Rendszerarchitektúrák laboratórium

Digitális jelfeldolgozó processzorok 1.-2.

# Mérési útmutató

Molnár Károly, Sujbert László

2009.

## 1. Bevezetés

Beágyazott információs rendszerekben gyakran alkalmaznak jelfeldolgozó processzorokat (DSP-ket). Ezeknek általában az a feladatuk, hogy a jelfolyamba beavatkozva ténylegesen valamilyen jelfeldolgozást valósítsanak meg, vagy egy magasabb szintű eljárás számára (pl. beszédfeldolgozás) készítsék elő a jelet. Hasonlóan fontos szerephez jutnak a jelprocesszorok szabályozó rendszerekben, ahol a szabályozó algoritmus a DSP-n fut. Az első mérés során egy fixpontos DSP programozását, fejlesztői környezetét, a jelfeldolgozó programok futásának tesztelését ismerhetjük meg.

A témakörről bővebb információt az alábbi forrásokban található:

**Tárgyak:**

- *Jelfeldolgozó Processzorok Alkalmazása.*
  A tárgy honlapja: http://www.mit.bme.hu/oktatas/targyak/vimm9318/jegyzet/index.html
- *Információfeldolgozás.*
  A tárgy honlapja: http://www.mit.bme.hu/oktatas/targyak/vimim237/
- *Discrete-Time Signal Processing (MIT OpenCourseWare).*
  Link: http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-341Fall-2005/LectureNotes/index.htm

**Letölthető könyvek:**

- *The Scientist and Egineer's Guide to Digital Signal Processing.*
  Link: http://www.dspguide.com
- *Numerical Recipes in C.*
  Link: http://www.nrbook.com/a/bookcpdf.php

A Rendszerarchitektúrák laboratórium tárgyban két mérés foglalkozik a DSP-kkel. Az első mérés során megismerkedünk egy egyszerű jelfeldolgozó rendszerrel és annak fejlesztői környezetével. A második mérés során megismerkedünk egy valós idejű operációs rendszerrel és annak alkalmazását DSP processzoron.

# 2. Elméleti összefoglaló

A DSP-k mikroprocesszorok, és általában tartalmaznak minden olyan funkciót, amelyet egy általános célú mikroprocesszor is tartalmaz. A DSP-k azonban igen fejlett – esetleg különféle célokra több – aritmetikai egységgel is rendelkeznek. A DSP-k utasításkészlete olyan, hogy támogassák a leggyakrabban előforduló műveletek minél gyorsabb végrehajtását. Az alábbiakban a legfontosabb DSP tulajdonságokat tekintjük át.

A mérések során a DSP programozása alapvetően C nyelven történik, de szükség lehet assembly nyelvű kódrészletek megírására is. Ez akkor indokolt, ha a megvalósítandó feladatnak szigorú valós idejű követelményeknek kell megfelelnie, vagy ha a processzor lehetőségeit maximálisan ki akarjuk használni.

## 2.1. Struktúra, memória

A DSP-k a szokásos Neumann-architektúra helyett ún. Harvard-architektúrával rendelkeznek. Ez azt jelenti, hogy külön memória szolgál az adat és a program tárolására, sőt egyes DSP-kben két külön adatmemória is van. Ez a megosztás, és az ezzel járó külön buszrendszer teremti meg a lehetőségét annak, hogy egy utasításciklus alatt fizikailag egy időben lehessen hozzáférni az utasításkódhoz, és akár több operandushoz is. A mérésben szereplő ADSP-BF537 DSP FIR szűrő programjának elemzésekor látni fogjuk, hogy egy utasításciklus alatt a processzor összeszoroz két számot, hozzáadja egy harmadikhoz és a szorzó egység bemeneti regisztereibe beolvassa a memóriából a következő két tényezőt. Ez a *Multiply and Accumulate (MAC)* művelet, amelyet tipikusan minden DSP végre tud hajtani. A művelet egy cikluson belüli végrahajtásának lehetőségét a nagymértékben párhuzamosított architektúra adja. Már a korai DSP-kben is volt önálló szorzó egység, külön címaritmetika (lásd később), stb. Az adatok áramlását több párhuzamos busz szolgálja ki.

A DSP-k fontos eleme a belső vagy on-chip memória. A fenti nagy párhuzamosság ugyanis csak a processzoron belül igaz teljes egészében, ha a DSP külső memóriához fordul, általában már bizonyos hozzáférések csak szekvenciálisan valósíthatók meg. Programozásnál tehát ügyelni kell arra, hogy az egyes változókat, de különösen a tömböket milyen memóriaterületre tesszük. A modern DSP-kben általában néhány tíz Kszó belső memória van, ami elegendő a legtöbb on-line alkalmazáshoz.

## 2.2. Aritmetika

A modern DSP-kben összeadó (kivonó), szorzó, léptető, logikai egységek vannak, ezek biztosítják a vonatkozó műveletek egy utasításciklus alatti végrehajtását. általában létezik egy olyan egység, amely korlátozott pontosságú (pl. 9 bites) osztást is képes végrehajtani. Az ehhez tartozó utasítás egy iterációs osztó rutin alapja lehet. A processzor számábrázolási pontosságának megfelelő osztást kb. 10 utasítás alatt lehet végrehajtani. Az osztás tehát időkritikus művelet, ha előre ismert számmal kell osztani, célszerű inkább a reciprokával szorozni.

Az adott processzorra jellemző, hogy az aritmetikai egységei milyen bitszámú adatot fogadnak, illetve az eredmény milyen bitszámú regiszterben képződik. Ebből a szempontból (is) elkülönülnek a fixpontos és a lebegőpontos processzorok, ezért a továbbiakban ezeket külön tárgyaljuk.

### 2.2.1. Fixpontos számábrázolás

A számábrázolás nevét adó „fix pont” azt jelenti, hogy a „kettedespont”, tehát az egészeket szimbolizáló számjegyeket a törtrésztől elválasztó pont helyzete nem változik, a memóriában vagy regiszterben tárolt szám egy helyi értéke állandó. Ez a helyi érték azonban többféle lehet. Tegyük fel, hogy a szám 16 bites, mint a legtöbb fixpontos DSP-ben. A számábrázolás lehet:

- előjel nélküli egész, ebben az esetben az ábrázolt számok: $0..2^{16}-1$,
- előjeles egész, ebben az esetben az ábrázolt számok: $-2^{15}..2^{15}-1$,
- előjel nélküli törtszám, ebben az esetben az ábrázolt számok: $0..1-2^{-16}$,

- előjeles törtszám, ebben az esetben az ábrázolt számok: $-1..1-2^{-15}$.

Az előjeles számokat általában kettes komplemens módon ábrázolják. Az egész számok ábrázolása triviális. Törtszámok esetében a legfelső bit helyi értéke (ha nem volt előjelbit) 1/2, a legalsóé $2^{-16}$. Előjeles törtszámok esetében a legfelső bit az előjelbit, ezért a legalsó bit helyi értéke $2^{-15}$.

Az, hogy a számábrázolás milyen, nem csupán intervallum-hozzárendelési kérdés. A számábrázolás akkor válik fontossá, ha két számot összeszorzunk, és az eredményt értelmezni akarjuk, illetve helyesen eltárolni vagy továbbszámolni vele. Két bináris szám összeszorzásakor az eredmény bitszáma a két bitszám összege. A követhetőség kedvéért most 3 bites számokon mutatjuk be a számábrázolás jelentőségét.

