Analóg eszközök

A mikrokontrollerekben található analóg eszközök nagyban elősegítik a külvilággal érintkező mérő és beavatkozó rendszerek integrációját, mivel sok analóg funkció elvégezhető közvetlenül a processzorral további külső alkatrészek használata nélkül is. Ez azért fontos, mert egy beágyazott rendszer nagyon sok esetben működik olyan feladatkörben, ahol bizonyos folyamatokat leíró analóg mennyiségeket kell tudni mérni és értelmezni, digitálisan feldolgozni és az eredmény alapján adott esetben ismét analóg módon beavatkozni a rendszer működésébe (pl. szabályozókörök). Emiatt már a legtöbb egyszerűbb mikrokontroller is tartalmaz legalább egy **analóg-digitális átalakítót (ADC)**, ami lehetővé teszi egy adott határok között változó analóg *feszültségérték* adott felbontással és frekvenciával történő kvantálását és mintavételezését. Ez egyrészt azt vonja maga után, hogy a közvetlen mérést végző szenzornak feszültség kimenetet kell adnia (vagy az egyéb mennyiséget feszültséggé kell alakítani), másrészt megfelelő analóg „előtét" elektronikával biztosítani kell, hogy a mintavételezés következtében ne jelenjenek meg olyan torzító összetevők a jel digitális formájában, amelyek az analóg jelben nem voltak jelen és digitális szűrővel már nem távolíthatók el. Ez az átlapolódás vagy *aliasing* jelensége, további információ és szemléltetés többek között az alábbi linkeken érhető el:

- http://www.maximintegrated.com/en/app-notes/index.mvp/id/928
- http://www.photocritic.org/temporal-aliasing-in-video/
- https://www.youtube.com/watch?v=mODqQvlrgIQ

Az ADC mellett további hasznos analóg perifériák a **digitális-analóg átalakító (DAC)**, ami analóg feszültségértékek adott határok között adott felbontással történő előállítását teszi lehetővé a környezetbe történő beavatkozáshoz, valamint a **komparátor**, ami két analóg érték nagyságrendi összehasonlítására használható. Az általunk használt STM32 mikrokontrollerekben az alábbi analóg perifériák találhatók meg:

STM32F407VG

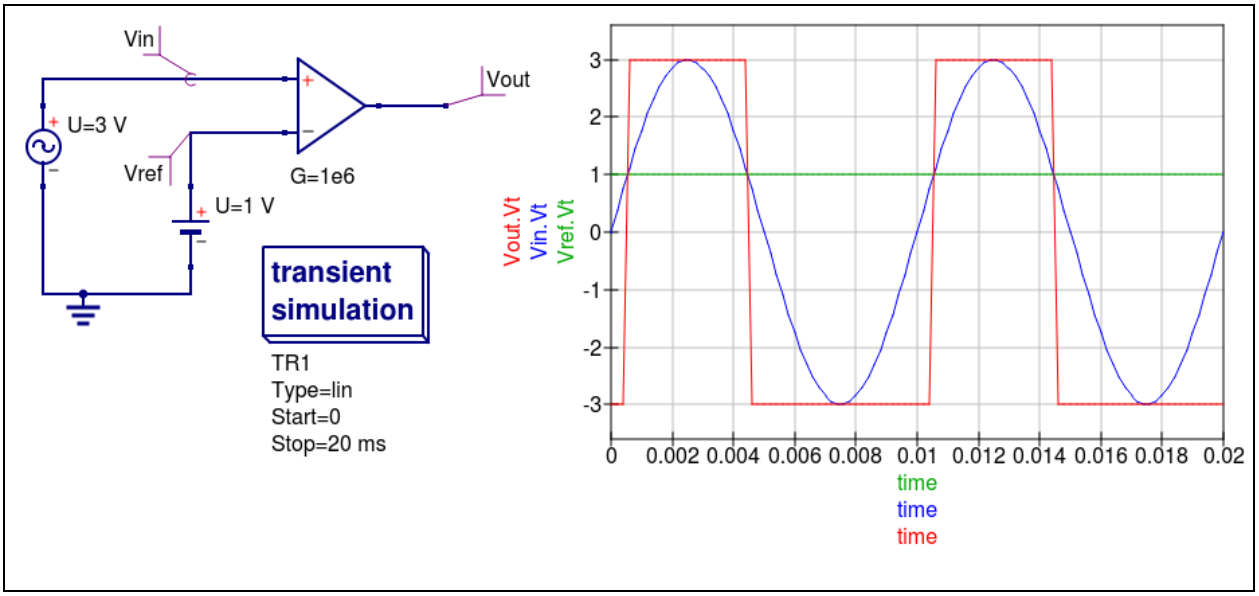
- 3×12-bit, 2.4 MSPS Successive Approximation ADC
- 2×12-bit DAC

STM32F373VC (mixed-signal)

- 1×12-bit Successive Approximation ADC
- 3×16-bit Sigma-Delta ADC
- 3×12-bit DAC
- 2×fast rail-to-rail analog comparators with programmable input and output

