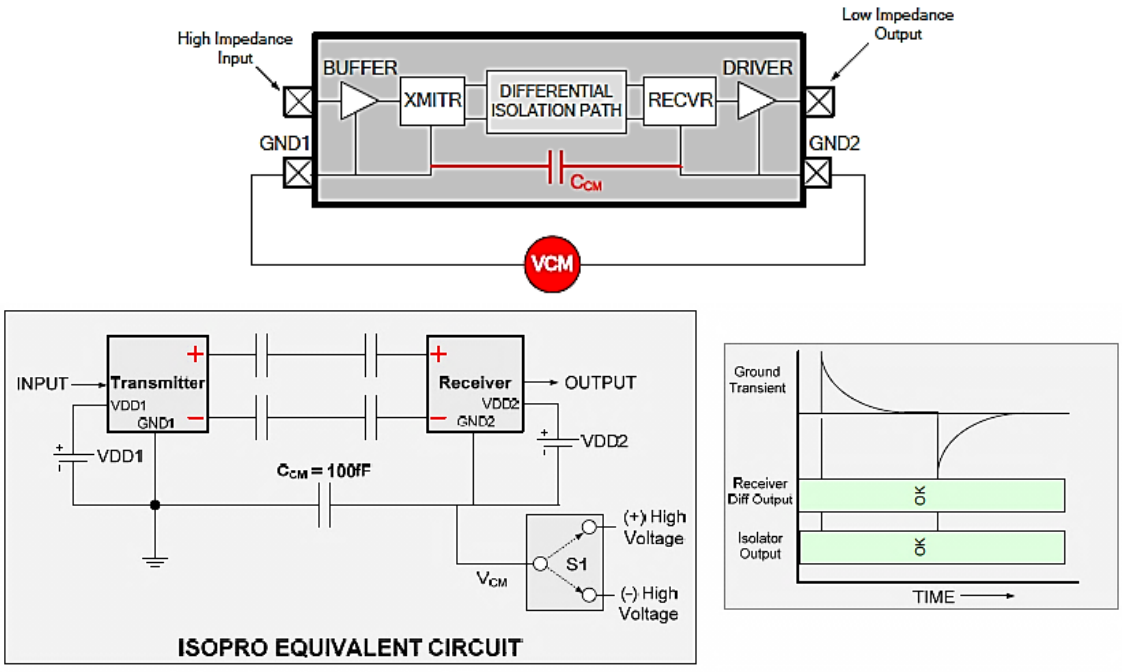


Jelkésleltetési idők – optocsatolók (A-F) vs ISOpro

A közös modusú jelek tranziens elnyomásában a differenciális jelátvitel „verhetetlen” a hagyományos optocsatolókhoz képest: az ekvivalens kapacitás kb. az 5..10-ed része az optocsatolóknál – árnyékoló lemez alkalmazásával – elérhető értéknél.

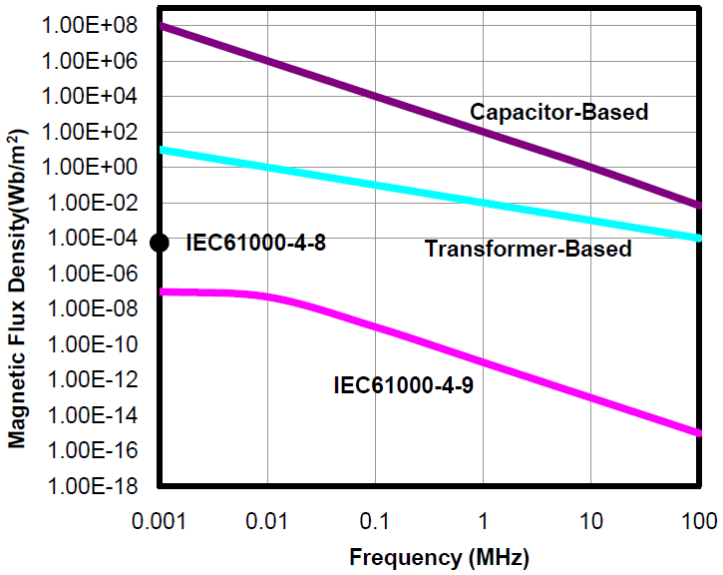


Közös modusú zavarelnyomás – optocsatoló esetében

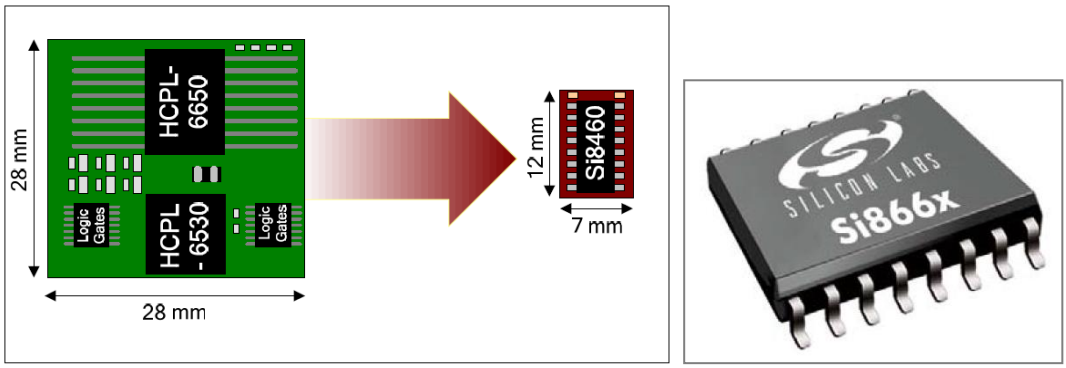


Közös modusú zavarelnyomás – RF differenciális jelátvitel mellett (1fF = 10^{-15}F)

A gyártó által publikált mágneses tér érzéketlenség jó összehasonlítási alap a bemutatott induktív és kapacitív elven működő eszközökre a szabvány előírásaihoz képest.



Mágneses tér immunitás az egyes leválasztási módok esetében ($1Wb/m^2=1T$)



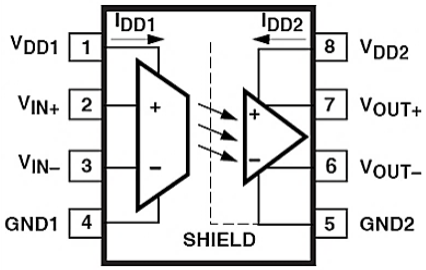
Helyszükséglet: 6 csatornás CMOS jelszintű leválasztó fokozat

Az eddig bemutatott megoldások mindegyike kétállapotú digitális jelek leválasztására használható. A leválasztó áramkörök nemlineáris, nagy paraméterszórású karakterisztikái nem alkalmasak pontosabb analóg jelek átvitelére. Az analóg jelek galvanikus leválasztása lényegesen drágább és ebből adódóan lényegesen ritkábban előforduló megoldás – sokkal inkább alkalmazzuk azt a megoldást, hogy az analóg jeleket egy, a környezetüktől potenciálisan nem független külső A/D átalakítóval fogadjuk, vagy D/A átalakítóval állítjuk elő, és ezeket az eszközöket a digitális oldalukon (pl. a soros SPI buszon) választjuk le a mikrokontrollerről.

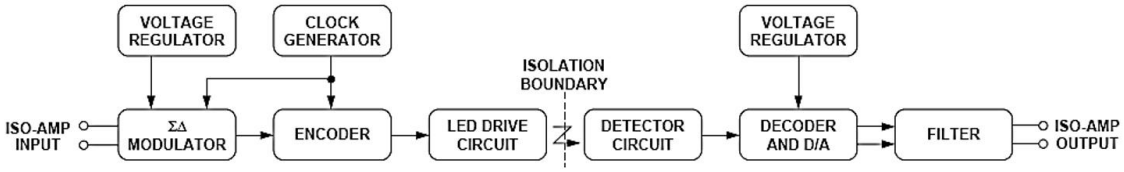
Amennyiben mégis szükséges lenne az analóg jelek galvanikus leválasztására, ún. leválasztó erősítőket kell alkalmaznunk. Ezek a méregdrága eszközök viszonylag nagy jelhűséggel (nemlinearitási hiba ≈ 0,1%), de messze nem ilyen pontos eredő erősítéssel

(általában 2..5%) választják le az analóg jeleket – pontosabb alkalmazások esetén utóbbi korrekciót és hitelesítést igényel.

Az eszközök egyik csoportja az analóg jelet digitalizálja, a digitális jelet választja le, majd átvitel után visszaalakítja analóg jellé. A HCPL-7850 áramkör 100 kHz sávszélességű jelek átvitelére alkalmas, primer oldalon egy Σ-Δ A/D átalakító digitalizálja a jelet, majd a másik oldalon D/A átalakító alakítja vissza azt. A hagyományos, monolitikus IC technológiára épülő áramkör eredő erősítésének pontossága 5%, nemlinearitása 0,1%.

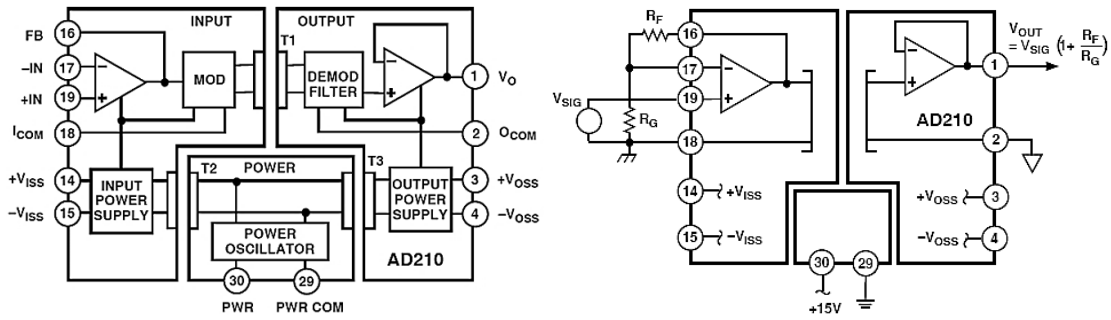


HCPL-7850



A HCPL-7850 belső felépítése

Az Analóg Devices AD210 áramköre induktív úton történő jelátvitelt alkalmaz modulációs-demodulációs technikával. Az előbbinél lényegesen nagyobb, hibrid áramkör érdekessége, hogy háromportos galvanikus elválasztást alkalmaz, egyetlen tápfeszültségből előállítja mindkét oldal működtetéséhez a segédfeszültségeket. Erősítés hibája 2%, nemlinearitása 0.012%.



AD210 funkcionális blokkdiagram és alkalmazási példa (A_u > 1)

4.6 Külső memória elemek illesztése

Beágyazott rendszerekben sokszor van szükségünk külső adattároló elemekre. Ezek célja általában kettős, és ennek megfelelően két különböző memóriatípust is használunk a feladatokhoz:

- egyik esetben **nagyobb adattömeget** kívánunk tárolni, amely tartalmát tekintve vagy változatlan, vagy változása a működési sebességhez képest nem túl gyakori. Előbbire lehet példa akár nagyobb tömegű programkód tárolása (melynek futtatását pl. DSP-k esetében úgyis belső RAM memóriából végezzük), nagyobb tömegű konstans adat (érzékelő karakterisztikák, aritmetikai segédtáblázatok, stb.), de ide sorolhatjuk egy mérésadatgyűjtés során keletkezett adattömeg tárolását, utólagos kiolvasás vagy feldolgozás számára. Az írási műveletek gyakorisága ilyen adatoknál percekben, napokban vagy még nagyobb egységekben mérhető.
- a második esetben a tartósan (kikapcsolás után is megmaradó módon) tárolandó adatok mennyisége lényegesen kisebb, viszont ezeket kisebb egységekben és az előbb említettnél **nagyobb gyakorisággal** szeretnénk változtatni (pl. egy irányítás paraméterrendszere, utolsó beállítási állapot eltárolása, relé vagy pneumatikus szelep teljesített kapcsolásainak száma, stb.)

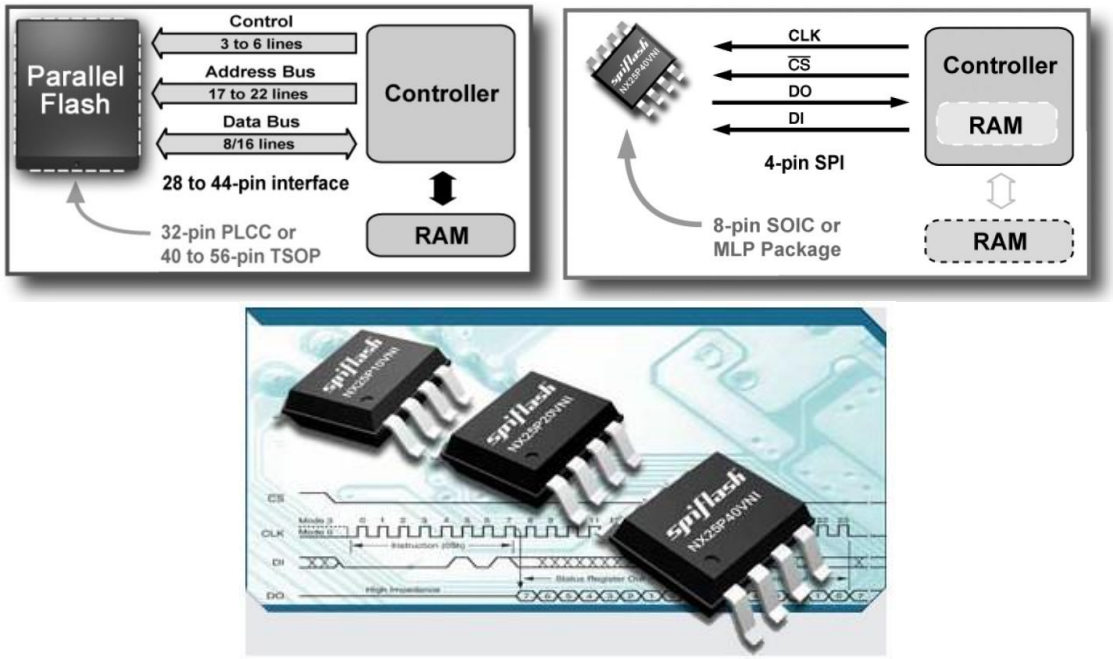
Egyik esetet se keverjük a normál RAM memória írási műveleteinek sebességével. A vázolt feladatra az EEPROM és a flash memóriaelemeket használjuk, melyek közös tulajdonsága, hogy az írási műveletek hossza lényegesen nagyobb, mint a RAM memóriák esetében. Ennek megfelelően normál programváltozók, verem, stb. kialakítására nem alkalmasak, az írási műveletek valamilyen mértékben károsítják az eszköz belső félvezető elemeit (a gate elektróda szigetelési ellenállásának letörésére alapozódnak belső műveletek), így az írási műveletek maximális száma valamennyi ilyen tároló elem esetében korlátozott.

Tároló típus	Tartalom megmarad	Írás sebessége	Írás alapegysége	Törlés alapegysége	Ár/cella	Irások max. száma
SRAM	nem	5-70 nsec	bájt	bájt	közepes	∞
EEPROM	igen	1-5 msec	bájt/lap	bájt	magas	1 000 000
FLASH	igen	5-10 μsec	bájt/lap	szektor	alacsony	10-100 000

A két nem felejtő tároló elem működési elve sok szempontból hasonló, a leglényegesebb különbség felhasználói szempontból az, hogy az EEPROM esetében egy-egy bájtot önmagában bármikor megváltoztathatunk, a flash esetében a törlés alapegysége jóval nagyobb (2-64 kbájt), ez a hosszadalmas művelet a szektor/blokk valamennyi celláját törli (általában „1” állapotba), majd ezeket bájtos egységekben változtathatjuk a legközelebbi törlésig. A bájtonkénti törlés az EEPROM memóriát lassabbá, és jóval költségesebbé teszi.

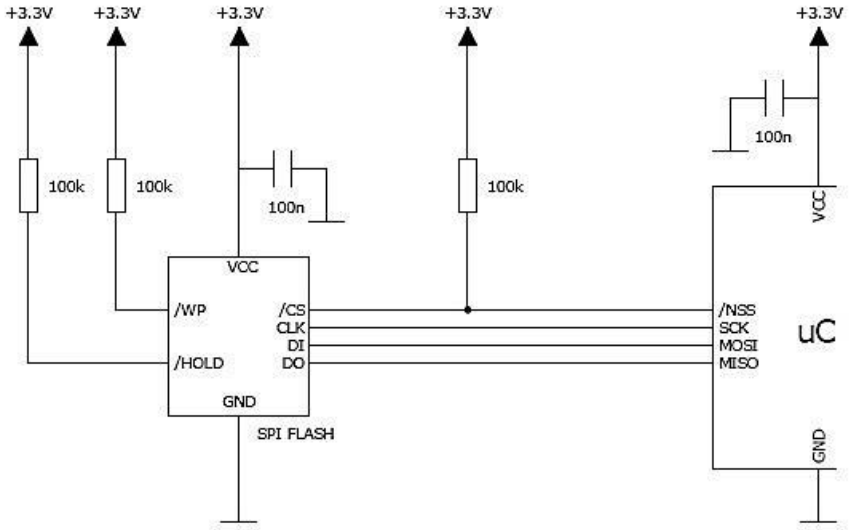
A tárolt adattömegek mérete miatt is a két memória alapvetően különböző illesztési felülettel rendelkezik. A műveletek sebességeihez a korábban már megismert soros

adatátviteli interfészek tökéletesen megfelelnek, csökkentve ezzel a kommunikációs interfész kialakításához szükséges lábak számát. Viszont amíg a flash memóriák nagysebességű (70 MHz !) SPI felületen keresztül kommunikálnak a mikrokontroller megfelelő egységével (közel 10 Mbájt/sec), addig az EEPROM-ok vagy lassabb SPI (1-5 MHz), vagy I^2C kommunikációs felülettel rendelkeznek (400 kbit/sec).



A soros flash memóriák nagysebességű SPI interfésszel rendelkeznek

A következő ábra egy SPI interfésszel rendelkező soros flash memória illesztését mutatja mikrokontrollerhez, a megoldás nem igényel külön magyarázatot.



Soros flash memória illesztése mikrokontrollerhez

A nagy adattömeg (1-64 Mbit = 128 kbájt – 8 Mbájt) egyetlen parányi 8 lábú integrált áramkörbe „belefér”, az írási és törlési műveletek esetében hozzávetőlegesen az alábbi időkkel kell számolnunk:

Symbol	Parameter		Min	Typ	Max	Units
t_{PP} (1)	Page Program Time (256 Bytes)			1.2	5	ms
t_{BP}	Byte Program Time			7		µs
t_{BLKE} (1)	Block Erase Time	4 Kbytes		50	200	ms
		32 Kbytes		250	600	
		64 Kbytes		400	950	
t_{CHPE} (2)	Chip Erase Time			6	14	sec
t_{WRSR} (2)	Write Status Register Time				200	ns

SPIflash írási és törlési paraméterei

Lassabban írható, de alapegységekben (bájtonként) törölhető tárolóként az EEPROM memóriákat használjuk.

Symbol	Parameter	Min	Max	Units
f_{SCL}	Clock Frequency, SCL		400	kHz
t_{LOW}	Clock Pulse Width Low	1.2		µs
t_{HIGH}	Clock Pulse Width High	0.6		µs
t_{WR}	Write Cycle Time		5	ms
Endurance (2)	5.0V, 25°C, Page Mode	1M		Write Cycles

EEPROM írási sebessége és paraméterei

Az illesztés az SPI vagy az I^2C interfészeknél megszokott módon történhet, azonban ezeknél az elemeknél egy egészen más, kezeléstechnikai problémára is gondolnunk kell a rendszer tervezésekor.

A problémát az okozza, hogy az EEPROM tulajdonságait az esetek többségében arra használjuk, hogy rendszerállapotot (paraméterek aktuális értékét, konfigurációt, utolsó állapotot, stb. tárolunk úgy, hogy a tároláshoz több 10 vagy 100 bájtnyi információt kell eltárolnunk. Ezek az adatok összetartoznak, egy állapot folytatásához, helyreállításához valamennyire szükségünk van, a töredékükkel nem tudunk mit kezdeni. Egy **konzisztens adatokat** tartalmazó blokkról beszélünk tehát, amelynek tárolása alatt bekövetkező rendszerleállás (tápkimaradás, reset) esetén a folytathatóság kérdőjeleződik meg, ha hibás, félig mentett adatsorozat alapján kíséreljük meg azt. Olyan megoldást kell kialakítanunk, amely garantálja, hogy a blokk felhasználásakor el tudjuk dönteni a legfontosabb kérdést: a benne található adatok összetartoznak (konzisztensek) és nem hibásak (nem egy félbeszakadt, hibás mentés eredményei).

Az elmondott probléma áthidalására alapvetően azt a megoldást kell alkalmaznunk, hogy két adatblokkot alakítsunk ki: az egyik, **az utolsó érvényes állapotot** (mentést) jelenti a

felhasználó programrészek számára, a másik képezze a **következő mentési műveletek** tárgyát oly módon, hogy ezen műveletek sikeres lezárásáig ezt a blokkot ne tekinthessük érvényesnek. Az érvényesség az írási folyamat bármely helyzetben való megszakadása esetén is eldönthető kell legyen a felhasználó programrész számára.

Alapesetben két teljesen azonos felépítésű blokk kialakítása a célszerű, melyek között pl. az első bájton található egyetlen írási művelettel változtatható sorszám döntse el a blokkok érvényességét (a nagyobb szám a későbbi, tehát ő az érvényes). A két sorszám szomszédos szám kell legyen, a 255 – 0 számpárost is szomszédoknak tekintve (melyek közül a 0 a „nagyobb", azaz a későbbi). Végezzük el a következő gondolatkísérletet egy bájtos szervezésű EEPROM esetében:

- legyen a két blokk sorszáma egy adott időpillanatban „i" és „i+1"
- következő mentésünket az „i" blokkba írjuk, a blokk számát egyelőre nem változtatva meg. Bárhol is szakad meg a folyamat, az „i+1" blokk „későbbi", és ő van érvényben
- adataink mentése után át kellene állítanunk a sorszámot „i+2"-re. Ez az írási művelet megszakadhat, és eredményeként a sorszám helyett szinte minden érték előfordulhat (egy cella fizikailag félig törölt, vagy félig beírt állapotában szinte semelyik érték sem zárható ki 0..255 között)
- következő felhasználáskor a nem szomszédos sorszámok alapján csak azt tudjuk megállapítani, hogy baj történt az utolsó írási műveletnél, de nem tudjuk, melyik blokkunk sorszáma helyes.
- Képezzünk mindig egy olyan ellenőrző összeget is a blokkokra (beleértve a sorszámot is), amely ellenőrizhetővé teszi a blokkban található adatok érvényességét. Előbbi mentésünknél azonnal megállapítható lenne, hogy kettőjük közül melyik tartalmaz helyes adatokat, és melyik mentése szakadt félbe.
- a sorszám beírása előtt tehát mentsük az új sorszám szerint kiszámított ellenőrző összeget (ami a régi sorszámmal még hibás blokkot mutat), majd utolsóként mentsük a sorszámot. Egészen eddig a lépésig senki sem akarja majd használni blokkunkat, mivel sorszáma kisebb a másiknál. Ennek a sorszámnak a mentése viszont ha félbeszakadna, bármilyen más sorszám rögzítése esetén az ellenőrző összeg nem lesz helyes.

A megoldás működik, de a felhasználás meglehetősen bonyolult ellenőrző algoritmust kíván meg:

- olvassuk be mindkét blokk sorszámát!
- a nagyobbik sorszámú blokknak olvassuk be az összes elemét és ellenőrizzük az ellenőrző összeg helyességét!
- ha ez helyes volt, akkor ebből, ha nem, akkor a másik blokkból vegyük az adatot.