Tekintsük először az 100, 001 előjel nélküli számokat! Ezek önmagukkal vett szorzata szerepel a következő táblázatban.

| operandus | szorzat egész számként | szorzat törtszámként |
|---|---|---|
| **100** (4, 1/2) | 010 **000** (16) | **010** 000 (1/4) |
| **001** (1, 1/8) | 000 **001** (1) | **000** 001 (1/64) |

A táblázatban a vastagon szedett számok mutatják az eredeti helyi értékeknek megfelelő számmezőt, zárójelben pedig a számok decimális megfelelője szerepel. Látható, hogy az első esetben az egész, a második esetben a törtszámábrázolás kivezet az ábrázolt tartományból. Egész számábrázolás esetén a magasabb, törtszámábrázolás esetén az alacsonyabb helyi értékek felé „nő" a szorzat bitszáma. Egész számábrázolás esetén általában az alsó, törtszámábrázolás esetén a felső szót visszük tovább.

Most tekintsük a 010-át mint előjeles számot! A következő táblázatban ismét az önmagával vett szorzat szerepel. Jól látható, hogy az előjel nélküli esethez képest újabb probléma vetődött fel:

| operandus | szorzat egész számként | szorzat törtszámként |
|---|---|---|
| **010** (2, 1/2) | 000 **100** (4) | **000** 100 (1/8) **!** |
| | | **001** 000 (1/4) |

a szorzás eredménye itt nem is ad jó „bitmintát", a helyes eredményhez (amely a második sorban szerepel) el kell tolni a szorzatot egy helyi értéket balra.

A szorzó egységben tehát a számábrázolástól függő operációt kell végrehajtani. A legtöbb processzoron programozható, hogy a szorzó egység milyen számként kezelje az operandusokat. Jelfeldolgozási célra legtöbbször az előjeles törtszámábrázolást alkalmazzuk, ugyanis ez illeszkedik jól a fizikai képhez (a jel az A/D átalakító tartományán belül van), és a szorzat figyelembe nem vett része hiba jellegű mennyiség. általában lehetőség van kerekítésre is, legtöbbször szintén többféle kerekítés állítható be.

Fixpontos számábrázolás esetén a 16 bit – mint említettük – általános, de nagyobb pontossági igények esetén nem kielégítő. Ilyen problémák merülnek fel pl. IIR szűrők megvalósításakor. Léteznek olyan DSP-k, amelyek magasabb bitszámú (pl. 24, 32 bites) szavakkal dolgoznak.

C-ben történő programozás esetén a fordító elrejti előlünk ezeket a részleteket, és garantáltan jól működik, amennyiben szabványos C adattípusokkal dolgozunk (`float, double, int, long, short`). Ezek használata azonban sokszor közel sem optimális, mert pl. 32 bites lebegőpontos (`float`) számok szorzása a 16 bites fixpontos processzoron akár több száz assembly utasításra fordul le. A 16 bites processzorhoz leginkább a short típus használata illeszkedik (16 bites előjeles egész szám).

A szabványos C-ben nincs meg a fent említett törtszám ábrázolás, ezért az Analog Devices definiálta a `fract16` és `fract32` típusokat, amelyek 16 illetve 32 bites törtszámok ábrázolására hivatottak. Ezek továbbra sem szabványos C típusok, tehát pl. két fract16 összeszorzása a $*$ operátorral helytelen eredményre vezet! A probléma megoldására az Analog Devices biztosít C könyvtári függvényeket, amelyekkel a `fract16` és `fract32` típusú számok közötti műveletek elvégezhetők.

### 2.2.2. Lebegőpontos számábrázolás

A lebegőpont azt jelenti, hogy az ábrázolandó számot $m2^e$ alakban ábrázoljuk, ahol $m$ jelenti a mantisszát, $e$ az exponenst. Az exponenst mindig úgy állapítják meg, hogy a mantissza 0.5..1 közötti szám legyen. Mind az exponens, mind a mantissza általában előjeles szám, de természetesen a különböző kombinációk is előfordulnak. Ebben az esetben egész és törtszámok vegyesen is ábrázolhatók. Pl. a 2.5 decimális számot – 3-3 bites előjel nélküli exponenst és mantisszát feltételezve – 101 $2^{010}$ alakban ábrázolhatjuk (101 $2^{010} = (\frac{1}{2} + \frac{1}{8}) \cdot 2^2 = 0.625 \cdot 4 = 2.5$).

A lebegőpontos számábrázolással részleteiben nem foglalkozunk. Ennek oka egyfelől, hogy a mérés keretében nem foglalkozunk lebegőpontos DSP-vel, másrészt a lebegőpontos számábrázolás jóval kevesebb problémát vet fel, mint a fixpontos. Az is igaz viszont, hogy amennyiben probléma adódik a számábrázolással, az analízis és a megfelelő megoldás megtalálása nehéz feladat. (Gondoljunk csak arra, hogy a kvantálás itt nem egyenletes.)

A modernebb lebegőpontos processzorok megfelelnek az *IEEE* 754-es szabványában leírt formátumoknak, és igyekeznek az ott leírt kerekítési módokat alkalmazni. (pl. *IEEE single precision*: 9 bites előjeles exponens és 24 bites előjeles mantissza. Mivel a mantissza vezető bitje mindig 1, azt nem ábrázolják fizikailag, így 32 bites szavak elegendőek.)

C programokban megengedett a szabványos lebegőpontos számok használata (`float, double`), de a 2.2.1 pontban említett erőforráskihasználási okokból ezek használata nem célszerű.

## 2.3. Címzés

A DSP-kben is létezik direkt és indirekt címzés. Az alábbiakban az indirekt címzésről lesz szó.

A korábban említett nagy párhuzamosság eléréséhez az indirekt címzés és a vele összekapcsolódó címaritmetika is hozzájárul. Az indirekt címzésre néhány (általában 8..16) regiszter áll rendelkezésre. A címregiszterek értékének módosítására a címaritmetikai egységben kerül sor. A módosítás lehet inkrementálás/dekrementálás (címzés előtt vagy után); vagy más, a címregiszterekhez rendelt egyéb regiszterek alapján történő módosítás. Általában két ilyen „hozzárendelt" regiszter van: egy módosító vagy ofszet regiszter, amelynek 1-től különböző értékével lehet módosítani a címregisztert; illetve a címregiszter által címzett ún. cirkuláris buffer hosszát meghatározó regiszter.

A cirkuláris buffer egy olyan memóriaterület, amelynek elemeit ciklikusan olvassuk ki vagy ciklikusan írunk bele. Ilyen buffer lehet pl. egy FIR szűrő együtthatóit tartalmazó buffer, amelyre minden kimeneti minta kiszámolásakor szükség van. A címaritmetika gondoskodik arról, hogy a cirkularitás megvalósítása ne igényeljen plusz utasítást.

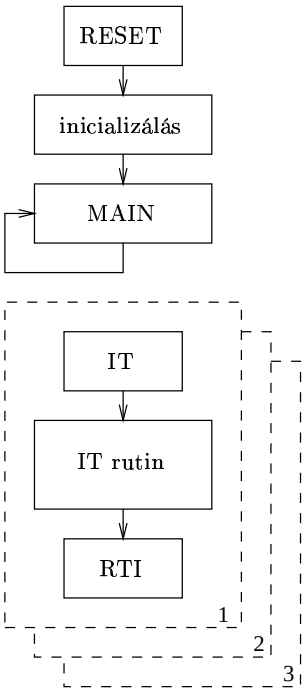
A diszkrét Fourier-transzformáció kiszámítására legtöbbször valamilyen FFT algoritmust alkalmaznak. Ennek tulajdonsága, hogy vagy a bemeneti, vagy a kimeneti vektor ún. bitfordított sorrendben van. Ehhez a DSP-k támogatást adnak, vagy a címbusz bitjeinek fizikai megfordításával, vagy a kezdőcím ügyes módosításával.