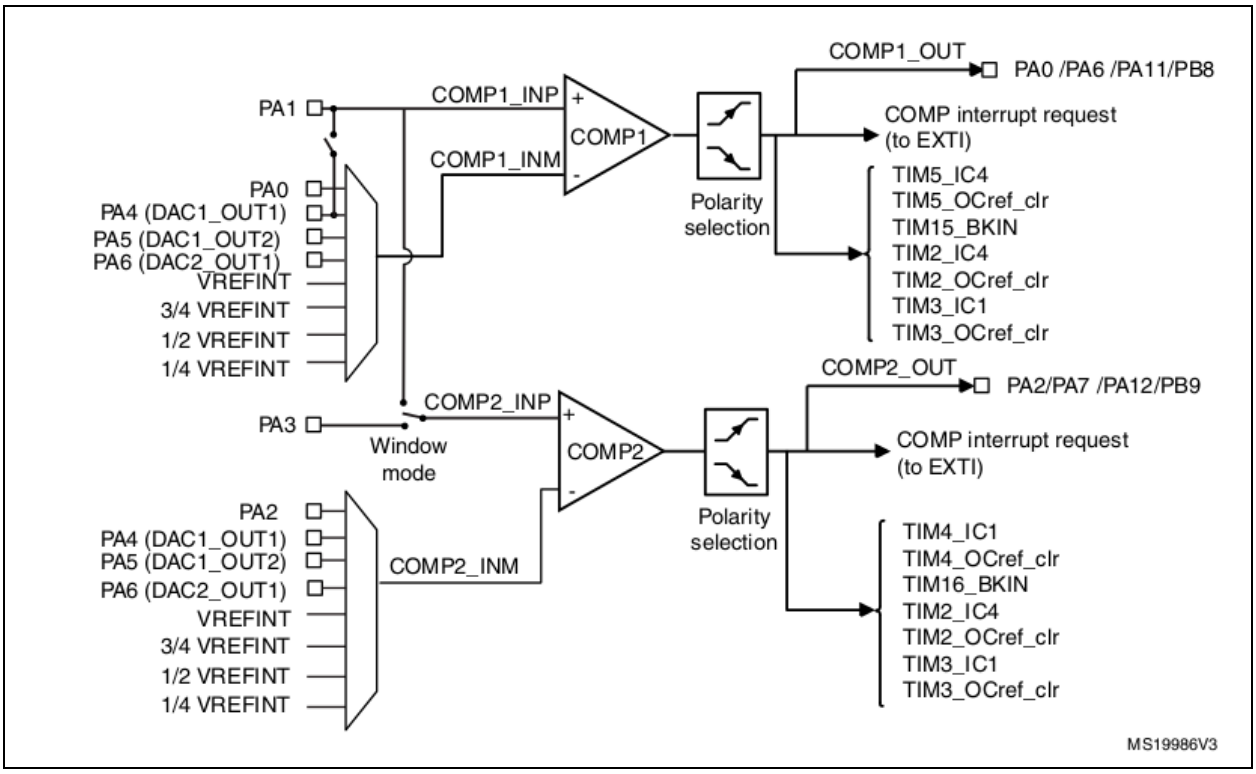
Komparátor

(Az F4-es processzorcsaládban nem található meg.) A komparátor lényegében egy műveleti erősítő, ami attól függően ad logikai 0 (negatív analóg táp) vagy logikai 1 (pozitív analóg táp) kimenetet, hogy a nem invertáló (+) bementére kapcsolt potenciál az invertáló (-) bemeneten lévő referencia értéknél kisebb vagy nagyobb. Ez a műveleti erősítők nagyon magas nyílt-hurkú erősítésének köszönhető, ami miatt már a bemenetek között lévő nagyon kis potenciálkülönbség esetén is szaturálódik a kimenet valamelyik (pozitív vagy negatív) táp irányába (további szemléletes információ itt érhető el: http://www.electronics-tutorials.ws/opamp/opamp_1.html). Az analóg komparátor működését az alábbi szimuláció szemlélteti.



Ábra 1 – Műveleti erősítő használata komparátorként. A szimulációban használt OpAmp tápfeszültsége +/- 3V, nyílt hurkú erősítése 10^6 volt. A szimuláció Qucs környezetben történt (http://qucs.sourceforge.net/).

Az ST mixed-signal processzoraiban (pl. STM32F373VC) lévő komparátorok a fenti funkciót a bemenetek és kimenetek nagyfokú konfigurálhatóságával valósítják meg az alábbi ábrának megfelelően:



Ábra 2 – Az STM32F373 mikrokontroller beépített komparátor blokkjának felépítése az eszköz adatlapja alapján.

Ahogyan az ábrán is látható, a két integrált komparátor „ablak” módban is használható, ekkor az analóg bemenet adott sávon belüli tartózkodása kísérhető figyelemmel. A komparátorok bemenete a belső referencia (bandgap feszültség, processzoronként külön mérve és tárolva, az ADC_IN17 csatornán érhető el) adott hányadai mellett a DAC-k kimenetei és egy-egy konkrét processzorlábon lévő (PA0, PA2) feszültségértékek is lehetnek. A kimeneteket GPIO lábakra (COMPx_OUT), külső interrupt vonalakra (EXTI, Sleep és Stop üzemmódból is felébreszti a processzort) és timerekre is át lehet irányítani (ekkor például folyamatos polling nélkül számolható, hogy a vizsgált analóg érték hányszor lépte át a referencia szintet).