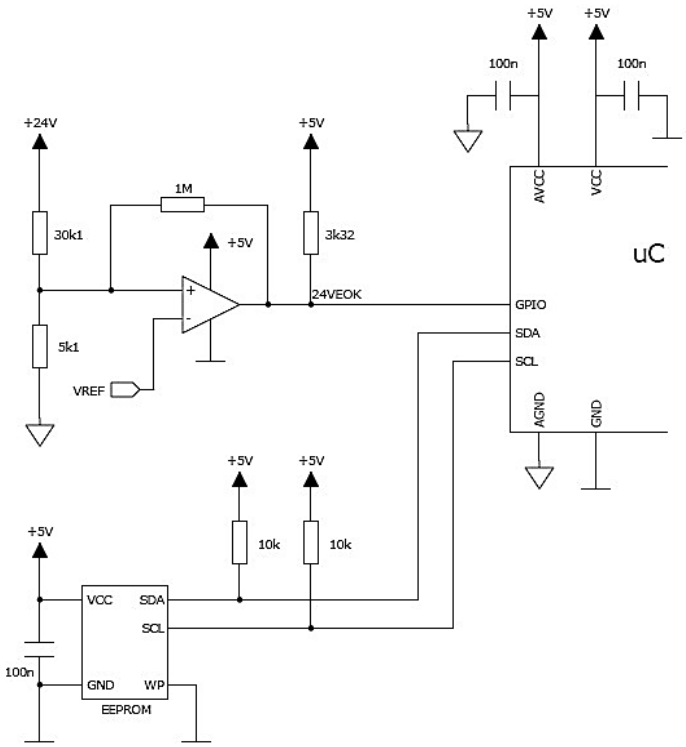
Sok alkalmazás számára nehézkes és lassú ez az algoritmus (egy-egy bájt beolvasása I^2C buszon 400 kbit/sec sebesség mellett 3 x 9 = 27 bit, azaz 67.5 μsec időbe telik), szeretnénk inkább elsőre „biztosra" menni a beolvasás során. Ez csak akkor lesz lehetséges, ha **előre garantálni tudjuk** egy-egy írási műveletünk sikeres befejezését, amihez viszont hardver

támogatás szükséges. Ebben az esetben ugyanis szintén a kettőzött blokkok gondolatmenetét követve az előző algoritmus a következőképpen alakul:

- Egy, a későbbiekben egyetlen írási művelettel módosítható mutató (pointer) segítségével határozza meg felhasználó programunk, hogy éppen melyik blokk érvényes (pl. 1 vagy 2). Új adatblokk mentésekor itt sem bántjuk sem az éppen érvényes blokk adatait, sem a mutatót, hanem az éppen „nem érvényes" másik blokkba kezdünk menteni. Ha ez a folyamat félbeszakadna, senki sem fog tudomást szerezni megkezdett műveletünkről, a következő mentést ugyanitt fogjuk kezdeni. Amennyiben az adatok írása sikeresen lezárult, már csak a bevezetőben említett pointerünket kell átállítanunk az új adatblokkra, ezt követően pedig a korábbi blokk lesz felülírható. A pointer átállítása egyetlen (fontos, hogy egyetlen!) írási művelet kell legyen, ennek a félbeszakadása nem szabad bekövetkezzen.

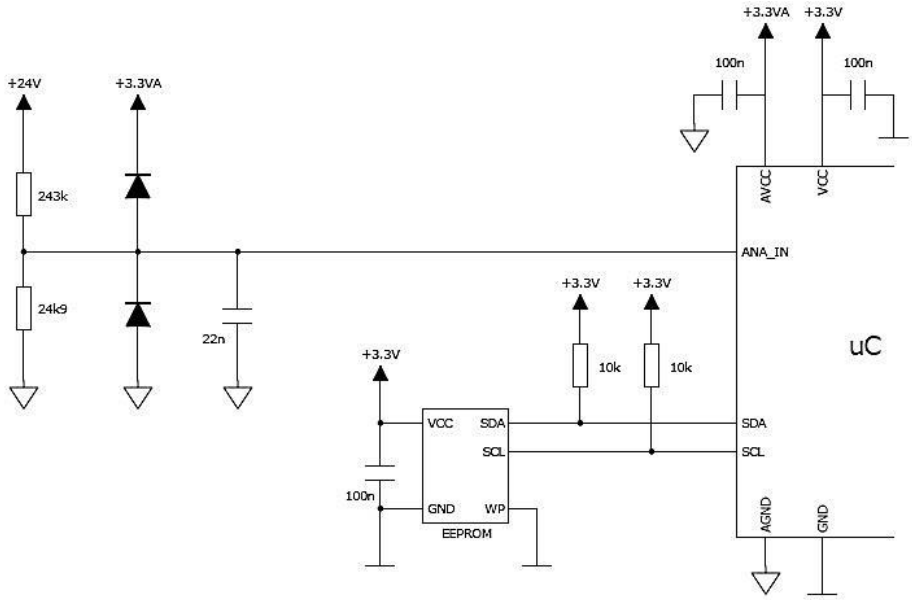
A megoldáshoz hardver kapcsolással ki kell alakítanunk egy olyan komparátor áramkört, amely egységünk primer (tápegység előtti) tápellátását figyeli. Áramkörileg könnyen megvalósítható az az előírás, hogy a primer tápellátás kimaradását követően a tápegység energiatároló elemei (elsősorban a primer és a szekunder oldalon lévő puffer kondenzátorok) kb. 20-50 msec időtartamig még ne engedjék meg a logika tápfeszültségének a leállítás (reset) küszöbfeszültségéig való lecsökkenését. Egy ilyen komparátor áramkör látható következő áramkörrészletünk bal oldalán középen. Az áramkör primer tápfeszültséget figyelő osztón előálló feszültséget egy VREF=2,5V-os referenciafeszültséggel hasonlítja össze folyamatosan, és amennyiben a 24V primer tápfeszültség 17V alá csökken ($17V \cdot 5{,}1k/35{,}2k = 2{,}46V$), az áramkör a processzor egy logikai bemenetén jelzést ad. Segítségével az algoritmus a következőképpen alakítandó:

- az éppen változtatható blokk adatait frissítjük (még egy ellenőrző összeget is a végére tehetünk a biztonság növelése érdekében),
- beolvassuk a primer tápfeszültség meglétét jelző bemenetet. Amennyiben az aktív, biztosak lehetünk abban, hogy a következő 10-20 msec-on belül nem következik be a tápellátás összeomlása, az EEPROM írásának idejére letiltott megszakítás-kéréssel (néhány μsec) kezdeményezzük a pointer átállítását az új adatblokkra. Az írási művelet fizikai végrehajtása (kb. 5 msec) alatt a kontroller már dolgozhat tovább akár más feladaton is, vagy várakozhat a művelet befejezésére. Inaktív előjelző bemenet (tápkiesés) esetén a műveletet el sem kezdjük, a régi adatblokk marad érvényben („már nem volt időnk menteni").



Külső EEPROM illesztése mikrokontrollerhez (digitális bemenettel)

A primer tápellátás kiesésének a bemutatott módon való előjelzését egy analóg bemenet segítségével még rugalmasabban oldhatjuk meg, mivel ilyenkor még a tápfeszültség figyelés komparálási szintjét is állíthatjuk programrendszerünkben.



Külső EEPROM illesztése mikrokontrollerhez analóg bemenet segítségével

4.7 Soros adatátviteli interfészek

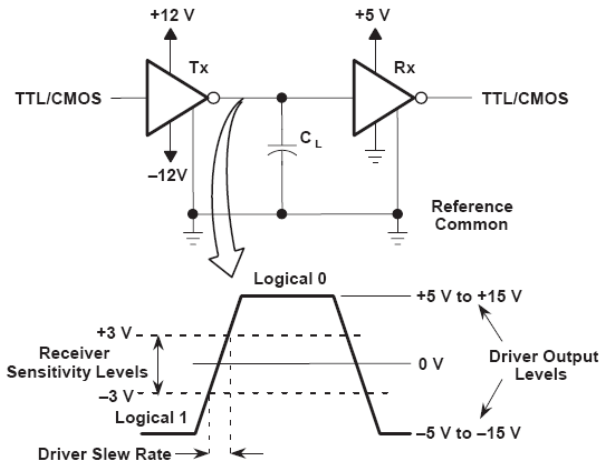
Az aszinkron soros adatátvitel tárgyalásakor már megemlítettük, hogy az ott ismertetett logikai szintű jelsorozat a különböző szabványok szerint más fizikai jelszinten kerül továbbításra a fizikai egységek között.

4.7.1 EIA RS232C

A mai szemmel nem túl nagy adatátviteli sebességre kialakított EIA RS232C (**E**lectronic **I**ndustries **A**lliance **R**evised **S**tandard **232** Version **C**) szabvány viszonylag nagy jelszint-távolságot ír elő a kétállapotú jelek átvitelére (nagyobb sebességek esetén már a megkövetelt slew rate is gondot okozhatna a meghajtó áramköröknek). A pont-pont jellegű kapcsolat legfontosabb előírásai a következők:

Paraméter	Specifikáció	Megjegyzés
Működés módja	pont-pont (single ended)	egy adó, egy vevő egy vonalon
Max. kábel kapacitás	2500 pF	15m kábel
Max. bitsebesség	20 kbps	-
Max. közös módusú feszültség	± 25V	-
Meghajtó kimenő feszültség	terheletlenül: ≤ ± 25V terhelten: ± 5V .. ± 15V	-
Terhelő impedancia	3 .. 7 kΩ	-
Meghajtó rövidzárási áram	≤ 100 mA	-
Vevő érzékenység	± 3V	Minimális bemenőjel szint
Zajtávolság	± 2V	VOHmin - VIHmin

EIA-RS232C specifikáció

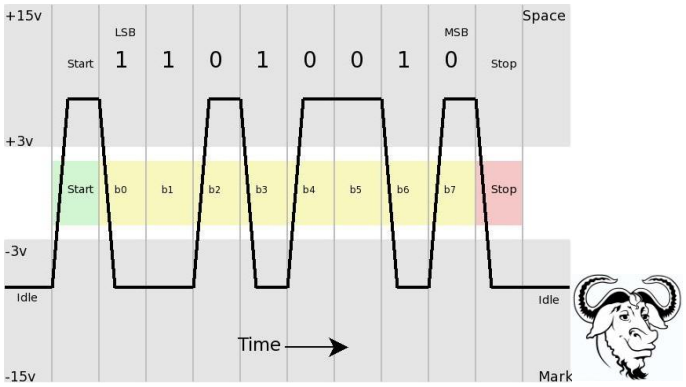


Logikai-fizikai-logikai jeltranszformáció

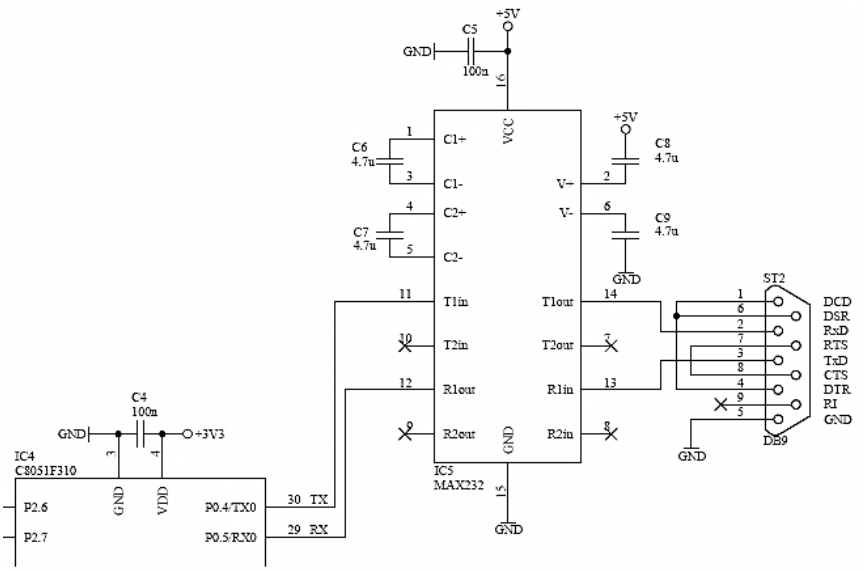
Fontos tisztáznunk, hogy **aszimmetrikus jelátvitellel** (angolul single-ended signalling) van dolgunk, az információ hordozója egy feszültség, amely az egyetlen jelvezetéken egy fix vonatkoztatási ponthoz képest változik. A megítélés szempontjából érdektelen, hogy ez a feszültség pozitív és negatív értékeket is felvehet a vonatkoztatási ponthoz képest.

Vegyük észre a következő fontos tulajdonságokat:

- Az eredeti specifikáció még max. 20 kbps (19200 Baud) sebességben gondolkozott, ma már ennél nagyobb bitsebességeket is használunk (38400, 57600, 115 200)
- A vonali jelszintek nem logikai jelszintek, a logikai „1” szint (Mark) a vonalon negatív, a logikai „0” szint (Space) a vonalon pozitív feszültségértékeknek felel meg.

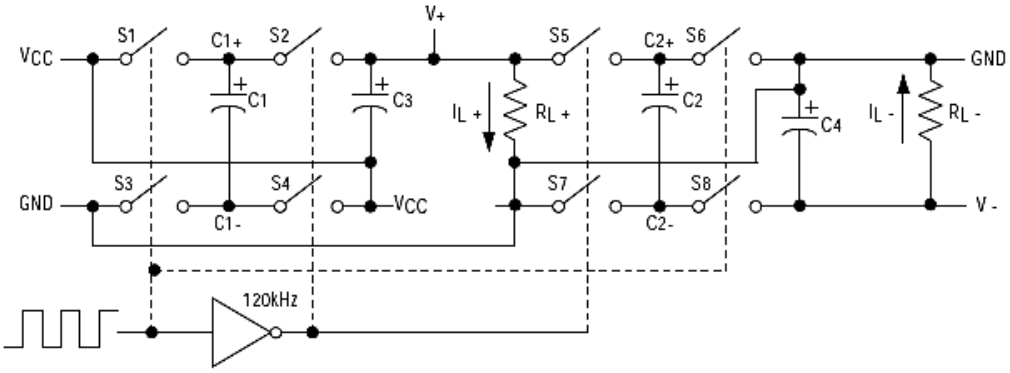


Az RS232C interfész fizikai jelszintjei



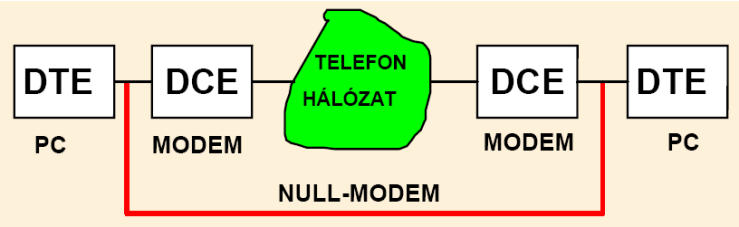
RS 232C fizikai jelszint illesztő kapcsolás

A hardver tervező szemével nézve a specifikációt azonnal az jut eszünkbe, hogy a működtetéshez negatív tápfeszültségre lesz szükségünk (-3..-15V). Ezt a problémát a gyártók már régen megoldották: a +5V tápfeszültségből feszültség kétszerező kapcsolásokkal ±10V-ot állítanak elő, melyhez segítségül mindössze 4 db kapacitást kérnek az áramkörön kívül (a kapcsolási frekvencia növelésével a kezdeti 22µF-os kondenzátorok helyett ma már 0,1µF-ot igénylő megoldásokat használunk).



Feszültségkétszerező kapcsolás az RS232 meghajtókban

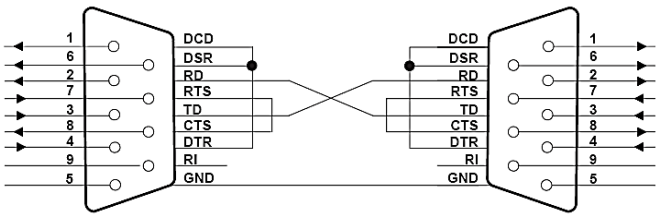
Az eredeti RS232 szabvány a nagy távolságban lévő egységek kapcsolatát még modemek és analóg telefonhálózat segítségével definiálta (innen érthető meg a definícióban szereplő 20 kbps korlát). A számítógép (DTE: **D**ata **T**erminal **E**quipment) modemen (DCE: **D**ata **C**ircuit terminating **E**quipment) keresztül kapcsolódott az egységeket összekötő telefonhálózatra. A számítógép és a modem között az adatvezetékeken túl ún. modem vezérlő jelek irányították az adatforgalmat. Két PC összekötésekor a modemek kihagyásával („null-modem") kötjük össze az egységeket, ilyenkor a modemvezérlő jelek általában nem használatosak.



Egységek kapcsolódása az RS232 szabvány szerint

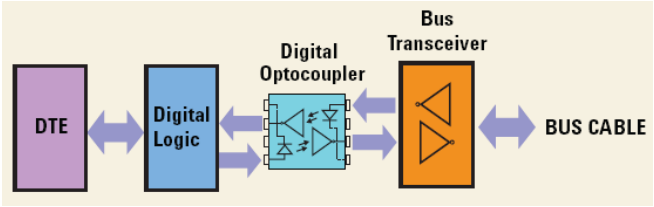
Jelnév	Sub-D9 Pin	Magyarázat
TD	3	(*Transmit Data*) UART adó kimenete
RD	2	(*Receive data*) UART vevő bemenete
GND	5	(*Ground*) Jelföld
/RTS	7	(*Request To Send*) DCE jelzi, hogy vevője kész az adatátvitelre
/CTS	8	(*Clear To Send*) DTE fogadja DCE visszajelzését, a vevő készenlétéről. Inaktív állapota blokkolja a DTE adójának a kimenetét.
/DTR	4	(*Data Terminal Ready*) DTE jelzi, hogy be van kapcsolva
/DSR	6	(*Data Set Ready*) DCE visszajelez, hogy csatlakoztatni tudta DTE-t az analóg telefonvonalra
/DCD	1	(*Data Carrier Detect*) DCE jelzi DTE-nek, hogy érzékeli a vonalon a túloldali DCE vivőfrekvenciáját
/RI	9	(*Ring Indicator*) DCE hívást (csengetés) érzékel

Jelvezetékek értelmezése az RS232 interfészen



A SUB-D9 csatlakozók jelkiosztása RS232C interfész esetén
(3 vezetékes null-modem kábel)

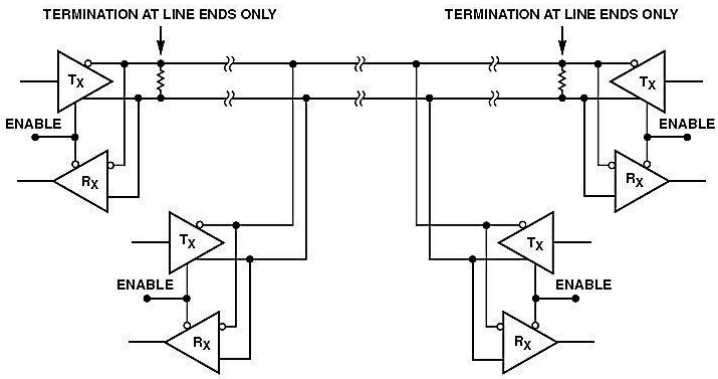
Amennyiben a vonalat galvanikusan le szeretnénk választani számítógépünk logikai részeitől, a leválasztás mindig logikai oldalon – a UART és a meghajtó között – történik.



Galvanikus leválasztás soros interfészeknél

4.7.2 EIA RS485

Az RS485-ös szabvány szemben az RS232C aszimmetrikus felépítésével egy sokkal nagyobb sebességű differenciális kapcsolatot definiál. A kapcsolat a szabvány értelmezése szerint félduplex multipont kapcsolat. Egyik legelterjedtebb alkalmazása, hogy a népszerű Profibusz kapcsolat is ezt a fizikai réteget használja félduplex üzemben (master-slave kapcsolat).

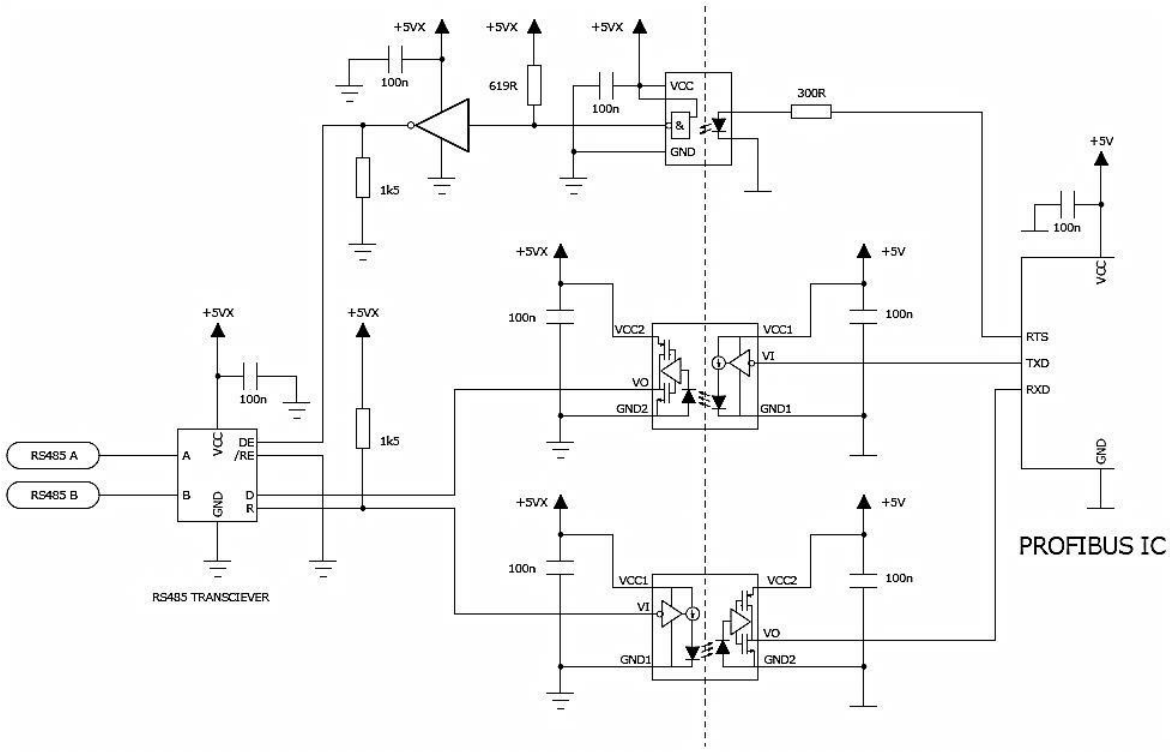


RS485 – félduplex kapcsolat

Ebben az esetben **szimmetrikus jelátvitellel** (angolul differential signalling) van dolgunk, az információ hordozója egy különbségi feszültség, amely két hasonló tulajdonságokkal rendelkező jelvezeték között mérhető. Tipikus megoldás, hogy egy sodrott érpár egyik vezetékén az információt hordozó jel, másik vezetékén annak az ellentettje (komplemense) kerül továbbításra.