## 2.4. Programszervezés

Jelfeldolgozó programok gyakran tartalmaznak olyan ciklusokat, amelyek sokszor futnak le két mintavétel között. Ezeknél a ciklusváltozó vizsgálata nagyon lassítaná a program futását. Ilyen esetekben hasznos a DSP-k hardver alapú ciklusszervezése, amely azt jelenti, hogy van egy dedikált hadver egység, amely a ciklusok gyors futtatását biztosítja. Ennek használatakor nem kell külön gondoskodni ciklusváltozóról, csak a ciklus elejét és végét, valamint a ciklus lefutásainak számát kell megadni. C programokban előforduló `for` ciklusok esetén a fordító megkísérli kihasználni ezt a lehetőséget.

A gyors végrehajtást szolgálják a párhuzamos utasítások. Általában azok az utasítások, amelyek külön egységet és külön buszt használnak, párhuzamosan is végrehajthatók.

A DSP-kben fontos szerepe van a megszakításoknak. Legtöbbször az A/D átalakító is megszakítást kér, ha egy újabb érvényes adat van a kimenetén. A megszakítások vektorosak, prioritásuk a forráshoz kötött, de egyes esetekben programozható.




1. ábra. Tipikus DSP programszervezés

A jelfeldolgozó processzorok programszervezésének alapesetét mutatja az 1. ábra. A program futása az inicializáló résszel kezdődik: az inicializálás jelenti a processzor beállításainak rögzítését (pl. IT prioritások, számábrázolási mód), de itt kapnak kezdőértéket a változók is. Az inicializálás alatt a megszakítsok tiltva vannak.

Miután az összes beállítás megtörtént, és a megszakítások is engedélyezve lettek, a „főprogram" következik, amely azonban egy üres ciklus, amely nem csinál semmit, csak magára ugrik vissza. Amíg megszakítás nem érkezik (pl. az A/D átalalkító felől), addig a processzor vár. Megszakítás esetén a processzor a kérést kiszolgálja, majd visszatér a „főprogram"-ba.

Jelfeldolgozó programoknál az adatvesztés úgy kerülhető el, hogy az IT rutin biztosan hamarabb lefut, mint hogy a következő IT kérés megérkezik. A jelfeldolgozó algoritmusok (átlagolás, szűrés, valamilyen szabályozási feladat) mindig felírhatók úgy, hogy egy tetszőleges időpontban a bemenet aktuális és korábbi értékeiből hogyan határozható meg a kimenet aktuális mintája. Például egy 4-minta hosszú csúszó ablakos átlagolás így írható fel:

$$y[k] = 0.25 * (x[k] + x[k-1] + x[k-2] + x[k-3])$$

ahol $y[.]$ a kimeneti, $x[.]$ a bemeneti mintákat jelöli. Az IT kérés ilyen esetekben az A/D átalakítótól érkezik, amikor egy új minta rendelkezésre áll. Ekkor zz IT rutinban kell az algoritmust megvalósítani.

Ilyen programszervezés mellett az indokolt, hogy a RESET után egyszer lefutó inicializálást C nyelven írjuk meg, mivel ennek futási ideje nem kritikus. A jelfeldolgozást végző rutin esetén viszont előfordulhat hogy a C nyelven írt rutin futási ideje nagyobb lenne mint a mintavételi idő, ekkor ezt assembly nyelven kell megírni, törekedve a processzor párhuzamosítási és egyéb speciális hardver gyorsítási lehetőségeink teljes kihasználására.

Az ilyen egyszerű programszervezés természetesen csak akkor használható a gyakorlatban, ha a DSP a jelfeldolgozó algoritmuson kívül nem végez számottevő más tevékenységet. A technológia fejlődésével a DSP-k képességei is folyamatosan bővülnek, így természetes hogy egyre több és komplexebb feladatot végeznek el. Továbbá megfigyelhető a DSP processzorok és az általános célű mikorvezérlők konvergenciája, ezek az eszközök egyre több hasonlóságot mutatnak. Ennek

megfelelően a DSP-k el tudják látni egy hagyományos mikrovezérlő feladatait is a jelfeldolgozási algoritmusok mellett. Ekkor a fent vázolt programstruktúra nem elegendő, valamilyen átgondoltabb, bevált programszervezési módszerre van szükség (pl. IT-vel kiegészített ciklikus programszervezés, ütemezett függvények módszere, valós idejű operációs rendszer). A második mérés során az Analog Devices saját valós idejű operációs rendszerével a VisualDSP Kernel-lel (VDK) fogunk megismerkedni.

# 3. Az ADSP BF537 jelfeldolgozó processzor

**Felhasználói kézikönyvek**

- *ADSP-BF537 Blackfin Processor Hardware Reference.*
  Link: http://www.analog.com/static/imported-files/processor_manuals/bf537_hwr_Rev3.2.pdf
- *C/C++ Compiler and Library Manual for Blackfin Processors.*
  Link: http://www.analog.com/static/imported-files/software_manuals/50_blackfin_cc.rev5.1.pdf
- *Kernel (VDK) User's Guide.*
  Link: http://www.analog.com/static/imported-files/software_manuals/50_vdkm_rev3.2.pdf

A processzor az Analog Devices cég terméke, a Blackfin család tagja. Ez a család 16-bites fixpontos DSP-ket tartalmaz, amelyek már rendelkeznek mindazokkal a képességekkel amelyeket egy általános célú mikrovezérlőtől is elvárunk. Az előző fejezetben említett konvergencia a Blackfin DSP család esetében teljesen nyilvánvaló, pl. a gyártó ezt a családot már nem is DSP-nek, hanem Embedded Processor-nak nevezi.

Az alábbiakban az elméleti összefoglalóban megismert sorrendben tekintjük át a tulajdonságait. A programozáshoz, hibakereséshez természetesen szükség van a felhasználói kézikönyvekre, itt csak a legfontosabb ismereteket közöljük.

## 3.1. Struktúra, memória

A processzor egy program- és egy adatmemóriát címez meg. A processzor 48 KByte belső programmemóriát és 64 KByte adatmemóriát tartalmaz. Az utasítások 16 és 32 bitesek lehetnek, az adatok pedig 8, 16 vagy 32 bitesek. A processzor on-chip memóriájával bővebben a programozási kézikönyv foglalkozik.

## 3.2. Aritmetika

Az aritmetikai műveleteket 3 különböző egység (ALU – aritmetikai-logikai, MAC – szorzó-összeadó, Shifter – léptető) végzi. Ezek részletes leírása a programozási kézikönyvben található meg. Assembly programozás esetén nagyon fontos ezen kívül a 15. fejezetben bemutatott szintaktika betartása, mert esetenként nagyon kötött hogy az egyes műveletekhez melyik regiszterek használhatók.

## 3.3. Címzés

A címzésre használható regisztereket csak assembly nyelvű programozás esetén kell közvetlenül kezelnünk, C programozás esetén a fordító gondoskodik erről.

Az indirekt címzésre a P0..5 regiszterek használhatók. Cirkuláris bufferek megvalósítására a B0..3, I0..3, L0..3 és az M0..3 regisztereket kell használni. A buffer kezdőcímét az egyik B (Base) regiszterben, hosszát a hozzá tartozó L (Length) regiszterben, a léptetés értékét pedig valamely M (Modify) regiszterben kell megadni. Az I (Index) regiszterben található az éppen aktuális memóriacím, tehát léptetéskor ez a cím változik az M módosító regiszter értékével.