A beépített komparátorok további hasznos funkciói a programozható hiszterézis és a négyféle teljesítmény/fogyasztás beállítási lehetőség. A hiszterézis zajos bemenet esetén megakadályozza a kimenet gyors ugrálása miatt bekövetkező hamis detekciókat (STM32F373 Reference Manual, 15.3.5. fejezet). A teljesítmény/fogyasztás beállítása értelem szerűen az

egység sebességét (jelpropagáció) és fogyasztását határozza meg (a két tulajdonság fordítottan arányos egymással). Lehetséges alkalmazások pl.: küszöb detekció, nullátmenet detekció, Schmitt trigger, analóg feszültség detekció.

Programozása (stm32f37x_comp.[h/c])

Beállító struktúra:

```
typedef struct
{
  uint32_t COMP_InvertingInput;     /*!< Selects the inverting input of the comparator.
                                        This    parameter    can    be    a    value    of    @ref
                                        COMP_InvertingInput */

  uint32_t COMP_Output;             /*!< Selects the output redirection of the comparator.
                                        This parameter can be a value of @ref COMP_Output */

  uint32_t COMP_OutputPol;          /*!< Selects the output polarity of the comparator.
                                        This    parameter    can    be    a    value    of    @ref
                                        COMP_OutputPolarity */

  uint32_t COMP_Hysteresis;         /*!< Selects the hysteresis voltage of the comparator.
                                        This parameter can be a value of @ref COMP_Hysteresis
                                        */

  uint32_t COMP_Mode;               /*!< Selects the operating mode of the comparator
                                       and allows to adjust the speed/consumption.
                                        This parameter can be a value of @ref COMP_Mode */
}COMP_InitTypeDef;
```

Vezérlő függvények:

```
// inicializálás
void COMP_Init(uint32_t COMP_Selection, COMP_InitTypeDef* COMP_InitStruct);

// beállító struktúra alapértelmezésbe állítása
void COMP_StructInit(COMP_InitTypeDef* COMP_InitStruct);

// komparátor engedélyezése vagy kikapcsolása
void COMP_Cmd(uint32_t COMP_Selection, FunctionalState NewState);

// az 1-es komparátor (COMP1) bemenetének nagyimpedanciára kapcsolása (PA1 -> PA4)
void COMP_SwitchCmd(FunctionalState NewState);

// kimeneti szint (magas vagy alacsony) lekérdezése
uint32_t COMP_GetOutputLevel(uint32_t COMP_Selection);

// ablak funkció engedélyezése vagy tiltása
void COMP_WindowCmd(FunctionalState NewState);

// az adott komparátor konfigurációjának zárolása (csak system reset oldja fel)
void COMP_LockConfig(uint32_t COMP_Selection);
```

ADC

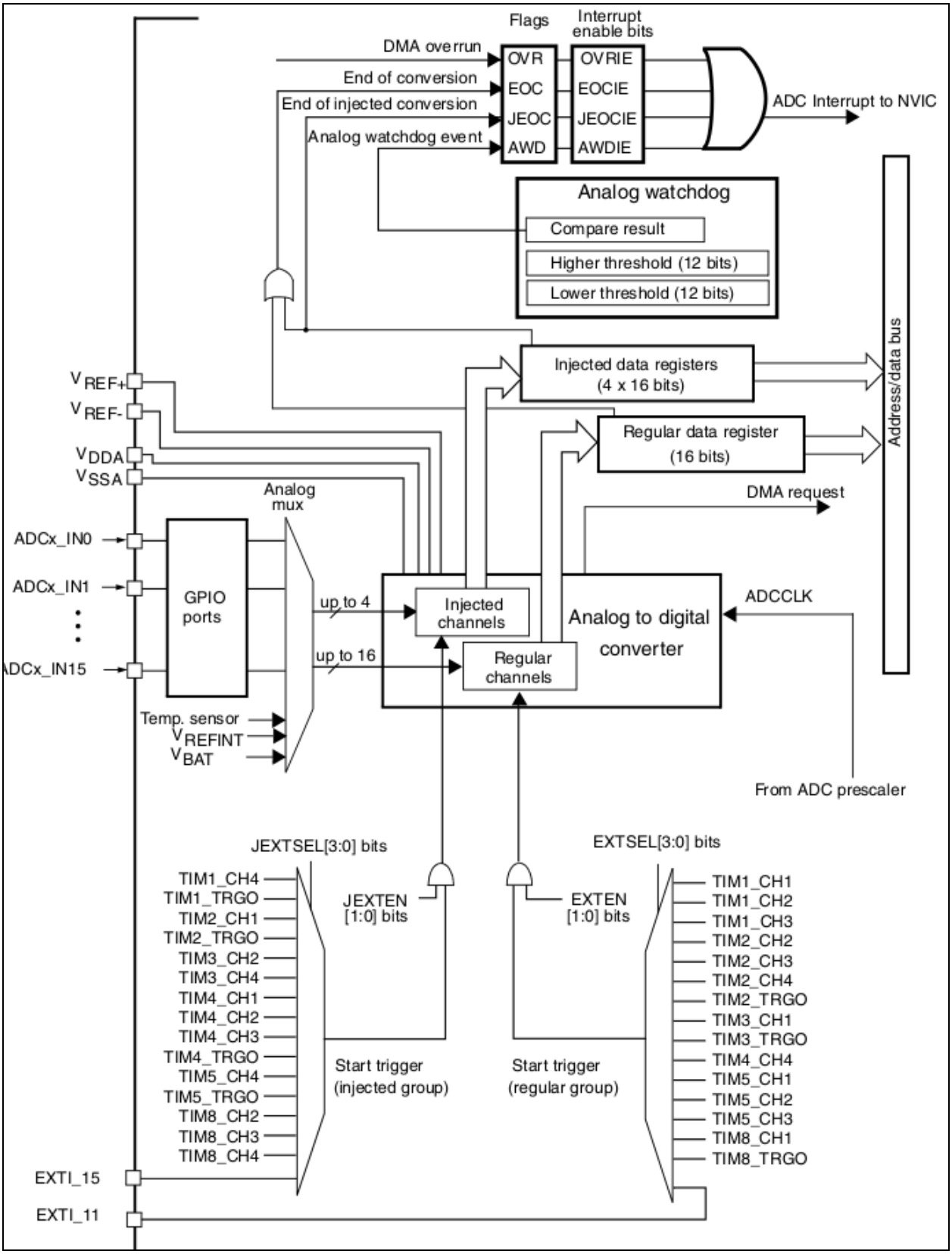
A folyamatos közelítésű (successive approximation register, SAR) ADC-k minden konverziókor a bemeneti analóg jelet az elérhető kvantálási szinteken végigiterálva egy bináris keresési algoritmus alapján digitalizálják. Működési sebességük a több MHz-et is elérheti, felbontásuk legtöbbször 8-16 bit között van. Előnyük az alacsony fogyasztás mellett elérhető elfogadható felbontás és kis helyfoglalás, ezért a legtöbb mikrokontrollerbe integrált ADC ezt az architektúrát alkalmazza. A SAR ADC-ről további szemléletes információ itt érhető el: http://www.maximintegrated.com/en/app-notes/index.mvp/id/1080.