	RS232	RS485
Differential	no	yes
Max number of drivers Max number of receivers	1 1	32 32
Modes of operation	half duplex full duplex	half duplex
Network topology	point-to-point	multipoint
Max distance (acc. standard)	15 m	1200 m
Max speed at 12 m Max speed at 1200 m	20 kbs (1 kbs)	35 Mbs 100 kbs
Max slew rate	30 V/µs	n/a
Receiver input resistance	3..7 kΩ	12 kΩ
Driver load impedance	3..7 kΩ	54 Ω
Receiver input sensitivity	±3 V	±200 mV
Receiver input range	±15 V	-7..12 V
Max driver output voltage	±25 V	-7..12 V
Min driver output voltage (with load)	±5 V	±1.5 V

RS232 - RS485 összehasonlító paraméterek



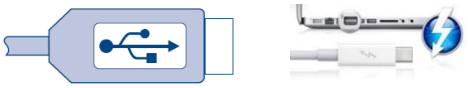
Galvanikusan leválasztott Profibus (RS-485) interfész

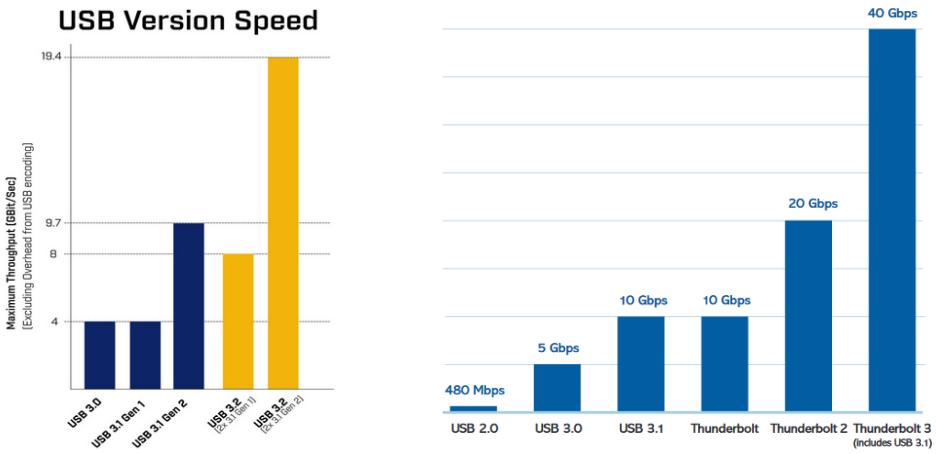
A nagysebességű (12 Mbps) átvitelhez a Profibusz rendszer saját protokollvezérlőt (SPC3: **S**iemens **P**rofibus **C**ontroller) alkalmaz. A kapcsolási vázlaton jól követhető az adó és a vevő vonalak nagysebességű, aktív optocsatolóval történő leválasztása és a differenciális adóvevő áramkör félduplex üzemmódnak megfelelő irányvezérlése (az SPC3 az /RTS modemvezérlő jelet használja erre a célra).

4.7.3 USB interfész

Az USB (Universal Serial Bus) interfészt 1995 novemberében mutatta be az Intel, Microsoft, Philips és a U.S.Robotics cég együttesen (utóbbi a megbízható modemjeiről volt híres a PC felhasználók körében). Az alapvetően egy PC és perifériái közötti adatkapcsolatra kialakított aszimmetrikus felépítésű, faszerkezetű összeköttetés lényege a **host** és az egységek (**device**) között kialakított logikai csatornákon (**pipe**) való kommunikáció. Eddig nyolc verziója látott napvilágot:

- az 1995-ben publikált **USB 1.0** verzió (1.5 majd 12 Mbps)
- az 1998-van kiadott **USB 1.1** (alapvetően az előző verzió hibáinak javítása)
- a 2000-ben megjelent **USB 2.0** (480 Mbps)
- 2008. november 17-én közzétették az **USB 3.0** szabványt, amelynek adatátviteli sebességét a szabvány 4.8 Gbps-ban adja meg. Az új technika napjainkban már lassan, de biztosan terjed.
  [Zárójelben meg kell említsük, hogy az Intel 2009-ben bemutatta az eredetileg optikai jelátvitelen alapuló (Light Peak), majd az egyszerűbb kivitelezhetőség érdekében rézvezetékre cserélt Thunderbolt (villámcsapás) névre keresztelt 4 vezetékes adatátviteli technikát, amely 10 Gbps sebességével komoly konkurense lett az USB 3.0-s kapcsolatoknak.]
- 2013. július 26-án jelentették be az **USB 3.1** szabványt, amely a (fizikai) kódolási séma megváltoztatásával kétszeresére, 10 Gbps-re növeli az adatátvitel sebességét (sajnos új csatlakozók és kábelek alkalmazásával). Viszont a szintén 2013-ban megjelent Thunderbolt 2 kapcsolat két csatorna összeolvasztásával már 20 Gbps sebességgel kommunikál, és az Intel ígérete szerint a kommunikáció sebessége hamarosan 100 Gbps lesz !
- 2017. július 25-én bejelentették az **USB 3.2** szabvány létrejöttét (20 Gbps), a nagyobb sávszélességet egy helyett két adattovábbító sáv (lane) biztosítja. A Thunderbolt 3 interfészek ekkor 40 Gbps sebességgel kommunikálnak. Az USB 3.2 változat használathoz új (USB-C) csatlakozó és kábel szükséges.



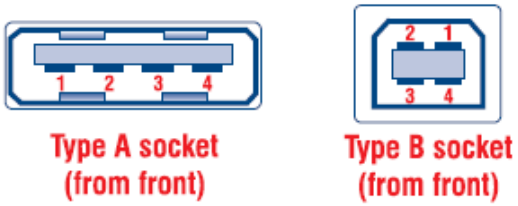


USB vs Thunderbolt

- Az **USB4** változat specifikációját 2019. augusztus 19-én adták ki, az implementálása majd két évet igényelt, így az új változatot használó első eszközök 2021-től jelentek meg (MacBook Pro M1, Orico Enclosure külső SSD tároló csatlakoztatására, stb.). A Thunderbolt 3 specifikáción alapuló, USB-C csatlakozóra és kábelre épülő interfész 40 Gbps sebességű kommunikációt tesz lehetővé.
- 2022. október 18-án megjelent az **USB4 2.0** verzió, amely újra megduplázta a sávszélességet az alapfrekvencia kismértékű növelésével, viszont az eddigi bináris adatátvitel helyett a PAM3 kódolást vezette be (ennek részleteivel jelen tantárgy keretei között nem foglalkozunk). Az adatátviteli sebesség 80 Gbit/s-ra (az egyik sávon simplex adatátvitelt alkalmazva 120 Gbit/s-ra) emelkedett.

Az interfész fontos tulajdonsága, hogy kialakításában és kezeléstechnikájában támogatja az egységek működés közben (tápfeszültség alatt) történő csatlakoztatását, ill. bontását (**hot-plug**). A csatlakozók érintkezőinek kialakítása garantálja előbb a tápfeszültség, és csak ezt követően a jelvezetékek elektromos kapcsolatát (ez ugyanis az áramkörök károsodásának egyik leggyakoribb oka feszültség alatt történő csatlakoztatás esetén). Az USB interfész a perifériák sokasága révén többféle csatlakozótípust használ, ezek közül a legismertebbek az alábbi ábrán láthatók.

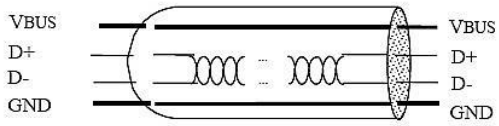
USB standard, mini és micro csatlakozók



Pin connections	
Pin No.	Signal
1	+5V Power
2	– Data
3	+ Data
4	Ground

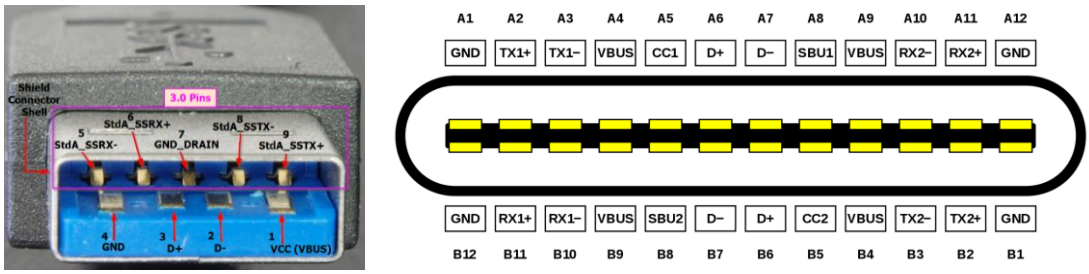
Az USB csatlakozók érintkezőinek kiosztása

Az eredeti USB interfész alapvetően 4 vezetékre épül, kettő ebből a tápvezeték (5V, 100 ill. 500mA max.), kettő pedig a differenciális adatvezeték (D+ és D-). Két szomszédos állomást (amely lehet USB hub is) max. 5 m hosszúságú vezeték köthet össze. A differenciális jelvezetéken a jelszintek 0-0,3V / 2,8-3,6V (1,5 és 12 Mbps) ill. ±400mV (480 Mbps).



Az USB interfész jelvezetékei

Az USB 3.0 szabvány kibővítette az eredeti 4 pólusú (+1 árnyékolás) csatlakozót további 5 segédpólussal: SSRX+, SSRX-, SSTX+, SSTX-, GND. Az eredeti 4 pólust a kompatibilitás miatt tartották meg, ezeken csak a HS (2.0) sebességkategóriáig lehetséges kommunikáció, az új SS (3.0) és SS+ (3.1) kommunikációhoz már külön-külön differenciális adó- és vevő érpárok tartoznak. Ezekből a legújabb 24 pólusú USB-C csatlakozóban és kábelben kettő is található (3.2, USB4 és USB4 2.0). Ennek megfelelően a 2.0 feletti szabványok szerinti egységekhez már speciális új adatkábelek is szükségesek.



Az USB 3.0 szabvány kibővített csatlakozója és az USB-C csatlakozó

A buszon folytatott kommunikáció logikája és protokollja nem egyszerű. A perifériák ún. **végpontokat** tartalmaznak, minden végpont **egy be- vagy kimeneti adatcsatorna** a host szempontjából. A végpont funkcióhoz kötődő fogalom, pl. egy webkamera két végpontot tartalmaz az egymástól független video- és audiojelek továbbítására. Egy USB készülék max. 15 bemeneti és 15 kimeneti végpontot tartalmazhat. Minden végpontot a hosttal egy-egy **logikai csatorna (pipe)** köt össze, ezek lehetnek egy- és kétirányúak is.

Az USB szabvány nyolcféle sebességkategóriát definiál, ezek:

- Low-Speed (LS): 1.5 Mbps – USB 1.0
- Full-Speed (FS): 12 Mbps – USB 1.1
- High-Speed (HS): 480 Mbps – USB 2.0
- Super-Speed (SS): 4.8 Gbps – USB 3.0 (később átnevezték USB3 Gen 1-re)
- Super-Speed+ (SS+): 10 Gbps – USB 3.1 (később átnevezték USB3 Gen 2-re)
- Super-Speed+ (SS+): 20 Gbps – USB 3.2 (később átnevezték USB3 Gen 2x2-re)
- USB4 (USB^{40}): 40 Gbps – USB 4.0 vagy USB4 (másik neve USB4 Gen 3x2)
- USB4 (USB^{80}): 80/120 Gbps – USB4 2.0 (másik neve USB4 Gen 4x2)

Az egyes sebességkategóriákhoz fontos időegységek (frame rate) tartoznak, melyek az adatátvitelnél és a lekérdezések ütemezésénél fontosak lesznek: ez LS és FS esetében 1 msec, HS, SS és SS+ sebességeknél 125 μsec (az előző 1/8-a). Az USB3 változatoktól a frame-ek szerepét az ITP-k (Isochronous Timestamp Packet) veszik át.

A kommunikáció csomagok formájában kerül lebonyolításra, az adatcsomagok négyféle típusúak lehetnek:

- az **időkritikus (isochrone) csomagok** állandó adatsebességet valósítanak meg, a teljes rendelkezésre álló sávszélességének előre egyeztetett részét (általában 90%-át) használja ez az átviteli tipus adott késleltetéssel. Az isochronous adat folytonos és valós idejű a keletkezésében, a továbbításában és a felhasználásában is (tipikus példák az audio- és videoadatok). Akár az adatátviteli sebesség kisebb a szükségesnél, akár késleltetve jut el az adat a célba, minőségromlás lép fel. Az izoszinkron átvitelnél a potenciális tranziens hibák javítása nem történik meg, azaz az elektromos adatátvitelben fellépő hibákat a hardver nem korrigálja a csomagok ismétlésével. A gyakorlatban az USB bithiba valószínűsége olyan kicsi, hogy ez általában nem okoz semmilyen nehézséget. Az ütemezés alapját a már említett időegységek (frame rate) képezik.

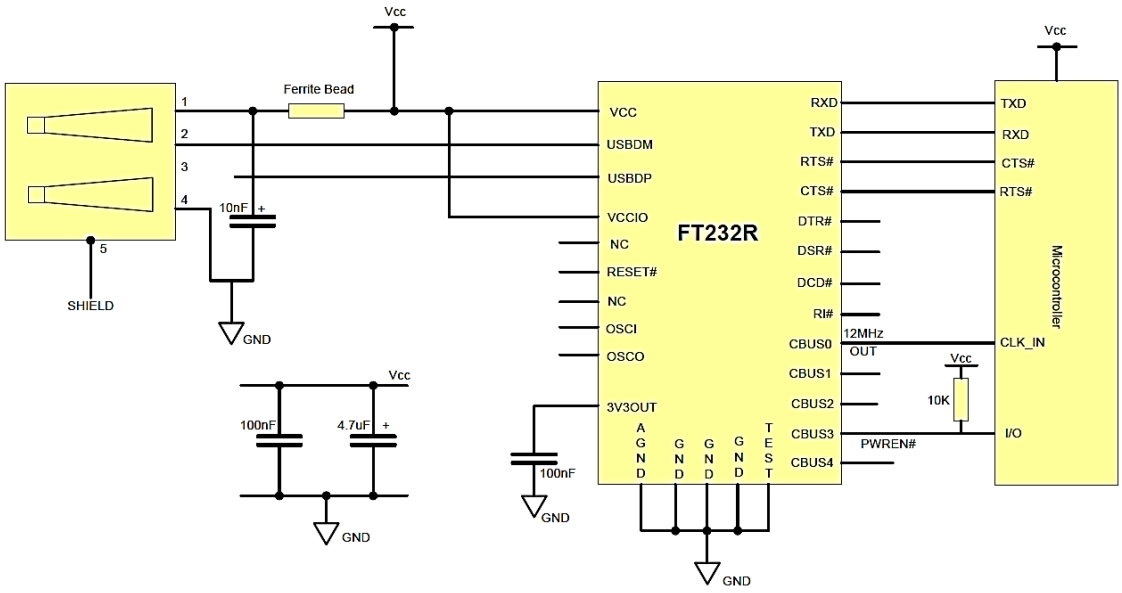
Spec.	Bitsebesség [Mbit/s]	Sáv (lane)	Frame/ μFrame [μs]	Keretidő 90%-a [μs]	Burst szám	Burst csomagok száma	Csomag mérete [Bájt]	Átvitt bájtok száma per frame	Kódolás	Átvitt bitek száma per frame	Időszükséglet overhead és FEC nélkül [μs]
USB 1.0	1,5	1	1000	900	1	5	32	160	-	1 280	853,3
USB 1.1	12	1	1000	900	1	5	256	1 280	-	10 240	853,3
USB 2.0	480	1	125	112,5	1	6	1 024	6 144	-	49 152	102,4
USB 3.0	5 000	1	125	112,5	3	16	1 024	49 152	8b/10b	491 520	98,3
USB 3.1	10 000	1	125	112,5	6	16	1 024	98 304	128b/132b	811 008	81,1
USB 3.2	10 000	2	125	112,5	6	16	1 024	196 608	128b/132b	1 622 016	81,1
USB4	20 000	2	125	112,5	6	32	1 024	393 216	128b/132b	3 244 032	81,1
USB4 2.0	25 600	2	125	112,5	6	64	1 024	786 432	PAM3+FEC	6 291 465	77,5

A hasznos adatsebesség becsléséhez figyelembe kell vegyük, hogy a keretek az adatokon kívül járulékos karaktereket (START, STOP, CRC) is tartalmaznak, valamint az USB 3.0-nál már adatkódolást (3.0: 8b/10b, a 3.1-től 128b/132b) is alkalmaznak, így a hasznos adatsebességnél csak 90-95%-kal számolhatunk. Az USB4 2.0 változat már PAM3 modulációt és hibajavító kódot (FEC – Forward Error Correction) alkalmaz, ami még bonyolultabbá teszi a bit-bájt átváltási arányt és az időegység alatt átvitt hasznos adatmennyiség számítását.

- a **terjedelmes (bulk) adatcsomagok** nem időkritikus, nagy adattömegek továbbítására alkalmas alacsony prioritású egységek. Akkor kerülnek továbbításra, ha az időkritikus és a megszakítási csomagok már kiszolgálásra kerültek (scanner, printer stb. adatok)
- a **megszakítási (interrupt) csomagok** aszinkron időpontban keletkező kis adattartalmú csomagok melyek lekérdezése megadott ciklikus időközökben (a korábban ismertetett időegységenként) történik, ezekkel kérhetnek az egységek a hosttól kiszolgálást
- a **vezérlési (control) csomagok** pl. az elemek azonosításához, címkiosztásához szükségesek, kölcsönösen nyugtázott oda-vissza adatforgalmat valósítanak meg.

Idő hiányában a protokoll részleteivel nem tudunk foglalkozni, erre a mesterképzés megfelelő tantárgyai térnek majd ki részletesebben.

Beágyazott rendszerünknek egy host számítógéphez USB interfészen való kapcsolásához kész céláramkörök állnak rendelkezésünkre, melyek **egyik oldalukon** a szabványos USB felületet állítják elő, önmagukban teljes mértékben lekezelik az USB protokollt, **másik oldalukon** pedig egy párhuzamos vagy egy soros (aszinkron vagy SPI) felületet mutatnak a felhasználó felé, melyen keresztül az áramkört egyszerűen rendszerünkhöz illeszthetjük. A soros felület még aszinkron kapcsolat esetében is – lévén az adattovábbítás logikai jelszinteken zajlik – nagy sebességet tesz lehetővé, 110 bps - 3 Mbps lehet.

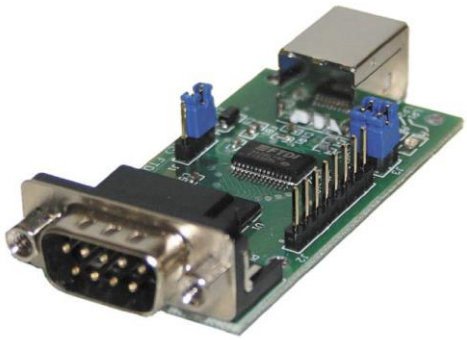


FTDI chip aszinkron soros illesztése mikrokontrollerhez

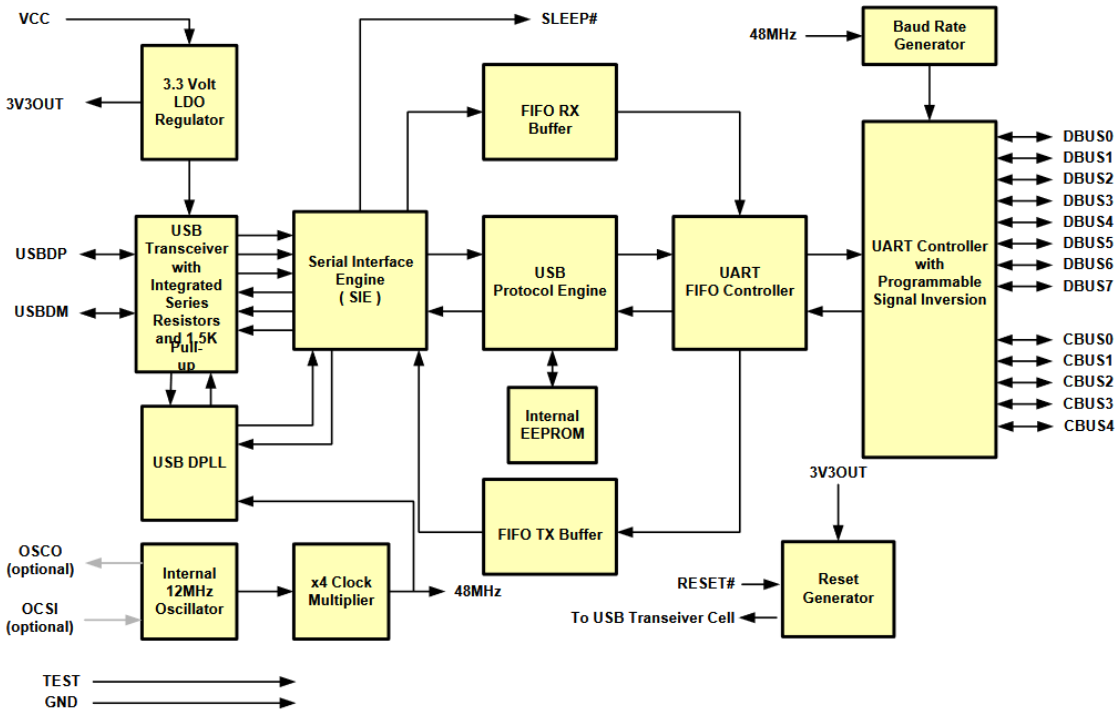
A **F**uture **T**echnology **D**evices **I**nternational Ltd. által gyártott eszközcsalád (rövid nevükön az FTDI-chipek) a tulajdonságok és illesztések széles skáláját kínálva rendkívül egyszerű USB interfész kialakítását teszik lehetővé egy-egy kisebb beágyazott rendszer számára. Mind a mikrokontroller, mind a host PC „láthatja” a kommunikációs felületet (virtuális) soros portként vagy párhuzamos felületen, a két oldal között csak a

kommunikációs sebességet kell összehangoljuk, mint egy aszinkron soros átvitel esetében, a protokoll lebonyolítását a PC driver és az FTDI chip már elvégzik egymással.

USB-UART konverterek FTDI chippel



Az EVAL232R fejlesztő modul



Az FT233R chip belső blokvázlata

Mikrokontroller alapú rendszerek

Elektronikus jegyzet

5. fejezet

Készítette: Dr. Tevesz Gábor c. egyetemi tanár
BME Automatizálási és Alkalmazott Informatikai Tanszék
1117. Budapest, Magyar tudósok körútja 2.
Q ép. B szárny II. em. B216.
Tel: 463-2881
Fax: 463-2871 (adm.)
Mail: tevesz@aut.bme.hu

Hallgatják: Villamosmérnöki és Informatikai Kar
Nappali tagozat
Villamosmérnöki alapszak (BSc)
III. évfolyam, 5. félév
Beágyazott és irányító rendszerek specializáció hallgatói

COPYRIGHT

TARTALOMJEGYZÉK

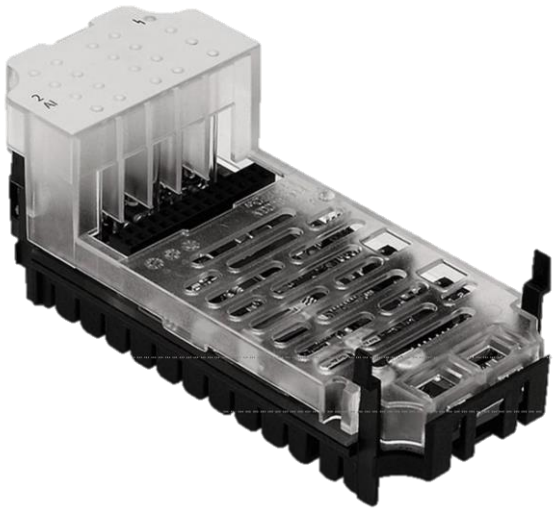
5. BEÁGYAZOTT RENDSZEREK TERVEZÉSÉNEK ALAPELVEI ÉS LÉPÉSEI

Ebben a fejezetben azt vizsgáljuk, hogy az eddig megismert eszközök és alapelvek segítségével hogyan történik egy-egy konkrét beágyazott rendszer megtervezése, milyen lépések vezetnek a feladat megfogalmazásától a tényleges rendszer kialakításáig. Először egy egyszerűnek tűnő feladat megvalósítását kísérjük végig egy esettanulmány keretei között, ezt a hardver tervezés általános szabályainak vizsgálatával folytatjuk, majd az áramkör élesztés, programozás és tesztelés eszközeinek áttekintésével zárjuk a lépések sorát.