## 3.4. A fejlesztőrendszer

A DSP kártya programozása a VisualDSP++ integrált fejlesztői környezetben történik. Ennek egyik fő része a projekt editor, ahol a DSP program forrásfájljainak szerkesztése történik. Az egymáshoz rendelt forráskód fájlok alkotnak egy projektet, amelyet a VisualDSP++ fordít le, linkel és egy DSP-re letölthető fájlt generál. A forráskód fájlok készülhetnek assembly, C vagy C++ nyelven. A Linker számára az ún. Linker Description File (.ldf) tartalmaz utasításokat.

A fejlesztői környezet másik fontos része a debugger, amely lehetőség biztosít a megírt DSP programok kipróbálására futás közben. Lehetőség van a program futásának részletes vizsgálatára, töréspontok helyezhetők el, lehet léptetve végrehajtani az utasításokat, a memóriaterületek valamint a rendszer regisztereinek tartalma folyamatosan monitorozható. A debugger "bementi adata" a projekt editor által előállított, a DSP-re letölthető fájl. A DSP program futtatható szimulátorban, vagy emulátor segítségével fizikai hardveren is. Ennek kiválasztása a Session menüben történik.

# 4. Laboratórium: 1. mérés

A mérés során az ADSP BF537 EZ-KIT Lite fejlesztői kártyát programozzuk emulátor segítségével. A kártya USB porton csatlakozik a PC-hez. A DSP kártya tartalmaz egy ADSP BF537 600 MHz-es DSP-t, egy AD1871 sztereo DAC-t, valamint egy AD1854 sztereo ADC-t, amelyek jelei Jack aljzatokra vannak kivezetve. Mind az ADC, mind a DAC 48 kHz mintavételi frekvencián üzemel. A kártya sok egyéb funkcionális egységet is tartalmaz, de a mérés szempontjából csak ezek lényegesek.

## 4.1. Példa alkalmazások

A példa alkalmazások bemutatják az ADSP-BF537 legfontosabb utasításainak használatát és a programozási modellt. A programok ismertetése kapcsán kitérünk néhány, a fejlesztőrendszerre jellemző tulajdonságra is.

Az első példa a LabFrame projekt. Nyissuk meg a projektfájlt (LabFrame.dpj). A projekthez tartozó forrásfájlok az alábbiak (a fontosabbak forráskódját ld. a 4.2. fejezetben):

- **main.c** – Ez a főprogram, RESET után ez fut le először. Meghívja az inicializáló függvényeket, amelyben a megszakítási rutinok is inicializálásra kerülnek, majd végtelen ciklusba kerül.
- **Initialize.c** – A hardver inicializását végző kódot tartalmazza.
- **ISR.c** – A különböző megszakításokhoz rendeli hozzá a kiszolgáló rutinokat. Jelen esetben csak az AD-hoz tartozó megszakítást rendeli hozzá a Process_data függvényhez, azaz ha az AD-n egy új minta áll elő, akkor meghívódik a Process_data függvény.
- **Process_data.c** – Ez tartalmazza a Process_data függvényt, amely tehát a minták feldolgozását végzi. Ebben a függvényben célszerű megvalósítani a jelfeldolgozó algoritmust. Jelen esetben a bejövő mintákatat változtatás nélkül átmásoljuk a kimenetre, azaz a program egy egyszerű jeltovábbítást hajt végre.

A projekt fordításához a parancssorból a Build All parancsot kell választani, ami lefordítja a projektet. Ha a Session megfelelően van beállítva (ADSP BF-537 EZKIT lite via Debug Agent), akkor sikeres fordítás esetén le is tölti a programot a DSP-re. Ekkor látjuk hogy a futás megáll az Init függvénynél, és a processzor Halt állapotba kerül. (A processzor állapotát a képernyő alján, jobbra láthatjuk.) Ekkor a Run parancs kiválasztásával futtathatjuk a programot. A futás ezután megállítható, töréspontot helyezhetünk el, stb. A Debug menüben pedig a processzor memóriáját, regisztereit, változóit vizsgálhatjuk.

Második példánk a LabFrame_FIR_3 projekt (LabFrame_FIR_3.dpj), ami az előzőtől csak a Process_data függvényben különbözik. Ez az alkamazás egy FIR szűrőt valósít meg. A FIR szűrő algoritmusa a következő:

$$y(n) = \sum_{i=0}^{N-1} w(i)x(n-i),$$

ahol $y(n)$ a kimenet az $n$. időpillanatban, $x(n-i)$ jelenti a bemenet aktuális és késleltetett értékeit, $w(i)$ pedig a szűrőegyütthatók. A szűrőnek tehát $N$ együtthatója van. A fenti képlet szerinti számítást kell tehát a DSP-nek elvégeznie a megszakítást kiszolgáló rutinban. A fenti művelet, az ún. diszkrét konvolúció kitüntetett jelentőségű a digitális jelfeldolgozásban. Használatáról lásd pl. *The Scientist and Engineer's Guide to Digital Signal Processing* könyv 6. fejezetét (http://www.dspguide.com/).

- **Process_data_FIR_asm.c** – Ez a fájl tartalmazza a beérkezett adatok feldolgozását végző `Process_data()` függvényt. A bementi adatokat az input tömbben tárolja, a szűrőegyütthatókat pedig a coefs tömbbe olvassa be egy külső fájlból. A `section` előtagok biztosítják hogy a két tömb különböző memóriaterületre kerüljön, hogy egyidejűleg lehessen olvasni őket a konvolúció során. A konvolúciót a `conv_asm()` függvény végzi, amelynek forráskódja a conv_asm.asm fájlban található.

- **conv_asm.asm** – A `conv_asm()` függvény assembly nyelven készült, annak érdekében hogy kihasználja processzor párhuzamosítási lehetőségeit a számítás során. A függvény részletes működése a kommentekből érthető meg.

## 4.2. Forrásfájlok

## 4.3. main.c

```
/*********************************************************************/
#include "LabFrameFIR.h"
#include <sysreg.h>
#include <ccblkfn.h>
/*********************************************************************/
Variables

Description:  The variables ChannelxLeftIn and ChannelxRightIn contain
the data coming from the codec ADC (AD1871).  The (processed)
playback data are written into the variables
ChannelxLeftOut and ChannelxRightOut respectively, which
are then sent back to the DAC (AD1854) in the SPORT0 ISR.
*********************************************************************/
// left input data from AD1871
int iChannel0LeftIn, iChannel1LeftIn;
// right input data from AD1871
int iChannel0RightIn, iChannel1RightIn;
// left ouput data for AD1854
int iChannel0LeftOut, iChannel1LeftOut;
// right ouput data for AD1854
int iChannel0RightOut, iChannel1RightOut;
// SPORT0 DMA transmit buffer
int iTxBuffer1[2];
// SPORT0 DMA receive buffer
int iRxBuffer1[2];

//-------------------------------------
// Function:  main()
// Description:  After calling a few initalization routines, main() just
// waits in a loop forever.  The code to process the incoming
// data can be placed in the function Process_Data() in the
// file "Process_Data.c".
//-------------------------------------
void main(void)
{
         Init_Flags();
         Audio_Reset();
         Init_Sport0();
         Init_DMA();
         Init_Interrupts();
         Enable_DMA_Sport0();

         while(1);
}
```