Az STM32F407-es processzorban 3 db 12 bites SAR ADC található 19 multiplexelt csatornával, ezekből 16 külső, 2 belső és egy Vbat (akkufeszültség mérésére). A perifériában többféle konverziós mód áll rendelkezésre (egyszeri, folytonos, pásztázó és nem folytonos). A konverziós eredmény egy 16 bites adatregiszterben kerül tárolásra, amelyben állítható az adatok balra vagy jobbra ütköztetése (left-aligned vagy right-aligned).

Az ADC főbb tulajdonságai:

- konfigurálható 6, 8, 10 és 12 bites felbontás
- interrupt generálás reguláris vagy injektált konverzió végén, analog watchdog eseménykor
- csatornánként állítható mintavételezési idő (ameddig az ADC tartja a jelet, nem keverendő össze a mintavételi frekvenciával)
- külső trigger lehetőség konverzió indítására
- Dual/Triple működési mód több ADC-vel rendelkező eszköz esetén
- DMA-s vezérlési lehetőség

A periféria blokkdiagramja a következő oldalon látható. Az eszköz középpontjában a tényleges ADC áll, ezt egészíti ki a bemeneti multiplexer (Analog mux), a két konverziós trigger multiplexer (Start trigger), az adatregiszterek és az interrupt generálásért felelős blokk az Analog watchdoggal. Az ADC meghajtása külön analóg (ADC működés) és digitális (regiszter elérés) órajelekkel történik, ahol az analóg órajel az APB2 busz órajelének leosztásával (f_{PCLK2} / 2, /4, /6, /8) jön létre, míg a digitális órajel értéke f_{PCLK2}.



Ábra 3 – Az STM32F407 mikrokontroller beépített ADC-jének blokkdiagramja az eszköz adatlapja alapján.

Az ADC-vel végzett adatkonverzió kétféle csoportba osztható: reguláris és injektált. Reguláris csoportba legfeljebb 16, injektált csoportba legfeljebb 4 konverzió tartozhat, a csoportokon belül a csatornák sorrendje tetszőleges.

Működési módok (egy ADC-re, az AN3116 AppNote alapján)

A processzorba integrált ADC az alábbi (egy ADC-t igénylő) működési módokban használható az aktuális feladatnak megfelelően:

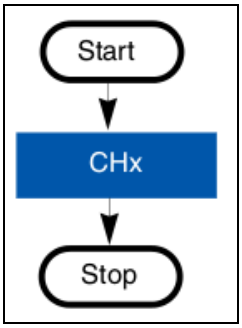
- Single-channel, single conversion
- Multichannel (scan), single conversion
- Single-channel, continuous conversion
- Multichannel, continuous conversion
- Injected conversion

Az alábbi tulajdonságok mindegyik működési módra érvényesek:

- a konverzió indítása triggerelhető szoftveresen vagy hardveresen (EXTI vagy timer)
- az egyes csatornákra a mintavételi idő megadható ADC órajelciklusban
- interrupt generálható konverzió végén, vagy ha egy konvertált érték egy általunk megadott sávon kívül van (Analog watchdog)

Single-chanel, single conversion

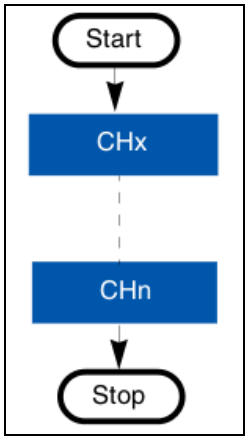
A legegyszerűbb ADC működési mód, amikor az ADC egyetlen mérés végez és a konverzió után leáll. Alkalmas lehet például rendszerindításkor egy analóg feltétel (pl. tápfeszültség) ellenőrzésére.