5.1 Esettanulmány: hőmérséklet mérése mikrokontrollerrel

Ebben a fejezetben egy ipari irányítórendszer hőmérséklet szenzort illesztő moduljának tervezési lépéseivel ismerkedünk meg.

5.1.1 A feladat megfogalmazása

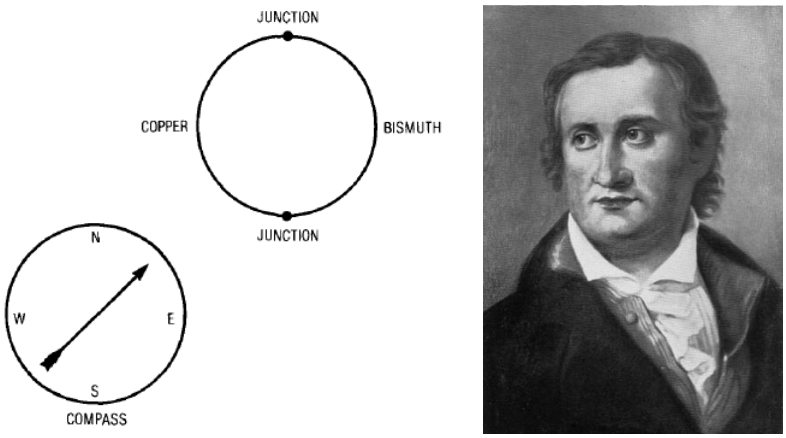


A Festo cég előző esettanulmányunkban már megismert CPX folyamatirányító rendszeréből régóta hiányzott egy olyan, magasabb hőmérsékletek mérésére is alkalmas intelligens modul, amely a hőmérséklet mérését hőelemek segítségével teszi lehetővé. A hőelemek, mint ismeretes, igen széles hőmérséklettartományok (kb. -200°C ..+1800°C) mérését teszik lehetővé meglehetősen igénytelen szenzor-kialakítással, relatíve kis termikus időállandók biztosításával. A feladat tehát egy intelligens modul kifejlesztése volt, amely 4 független helyen történő hőmérséklet mérését teszi lehetővé különböző hőelem-típusok segítségével, csatornánként legalább 4 mérés/másodperc ciklusidőt biztosítva. A szabadon (csatornánként) konfigurálható hőelem-típusok B, E, J, K, N, R, S és T, az adott típusra a szabvány (IEC 60584) által előírt méréstartományokkal. A modul valamennyi csatornán szenzorhiba (szakadás, rövidzár), paraméterezhető alsó és felső határérték figyelést kell megvalósítson. A modulnak természetesen illeszkednie kell a CPX rendszer moduljainak sorába, tehát a CBUS-ra illesztve standard intelligens I/O modulként kell megjelennie a rendszer moduljainak sorában.

5.1.2 A mérési elv

Bár a mérés elve más tantárgyakból már ismert a hallgatóság számára, a továbbiak jobb érthetősége érdekében ismételjük át röviden a hőelemmel történő hőmérsékletmérés alapelveit!

Thomas Johann Seebeck (1770-1831) Észtországban született német orvos-fizikus 1821-ben fedezte fel a termoelektromos (hőelem-) effektust. Seebeck észrevette, hogy egy olyan zárt áramkörben, melynek részei különböző fémekből tevődnek össze, áram indul meg, ha a fémek érintkezési pontjai különböző hőmérsékletűek. A felfedezés alapja az volt, hogy két különböző anyagú (réz és bizmut) fémszalagból kialakított vezető hurok csatlakozási pontjainak melegítésekor a hurok közelében elhelyezett iránytű kitért a szokásos É-D irányból az áramhurok által létrehozott mágneses tér hatására.

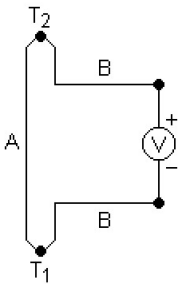


Seebeck kísérlete az áramhurokkal és az iránytűvel

A termoelektromos effektus révén keletkező feszültség függ a fémek anyagától és a végpontok hőmérsékletkülönbségétől.

A Seebeck-feszültség keletkezése

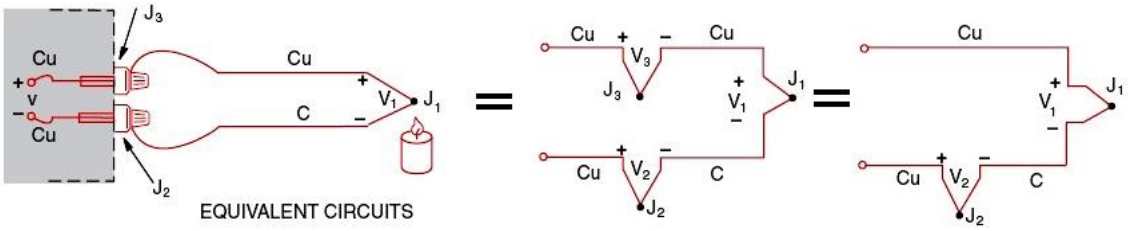
Az ún. termoelektromos (vagy Seebeck-) feszültség definíciója a következő: legyenek A és B különböző fémek, melyek csatlakozási pontjain a hőmérséklet T_1 és T_2, ekkor



$$V = S_{BA} * (T_2 - T_1)$$

A termoelektromos (Seebeck-) feszültség definíciója

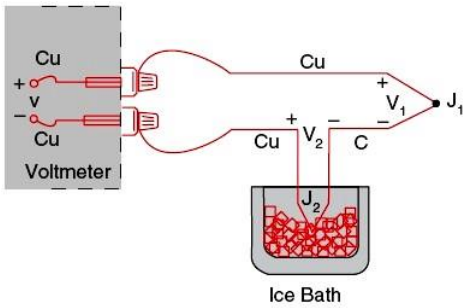
ahol $S_{BA}=S_B-S_A$ a fémek ún. termoelektromos, vagy Seebeck-együtthatói. Ez a feszültség azonban nem mérhető meg ilyen egyszerű módon, mivel a fémeknek a mérőműszerhez való csatlakoztatásával újabb hőelemeket hozunk létre (J_2 és J_3), és ezzel „meghamisítjuk" a mérni kívánt feszültséget.



A termoelektromos feszültség megmérése nem is olyan egyszerű

$$V = S_{CuC} * (T_1 - T_2)$$

Hozzunk létre egy, az eredeti definíció szerinti áramkört! Egyelőre az egyik alkalmazott fém egyezzen meg a műszer belsejében található fém típusával, ily módon a csatlakozási ponton nem keletkezik további hőelem.



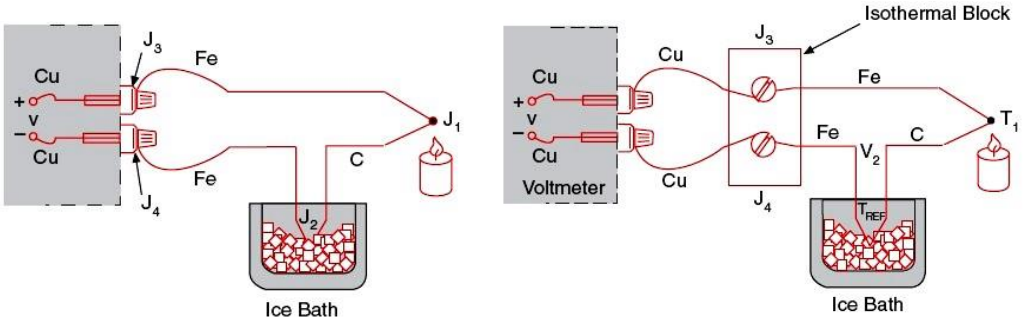
Így mérhetjük meg a termoelektromos feszültséget

Vegyük észre, hogy mindkét elrendezésben pontosan az eredeti definíciónak megfelelő áramkört alakítottunk ki, ahol $T_2 = 0°C$, ily módon

$$V = S_{CuC} * (T_1 - 0°C) = S_{CuC} * T_1$$

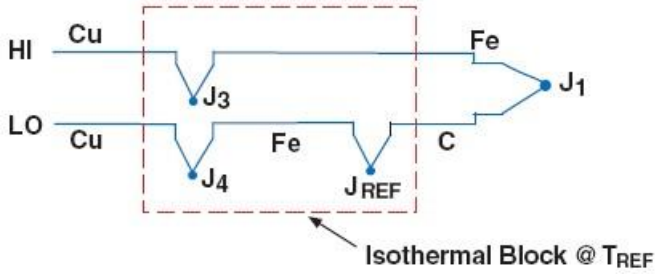
Egy adott pont abszolút hőmérsékletének hőelemmel történő megméréséhez tehát egy referenciahőmérsékletre van szükségünk, ami a fenti esetben az olvadó jég hőmérsékletének megfelelően 0°C volt. A referenciapont hőmérsékletét a továbbiakban **hidegpontnak**, a mérési eljárást pedig **hidegpont-kompenzációnak** nevezzük.

A kialakítás szempontjából azonban közömbös, hogy az alkalmazott fémek egyeznek-e a műszer belsejében található fémmel (réz), amennyiben feltételezhetjük, hogy a fémek találkozási pontjai azonos hőmérsékletűek. Ekkor ugyanis a J_3 és J_4 csatlakozási pontokon keletkező azonos nagyságú, de ellentétes értelmű feszültségek kiejtik egymást.



Az azonos hőmérsékletű csatlakozási pontok feszültségei kiejtik egymást

Innen már csak egy lépés szükséges a kellemetlen „jégfürdő” elhagyásához: tételezzük fel ugyanis, hogy azonos hőmérsékletre hozzuk J_2, J_3 és J_4 csatlakozási pontjainkat!

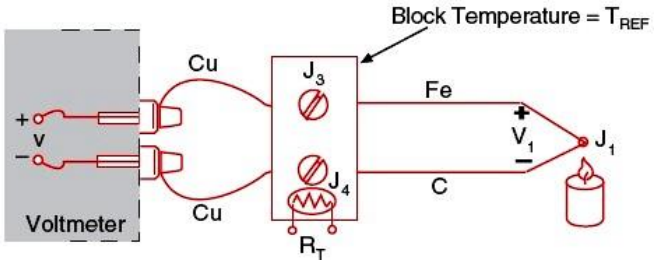


A hidegpont és a csatlakozási pont összeolvasztása egyetlen blokká

Ekkor értelemszerűen a termofeszültség

$$V = S_{FeC} * (T_1 - T_{REF})$$

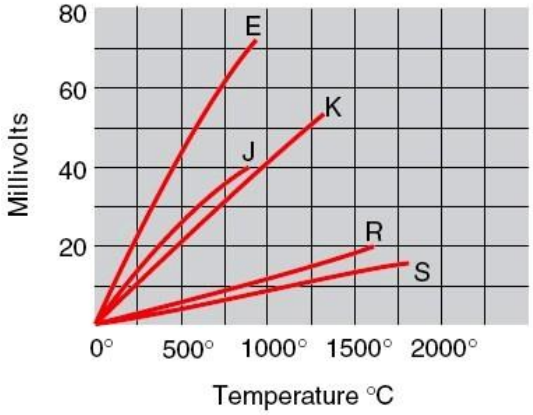
hiszen a J_3 és J_4 pontokon keletkező feszültségek továbbra is kiejtik egymást. Azt is vegyük észre, hogy az alsó ágban lévő vas vezeték (J_4 és J_{REF} között) elhagyható, hiszen a vezeték mindkét végén azonos a hőmérséklet, így rajta termoelektromos feszültség nem keletkezhet. Ez viszont azt jelenti, hogy a fenti összefüggés érvényben marad az alábbi egyszerűsített elrendezés esetén is:



$$V = S_{FeC} * (T_1 - T_{REF})$$

Hőmérsékletmérés a csatlakozási pont hőmérsékletének mérésével

A hosszú logikai levezetés után tehát összefoglalhatjuk az eredményt: a hőelem sarkait azonos hőmérsékletű pontokon csatlakoztatjuk a mérőműszerhez. Ennek a pontnak meg kell mérnünk az abszolút hőmérsékletét (pl. ellenállás-hőmérővel), melynek segítségével kompenzálnunk kell a mért feszültségérték alapján kapott eredményt. Ennél a mérési elrendezésnél a csatlakoztatási pont tölti be a hidegpont szerepét.



Type	Metals +	−
E	Chromel vs. Constantan	
J	Iron vs. Constantan	
K	Chromel vs. Alumel	
R	Platinum vs. Platinum 13% Rhodium	
S	Platinum vs. Platinum 10% Rhodium	
T	Copper vs. Constantan	

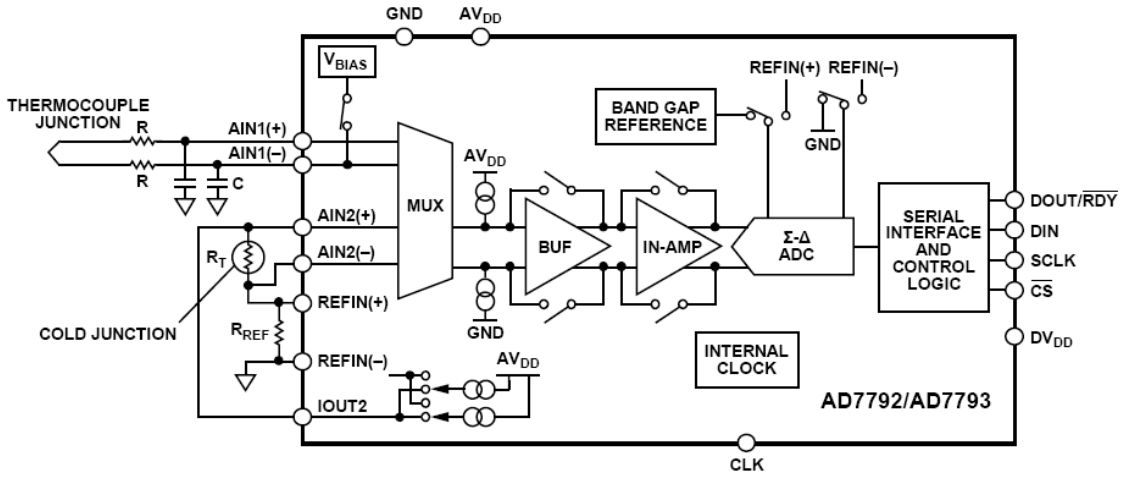
Különböző hőelem-típusok jellegzetes karakterisztikái

A gyakorlatban különböző hőelem-típusokat alkalmaznak. Minél meredekebb az érzékelő karakterisztikája, annál kisebb problémát jelent az igen alacsony termofeszültség nagypontosságú mérése. Sajnos a legnagyobb hőmérséklethatárok között pont a „leglaposabb” karakterisztikájú hőelemek alkalmazhatók.

5.1.3 A mérés menete

A hőelemmel való hőmérséklet meghatározás feladata tehát egyrészről egy kis értékű (néhány, vagy néhány tíz millivoltos) feszültség nagypontosságú megmérése és a hidegpont hőmérsékletének hasonló pontossággal történő meghatározása. Lévén a hidegpont hőmérséklete általában csak szűk tartományban változik, itt leginkább a pontos hőmérséklet-meghatározás a feladat, amit például ellenállás-hőmérő segítségével tudunk megmérni.

A feladathoz az áramkörgyártók a feladathoz legjobban illeszkedő céláramköröket fejlesztettek ki. Egy ilyen áramkör az Analog Devices AD7792 típusú áramköre.



Hőelemes mérésekhez alkalmazható átalakító kapcsolás (Analog Devices)

Az áramkör a **hőelem feszültségének** méréséhez belső programozható erősítőt (in-amp), az alacsony feszültségszint miatt el nem hanyagolható nagyságú zajfeszültségeket jól kiszűrő Σ-Δ elvű analóg-digitális átalakítót és ehhez pontos belső referenciafeszültséget (band-gap reference) tartalmaz. Előállítja az A/D átalakításhoz szükséges órajelet és SPI interfészen keresztül kommunikál a felügyelő mikrokontrollerrel. A digitalizált feszültségérték (0x8000 eltolással):

$$X_{TC} = \left(\frac{U_{TC}}{U_{ref}} * G + 1 \right) * 2^{15}$$

ahol (G a programozható erősítő erősítése). A V_{BIAS} segédfeszültséggel a bemenő jeltartomány közepére emelt hőelem feszültség ily módon pozitív és negatív feszültségek kiértékelésére egyaránt alkalmas.

Fentieken túl egy igen figyelemreméltó kapcsolást tartalmaz az ellenállás-hőmérővel történő hőmérsékletmérés támogatására. Az I_{OUT2} kimenethez konstans áramerősséget szolgáltató áramgenerátor csatlakoztatható, melyet átfolyatva az ellenállás-hőmérőn, annak ellenállás-változását lineáris összefüggés szerint feszültségváltozássá alakítjuk, amit a bemeneti multiplexer átkapcsolásával ismét az előbbi nagypontosságú A/D-átalakító segítségével tudunk digitalizálni. A mért feszültség változása

$$U_{RTD} = R_T \cdot I = R_0 (1 + \alpha \cdot t) \cdot I$$

összefüggés szerint azonban α, R_0 és I lineáris függvénye is. Ebből az ellenállás-hőmérő hőmérsékleti együtthatója és névleges értéke nagy pontossággal ismert, viszont hasonló (abszolút) pontosságú áramgenerátort nehéz lenne előállítani. Ezért ezt az áramot nem

csak a mérendő ellenálláson, hanem a nagy pontosságú R_{REF} ellenálláson is átfolyatjuk, és az ezen az ellenálláson eső feszültséget használjuk referencia feszültségként az A/D átalakításhoz. Az ellenállás-hőmérő sarkain mért digitalizált feszültségérték

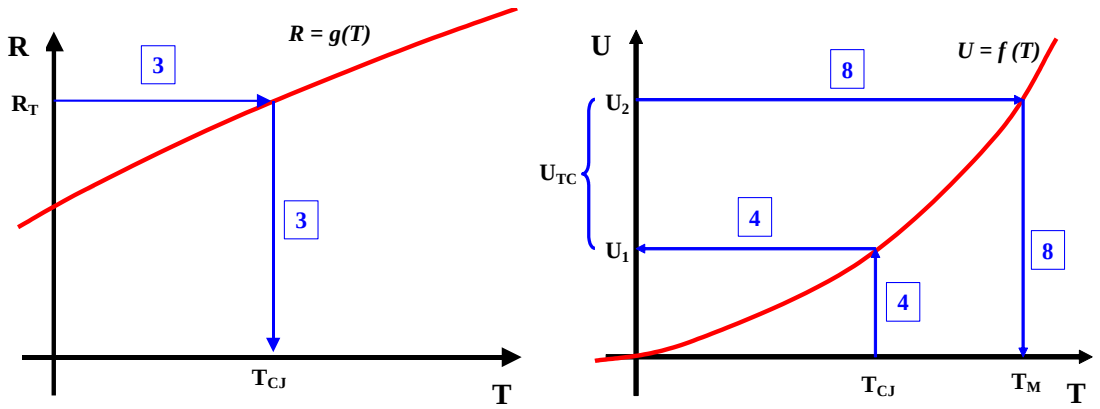
$$X_{RTD} = \frac{U_{RTD}}{U_{REF}} * G * 2^{16}$$

(G itt is a programozható erősítő erősítése), amibe az $U_{REF} = I*R_{REF}$ és az $U_{RTD} = I*R_T$ összefüggéseket behelyettesítve

$$R_T = \frac{X_{RTD}}{G \cdot 2^{16}} * R_{REF} = R_0 \cdot (1 + \alpha \cdot t)$$

Láthatóan a digitalizált feszültségérték és ily módon a hidegpont hőmérséklete független lesz az áramerősség tényleges nagyságától, azaz abszolút pontossága nem szükséges a méréshez. A mért ellenállásértékhez tartozó hőmérséklet alapján azonban még kompenzálnunk kell a hőelem sarkain mért feszültséget a következő algoritmus szerint:

1) Megmérjük az R_T ellenállás értékét: X_{RTD}
2) Kiszámítjuk ebből az ellenállás-hőmérő abszolút ellenállását: $R_T = R_{REF} * X_{RTD} / 2^{16}$
3) R_T értékéből meghatározzuk a hidegpont hőmérsékletét: $T_{CJ} = g^{-1}(R_T)$
4) Kiszámítjuk a kompenzációs hőmérséklethez tartozó feszültséget: $U_1 = f(T_{CJ})$
5) Megmérjük a hőelem sarkairól érkező feszültség digitalizált értékét: X_{TC}
6) Kiszámítjuk az ehhez tartozó feszültségértéket: $U_{TC} = U_{ref} * (X_{TC} / 2^{15} - 1) / G$
7) Kiszámítjuk a kompenzált feszültségértéket: $U_2 = U_1 + U_{TC}$
8) U_2 alapján meghatározzuk a mérendő hőmérsékletet: $T_M = f^{-1}(U_2)$



R=g(T) és U=f(T) karakterisztikák a hőmérséklet számításához

5.1.4 A szenzorkarakterisztikák közelítése

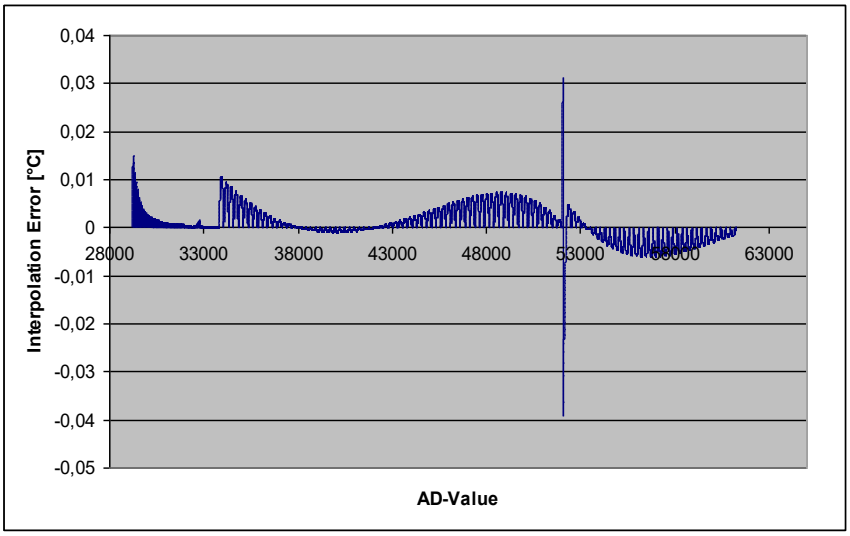
A fenti két karakterisztika a hőmérséklet nemlineáris függvényei. Pontosabb mérésekhez mind az ellenállás-hőmérő, mind a hőelem karakterisztikáját polinóm egyenletekkel közelítik, R_T esetében harmadfokú, U_{TC} esetében kilencedfokú (!) polinómokkal kell számolnunk a szabványok által megadott pontosságok eléréséhez. Ilyen bonyolultságú műveletek megfelelő pontossággal történő elvégzése nagypontosságú lebegőpontos műveletvégzések sorozatát követelné meg, ezért a kisebb mikrokontrollerek általában adott felbontású táblázatokat használnak a függvényértékek leírására, a pontok közötti értékeket pedig lineáris interpoláció segítségével közelítik. Az egyes hőelem típusok jellemzőit a következő táblázat foglalja össze:

TC Type	T_{min} [°C]	T_{max} [°C]	U_{min} [mV]	U_{max} [mV]	Gain	FS [+/- mV]	1 LSB [uV]	A/D max
T	-200	400	-5,603	20,872	32	36,563	1,116	**0xC90E**
S	0	1768	0,000	18,693	32	36,563	1,116	**0xC16E**
R	0	1768	0,000	21,101	32	36,563	1,116	**0xC9DB**
N	-200	1300	-3,990	47,513	16	73,125	2,232	**0xD327**
K	-200	1370	-5,891	54,819	16	73,125	2,232	**0xDFF0**
J	-200	1200	-7,890	69,553	16	73,125	2,232	**0xCB20**
E	-200	900	-8,825	68,787	16	73,125	2,232	**0xF9B9**
B	400	1820	0,787	13,820	64	18,281	0,558	**0xE0BF**