### 4.4. Process_data.c

```
#include "Talkthrough.h"
//--------------------------------------------------------------------------
// Function:  Process_Data()
//
// Description:  This function is called from inside the SPORT0 ISR every
// time a complete audio frame has been received.  The new
// input samples can be found in the variables iChannel0LeftIn,
// iChannel0RightIn, iChannel1LeftIn and iChannel1RightIn
// respectively.  The processed data should be stored in
// iChannel0LeftOut, iChannel0RightOut, iChannel1LeftOut,
// iChannel1RightOut, iChannel2LeftOut and iChannel2RightOut
// respectively.
//--------------------------------------------------------------------------
void Process_Data(void)
{
         iChannel0LeftOut = iChannel0RightIn;
         iChannel0RightOut = iChannel0RightIn;
}
```

## 4.5. Process_data_FIR_asm.c

```
#include "LabFrameFIR.h"
#define N 401
         // Szűrőegyütthatók száma
section("L1_data_b") fract16 coefs[N] = #include "bp_fract16.dat";
         // Szűrőegyütthatők beolvasása külső fájlból
section("L1_data_a") fract16 input[N];
         // A "section" előtag szabja meg, hogy a deklarált változó melyik
         // memóriterületre kerüljön.  A coef és input buffereket érdemes
         // különböző területekre tenni, hogy a konvolúciónál párhuzamosan
         // lehessen beolvasni az értékeiket.
int i = 0;
extern int conv_asm( fract16 *, fract16 *, fract16 * );
//-------------------------------------//
// Function:  Process_Data()
//-------------------------------------//
void Process_Data(void)
{

         iChannel0LeftOut = iChannel0RightIn;
                  // A bal kimeneti csatornán változtatás nélkül
                  // megjelenik a jobb bemeneti csatorna jele

         input[i] = iChannel0RightIn >> 8;
                  // Konverzió 24-ről 16 bitre
         fract16 out = 0;

         out = conv_asm(coefs,&input[i],input);
                  // Konvolúció számítását végző függvény meghívása

         i++;
                  // Számláló növelése
         if (i==N){
                  i = 0;
                  }
                  // Számláló nullázása

         iChannel0RightOut = out <<8;
                  // Konverzió 16-ról 24 bitre }
```

### 4.6. conv_asm.asm

```
// ****************************************************** //
// ASSEMBLY CONVOLUTION FUNCTION
// Created by Karoly Molnar 2006.
// Declaration:
// extern int conv_asm( fract16 *, fract16 *, fract16 * );
// Usage:
// out = conv_asm(coefs,&input[i],input);
// ****************************************************** //
// Fontos megjegyzes:
// A két bemeneti vektor hosszát manuálisan kell megadni az
// N_TAPS konstans beállításával
// ****************************************************** //

#define N_TAPS 401
.section L1_code;
         // A kód az L1_code memóriaterületre kerül
.global _conv_asm;
         // A függvény globálisan hívható

_conv_asm:
         // Címke, a függvény kezdőcíme
         // A C függvény argumentumait az ASM függvény az R0, R1, R2
         // regiszterekben kapja meg (előírás)

         P0 = R0; // A P0-ba indexregiszterbe kerül a coeffs kezdőcíme
         I0 = R1; // Az I0 cirkuláris bufferbe az aktuálisan bejött minta címe
         B0 = R2; // B0-ba az input buffer kezdőcíme
         P1 = 2 * N_TAPS;
                  // A 2-vel szorzás oka, hogy 16 bites adatokat használunk
         L0 = P1;
         M0 = 1;
         R0 = 0;
         NOP;
         NOP;

         R1 = W[P0++] (Z);
                  // Indirekt címzés, R1-be íródik a P0 által mutatott érték
                  // majd P0 értéke egyel nő
         R2.l = W[I0++];
                  // Cirkuláris buffer használata, R2-be íródik az I0
                  // által mutatott érték majd P0 értéke egyel nő

         A0 = 0;  // Az akkumuláló regiszter nullázása
         P1 = N_TAPS - 1;
                  // N-1-re kell állítani, mert a ciklust N-1-szer futtajuk le

         LSETUP (mac_loop,mac_loop) LC0 = P1;
                  // Hardveres ciklusszervezés, a ciklus eleje és vége is
                  // ugyanaz a címke (mac_loop), azaz 1 utasítás van a ciklusban
```

```
        mac_loop:
        A0 += R1.l * R2.l || R2.l = W[I0 ++] || R1 = W[P0++](Z);
                  // Egy utasításon belül 1 szorzás, 1 összeadás
                  // és 2 memóriamozgatás (MAC)
        A0 += R1.l * R2.l;
                  // Az utolsó szorás és összeadás már cikluson
                  // kívüli, mert ekkor már nincs memórimozgatás
        R0.l = A0;
                  // Az visszatérési értéket az R0-regiszterben kell
                  // tárolni (előírás)
        RTS;

_conv_asm.end:
                  // Címke, a függvény vége
```

## 4.7. Mérési feladatok

Az alábbi mérési feladatok közül a mérésvezetővel egyeztetve egyet kell kiválasztani és megoldani.

1. *Decimáló FIR szűrő megvalósítása.* Ebben az esetben a kártya kimenetén a matematikai mintavételi frekvencia kisebb, mint a bemenetén (pl. harminckettede). Ha a bemenetre adott jel kihasználta a teljes sávszélességet, egy szűrővel korlátoznunk kell a kimeneti mintavételi frekvencia szerint. A feladat szebb megoldása adható az eredeti programozási modell átszervezésével, ha a szűrő rutin a főprogramban fut.

2. *LMS algoritmus megvalósítása.* A referenciajel egyrészt a kártya egyik bemenetére, másrészt egy ismeretlen rendszer bemenetére kerül. Az ismeretlen rendszer kimenetét a kártya másik bemenetére vezetjük. A kártya valamelyik kimenete az algoritmus hibajele, amit oszcilloszkópon figyelhetünk meg. Az LMS-algoritmusról részletesebben a „Digitális jelfeldolgozás" c. tárgyban esett szó.

3. *Periodikus jelek (pl. háromszög-, fűrész-, négyszögjel) előállítása néhány Fourier-komponenséből.* Ehhez rendelkezésre áll egy C könyvtári függvény, amely egy tetszőleges szög szinuszát adja vissza. Ezt ciklikusan meghívva és a kártya kimenetére téve szinuszjelet kapunk. Ha a kiolvasás sebessége változik, a szinuszjel frekvenciája is változik. A kiolvasási sebességeket összehangolva a felharmonikusok előállíthatók, ez utóbbiak súlyozott összegzésével pedig előáll a kívánt periodikus jel.

4. *Fáziszárt hurok (PLL) megvalósítása.* A feladat megoldásához rendelkezésre áll egy C könyvtári függvény, amely egy tetszőleges szög szinuszát adja vissza. Ennek felhasználásával (a számábrázolási pontosságon belül) tetszőleges frekvenciájú szinuszjel előállítható. A feladat ennek a szinuszjelnek a frekvenciáját úgy változtatni, hogy az megegyezzen a bemenetre adott periodikus jel frekvenciájával.

5. *Egyszerű dallamot játszó program megírása.* A feladat megoldásához rendelkezésre áll egy C könyvtári függvény, amely egy tetszőleges szög szinuszát adja vissza. Ennek felhasználásával (a számábrázolási pontosságon belül) tetszőleges frekvenciájú szinuszjel előállítható. A feladat kidolgozása során ügyelni kell arra, hogy a játszott dallam könnyen változtatható legyen.

6. *Logaritmikus sweep-generátor megvalósítása.* A sweep-generátor olyan függvény- (általában szinusz-) generátor, amelynek frekvenciája egy adott frekvenciaintervallumban változik. Logaritmikus sweep-generátor esetében a frekvencia exponenciálisan változik, azaz logaritmikus léptékben ábrázolva lineárisan. A feladat megoldásához rendelkezésre áll egy C könyvtári függvény, amely egy tetszőleges szög szinuszát adja vissza. Ennek felhasználásával (a számábrázolási pontosságon belül) tetszőleges frekvenciájú szinuszjel előállítható. A frekvenciát érdemes a néhány 100 Hz nagságrendjébe beállítani, így a keletkezett jel hangszórón is ellenőrizhető.