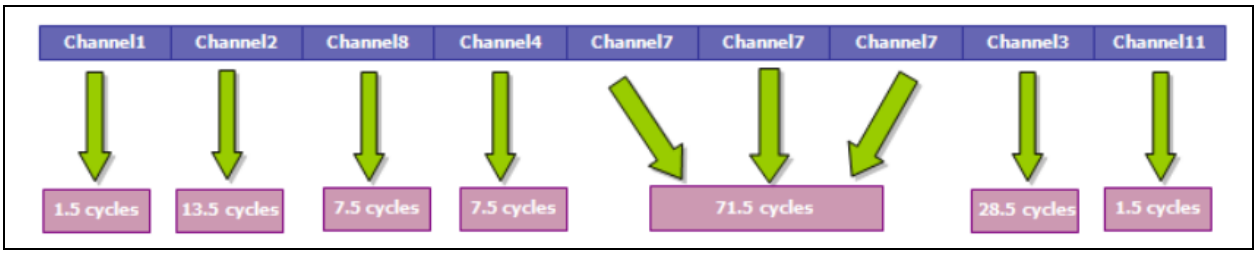
Ábra 4 – Egy csatorna, egy konverzió.

Multichannel (scan), single conversion

Ebben a módban az ADC egy indításra több csatornát képes konvertálni egymás után, megadott sorrendben és mintavételi idővel. Alkalmas lehet például több érték végigmérésére rendszerindításkor.



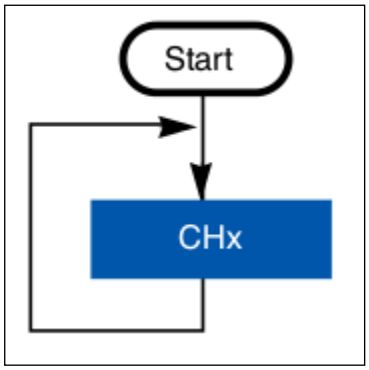
Ábra 5 – Több csatorna, egy konverzió.



Ábra 6 – Scan módban a csatornák sorrendje és mintavételi ideje egymástól függetlenül állítható.

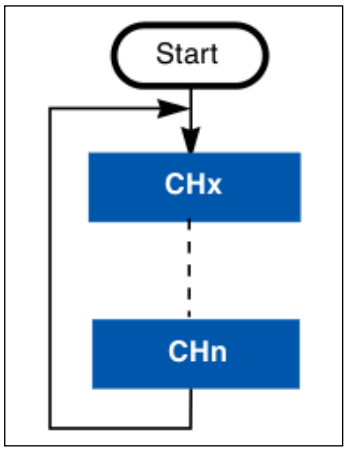
Single-channel, continuous conversion

Egy csatorna mérése folyamatosan, a processzor beavatkozása nélkül. Az adatregiszter kiolvasása történhet tetszőleges időpontban szoftveresen, vagy DMA használatával. Alkalmas lehet például az akkufeszültség folyamatos figyelésére vagy szabályozási feladatokra. Ha csak egy érték adott tartományon belül maradásának ellenőrzése a cél, az Analog wachdog funkcióval kombinálva teljesen automatizálható a detekció.



Ábra 7 – Egy csatorna, folyamatos konverzió

Multichannel (scan), continuous conversion

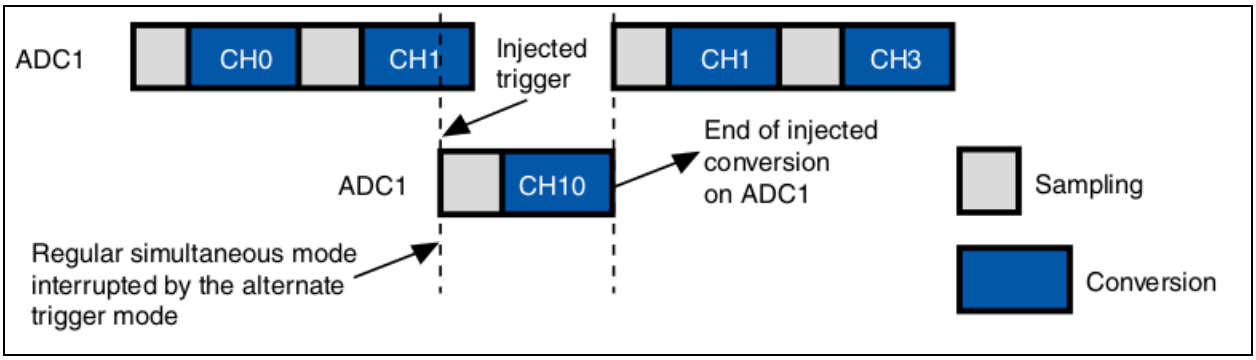


Több csatorna mérése folyamatosan, a processzor beavatkozása nélkül. A csatornák sorrendje és a mintavételi idő a korábban látottakhoz hasonlóan tetszőlegesen megadható. A konvertálási sor végén nem áll meg a működés, hanem a sor elejéről újraindul. Alkalmas lehet például több érték folyamatos figyelésére és szabályozási feladatokra.