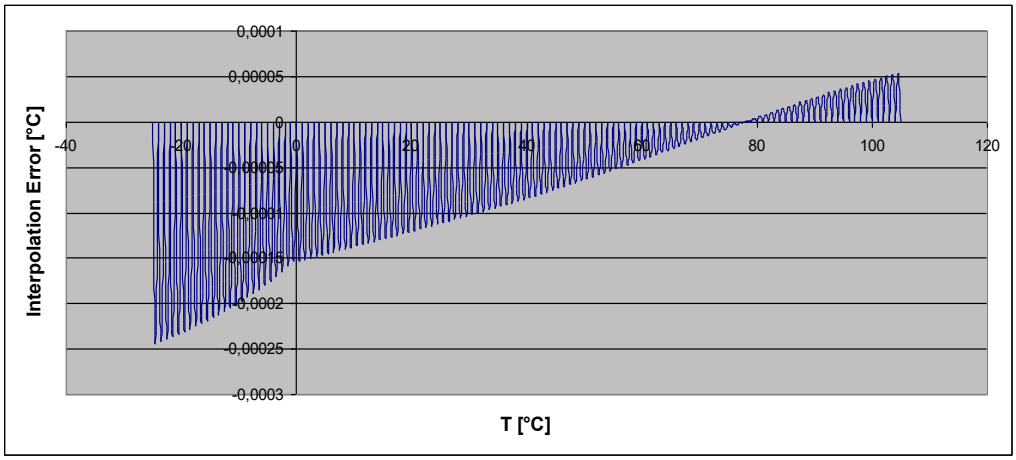
Az alkalmazott hőelemek jellemző paraméterei

A táblázatok tartalma előzetesen nagy pontossággal számítható, a kérdés csupán a tárolt pontok távolsága, ami a közöttük történő lineáris interpoláció maximális hibáját szabja meg. Minél sűrűbben tároljuk a pontokat, annál pontosabban tudunk számolni, viszont annál nagyobb táblázatméretekkel kell számolnunk. Összességében háromféle táblázattípust kell tárolnunk:

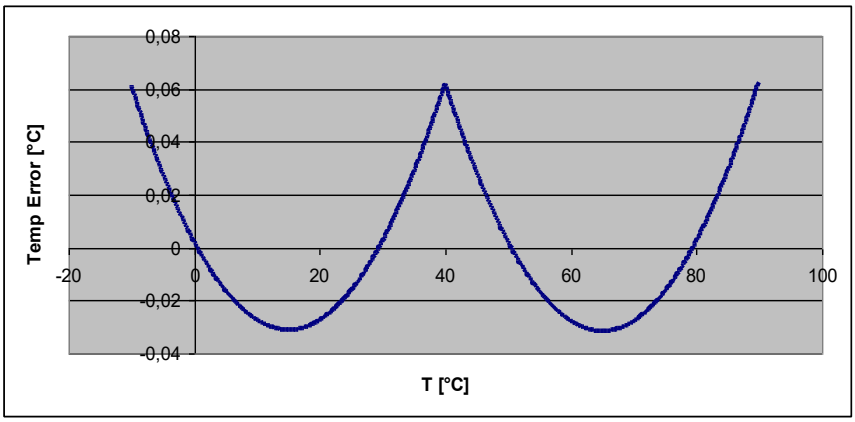
- A „normál” hőelem táblázatra [4] csak a hidegpont hőmérsékletének feszültséggé való visszakonvertálásához lesz szükségünk. Mivel a hidegpont fizikailag a modul elején található csatlakozóban helyezkedik el, hőmérséklete csak egy szűkebb hőmérséklettartományban mozoghat, így a –25..+105°C tartománynak megfelelően 1°C-os lépcsőkben tároljuk a feszültségnek megfelelő A/D értékeket (2 bájt).
- Az „inverz” hőelem táblázatra [8] a teljes tartományban szükségünk lesz. A számítások szerint itt egyetlen felbontással nem fedhető le a tartomány, mivel a karakterisztikák alsó határai felé közeledve durván eltérnek a lineáristól. Ezért célszerű egy közbenső (T_{mid}) hőmérséklet és két különböző lépésköz bevezetése. A tárolt adatok nagypontosságú hőmérsékletértékek (1/10000°C pontosság, 4 bájt).
- Az ellenállás-hőmérő karakterisztikájára [3] szintén csak a hidegpont hőmérséklet-tartományában lesz szükségünk. Ez a lényegesen lineárisabb összefüggés két egyenessel már igen jól közelíthető az adott tartományban.



Hőelem inverz karakterisztikájának hibaszámítása (AD érték $\Rightarrow T$: $T_M=f^{-1}(U_2)$)



Hőelem normál karakterisztikájának hibaszámítása ($T \Rightarrow$ AD érték : $U_1=f(T_{CJ})$)



Ellenállás-hőmérő karakterisztikájának hibaszámítása ($T_{CJ}=g^{-1}(R_T)$)

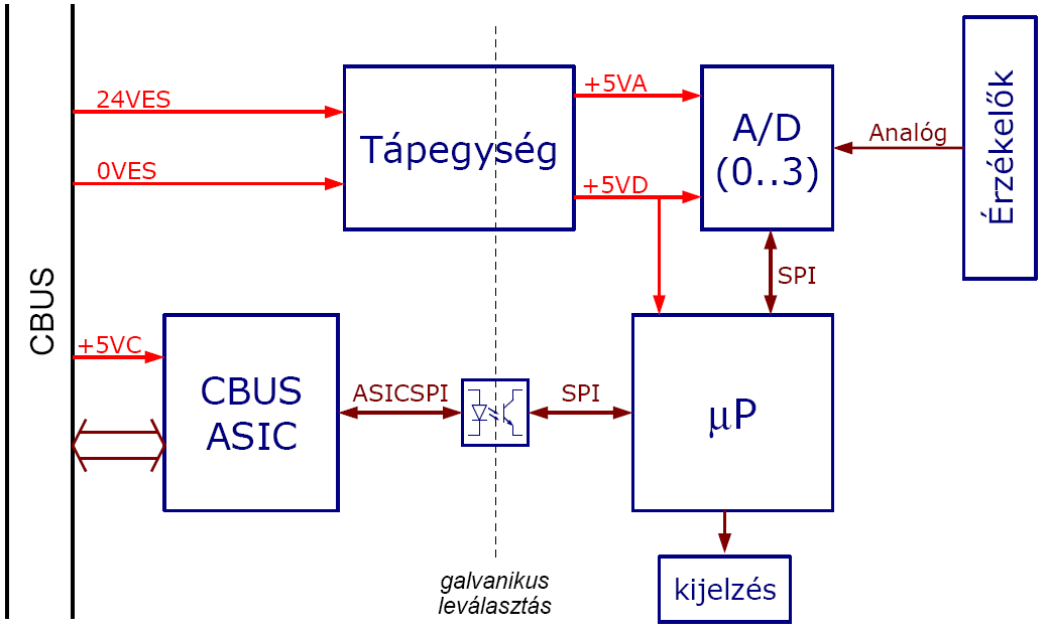
Érdekességként bemutatjuk a számítások eredményeként kiadódó táblázatok méreteit (a teljes táblázatrendszer mérete 9800 bájtra adódik):

Szenzor típus	Tmin [°C]	Tmid [°C]	Tmax [°C]	Δx1 [LSB]	Δx2 [LSB]	Max hiba [°C]	Adat-pont	Hossz [bájt]
T	-222	-55	422	64	256	0,026796	141	576
S	-21	387	1791	32	64	0,023872	313	1264
R	-22	288	1795	32	128	-0,031910	202	820
N	-221	63	1321	16	128	0,017997	329	1328
K	-223	125	1405	32	256	0,026963	249	1008
J	-220	44	1222	32	256	-0,039338	269	1088
E	-220	-40	919	64	256	0,033823	177	720
B	382	970	1848	64	256	0,024303	176	716
Összesen (1 adatpont = 4 bájt – long érték 1/10000°C-ban):							**1856**	**7520**

Az hőelem karakterisztikák inverz táblázatainak számított méretei

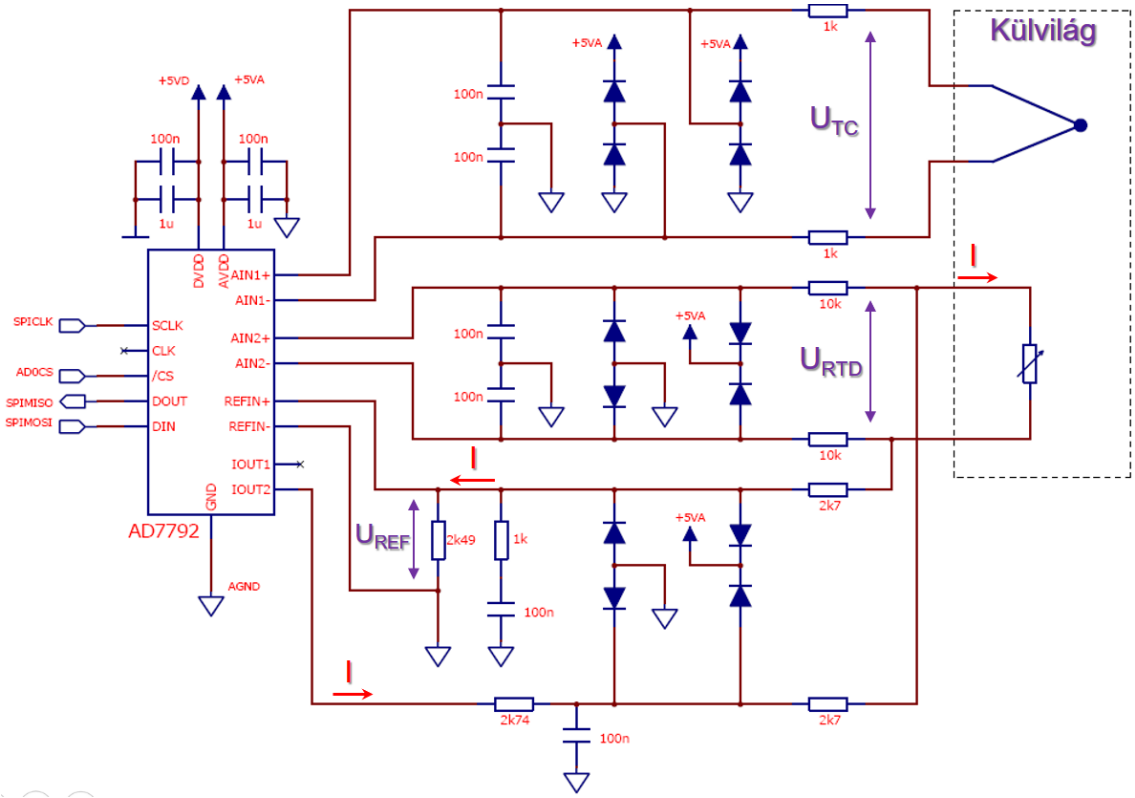
5.1.5 A mérési kapcsolás

Először a modul blokkvázlatát készítjük el. A modult a belső buszrendszerre – a rendszer felépítéséből adódóan – ismét a soros buszrendszer speciális protokolljának kezelését megvalósító céláramkör (CBUS ASIC) illeszti. A CPX rendszerben – mint a legtöbb folyamatirányító rendszer esetében – előírás, hogy a belső buszrendszer nem lehet galvanikus kapcsolatban a külvilágból érkező jelvezetékekkel (EMC szempontok miatt). Fontos kérdés, hogy ez az elválasztás hol valósuljon meg.



A hardver kapcsolás blokkvázlata

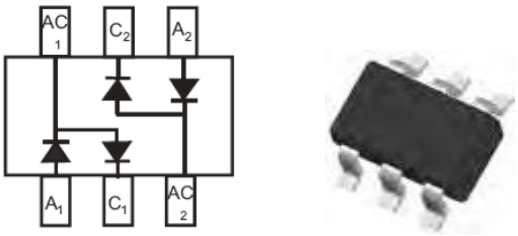
Az ASIC soros SPI felületen kapcsolódik a mikrokontrollerhez, hasonlóan a 4 hőelem illesztését megvalósító 4 db A/D átalakítóhoz. A CPX rendszer adottsága, hogy saját buszkezelő áramkörének tápellátását képes ellátni (CBUS5V), azonban ez a tápfeszültség érthető okokból (max. 48 modul lehet egy buszrendszeren) nem terhelhető további egységekkel. Az analóg bemenetek szintjén – a néhány millivoltos hőelem-feszültségek és a 200µA RTD-áramok mellett – a leválasztás teljesen illuzórikus. Reális műszaki megoldásként a leválasztás vagy az A/D átalakítók és a mikrokontroller, vagy az ASIC és a mikrokontroller közé javasolható (digitális jelek leválasztása a tanult módszerekkel). Az A/D-k leválasztása esetén 2 db 5V-os tápfeszültségre lenne szükségünk (µC és környezete ill. a leválasztott A/D-k számára), és min. 7 db digitális jelet kell leválasztanunk (SCK, SDI, SDO + 4 db /CS). Amennyiben a mikrokontroller és az ASIC közé tesszük a leválasztást, csak egy tápfeszültségre lesz szükségünk, és a leválasztandó vonalak száma 4+1 (SPI + 1 ASIC státuszvonal). A költségek minimalizálása érdekében az utóbbi megoldás mellett döntünk.



Az A/D átalakító és a szenzorokat fogadó kapcsolás

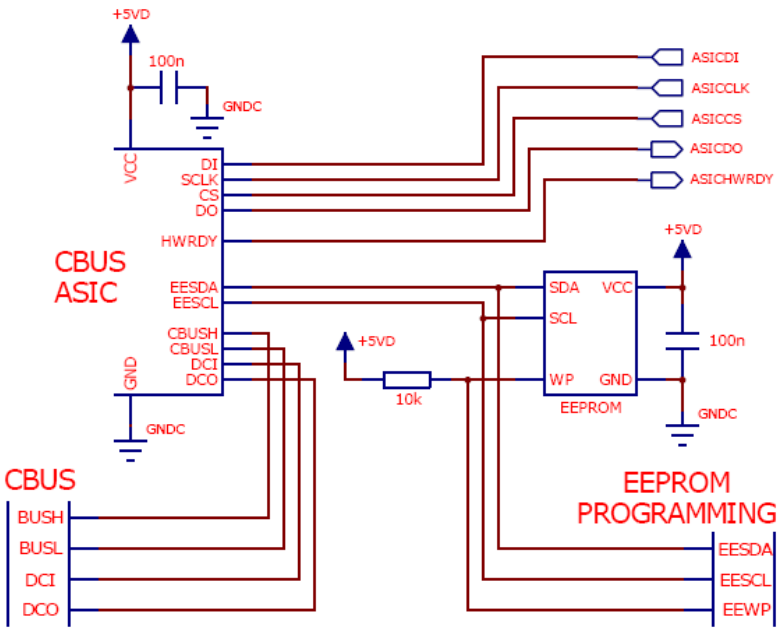
Az analóg bemeneti rész megoldásai következnek a választott IC felépítéséből és a mérés elvéből. Alapszabály, hogy a külvilágból érkező jelek mindegyikére megfelelő EMC védelmet kell biztosítanunk. Analóg jelek esetében erre egy bevált megoldás a min. 1 kΩ-os leválasztó ellenállást követő szűrőkondenzátor – vágódióda kombináció, ami egyrészről zajszűrést is megvalósít, másrészről a korlátozott túlfeszültségnek kitett analóg vonalakon érkező zaj energiáját a tápfeszültségek (+5V, GND) irányába vezeti el. A problémát általában az R-C időállandó megválasztása (ne lassítsa a bemenetet) és a

vágódiódák max. szivárgási árama okozhatja, ami meghamisíthatja a bemeneti feszültségértékeket. Szerencsénkre az előbbi feltétel nem okoz gondot (a termikus időállandók miatt a bemeneti jelváltozások lassúak, nincs szükségünk 1-2 kHz feletti sávszélességre), a második szempont megoldására pedig található megoldás: kaphatók az ilyen feladatokra kifejlesztett miniatűr dióda áramkörök, melyek 5nA szivárgási áramot garantálnak a jelvezetékek felé, miközben rövid ideig max 4A (1µsec) ill. max 1A (1 msec) csúcsáramot is képesek elviselni.



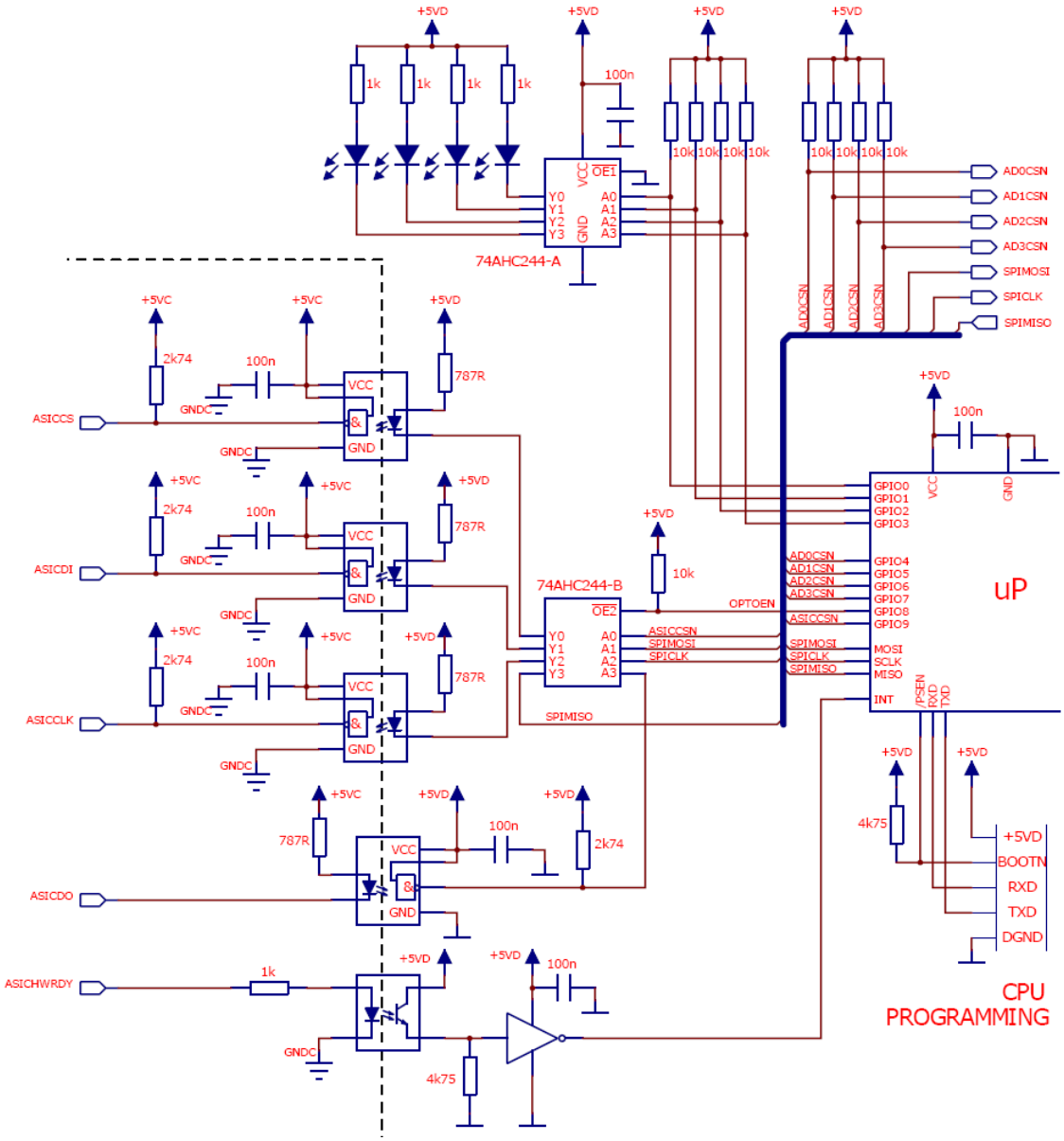
A BAV199DW vágódióda-áramkör

A kapcsolási rajzon látható 2.49kΩ-os ellenállás (a REFIN+ és REFIN- bemenetek között) valósítja meg a mindenkori RTD mérőárammal arányos referenciafeszültség előállítását, ez az ellenállás 0.05% pontosságú.



A buszkommunikációs egység és környezete

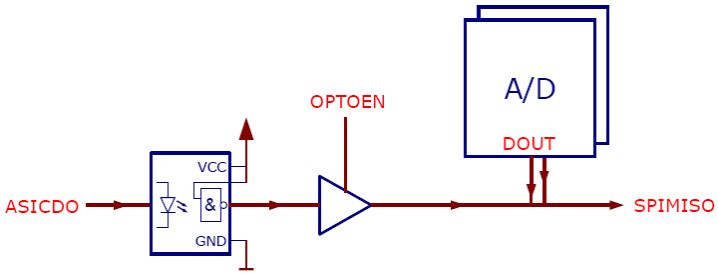
A buszkommunikációt megvalósító ASIC áramkör kapcsolása nem takar semmi különlegeset: SPI interfészen keresztül kapcsolódik a mikrokontrollerhez, és egy I^2C felületű EEPROM-ból olvassa be a modul (rendszer-) tulajdonságait. Az EEPROM tartalmát beültetés után programozzák be gyártáskor (ISP).



A processzor kapcsolata környezetével

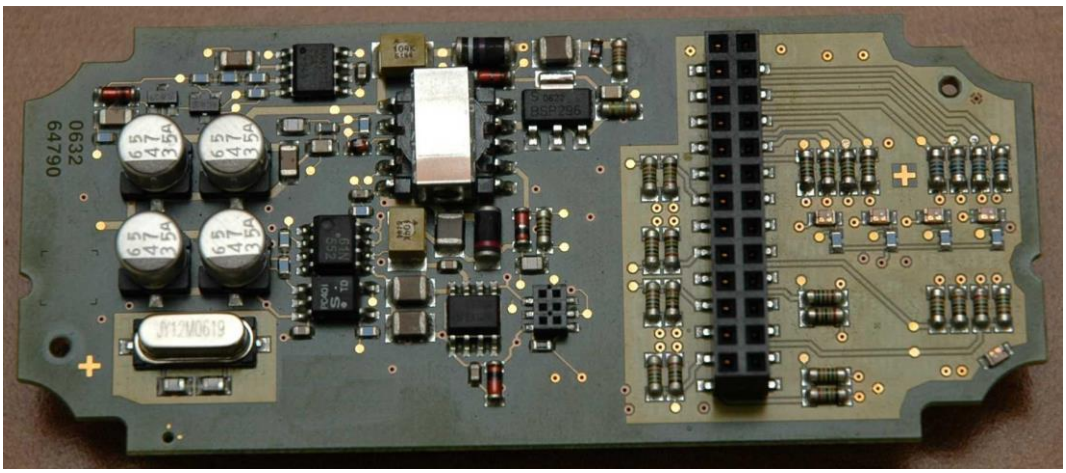
A modul intelligenciáját az Atmel cég AT89C51ED2 mikrokontrollere biztosítja. Ez a típus egyike az 5V tápfeszültségű, 64 kbájt flash memóriát tartalmazó 51-es magot tartalmazó mikrokontrollereknek, melyek ISP programozást és selfupdate lehetőséget biztosítanak. Az A/D átalakítók 5V tápfeszültség igényét figyelembe véve célszerű 5V tápfeszültségű típust választanunk, ezzel megspóroljuk egy 3.3V tápegység kialakítását. A feladat bonyolultsága (hasonló feladatokkal összehasonlítva) 20-30 kbájt méretű firmware kód méretet valószínűsít, a szenzortáblák tárgyalásánál már láttuk, hogy min. 10 kbájt konstans adatterületre is szükségünk lesz (célszerű esetleg ebben még némi tartalékot is fenntartani az utólagos bővítések céljára).