7. *Valószínűségsűrűségfüggvény-mérő megvalósítása.* A kártya bemenetére adott jel valószínűség-sűrűségfüggvényét kell oszcilloszkópon megjeleníteni. A feladat két részből áll: egy tömbben tárolni kell a bemenő jel hisztogrammját, és ezt a tömböt periodikusan ki kell olvasni a kimenetre, amit oszcilloszkópra vezetünk.

8. *Medián szűrő megvalósítása.* Az $N = 2K + 1$ hosszúságú ($K$ egész szám) medián szűrő a bemenetre érkezett utolsó $N$ mintát dolgozza fel, és számolja ki az aktuális kimenetet (csakúgy, mint a FIR szűrő). Az aktuális kimenet számításához nagyság szerint sorba kell rendezni a mintákat, és a kimenet a rendezett $N$ elemű tömb $K + 1$. (középső) eleme.

9. *FFT analizátor.* Adattömb FFT-jét kiszámító C könyvtári függvény segítségével kell előállítani egy bejövő jel spektrumát, és azt a kimeneten oszcilloszkópon megjeleníthető formában kiadni.

10. *Saját feladat kidolgozása.* Lehetőség van arra is, hogy a mérőcsoport saját ötletét valósítsa meg. A feladatot előzetesen egyeztetni kell a mérésvezetővel, hogy annak nagyságrendje megfeleljen a mérésnek (ne legyen sem túl egyszerű, sem túl bonyolult).

# 5. Laboratórium: 2. mérés

A mérés során megismerkedünk a VisualDSP Kernel (VDK) valós idejű operációs rendszerrel, majd egy példaprogramot tekintünk át a már ismert ADSP BF537 EZ-KIT Lite fejlesztői kártyán. Végül egy saját feladat önálló megoldására kerül sor.

Az elméleti összefoglalóban láttuk, hogy a Blackfin DSP család tagjai már rendelkeznek mindazokkal a képességekkel és erőforrásokkal amelyeket egy általános célú beágyazott rendszerekben használatos processzortól elvárhatunk. Ilyen módon egyidejűleg több különböző és önmagában is komplex feladatot végez el a DSP (jelfeldolgozó algoritmusok, vezérlési feladatok, kommunikációs feladatok, felhasználói interfész, különböző perifériák kezelése). Ennek szoftveres támogatásához szükséges valamilyen tudatos szoftvertervezési módszer, a különböző feladatok kézi összefésülése ("spagetti kód") a legegyszerűbb esetektől eltekintve nem elegendő. Ennek egyik oka az ilyen kód skálázhatatlansága. A másik probléma a nehéz debuggolás, ugyanis a különböző szálakat nem lehet a többitől függetlenül debuggolni valós futási környezetben. Ilyen módon a valós időzítési viszonyokat nehéz kideríteni. Harmadik problémát a valós idejű követelmények jelentik, szisztematikus tervezés nélkül ennek biztosítása reménytelen.

Kijelenthető tehát, hogy DSP alapú beágyazott rendszerek esetén is egy bizonyos bonyolultság felett szükséges valamilyen operációs rendszert használni. Ez minimum egy saját fejlesztésű keretprogram, de biztonságosabb és gyorsabb egy készen rendelkezésre álló kipróbált, támogatott operációs rendszert használni. A Blackfin család esetén több operációs rendszer közül választhatunk:

- uClinux
- $\mu$C/OS II
- ThreadX
- RTXC Quadros
- Unicoi Fusion
- Nucleus
- Green Hills Software ($\mu$-velOSity, velOSity, INTEGRITY)

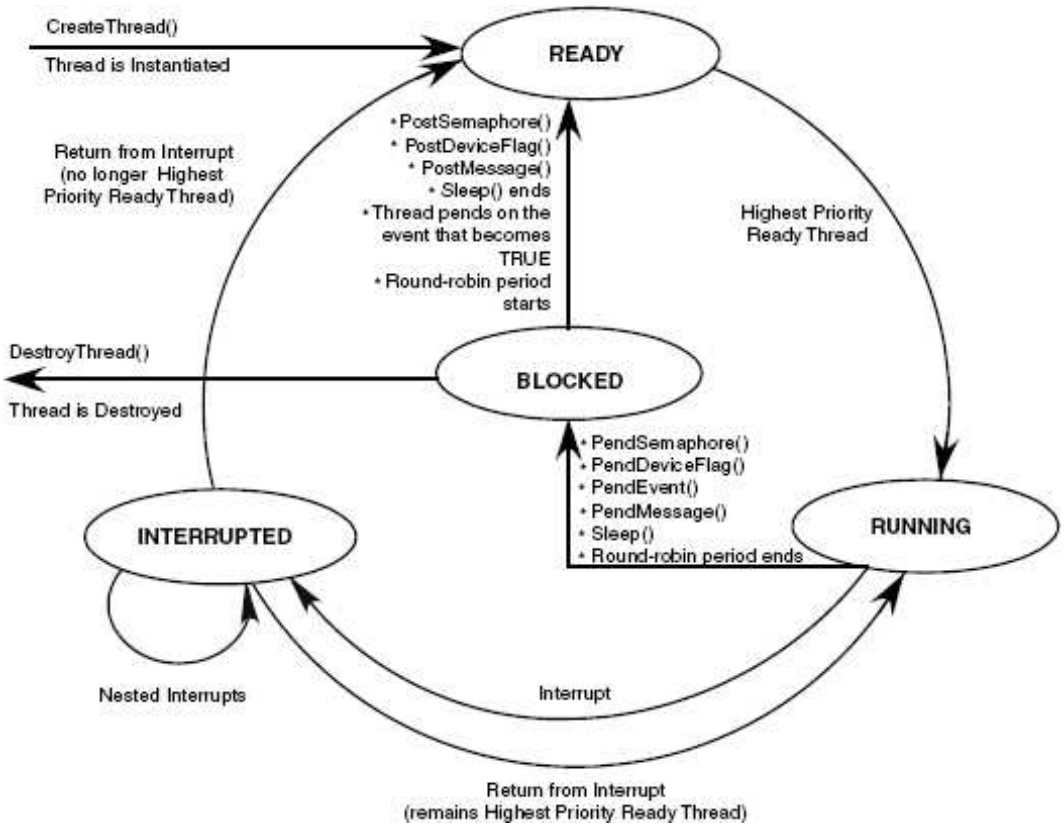
## 5.1. A VisualDSP Kernel (VDK)

A VisualDSP Kernel (VDK) az Analog Devices cég saját fejlesztésű valós idejű operációs rendszere. A VDK portolva van az Analog Devices több DSP családjára:

- Blackfin (ADSP-BFxxx)
- SHARC (ADSP-21xxx)
- TigerSHARC (ADSP-TSxxx)

A VDK a VisualDSP++ fejlesztői környezetbe szervesen integrálva van, így a VDK mint operációs rendszer választása előnyös ha amúgy is a gyári VisualDSP++ fejlesztői környezetet használjuk.