Ábra 8 – Több csatorna, folyamatos konverzió

Injected conversion

Az eddig vizsgált működési módokban a mérési csatornák mindig reguláris csoporthoz tartoztak. Az ADC a még összetettebb működés hatékony biztosítása érdekében lehetőséget nyújt injektált konverzió elvégzésére is, melynek lényege, hogy az éppen futó reguláris konverzió egy esemény hatására (pl. külső trigger) megszakítható egy nem reguláris, injektált konverzióval, amely után a periféria folytatja a reguláris csoporton történő működést.



Ábra 9 – Injektált konverzió szemléltetése.

Ennek a módnak a lényege, hogy egy futó konverziós folyamat mellett ugyanazzal az ADC-vel kezelhetővé válik egy nem determinisztikus (pl. gombnyomás hatására bekövetkező) mérési művelet elvégzése.

Direct Memory Access (DMA) röviden

A processzorba integrált mindegyik ADC periféria (ADC-nként) egyetlen adatregiszterrel rendelkezik. Ez azt vonja maga után, hogy ha a fent bemutatott működési módokban (a single-channel, single conversion kivételével) minden adatot el szeretnénk tárolni a működés során, még azelőtt kell kiolvasnunk az ADC adatregiszterének értékét, hogy a következő konverzió megtörténne. Nagyon kis sebességeknél ez szoftveresen még megtehető, de ez sem lesz hatékony megoldás. Jó hír viszont, hogy mindegyik STM32 mikrokontroller rendelkezik DMA egységgel, melynek célja a processzortól független adatmozgatás adott memóriaterületek között (innen az elnevezés is). DMA átvitel beállítható két memóriacím között is (változók értékeinek gyors másolása), de gyakoribb a periféria és memória közötti átvitel, amikor adott eseményre megadott számú és méretű adat mozgatható a periféria adatregisztere és adott memóriaterület között, elkerülve ezzel az esetleges adatvesztéseket.

Az ADC esetén például a DMA felkonfigurálható úgy, hogy az ADC adatregiszter értékét minden egyes konverzió végét jelző interrupt (EOC) esetén másolja egy adott méretű tömbbe a változóban, gondoskodva a tömbindex növeléséről is. Ezáltal adatvesztés nélkül megoldható az adatok tárolása úgy, hogy az még processzoridőt se használjon el.

Programozása (stm32f4xx_adc.[h/c])

Beállító struktúrák:

```
typedef struct

{

  uint32_t ADC_Resolution;                /*!< Configures the ADC resolution dual mode.
                                               This   parameter   can   be   a   value   of   @ref
                                                ADC_resolution */

  FunctionalState ADC_ScanConvMode;       /*!< Specifies whether the conversion
                                               is performed in Scan (multichannels)
                                               or Single (one channel) mode.
                                               This parameter can be set to ENABLE or DISABLE */

  FunctionalState ADC_ContinuousConvMode; /*!< Specifies whether the conversion
                                               is performed in Continuous or Single mode.
                                               This parameter can be set to ENABLE or DISABLE. */

  uint32_t ADC_ExternalTrigConvEdge;      /*!< Select the external trigger edge and
                                               enable the trigger of a regular group.
                                               This parameter can be a value of
                                               @ref
                                               ADC_external_trigger_edge_for_regular_channels_con
                                               version */

  uint32_t ADC_ExternalTrigConv;          /*!< Select the external event used to trigger
```

```
                                        the start of conversion of a regular group.
                                        This parameter can be a value of
                                        @ref
                                        ADC_extrenal_trigger_sources_for_regular_channels_
                                        conversion */

  uint32_t ADC_DataAlign;               /*!< Specifies whether the ADC data  alignment
                                        is left or right. This parameter can be
                                        a value of @ref ADC_data_align */

  uint8_t  ADC_NbrOfConversion;         /*!< Specifies the number of ADC conversions
                                        that will be done using the sequencer for
                                        regular channel group.
                                        This parameter must range from 1 to 16. */
}ADC_InitTypeDef;
```

```
typedef struct
{
  uint32_t ADC_Mode;                    /*!< Configures the ADC to operate in
                                        independent or multi mode.
                                        This   parameter   can   be   a   value   of   @ref
                                        ADC_Common_mode */

  uint32_t ADC_Prescaler;               /*!< Select the frequency of the clock
                                        to the ADC. The clock is common for all the ADCs.
                                        This   parameter   can   be   a   value   of   @ref
                                        ADC_Prescaler */

  uint32_t ADC_DMAAccessMode;           /*!< Configures the Direct memory access
                                       mode for multi ADC mode.
                                        This parameter can be a value of
                                        @ref   ADC_Direct_memory_access_mode_for_multi_mode
                                        */

  uint32_t ADC_TwoSamplingDelay;        /*!< Configures the Delay between 2 sampling phases.
                                        This parameter can be a value of
                                        @ref ADC_delay_between_2_sampling_phases */
}ADC_CommonInitTypeDef;
```