A mikrokontroller típus az ISP programozást soros felületen teszi lehetővé, ezért alakítjuk ki az alsó Tx/Rx vonalakra épülő ISP csatlakozót. A lokális (csatornánkénti) hibajelzést ellátó LED diódák meghajtására a mikrokontroller kimeneti árama nem elegendő, ide meghajtó áramkört kell alkalmaznunk. Ugyanez igaz az SPI felület galvanikus leválasztását biztosító optocsatolók LED diódáinak meghajtása esetén.



Az SPI buszra kapcsolódó egységek

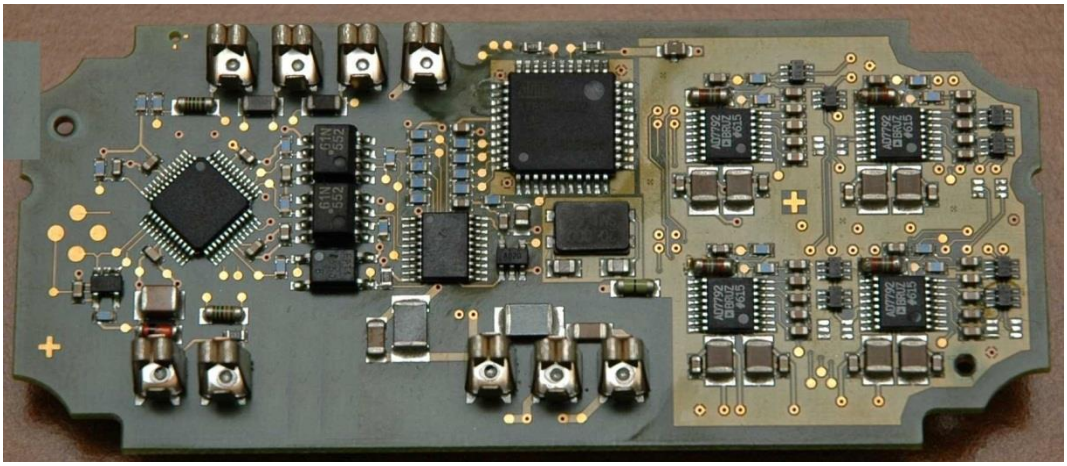
Mivel az SPI buszon összesen 5 résztvevő is található (ASIC + 4 A/D átalakító), tekintettel kell lennünk arra, hogy az SPI busz MISO jelvezetéke (amely az inaktív slave egység felől nagyimpedanciás állapotot feltételez) az optocsatolóval történő leválasztás után folyamatosan meghajtásra kerülne, így ezt egy külön – háromállapotú kimenettel rendelkező – kapuáramkörrel kell illesztenünk az SPI buszra. Időzítési okokból (mivel több jelvezetéket is ez a kapuáramkör hajt meg) az engedélyezést nem végezhetjük magával a kiválasztó jellel (/ASICCSN), hanem egy külön vezérelhető kimenetet (OPTOEN) használunk a célra.



A megvalósított áramkör (felülnézet)

A vázolt elképzelések szerint megtervezet áramkörök különösebb változtatások nélkül működőképesek voltak. Az áramkör képein jól látszik mindkét oldalon a digitális rész (felülnézetben baloldal) elkülönülése az analóg egységektől (jobboldal). A digitális részhez soroljuk a felülnézeti képen látható galvanikusan leválasztott tápegységet, a

buszkommunikációs egység és a mikrokontroller az alulnézeti képen láthatók. A két egység két szigetet képez, melyek között az optocsatolók teremtenek kapcsolatot.



A megvalósított áramkör (alulnézet)

A hidegpont hőmérsékletét a modul csatlakozójában mérjük. A csatlakozó 5 bekötési pontja közel azonos hőmérsékletűnek tekinthető, a valamennyi ponthoz kb. azonos távolságra levő középső ér helyén található az a miniatűr ellenállás-hőmérő (az ábrán fehér színnel), melynek kivezetéseit azonnal ugyanennek a csatlakozónak két pólusán vezetjük vissza. A másik két pólus lesz a hőelem két csatlakozási pontja.

Hidegpont a csatlakozóban

5.2 A hardvertervezés alapelvei

Az elsajátított ismeretek alapján elkezdhetjük megtervezni a kitűzött feladatot ellátó mikrokontrolleres kapcsolást. Ennek lépései a következők:

- A **rendszerterv** (feladatterv) összeállítása
  A feladat megértése, a hardver-szoftver feladatok előzetes vizsgálata, a megvalósítási lehetőségek elemzése, a megoldások kiválasztása, algoritmusok felállítása (mintaáramkörök, mérések, mintaprogramok, szimulációk). Nagyon fontos a rendszerszintű szemlélet, a hardver és a szoftver kapcsolatának és egymásra hatásának elemzése.
- A hardver kapcsolás **logikai vázlatának** (kapcsolási rajz) megtervezése
- **Mechanikai tervek** elkészítése
  Tipikusan nem az áramkört tervező villamos szakember feladata. A beágyazott rendszerek esetében viszonylag ritka a „majd elhelyezzük valahova az áramkört" jellegű konstrukció, sokkal inkább a fizikai lehetőségek szólnak bele sokszor még az áramkör kapcsolástechnikai felépítésébe is (pl. áramkörrészek eloszlása a rendelkezésre álló helyen, használható alkatrészek - max. magasság, csatlakozótípusok - a fizikai méretkorlátok által meghatározott módon).
- A **nyomtatott áramkör** fizikai megtervezése
  Nehéz „rabszolgamunka", ami nagyon meg tudja hálálni a tervező alaposságát és körültekintését. Indul az alkatrészek elhelyezésével, folytatódik az elektromos összeköttetések megtervezésével, és általában egy iterációba torkollik a követelmények végső ellenőrzése során. Ha a kapcsolást nem értő (pl. betanított) személy végzi, általában a végeredmény messze van az optimumtól. Az egyre jobb képességű autorouterek sok feladatot le tudnak venni a vállunkról, de a kialakítás feltételrendszerének bekonfigurálása általában a megoldással azonos nagyságú feladat.
- Az első **hardver prototípusok beültetése**. Ezt a tervezők sokáig maguk csinálták, az elemek fokozatos beültetésével, a tervezési hibák lehető legkisebb áramköri károkkal történő feltárására, az egyes egységek működésének lépésről lépésre történő mérésével. Napjainkban a nagyszámú és kézzel nehezen forraszthető (pl. BGA) felületszerelt alkatrész miatt már legtöbbször ültető automaták végzik.
- A **hardver prototípusok élesztése**, első tesztprogramok a működés elemzésére
  Ezt a feladatot általában ugyanaz a személy (csoport) végzi, aki a kapcsolást tervezte. Ekkor történik a működés elemzése, a tényleges paraméterek bemérése.
- A **programrendszer** fejlesztése.
  Optimális esetben a szoftvert már a meglévő hardveren fejlesztjük, ha nem, akkor ilyenkor következik a szoftver „élesztése". Minél nagyobb programrendszert írtunk próba nélkül, annál nagyobb a valószínűsége a bonyolult, egyre nehezebben felderíthető hibáknak.
- A hardver-, a program- és a teljes rendszer tesztelése
  Hardver teszt: a kapcsolás elektromos tulajdonságainak mérésekkel történő ellenőrzése és dokumentálása (tápáram-felvétel, tápfeszültség-tartomány, bekapcsolási áramcsúcs, tápkimaradás-tűrése, stb.), EMC, klíma, és mechanikai vizsgálatok.

Szoftver teszt: a leprogramozott algoritmus szisztematikus vizsgálata minden lehetséges bemenő állapotra (pl. a hőelemes esettanulmánynál valamennyi lehetséges A/D- értékre kapott válasz valamennyi szenzortípusnál, ezzel az algoritmus és a táblázatok minősítése)
Integritás teszt: a végleges modul (automatikus) tesztelése végleges szoftverrel végleges állapotában (trükkök, és szimulált „belenyúlások" nélkül)

- **Redesign**
  A feltárt hibák korrigálása, végleges kapcsolás és program kialakítása.
- **Gyártási dokumentáció** elkészítése
  Minden, ami a tervezett egység gyártásához, reprodukálásához és dokumentálásához tartozik. Pl. ide tartozik annak a rendszernek a pontos leírása és esetleg archiválása, amivel a rendszer fejlesztése történt. Mérési utasítások, alkatrészlisták, költségkalkulációk, stb. elkészítése.

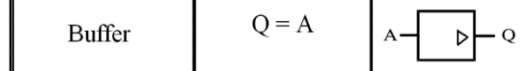
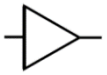
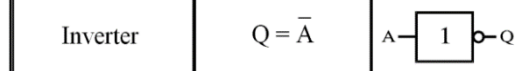
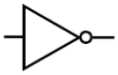
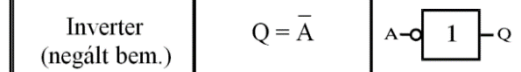
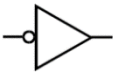
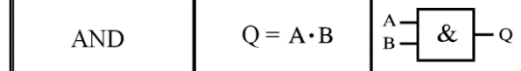
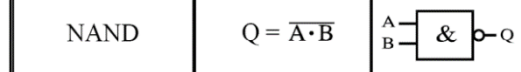
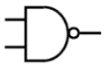
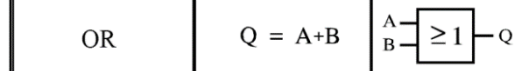
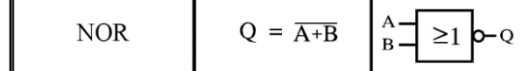
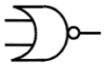
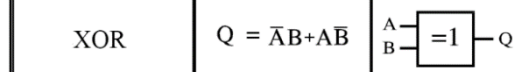
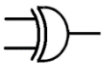
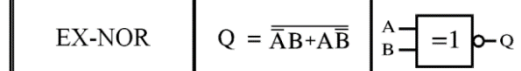
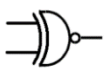
A folyamat (elsősorban a hardver megtervezése) sok olyan technológiai jellegű ismeretet igényel, melyekkel más tárgyak keretei között ismerkedtek meg részletesebben (pl. Elektronikai technológia). A kapcsolási rajz és az áramkörtervezés CAD eszközeinek ismeretét ezek alapján már ismertnek tekintjük, ezek ismertetésére itt nem térünk ki.

5.2.1 A feladatterv (rendszerterv) elkészítése

Mint azt már két korábban bemutatott esettanulmány kapcsán is láthattuk, a beágyazott rendszerek tervezésénél elengedhetetlen a feladat előzetes lebontása, annak eldöntése, hogy mit kell hardverrel, és mit szoftver program segítségével megvalósítanunk. A határok helyes megtalálása döntően befolyásolja a megoldás jóságát, ide értve annak árát, teljesítőképességét, alkalmazhatóságát, pontosságát, stb. A beágyazott rendszerek általában meglehetősen költségérzékenyek, a hardver felesleges túlméretezése a végterméket drágítja, a többletköltség minden egyes termék árában jelentkezni fog. Az „alulméretezés" viszont általában a pontosság, a sebesség és egyéb követelmények rovására megy, ha egyáltalán kompenzálhatók szoftver segítségével. Ügyeljünk tehát az előzetes vizsgálatok során a lehetőségek pontos feltérképezésére, szükség esetén mérések elvégzésével, programrészletek tesztelésével (pl. futásidők becslése kritikus esetben).

5.2.2 Kapcsolási rajz elkészítése

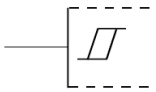
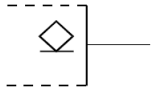
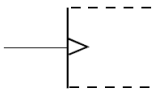
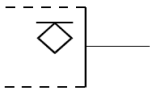
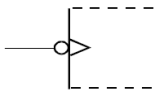
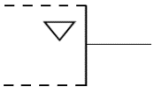
A kapcsolási rajzot napjainkban már kizárólag valamilyen CAD rendszerben készítjük el (OrCAD, PADS, Protel, Altium, Allegro, Eagle, stb.) Ennek oka a számítógépes támogatással való igen gyors feladatmegoldás, az automatikus ellenőrzések előnyei, az egyszerű változtathatóság, dokumentálhatóság és archiválhatóság lehetősége. Az igen költséges CAD rendszer kiválasztása a legritkább esetben a tervező lehetősége, ez általában vagy adott a munkahelyen, vagy a megrendelő szabja meg, milyen rendszerben kéri a fejlesztés végleges eredményeinek átadását (itt ne a nyomtatható ábrákra, hanem a későbbiekben tovább módosítható bináris állományokra és alkatrészkönyvtárakra gondoljon!)
A továbbiakban csak összefoglaló jelleggel néhány szabályt és jó tanácsot ismertetünk a kapcsolási rajz elkészítéséhez.

		IEC 60617	Német	Amerikai
Buffer	Q = A			
Inverter	$Q = \overline{A}$			
Inverter (negált bem.)	$Q = \overline{A}$			
AND	$Q = A \cdot B$			
NAND	$Q = \overline{A \cdot B}$			
OR	$Q = A+B$			
NOR	$Q = \overline{A+B}$			
XOR	$Q = \overline{A}B+A\overline{B}$			
EX-NOR	$Q = \overline{\overline{A}B+A\overline{B}}$			

Logikai kapuk szabványos jelölései (IEC 60617)

A CAD rendszerekkel való hatékony munka elsődleges feltétele az alkalmazandó elektronikus alkatrészek megléte az alkatrészkönyvtárakban. Mivel az alkatrészek ilyen célú leírása (az ún. „alkatrész generálás”) meglehetősen munkaigényes feladat – különösen, ha igényes grafikával és tulajdonságainak az ellenőrzések és a szimulációk számára is elegendő részletességével írtuk le őket –, ezeket a tervezők szívesen veszik kész könyvtárakból, a gyártók által kibocsátott alkatrészkönyvtárakból. Emiatt, és a nehezen változó szokásrendszer miatt is jellemző a logikai kapcsolási vázlatokra, hogy meglehetősen vegyes elemjelölést alkalmaznak. A korábbi német (DIN = Deutsches Institut für Normung) és amerikai (ANSI = American National Standards Institute) jelölések keveredése jellemző egészen napjainkig, jóllehet **Magyarországon is kötelező érvényűek** az európai országok által elfogadott IEC 60617 (International Electrotechnical Comission) 1996-os szabvány előírásai, melynek 12. fejezete foglalkozik részletesen a logikai kapcsolások elemeinek jelöléseivel. A fenti táblázatban a legfontosabb alapelemek jelöléseit mutatjuk be mindhárom rendszerben.

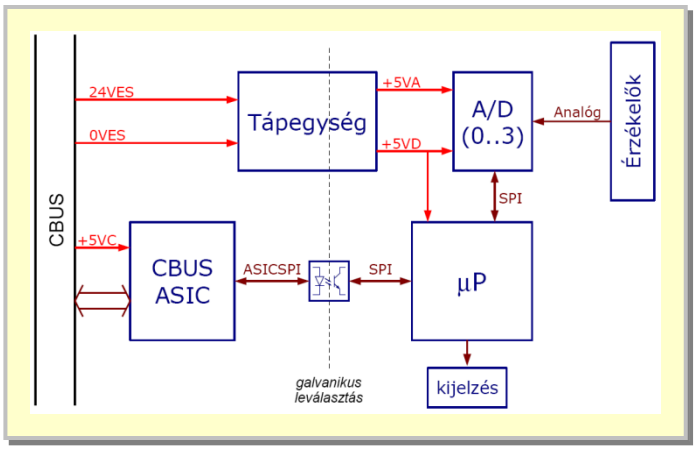
Szintén hasznos annak az ismerete, hogy a szabványos logikai elemek jelölésein a be- és kimenetek mellett előfordulhatnak olyan kiegészítő jelölések, melyek fontos tulajdonságokat takarnak. Ezek közül a legfontosabbakat mutatja be a következő ábra.

Bemenet	Jelentés	Jelentés	Kimenet
	Hiszterézises bemenet	O/C vagy O/D kimenet (aktív ”0” kapcsoló)	
	Dinamikus (élvezérelt) bemenet	O/E vagy O/S kimenet (aktív ”1” kapcsoló)	
	Dinamikus (élvezérelt) bemenet negálva (nem használják)	Háromállapotú (TS) kimenet	

Kiegészítő jelek logikai be- és kimeneteken

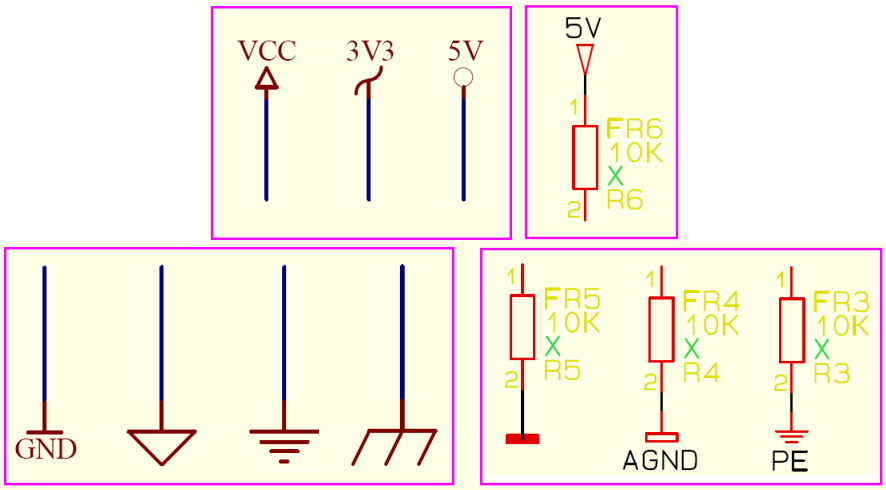
A következőkben néhány fontos (bevált) jó tanácsot ismertetünk kapcsolási rajzaink kialakításához.

- *Tegyük gyorsabban áttekinthetővé kapcsolási rajzunkat egy bevezető blokkvázlattal!*
  A digitális egységek kapcsolási rajzai általában igen sok nagybonyolultságú elemet tartalmaznak, a működés megértése (később esetleg a kapcsolás módosítása) nem egyszerű feladat bármilyen hozzáértő olvasó számára. Kialakítása során sok olyan gondolat, döntés, „trükk” kerül bele, melyek újbóli felidézése bizonyos idő elteltével saját magunknak is nehéz feladat lesz, nemhogy a részletes kapcsolást először látó, háttérgondolatainkról többnyire mit sem tudó kollégánk számára. A megoldás nem egy-egy kézikönyv mellékelése minden kapcsolási rajz mellé, hanem egy blokkvázlattal, dekódolási táblázatokkal, szöveges megjegyzésekkel a rajz megértésének segítése az olvasó számára.



A kapcsolási rajz első oldala mindig egy blokkvázlat legyen

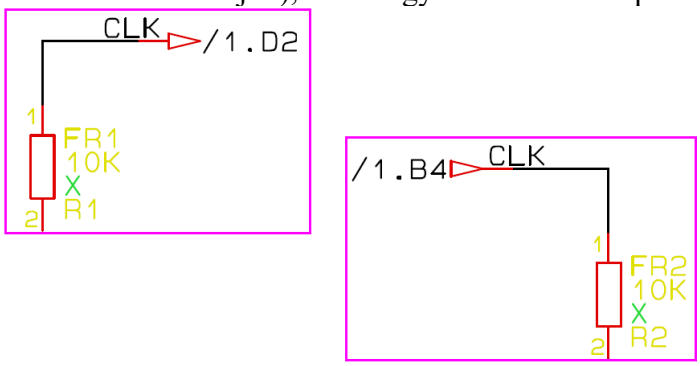
- *Válasszuk meg jól kapcsolási rajzunk tagolását (oldalakra bontását)!*
  Ne zsúfoljunk mindent egy oldalra, mert az eredmény áttekinthetetlen és nyomtathatatlan lesz. A logikailag egy csoportba tartozó elemek kerüljenek azonos oldalra! Kezdettől fogva tartsuk szem előtt, hogy a rajzot ki is kell nyomtatni, amihez alkalmas lapméret és nyomtató szükséges.
- *Ügyeljünk a táp- (power) szimbólumok megfelelő használatára!*
  Kapcsolási rajzunkon nagy számban fordulnak elő a tápfeszültség és jelföld csatlakozások. Itt a legfontosabb szabályok:
  - minden nyíl mellett jelenjen meg egyértelműen, hogy milyen feszültségszintet jelöl (azért is megy ki a divatból egyre inkább a VCC, mert általában egy kapcsolási rajzon több tápfeszültséget is használunk – 24V, 5VD, 5VA, 3.3V, stb.)
  - hasonló módon többféle jelföld szimbólum fordul elő (0V, GND, AGND, PE), ezek jelölésére célszerű különböző szimbólumokat használnunk
  - minden alkatrész mellett jelenjen meg egy tápszűrő kerámia kondenzátor (68-100nF), amely a nagyfrekvenciás zaj kialakulását és elnyomását nagymértékben befolyásolja. Fizikailag ezek a kondenzátorok párhuzamosan kapcsolódnak egy-egy tápfeszültségen belül, mégse különítsük el őket a kapcsolási rajz egyik sarkába, hanem mutassuk meg a rajzon egyértelműen, hogy melyik alkatrész szűrését hivatott ellátni!



VCC és GND szimbólumok a kapcsolási rajzokon

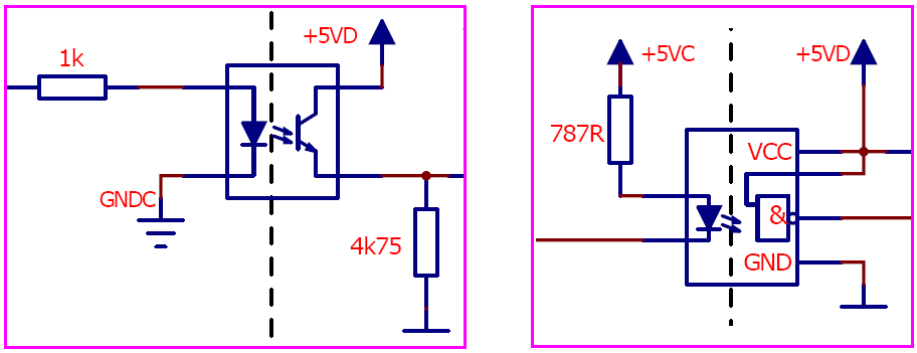
- *Ne hagyjunk „nyitott vezetékeket” a kapcsolási vázlaton!*
  Sajnos több CAD program is lehetővé teszi a vezetékek egy adott ponton történő „elhagyását” és egy tetszőleges másik helyen (adott esetben másik oldalon) történő folytatását. Ilyenkor az összekötési listában az összekötetlen vezetékvégek azonos jelként jelennek meg elnevezésük alapján, de a kapcsolási rajzot az ilyen megoldások olvashatatlanná teszik. A jobb programok általában felkínálják (a még jobbak megkövetelik) az ún. on-page (folytatás azonos lapon) és off-page (folytatás másik lapon) szimbólumok használatát. Segítségükkel nem csak azt tudjuk, hogy „nem felejtődött itt el” semmi, hanem a szimbólumok általában mutatják a jelterjedés

irányát és azt is, hogy a megfelelő csatlakozási pont(ok) helyileg hol (<oldalszam>.<XY-koordináta>) található(k). Ne éljünk vissza ezzel a lehetőséggel (azzal, hogy túl sokszor használjuk), mert nagyon nehezíti a kapcsolási rajz olvasását



On-page / off-page szimbólumok

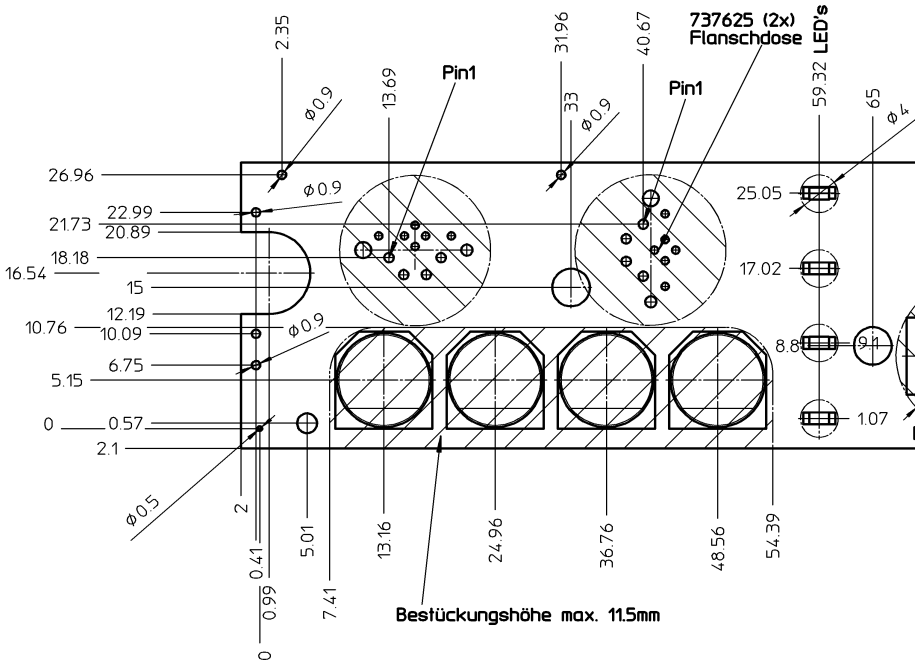
- *Vezetékcsoportok („buszok”) alkalmazása*
  Segítségükkel nagyban növelhetjük a „vezetékrengeteg” áttekinthetőségét, de csak akkor, ha logikailag összetartozó vezetékeket (adatvezetékek, címvezetékek, egy SPI busz különböző jelei, 8 db bemenet, stb.) fogunk össze egy-egy csoportba! A „mindent egy buszra” elv bosszús be- és kicsatlakozás keresgélésbe fog fulladni!
- *Ne üres „dobozokat” definiáljunk alkatrészekként!*
  Bár a logikai kapcsolatok leírását egy üres téglalap is kielégíti a CAD rendszer számára (megfelelő kivezetésekkel), törekedjünk a „beszédes” rajzszimbólumokra, még ha ezek elkészítése komoly többletmunkával is jár! Nem lehet úgy kapcsolási rajzot elemezni, hogy minden alkatrésznél megállunk egy katalógusban keresgélni: mit is csinál ez az alkatrész és melyik kivezetésének mi a funkciója.