Ha a VisualDSP++ fejlesztői környezetet megvásároltuk, akkor a VDK használata már ingyenes, royalty-mentes. A VDK "szorosan" van integrálva a fejlesztői környezetbe: ha egy új VDK projektet "varázsló" segítségével hozunk létre, akkor ez automatikusan generálja számunkra a szükséges fájlok vázait. A szoros integráció másik előnye, hogy a fejlesztés során külön ablakokban történik a VDK elemek megjelenítése, amely főleg debugolás során jelent nagy segítséget. A VDK forrásfájlok C, C++ és assembly nyelvűek lehetnek. (A mérés során C-ben fogunk programozni.) A VDK támogatja mindazokat a funkciókat amelyeket minden RTOS-tól elvárunk, de az egyes fogalmakat esetleg máshogy nevezik mint egyéb rendszerekben.



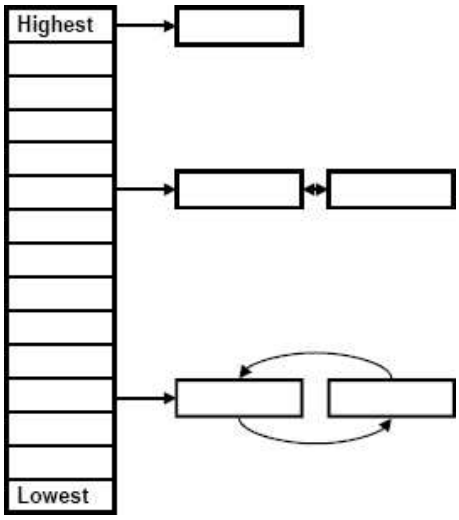

2. ábra. Szálak állapotai

### 5.1.1. Szálak (Threads)

VDK-ban végrehajtási szálnak (*thread*) nevezzük azon programrészeket amelyek egy összefüggő feladatot végeznek el (máshol ugyanezt taszknak nevezik). Az egyes szálak egymástól alapvetően függetlenül léteznek, egymással csak az operációs rendszer eszközein keresztül kommunikálhatnak (pl. szemaforok). Az objektum-orientált programozáshoz hasonlóan egy szál egy objektumhoz hasonlít, azaz a kódolás során nem magukat a szálakat definiáljuk, hanem szál típusokat (*thread type*), majd futás közben tetszőleges számú szálat hozhatunk létre egy adott típusból. A VDK nem korlátozza hogy egyidejűleg hány szál futhat, ennek csak a rendelkezésre álló belső memória szab határt. A szálak létrehozása bootoláskor történhet vagy futás közben. Logikus VDK előírás, hogy legalább egy szálat bootoláskor kell létrehozni.

Az egyes szálakat az egyéni ThreadID-k alapján tudjuk megkülönböztetni. Minden létrehozott szál saját stack-el rendelkezik, ahol a hívási paraméterek vannak eltárolva, valamint a C fordító is ide helyezi a szál lokális változóit. Ilyen módon az egyes szálak lokális változói nem keveredhetnek össze.

Minden szálhoz tartozik 4 alapvető függvény, ezen függvények vázai automatikusan generálódnak a VisualDSP++ fejlesztői környezetben a szál definiálásakor. Ezek az alábbiak: `InitFunction`, `DestroyFunction`, `RunFunction`, `ErrorFunction`. Az első kettő hasonló a C++-ban megszokott konstruktor és desktruktor függvényekhez. A szál futási idejének nagy részében a `RunFunction` függvény fut, amelynek tartalma gyakran egy végtelen cikluson belül van. Ha a `RunFunction` függvény futása véget ér, akkor az egész szál is törlődik. Ez tehát egyenértékű a `DestroyFunction` hívásával. A `RunFunction` függvényen belül a vezérlés átadható más szálaknak, erre szolgálnak a `Sleep()` és a `Yield()` operációs rendszer hívások. Az előbbi hatására az operációs rendszer határozott időre szünetelteti a szál futtatását, majd ezen idő leteltével újra futásra kész állapotba helyezi a szálat. A `Yield()` operációs rendszer hívás koopertív ütemezésnél kap szerepet.




3. ábra. Szálak prioritási szintjei

Minden VDK projektben automatikusan létrejön egy Idle szál, amelynek a prioritása 0 (legalacsonyabb), tehát ez fut akkor, ha más szál éppen nincs futásra kész állapotban.

Az interruptokat kiszolgáló szerviz rutinok (*Interrupt Service Routine, ISR*) a VDK-ban nem jelentenek külön szálat és nem is részei a szálaknak, azoktól külön kezelendők. Alapvetően ezek rövid lefutású függvények amelyek szemaforok vagy egyéb operációs rendszer eszközök segítségével kommunikálnak a szálakkal.

### 5.1.2. Prioritáskezelés - preemptív, kooperatív, időosztásos

A szálakhoz 31 különböző prioritási szintet rendelhetünk hozzá. Ebből következik, hogy lehetnek azonos prioritású szálaink. A szálak prioritása futási időben változhat. Az operációs rendszer ütemezésének működése alapvetően prioritás alapú preemptív, azaz mindig az a szál kapja meg a processzort, amelyik a legmagasabb prioritással rendelkezik a futásra készen álló szálak közül. A magasabb prioritású szálak tehát megszakítják az alacsonyabb prioritású szálak futását.

Ha több azonos prioritással rendelkező szál is futásra kész állapotban van, akkor a VDK két lehetőséget biztosít az ütemezésre, a a kooperatív ütemezést (*Cooperative Scheduling*), és a körforgó ütemezést (*Round-Robin Scheduling*). Kooperatív esetben az egyes szálak a `Yield()` függvény meghívásával explicit átadják a futás jogát más szálaknak. Körforgó ütemezés esetén az ütemező meghatározott hosszúságú időszeletet biztosít ez egyes szálaknak. A különböző ütemezési típusokat szabadon lehet keverni, tehát az egyes prioritási szinteken ez eltérő lehet. Az viszont ilyenkor is biztosított hogy egy magasabb prioritású szál mindig megszakítja az alacsonyabb prioritásúakat.

### 5.1.3. Kritikus és nem-ütemezett régiók

Beágyazott rendszerekről lévén szó, minden alkalmazásban felmerül annak az igénye, hogy bizonyos kritikus műveleteket nem megszakíthatóvá kell tenni. Erre a VDK két megoldást kínál, az egyik a kritikus régió (*Critical region*), a másik a nem ütemezett régió (*Unscheduled region*).

A kritikus régió az erősebb megoldás, ha ilyen kódrészlethez ér a futás, akkor az operációs rendszer ütemezője letiltásra kerül, valamint az összes interrupt is letiltódik. Ilyenkor tehát ameddig ki nem jut a program futása a kritikus szakaszból, addig mindenképpen az ő számára van fenntartva a processzor teljes számítási kapacitása. Ez egy nagyon erős eszköz, amelyet ajánlott a lehető legrövidebb ideig használni, lehetőleg csak atomikus műveletek elvégzésének idejére fenntartani (pl. egy globális változó értékének kiolvasása). A nem ütemezett régió hasonló ehhez, azzal a különbséggel hogy ilyenkor csak az ütemező van letiltva, de az interruptok azért kiszolgálásra kerülnek.

A kritikus és nem-ütemezett régiókba az alábbi API hívásokkal tudunk ki- illetve bekerülni. Az elnevezések stack kezeléshez hasonlatosak:

- VDK_PopCriticalRegion()
- VDK_PopUnscheduledRegion()
- VDK_PushCriticalRegion()
- VDK_PushUnscheduledRegion()

### 5.1.4. Szemaforok

VDK-ban a szálak közötti szinkronizációs és kommunikációs eszközök az egyéb RTOS-eknél szokásos megoldásokhoz hasonlóak. Ezek közül a szemafor (*Semaphore*) a legegyszerűbb, a szálak egymás közötti illetve szál és ISR közötti szinkronizációt tesz lehetővé. Általában arra használjuk a szemafort, hogy egy közös erőforrás használatát szabályozza. A szemafor alapesetben egy nulla vagy egy értékű változó. Ha a szemafor értéke 1, akkor hozzáférhetünk a közös erőforráshoz, ha 0 akkor nem. Ha egy olyan erőforrásról van szó amelyből több van, akkor a szemafor értéke mindig annyi, ahány még szabad erőforrás van. Ha lefoglalunk egy szabad erőforrást, akkor a szemafor értékét egyel csökkenteni kell.