Az alkatrészek szimbólumai segítsék az áttekintést

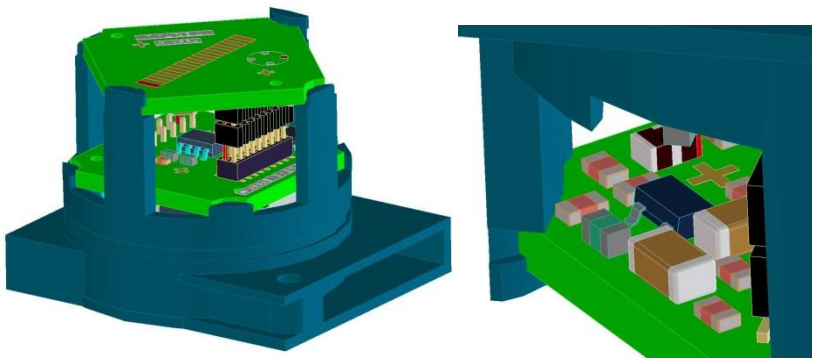
5.2.3 A mechanikai tervek elkészítése

Mint már említettük, a feladat tipikusan nem az áramkört tervező villamos szakember feladata, mégis ez a tervezési fázis szoros kölcsönhatásban van a tényleges áramköri tervek kialakításával. A tervezendő modul fizikai lehetőségei sokszor még az áramkör kapcsolástechnikai felépítését is befolyásolják, bizonyos alkatrészek cseréjére, esetleg a kapcsolás megváltoztatására is szükség lehet a méretkorlátokból adódóan.



Mechanikai méretrajz részlete

A mai korszerű számítógépes CAD rendszerek (a pontos, minden részletre kiterjedő alkatrészleírások, a fáradtságos munkával létrehozott precíz alkatrészkönyvtárak következtében) ma már arra is képesek, hogy az áramköri tervek első lépéseként létrehozott ún. alkatrész elhelyezési terv alapján az áramkör pontos, háromdimenziós képét is megmutassák nekünk, még mielőtt az áramkör ténylegesen legyártásra és beültetésre kerülne. Ezzel a lehetőséggel rengeteg tervezési hiba deríthető fel előre egy-egy komplexebb mechanikai kialakítású áramkör esetében. Sok esetben a kritikus helyekre kerülő alkatrészek elhelyezhetőségét, pozícióját és orientációját is ilyen segédeszközök segítségével határozzuk meg.



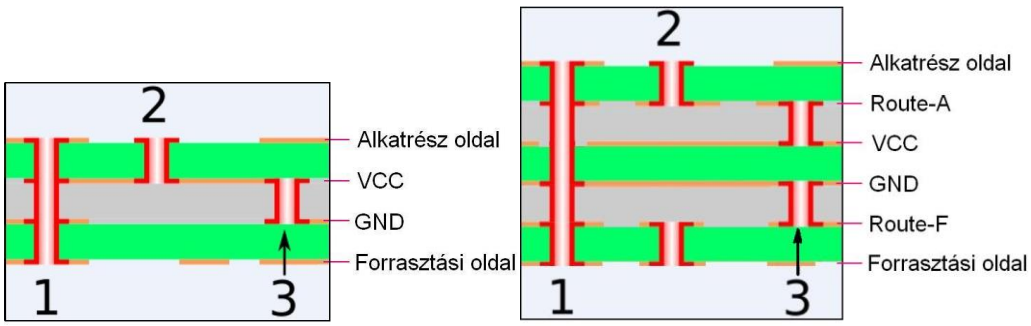
Az alkatrészek elhelyezési tervének ellenőrzése 3D szimuláció segítségével

5.2.4 A nyomtatott áramkör tervezése

Amint az előző pontban is említettük, a nyomtatott áramkör tervezésének első lépése az alkatrészek elhelyezése az áramköri lapon. A mai nagybonyolultságú áramkörök esetében ez egyáltalán nem lebecsülendő fontosságú feladat. Jól jegyezzük meg: egy jó alkatrész elhelyezési terv már fél megoldás az áramkör megtervezéséhez !

Nyomtatott áramköreinket napjainkban 2, 4, 6… rétegű kivitelben készítjük el. A rétegszám kiválasztásához a következő alapelvek az irányadók:

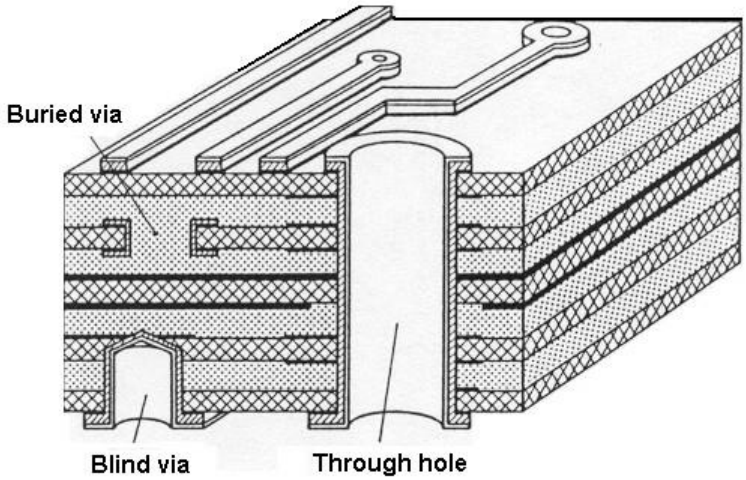
- A kétoldalas áramkörök előállítási költségei a legalacsonyabbak, viszonylag egyszerű technológiával előállíthatók két oldalon fóliázott lemezekből.
- Nagyobb alkatrészsűrűség esetén bajban leszünk a jelvezetékek és a tápvezetékek kialakításával. A teljes áramkört behálózó VCC és GND tápvezetékek vastag, széles rézfelületeket követelnének meg, aminek ellentmond, hogy nem „férünk el" a két oldal között ide-oda bujkáló jelvezetékekkel. Az oldalak váltása ráadásul mindig derékszögű törést jelent, ami reflexiós pont a nagysebességű jelvezetékek esetében.
- Mikrokontrollert tartalmazó digitális kapcsolásoknál szinte alapkövetelmény a legalább 4 réteg használata. Ezek közül a két belső a VCC és GND felület (plane), amely egy jó elektromos tulajdonságú, kisimpedanciás táphozzávezetést képes biztosítani, ha nem szabdaljuk fel kis részekre keskeny átvezetésekkel a részek között. Az alacsonyimpedanciás rézfelületek jó árnyékolást is biztosítanak jelvezetékeinknek, ezért mindenhol, ahol jelvezeték vagy alkatrész található, alatta a belső rétegen legyen rézfelület!



4 és 6 rétegű nyomtatott áramkör felépítése

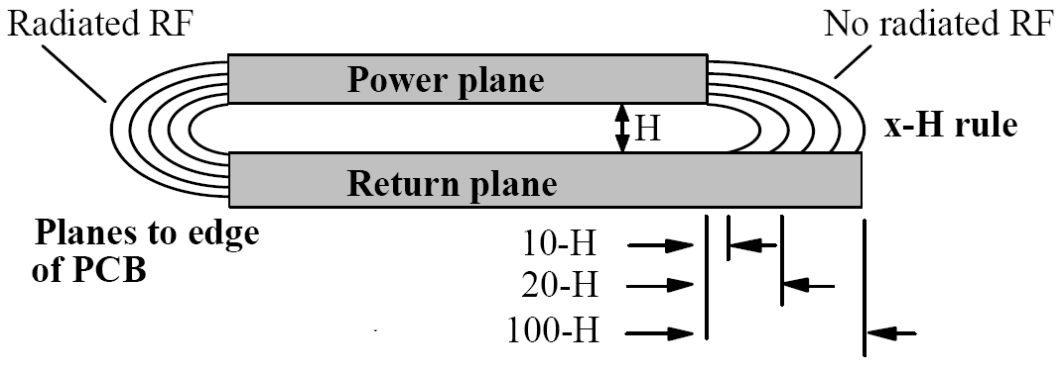
- Nagyobb sűrűségű kapcsolásoknál a külső rétegeket (mindkét oldalon) szinte beborítják az alkatrészek, a táprétegeket nem célszerű jelvezetéssel felszabdalni. Ilyenkor már 6 rétegű áramkörökön tervezünk, ahol a külső rétegek az alkatrészeké, az ezt követő rétegeken alakítjuk ki a jelvezetékeket, a középső rétegek a tápellátást szolgálják. Nagymértékben segíti a kialakítást, ha használhatunk ún. zsákviákat (blind via), melyek csak a szélső rétegek között biztosítanak átvezetést, ezért nem befolyásolják a túloldalon lévő alkatrészek elhelyezkedését és a jelvezetékek kialakítását. Ezek az átvezetések általában lézerrel fúrt, sokkal kisebb méretű (50 µm átmérőjű) furatok, mint az átmenő

furatok (0.4-0.5 mm). Még nagyobb segítséget nyújtanának az ún. eltemetett viák (burried via), melyekkel akár a tápréteghez is csatlakozhatnánk – ezek azonban lényegesen költségesebb nyák-technológiát igényelnek, ezért csak ritkán megengedett a használatuk.



Az átmenő, a zsák- és az eltemetett átvezető furat (via) kialakítása

Áramkörünk kialakításánál törekedjünk arra, hogy a tápellátást biztosító belső rézfelületek teljes egészében „fedjék le" jelvezetékeinket és áramköreinket. A kártya szélén mérések szerint rétegek között kialakuló erőtér a nagyfrekvenciás digitális zaj hatására komoly rádiófrekvenciás kisugárzóként (antennaként) jelenik meg, ami sérti az érvényben lévő EMC előírásokat. Kutatócsoportok vizsgálták a kialakuló tér tulajdonságait, és arra a következtetésre jutottak, hogy amennyiben a két réteg közötti távolság értéke „H", akkor az n x H mértékben kisebbre méretezett VCC plane erővonalai egyre inkább a nagyobb GND plane-ben záródnak, csökkentve ezáltal a kisugárzott tér erősségét.



A VCC és a GND plane helyes kialakítása (20H szabály)

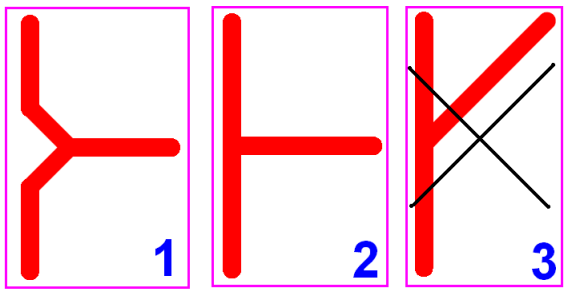
A mérések alapján

- 10H-tól kezdődően válik érzékelhetővé az RF kisugárzás csökkenése
- 20H-nál a kisugárzás 70%-a nyelődik el
- 100H-nál a kisugárzás 98%-a nyelődik el

H ≈ 100-150µ esetén az ún. 20H-szabály tehát azt írja elő számunkra, hogy VCC rétegünk rézfelülete körben mindegy 2 mm-rel legyen kisebb a GND felületnél, miközben valamennyi vezeték és alkatrész mindkét felület által fedett részen található.

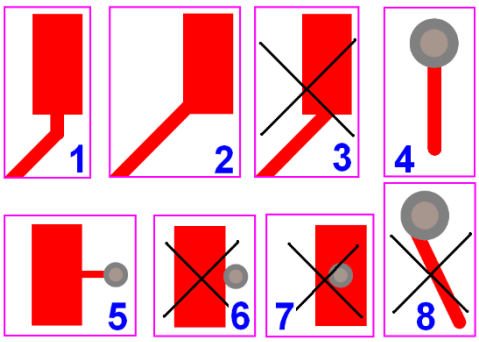
A vezetékek kialakításánál ügyeljünk a következő szabályok betartására:

- Ne alakítsunk ki derékszögű töréseket a vezetékekben, minden ilyen pont reflexiók forrása lesz. Elágazásnál legjobb az alábbi ábrán látható (1)-es, de legalább a (2)-es példának a követése.
- Ne alakítsunk ki hegyes (kb. 60° alatti) szögben egymáshoz csatlakozó rézfelületeket (3). Az elektrosztatikában tanult csúcshatás itt is érvényesül, a kialakuló térerő a lekerekítési sugár és a nyílásszög csökkenésével rohamosan (fordított arányban) nő!



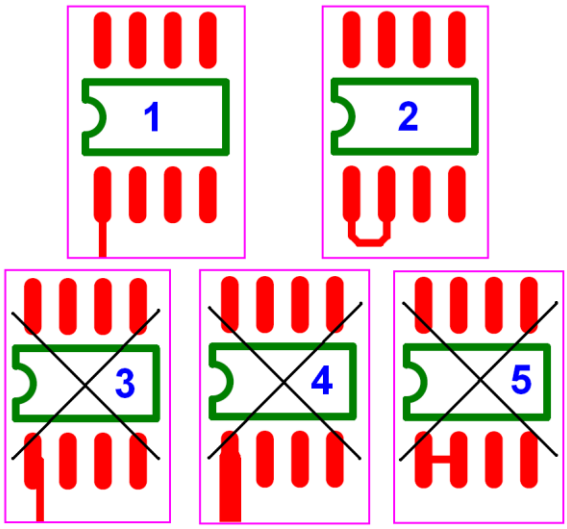
Vezetékek (route) elágazása

- Ugyanez az elv követendő az alkatrészlábakhoz és viákhoz való csatlakozás esetében (1, 2, 3, 4 és 8-as példák).



Padek és viák helyes csatlakozása a vezetékekhez

- Termikus problémákat okozhat az alkatrészlábak közvetlen lekötése átvezető furattal a nagyfelületű belső tápfelületekre. Az ilyen bekötéseket termikusan függetlenítsük az alkatrész lábát fogadó padtől (5), helytelenek a (6) és (7) kialakítások. Ezek további hátránya, hogy automatikus ültetés és forrasztás esetén a megolvadt forraszanyagnak az átvezető furatba való áramlása (az alkatrészláb bekötése és mechanikus rögzítése helyett) is gondokat okozhat.



Integrált áramkörök bekötési szabályai

- Integrált áramkörök lábait az (1) és a (2) példák alapján alakítsuk ki. Kerüljük a padek méretét megváltoztató vezetékbekötést (3 és 4 példák) és a két szomszédos láb „direkt” összekötését, amely forrasztás után egy véletlen zárlatnak fog kinézni.

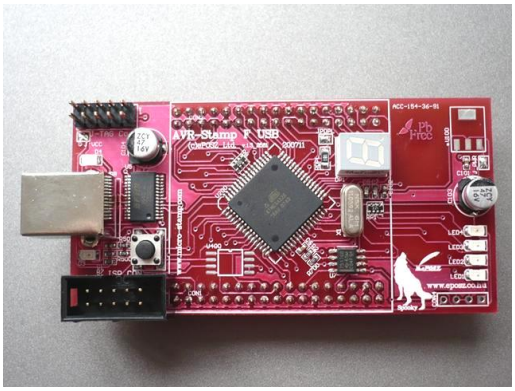
5.2.5 Az áramkör beültetése

Mint már említettük, napjainkban a nagyszámú és kézzel nehezen forraszthető (pl. BGA) felületszerelt alkatrész miatt már legtöbbször ültető automaták végzik.

5.2.6. Az áramkör élesztése, programozása

Korábbi (elsősorban a hagyományos, furatszerelt alkatrészeket tartalmazó) kapcsolások esetében a programozott áramkörök (mikrokontrollerek, memóriák, programozott logikák, stb.) előre programozott állapotban kerültek beültetésre az áramkörökbe. A fejlesztés idejére ezek többször programozható típusait (pl. EPROM) foglalatba helyezték, hogy kivehető, és speciális berendezéssel újraprogramozható legyen.

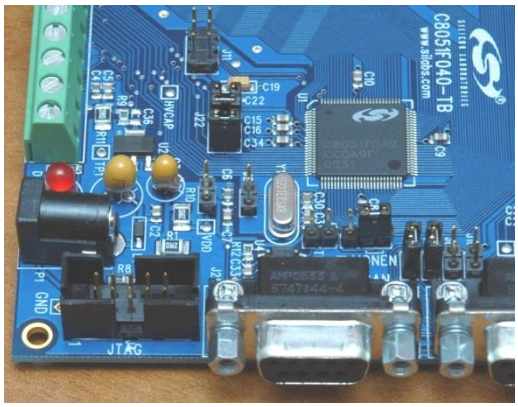
Napjainkban az olcsó tömeggyártás és az újfajta áramköri kialakítások (nagy lábsűrűségű felületszerelt áramkörök) már nem „tűrik el” az áramkör programozó készülékbe történő befogását és programozását – a mai követelményrendszer már a „mindennek nyers állapotban” való szerelhetőségét, és végleges (beforrasztott) állapotában való utólagos programozhatóságát követeli meg. Az ISP (**I**n **S**ystem **P**rogrammable) jelleg a mai áramkörök alaptulajdonságává vált, a programozásukhoz szükséges magasabb programozó feszültségeket az áramkörök már saját maguknak állítják elő. A csatlakoztatás a programozó hardverhez általában szinkron soros (SPI jellegű) felületen keresztül történik, ehhez egy 5-10 pólusú csatlakozó (VCC, GND, SCLK, DIN, DOUT, RST, stb.) elegendő.



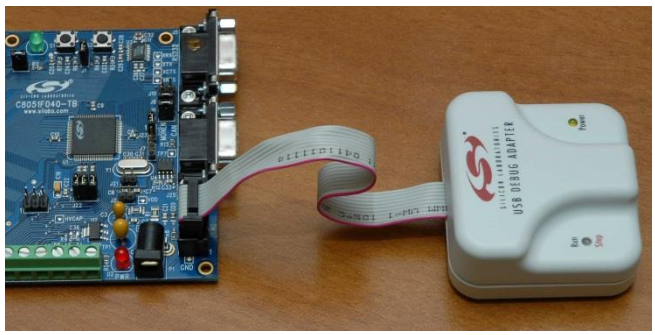
ISP csatlakozó a mikrokontroller mellett

Az egyszerűbb kialakítású ISP-felületek általában „csak” az adott áramkör programozását teszik lehetővé, ezen túlmenően többletszolgáltatást nem nyújtanak. A JTAG jellegű csatlakozási felületektől azonban már komolyabb előnyöket várhatunk, ezek kialakításával ismerkedjünk meg kicsit részletesebben!

Első ránézésre a képeken látható ISP és JTAG csatlakozófelületek kialakítása között nincs lényeges különbség. Az eltérés nem is a külső kialakításban, hanem az áramkörök belsejében rejtőzik.

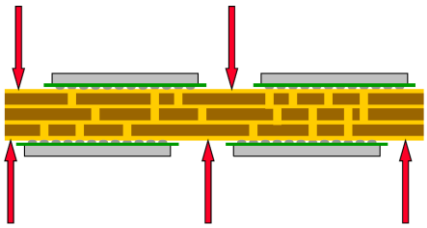


JTAG csatlakozó a mikrokontroller mellett



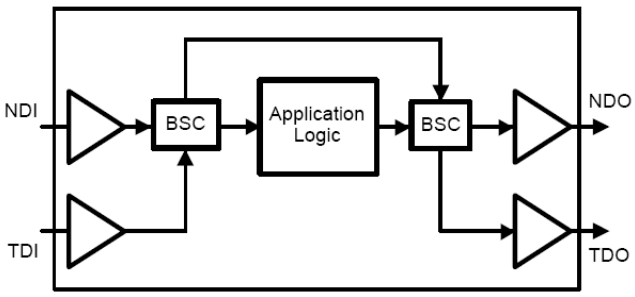
JTAG csatlakozó illesztőegysége (USB vagy COM)

A JTAG rövidítés a **J**oint **T**est **A**ction **G**roup tömörülés elnevezése. A társaság azt tűzte ki célul, hogy megoldják a nyomtatott áramköri berendezések minél általánosabb, beültetett állapotban történő tesztelésének a lehetőségét költséges, az egyre nagyobb alkatrészsűrűség és integráltsági fok miatt egyre inkább ellehetetlenülő tűágyas tesztelés nélkül. A magyarra „*peremfigyeléses teszt*”-nek lefordított – angol nevén boundary scan – módszert 1990-ben szabványosították (IEEE Std. 1149.1-1990). Napjainkban a boundary scan kifejezést és a JTAG betűszót azonos értelemben használják.



A többrétegű nyomtatott áramkörökön sűrűn elhelyezkedő nagyintegráltságú alkatrészek között már nincs hely elegendő tűágyas tesztpont elhelyezésére

A módszer az alkatrész minden lábához egy ún. „tesztcellát” (**BSC**: **B**oundary **S**can **C**ell) épít be még az alkatrészen beül, amely képes érzékelni, de akár felülbírálni is a láb szerepkörét és működését. A cellák az alkatrészen belül láncolatba fűződnek, majd kívül, az alkatrészek egymás között tovább láncolódnak.