A szemaforok kezelésére szolgáló API hívások:

- VDK_CreateSemaphore() - Létrehozás
- VDK_DestroySemaphore() - Törlés
- VDK_PendSemaphore() - Pend művelet: a szál megkísérel hozzáférni az erőforráshoz, azaz ellenőrzi hogy a szemafor nagyobb-e mint 0. Ha igen, akkor csökkenti a szemafor értékét egyel és folytatja a futást. Ha a szemafor értéke 0, akkor a szál futása blokkolódik, és addig így is marad ameddig a szemafor értéke 0 (ekkor mondjuk hogy *függőben van* a szál, angolul *pending*).
- VDK_PostSemaphore() - Post művelet: a szál befejezte a közös erőforráson végzett munkát, ezért felszabadítja a szemafort, azaz megnöveli az értéket egyel. Amennyiben vannak olyan függőben lévő szálak amelyek erre a szemaforra vártak és magasabb a prioritásuk mint a post-oló szál, akkor azok kezdenek futni.

A VDK támogatja periodikus szemaforok használatát is. Ha egy szemafor periodikus, az azt jelenti hogy az operációs rendszer bizonyos időközönként Post-olja az adott szemafort. Ezzel azt éri el, hogy az adott erőforrás ennyi időnként lesz csak elérhető. Ez egy elegáns módja a hozzáférések időbeli ütemezésének.

A szemaforokon kívül a VDK támogatja üzenetek (*messages*), mutexek, események (*events*) és eszköz flag-ek (*device flags*) használatát. Ezekről részletesen a VDK felhasználói kézikönyvében találhatunk részletes információkat, a méréshez nem szükségesek.

## 5.2. Példa alkalmazás forrásai

Az alábbi példaalkalmazás egy szálat, egy interruptot és egy szemafort tartalmaz. A `main` szál az UART0 kapcsolatot kezeli. Az egyetlen interrupt az `UART0 receive`, amely egy karakter beérkezését jelzi. A szemafor neve: `uart0_data_arrived`.

### 5.2.1. main.c

```
/* ============================================================================
*
* Description:  This is a C implementation for Thread main
*
* ============================================================================*/

/* Get access to any of the VDK features & datatypes used */
#include "main.h"
#include <stdio.h>
#include "globals.h"

#pragma file_attr("OS_Component=Threads")
#pragma file_attr("Threads")
void
main_RunFunction(void **inPtr)
{
        /* Put the thread's "main" Initialization HERE */
        int ii=0;
        while (1)
        {
                /* Put the thread's "main" body HERE */
                VDK_PendSemaphore(kuart0_data_arrived,0);
                printf("ii =      ii++;
                /* Use a "break" instruction to exit the "while (1)" loop */
        }

        /* Put the thread's exit from "main" HERE */
        /* A thread is automatically Destroyed when it exits its run function */
}

int
main_ErrorFunction(void **inPtr)
{
        /* TODO - Put this thread's error handling code HERE */
        /* The default ErrorHandler goes to KernelPanic */

        VDK_CThread_Error(VDK_GetThreadID());
        return 0;
}

void
main_InitFunction(void **inPtr, VDK_ThreadCreationBlock const *pTCB)
{
        /* Put code to be executed when this thread has just been created HERE */
```

```
        /* This routine does NOT run in new thread's context.  Any non-static thread
        * initialization should be performed at the beginning of "Run()."
        */

#define BAUD_RATE_115200 (65)
#define MAX_TEST_CHARS 5000

        volatile int temp;

        // set FERF registers
#if (__SILICON_REVISION__ < 0x0001) // Workaround silicon anomaly 05000212
        temp = *pPORTF_FER; //-possibly anomaly 05000157?
        *pPORTF_FER = 0x0003;
#endif
        *pPORTF_FER = 0x0003;
        // Configure UART0 RX and UART0 TX pins
#if (__SILICON_REVISION__ < 0x0001) // Workaround silicon anomaly 05000212
        temp = *pPORT_MUX; //-possibly anomaly 05000157?
        *pPORT_MUX = 0;
#endif
        *pPORT_MUX = 0;

/*****************************************************************************
*
* First of all, enable UART clock.
*
*****************************************************************************/
        *pUART0_GCTL = UCEN;

/*****************************************************************************
*
* Read period value and apply formula:  DL = PERIOD / 16 / 8
* Write result to the two 8-bit DL registers (DLH:DLL).
*
*****************************************************************************/
        *pUART0_LCR = DLAB | WLS(8);
        *pUART0_DLL = BAUD_RATE_115200;
        *pUART0_DLH = (BAUD_RATE_115200 >> 8);
/*****************************************************************************
*
* Clear DLAB again and set UART frame to 8 bits, no parity, 1 stop bit.
* This may differ in other scenarious.
*
*****************************************************************************/
        *pUART0_LCR = WLS(8);

/*****************************************************************************
*
* Finally enable interrupts inside UART module, by setting proper bits
* in the IER register.  It is good programming style to clear potential
* UART interrupt latches in advance, by reading RBR, LSR and IIR.
*
* Setting the ETBEI bit automatically fires a TX interrupt request.
*
```

```
*************************************************************************/
        temp = *pUART0_RBR;
        temp = *pUART0_LSR;
        temp = *pUART0_IIR;

        *pUART0_IER = ERBFI;

        *pSIC_IMASK |= 0x800;

}

void
main_DestroyFunction(void **inPtr)
{
        /* Put code to be executed when this thread is destroyed HERE */

        /* This routine does NOT run in the thread's context.  Any VDK API calls
        * should be performed at the end of "Run()."
        */
}
/* ======================================================================== */
```

### 5.2.2. EVT_IVG10.c

```
/* ========================================================================
*
* Description:  This is the Interrupt Service Routine called EVT_IVG10_Entry
*
* -------------------------------------
* Comments:
*
* ========================================================================*/

#include "VDK.h"
#include "globals.h"

#include <sys/exception.h>
#pragma file_attr("prefersMemNum=30")
#pragma file_attr("prefersMem=internal")
#pragma file_attr("ISR")

/* define VDK_REENTRANT_ISR macro if nested interrupts are allowed.  This
        definition can be done in a global project option if all interrupts allow
        nesting or in a particular interrupt file if there are any interrupts
        in the application that don't allow nesting */

#ifdef VDK_REENTRANT_ISR
        EX_REENTRANT_HANDLER(EVT_IVG10_Entry)
#else
        EX_INTERRUPT_HANDLER(EVT_IVG10_Entry)
#endif
{
```

```
        /* Insert your ISR code here */
        uart0_value = *pUART0_RBR;
        VDK_PostSemaphore(kuart0_data_arrived);
}

/* ========================================================================== */
```

## 5.3. Mérési feladatok

A mérés során egy olyan példaalkalmazást kell készíteni, amely legalább 2 szálat, 2 interruptot és szemaforokat használ. Az egyik szál valamilyen jelfeldolgozási feladatot valósítson meg, a másik pedig a vezérléssel és a felhasználóval történő kommunikációért legyen felelős. A konkrét feladatot a mérésvezetővel kell egyeztetni.

# Tartalomjegyzék