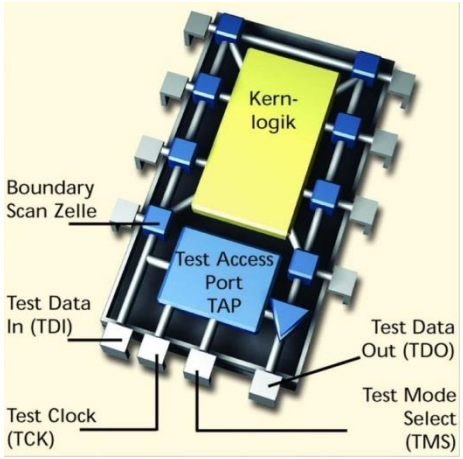
A JTAG (Boundary Scan)módszer alapgondolata (NDI, NDO: normal data I/O)

A tesztcella (BSC) képes a láb állapotát megfigyelni, lekérdezni, de a JTAG láncolaton keresztül egy speciális vezérlőnyelvvel (BSDL: Boundary Scan Description Language) vezérelhetők aktív állapotba is: meghajtva az adott lábat beállíthatják a bemenet állapotát, megmérhetik áramkörök közötti vezeték kapcsolatok meglétét, vagy zárlatait. Természetesen ilyenkor már áramkör specifikus leíró szekvenciára van szükség a működéshez (ami minden áramkörhöz az általános vezérlő nyelv segítségével előállítható). Normál alapállapotban a celláknak nincs hatásuk az áramkör működésére.

A JTAG port működéséhez mindössze 4+1 db logikai vezetékre van szükség:

- TCK (Test Clock): az SPI interfészhez hasonlóan a soros adatátvitel órajele
- TDI (Test Data In): a soros átvitel adatbemenete
- TDO (Test Data Out): a soros átvitel adatkimenete

- TMS (Test Mode Select): választás utasítás vagy adat átvitele között
- TRST (Test Reset): opcionális alaphelyzetbe állító jel

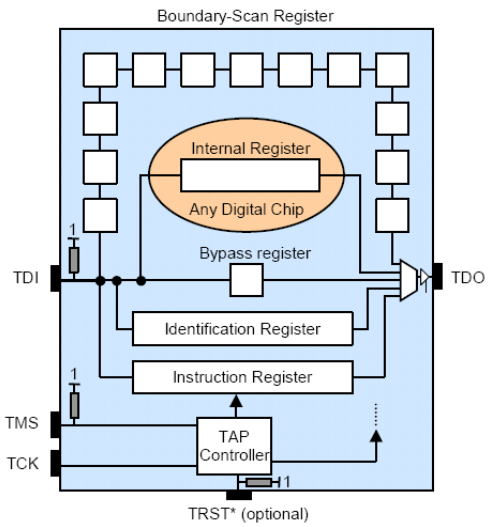


A tesztcellák elhelyezkedése és vezérlésük

A port „lelke” a TAP (Test Access Port) logika, amely kiértékeli a beérkezett parancsokat, és nekik megfelelően vezérli a tesztcellákat (BSC).

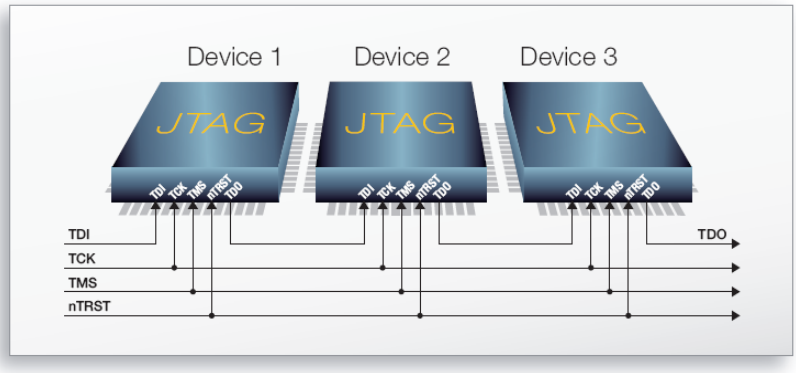
A következő blokkvázlaton jól látható, hogy a TAP és a köréje csoportosuló további logikai egységek (néha ezeket csak összevontan ábrázolják és együttesen TAP-nak nevezik őket, mint a fenti ábrán) képesek az adott áramkört

- a sorosan terjedő adatinformációba be- és kikapcsolni (bypass register),
- a lábak állapotát beállítani (preload) ill. lekérdezni (BSC láncolat körben),
- JTAG utasításokat (instruction) és
- azonosítót (ID = 32 bites gyártó, adott egység típus és verziószám kód) kezelni.



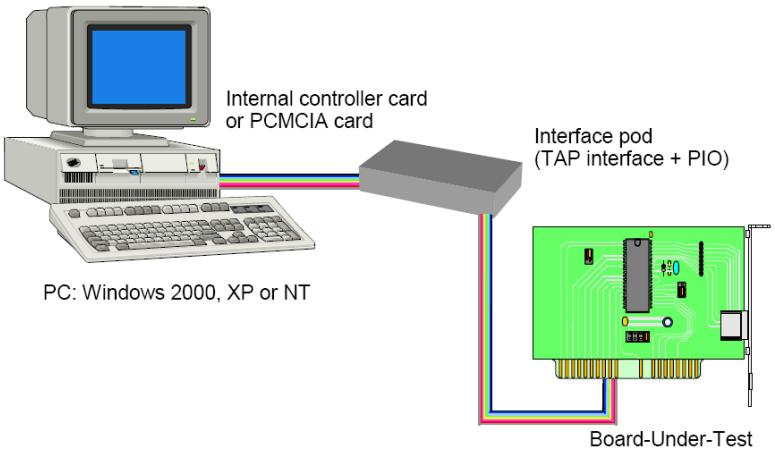
Boundary scan logika felépítése egy áramkörön belül

Természetesen a kialakítás nem csak egy, hanem több egység láncolt csatlakoztatását is lehetővé teszi egyetlen JTAG vezérlő egységhez. Az órajel, az üzemmód és az opcionális alaphelyzetbe állítás valamennyi egység számára közös, az adatok soros láncolatban terjednek az egységek között.



JTAG felülettel rendelkező egységek láncolása egy modulon belül

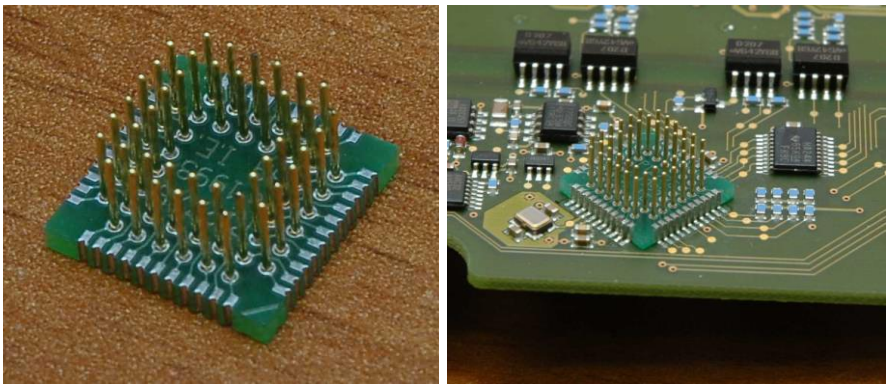
A bemutatott lehetőségből nem nehéz levonnunk azt a következtetést, hogy a JTAG felület – amennyiben létezik – egy-egy eszköz esetében mind az áramkör programozását (ISP), mind a fejlesztést támogató tesztelését (debugging), mind gyártás utáni áramköri tesztjét (ICT – In Circuit Test) lehetővé teszi megfelelő külső vezérlőegység segítségével.



JTAG interfész kezelésére alkalmas rendszer felépítése

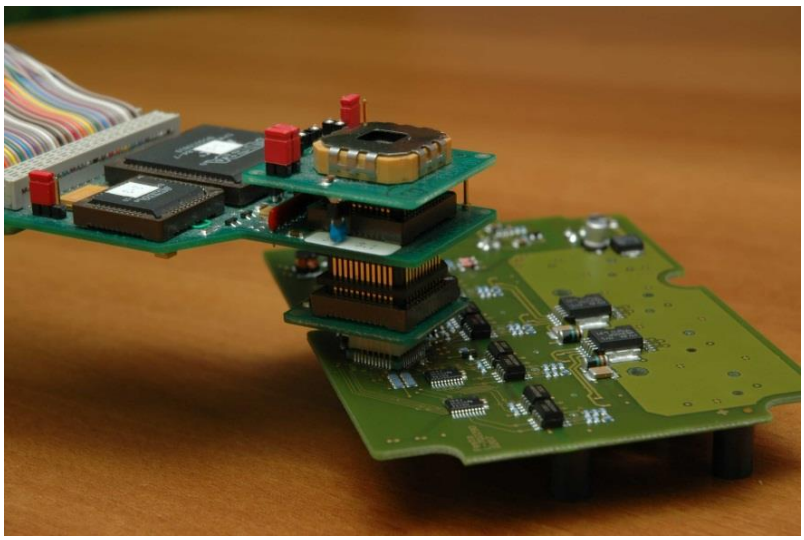
Speciális fejlesztési nehézséget jelentenek az OTP (One Time Programmable) memóriával ellátott mikrokontrollerek. Mint az alkalmazott memóriatípusoknál már említettük, nagyobb szériában gyártott mikrokontrollerek esetében számottevő költségcsökkentő tényezőként jelenhet meg a programmemóriának ez a legalacsonyabb árú lehetősége (pl. Infineon C505). Ilyen típus választásánál a fejlesztőnek két nehézséggel is szembe kell néznie:

- lévén a mikrokontroller memóriája nem újraprogramozható, keresnünk kell egy helyettesítő megoldást a fejlesztés idejére a processzor kiváltására. A végleges formájára kialakított áramkörünkön található (sűrűn elhelyezkedő) lábkivezetéseket csatlakoztathatóvá kell tennünk egy speciális „tűágy” adapterrel valamilyen alkalmas külső egység számára.



Tűágy-adapter OTP ROM memóriát tartalmazó mikrokontrollerhez

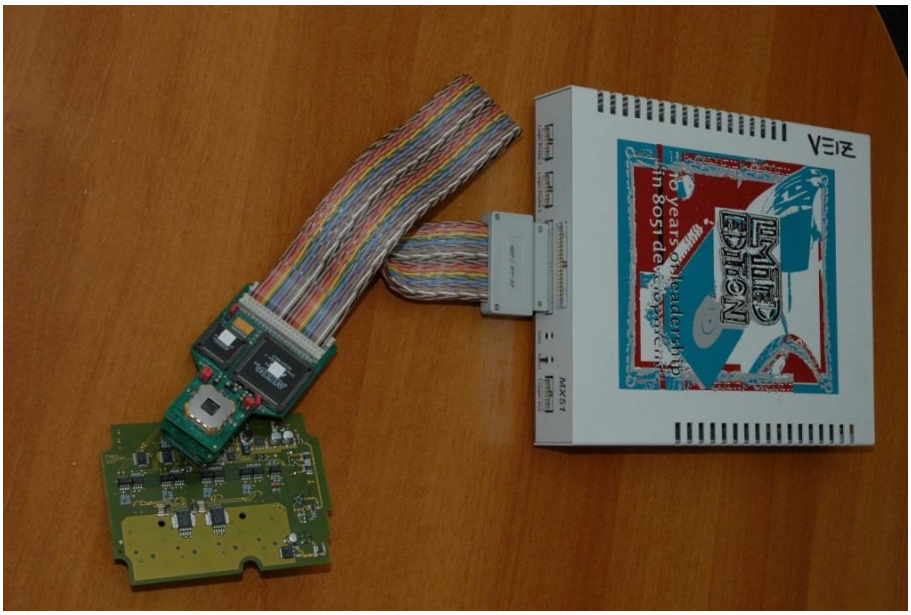
- Szükségünk lesz egy olyan, a komplett mikrokontrollert helyettesítő egységre (ún. emulátorra), amely az említett tűágy-adapterhez csatlakozva a véglegesen alkalmazandó mikrokontroller működését minden szempontból helyettesíteni (emulálni) képes a végleges áramkör számára.



A mikrokontroller helyére csatlakozó emulátorfej

A fenti ábrán jól látható **az eredeti mikrokontrollert** (legfelül) **is** tartalmazó megoldás, amely nagybonyolultságú programozható áramkörök (balra) és egy összetett adapter-rendszer segítségével csatlakozik a mikrokontroller „helyére” és helyettesíti azt működése minden tekintetében a fejlesztés alatt.

- A bemutatott emulátorfejet egy bonyolult logikát tartalmazó illesztőegység (az ún. emulátor) vezérli és illeszti egy külső fejlesztőrendszer elé. Az alábbi ábrán egy ilyen általános célú 8051-es emulátort (Hitex MX51) mutatunk be, mely többféle 8051-es mikrokontroller emulálására is alkalmas a megfelelő fejrész (és programrendszer) kiválasztásával.



C505 hardver emulátor (Hitex MX51)

Az emulátoron alapuló fejlesztés általában hasonló támogatásokat és szolgáltatásokat képes nyújtani a fejlesztő számára, mint a JTAG interfész. Nagy hátránya, hogy a precíziós tűágy-adapter általában többszöröse az alkalmazott mikrokontroller árának, az emulátorfej és az emulátor egység (hardver és fejlesztő környezet) pedig igen komoly – általában több ezer EUR nagyságrendű – beruházási költséget jelentenek. Még nagyobb gyártó cégek is leginkább akkor alkalmazzák, ha az adott modul nagy darabszámú gyártás előtt áll és az emulátor egységet több különböző modul fejlesztéséhez is fel tudják használni.

5.2.7 A programrendszer frissítése (update)

Az egyre inkább programozható logikákon és mikrokontrollereken alapuló áramkörök programrendszerének frissíthetősége (update) napjainkban az alapvető követelmények kategóriájába sorolódik. Ennek főbb okai a következők:

- A programrendszerek fejlesztése rendkívüli módon felgyorsult. A sokszor néhány hónapra leszűkülő fejlesztések mögött azonban egyre bonyolultabb rendszerek és algoritmusok találhatók, melyek tesztelése idő- és költségkorlátok miatt is egyre kevésbé tekinthető teljes körűnek. A programozott egységekben megbújó programhibák („bug”-ok) korrigálására a lehető legzökkenőmentesebb lehetőséget kell biztosítani.

- Egy-egy modul élettartama alatt a környezetében alkalmazott eszközök és a hozzájuk kapcsolódó algoritmusok is többnyire változnak, fejlődnek. Ezekhez a lekezelő programrendszereknek alkalmazkodniuk kell tudni.

Mindkét említett feltétel azonban burkoltan magában foglalja a következő nagyon fontos szempontot: a frissítendő modul többnyire a végleges alkalmazási rendszerben beépítve, folyamatos működésben található. Az alkalmazónak többnyire már az a leállás is problémát jelent, amennyi időre a teljes rendszer működését fel kell függesztenünk az adott modul programrendszerének frissítéséhez, **szó sem lehet** a modul kivételéről, kiszereléséről a rendszerből, vagy tokozásának („házának") kinyitásáról, a modul szétszereléséről. Utóbbi tevékenység legtöbbször a garancia azonnali elvesztését vonná maga után, ami miatt a felhasználó azonnal a gyártó szerviz-szakembereit hívná segítségül, ez viszont beláthatatlan költségeket eredményezne (különösen, ha a teljes rendszer leállásának költségeit is figyelembe vesszük).

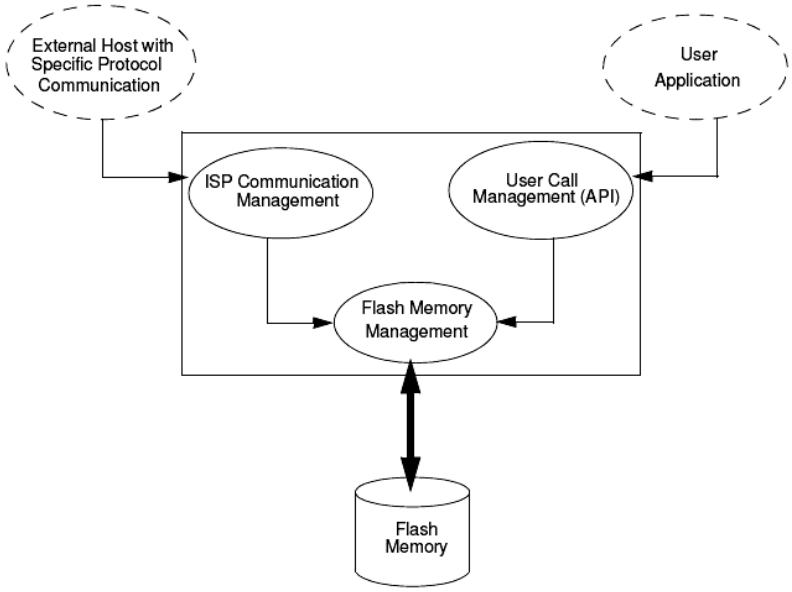
Fentiekből az következik, hogy a modul programrendszerének frissítését annak végleges működési helyzetében, üzemszerűen használt kommunikációs felületein keresztül kell elvégeznünk. Ez lehet egy tesztcélból kivezetett diagnózis felület (viszonylag ritkán fordul elő), az a kommunikációs csatorna, amin keresztül pl. egy host PLC üzemszerűen vezérli az adott egységet (pl. Profibus-DP), vagy egy „táv-update" – egy rövid időre lokálisan csatlakoztatott Internet-kommunikációs lehetőség formájában.

A programrendszer frissítésének (update) lényege, hogy lennie kell a rendszerben egy olyan programnak, amely futásképes a beágyazott firmware törlése és újraprogramozása alatt is. Ezt az ún. **bootloader programot** vagy a gyártók helyezik el a programozható eszközökben, vagy nekünk kell gondoskodnunk róla. Előbbi eset legnagyobb előnye a bootloader bármilyen körülmények közötti megléte a programozható eszközben. A frissítési eljárás (update) rémálma ugyanis a folyamat alatti zavar vagy tápkiesés – rosszabb rendszerekben ilyenkor a modul szétszerelése (még rosszabbakban a selejtezése) a következő lépés. Mi olyan programrendszer („saját bootloader") beépítésével tudjuk támogatni a folyamatot, melyet olyankor sem törlünk, mikor a firmware update folyamata zajlik. Egy tápellátás-kimaradást követően programrendszerünk tárolója lehet, hogy nem definiált állapotban lesz, bootladerünknek azonban ilyenkor is futásképesnek kell lennie.

Az elmondottak a fejlesztő számára általában azt jelentik, hogy az update alkalmazásonként egyedileg megoldandó feladat. A gyártók által beépített bootloaderek a legritkább esetben képesek kezelni azt az üzemszerűen is rendelkezésre álló kommunikációs felületet (és annak legtöbbször egyedi kommunikációs protokollját), melyről a mikrokontroller gyártójának fogalma sem lehet. Erre a megoldásra koncentrálva tehát leginkább a következő fontosabb szempontokat kell megvizsgálnunk az update lehetőségének biztosításához:

- mikrokontrollerünk programmemóriája (flash) törölhető-e saját programunk által úgy, hogy a törlést csak a memória egy részére alkalmazzuk, miközben a bootloadert tartalmazó rész változatlan kell maradjon?

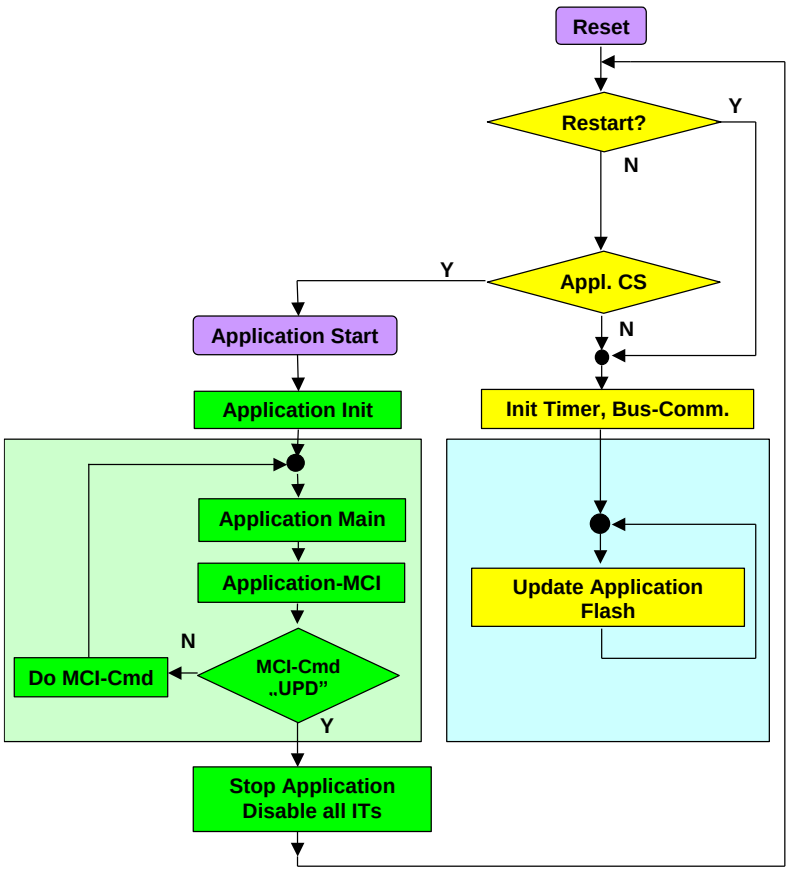
- elvégezhető-e a törlés és a programozás folyamata úgy, hogy közben ugyanazon memóriából programot is futtatunk (Harvard architektúra esetén nem megoldható a bootloader rész RAM-ból való futtatása, ami a működés alatti tápkimaradás miatt is veszélyes lenne)?
- kialakítható-e egy olyan adatátvitel a meglévő kommunikációs protokollon belül, amely lehetővé teszi nagyobb adattömegnek (magának a firmware kódnak) az átvitelét a kommunikációs csatornán?
- hogyan biztosítható egy „sérült” programrendszer esetén a bootloader automatikus indulása a rendszer indítása után a firmware biztonságos betöltése érdekében?



A flash memória programozásának felületei (AT89C51)

A mai korszerű mikrokontrollerek már lehetővé teszik (osztott flash vagy elkülönített segédmemória alkalmazásával) azt, hogy felhasználói program futtatása mellett más tartományba eső flash területek törölhetők ill. programozhatók legyenek (API = Application Programming Interface).

Amennyiben erre a lehetőséget megtaláltuk, már csak saját rendszerünkhöz illeszkedő bootloaderünket kell kialakítanunk. Egy lehetséges példát mutat a következő blokkdiagram:



Bootloader és alkalmazói programrész közös flash memóriában

Egy éppen induló modul („reset”) nem tudván, miért is indul most a programrendszerének végrehajtása, megvizsgálja, hogy újraindították-e őt vagy sem (ld. később). Ha nem, ellenőrzi az alkalmazói programrendszer meglétét és helyességét („Appl. CS”), amelyet az arról letárolt információk (pontos hossz, helyes ellenőrző összeg vagy CRC) birtokában képes elvégezni. Ha helyesnek találja ezeket a jellemzőket, el is indítja az alkalmazást és ezzel a bootloader rész (**sárga színnel jelölve**) be is fejezte működését. Az alkalmazói rész (**zöld színnel jelölve**) saját alapfeladatán túl az üzemszerűen rendelkezésére álló kommunikációs csatornán ciklikusan ellenőrzi, hogy érkezett-e számára firmware frissítési parancs. Ha nem, folytatja működését a modul alapfeladatának végrehajtásával. Ha igen, leállítja az általa beindított modulspecifikus folyamatokat, majd újraindítja a modul programrendszerét (tipikusan JP 0 – a normál Power-On reset utáni belépési pont). Ilyenkor a bootloader rész – észrevéve az újraindítás tényét – kikerüli az alkalmazás beindítását és csak az alkalmazás rész flash memóriájának újraprogramozására lesz hajlandó. Egy esetleges tápkimaradás a törlés vagy a programozás műveletei alatt az alkalmazás ellenőrzőösszegének sérülését kell eredményezze, és ez a legközelebbi indítás után ismét update fázisba vezérli a bootloader programot. Természetesen ez a programrész sohasem törlődik a memóriából, mivel programozása – gyártás után – csak ISP (vagy JTAG) felületen keresztül lehetséges.