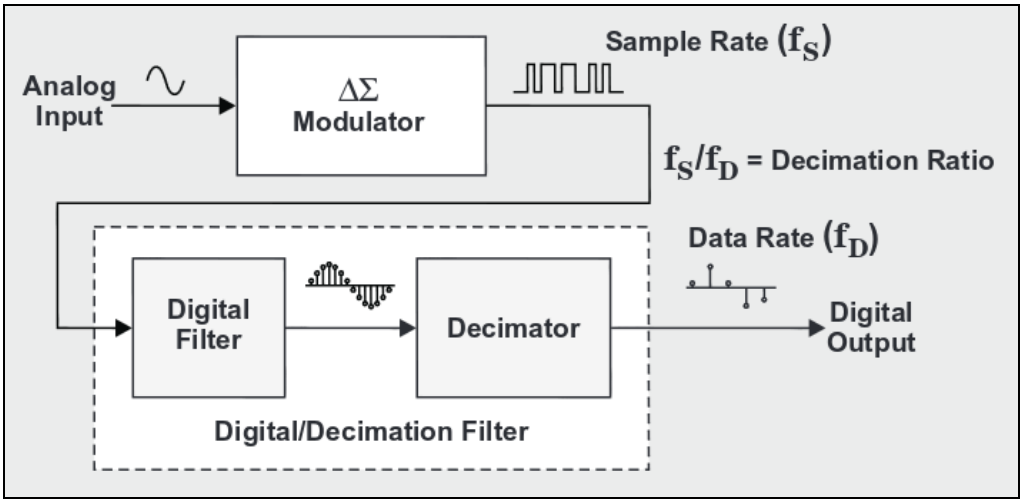
SDADC

A SAR mellett a másik széles körben használt architektúra a Sigma-Delta (vagy Delta-Sigma) ADC (SDADC), ami alapjaiban más megközelítést használ az analóg jelek digitalizálására. Felépítésében és működésében a SAR ADC-hez képest összetettebb, beépített zajformálást (noise-shaping), valamint digitális aluláteresztő és decimáló szűrőt tartalmaz. A zajformálás miatt az anti-aliasing előtét elektronikával szemben kevésbé szigorúak a követelmények, mint a SAR ADC esetén (pl. kisebb meredekségű analóg szűrő is elegendő). Működési (digitális kimeneti) sebessége alacsonyabb a SAR ADC-nél, viszont magasabb felbontás mellett jobb jel-zaj arány (SNR) elérését teszi lehetővé. Általában precíziós mérésekhez használják.

Az architektúráról további részletesebb leírás az alábbi linkeken található:

- http://www.maximintegrated.com/en/app-notes/index.mvp/id/1870
- http://www.ti.com/lit/an/slyt423/slyt423.pdf
- http://www.ti.com/lit/an/slyt438/slyt438.pdf

A Sigma-Delta ADC elvi felépítését az alábbi ábra mutatja:

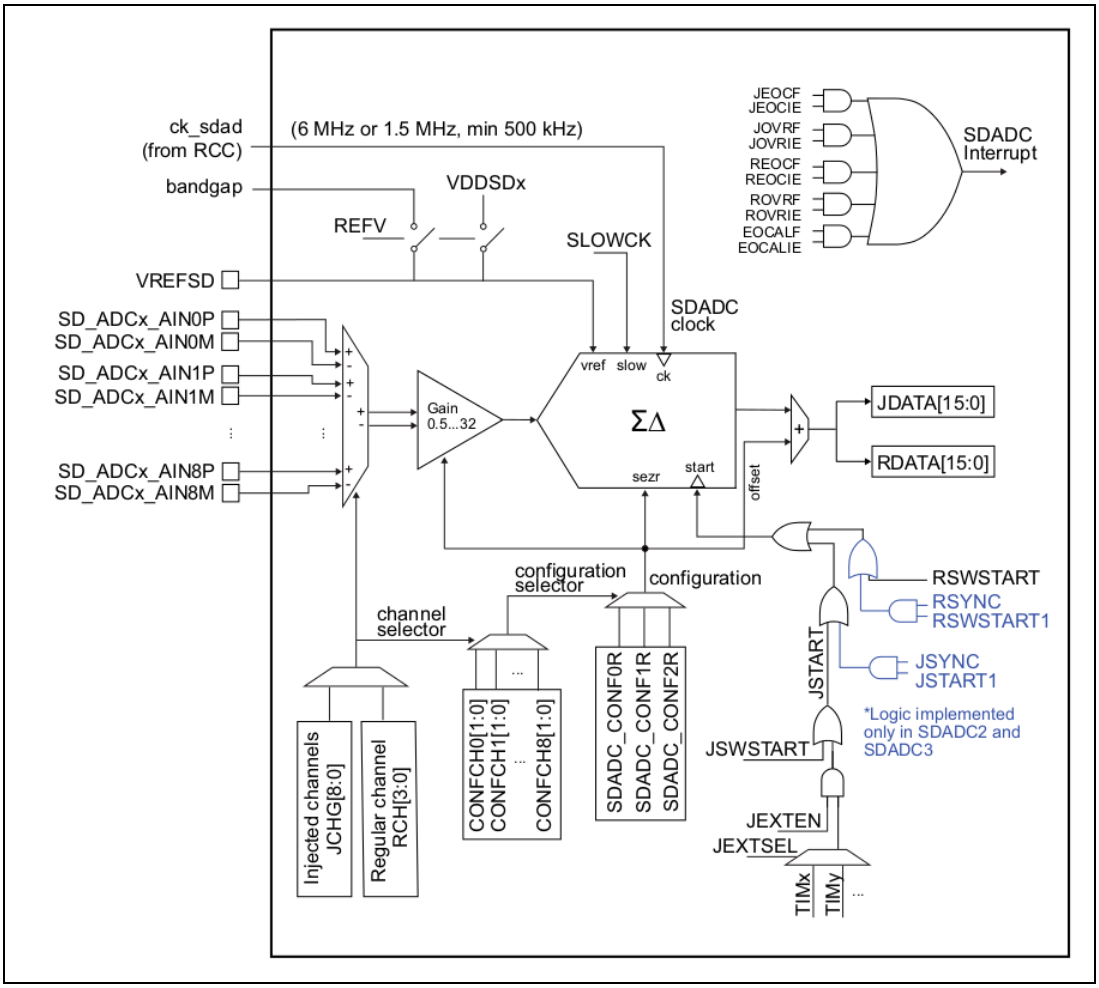


Ábra 10 – Sigma-Delta ADC blokkdiagramja. A ΔΣ modulátor túlmintavételezés mellett zajformálást végez, míg a digitális rész (szűrés és decimálás) a végleges digitális kimenet előállításáért felel.

Az STM32F373 processzorban található SDADC főbb tulajdonságai:

- 16-bites felbontás
- SDADC-nként 5 differenciális vagy 9 monopoláris bemenet
- több csatorna mérése esetén 16.6 ksps, egy csatorna mérése esetén 50 ksps sebesség
- beépített 7 fokozatú előerősítő (0.5x – 32x)
- konverzió indítása: szoftveresen, timerrel, külső eseményre vagy egy másik SDADC-vel
- DMA-s vezérlési lehetőség
- 3 kis fogyasztású működési mód: *Slow*, *Standby* és *Power down*

Az eszköz blokkdiagramja a következő oldalon látható. A középpontban itt is maga az ADC található, ezt egészíti ki többek között a bemeneti multiplexer, a programozható előerősítő és a kimeneti offset kompenzáló egység. Az SDADC órajelének legalább 500 kHz-en kell járnia, de ajánlott 6 MHz-en meghajtani (ez lesz a tényleges futási sebesség, így befolyásolja pl. a ΔΣ modulátor működését is).



Ábra 11 – Az STM32F373 mikrokontrollerben található SDADC felépítése.

Programozása (stm32f37x_sdadc.[h/c])

Beállító struktúrák:

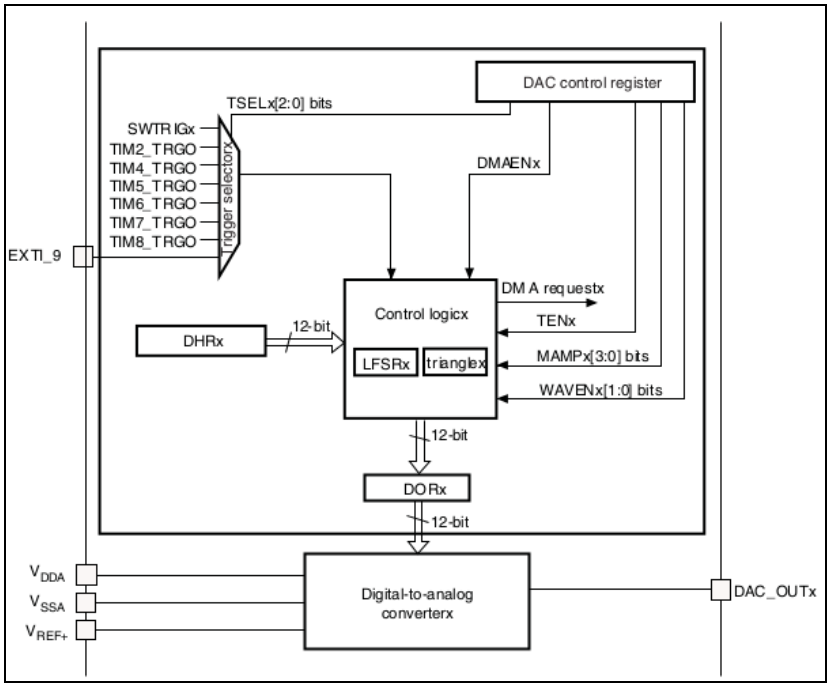
```
typedef struct
{
  uint32_t SDADC_Channel;                    /*!< Select the regular channel.
                                                  This parameter can be any value of @ref
                                               SDADC_Channel_Selection */

  FunctionalState SDADC_ContinuousConvMode;  /*!< Specifies whether the conversion is performed
                                               in Continuous or Single mode.
                                                  This parameter can be set to ENABLE or DISABLE.
                                               */

  FunctionalState SDADC_FastConversionMode;  /*!< Specifies whether the conversion is performed
                                               in fast mode.
                                                  This parameter can be set to ENABLE or DISABLE.
                                               */
}SDADC_InitTypeDef;
```

```
typedef struct
{
  uint32_t  SDADC_InputMode;          /*!< Specifies the input structure type (single ended,
                                           differential...)
                                      This parameter can be any value of @ref SDADC_InputMode */


  uint32_t SDADC_Gain;           /*!< Specifies the gain setting.
                                      This parameter can be any value of @ref SDADC_Gain */

  uint32_t SDADC_CommonMode;     /*!< Specifies the common mode setting (VSSA, VDDA, VDDA/2).
                                      This parameter can be any value of @ref SDADC_CommonMode */

  uint32_t SDADC_Offset;         /*!< Specifies the 12-bit offset value.
                                      This parameter can be any value lower or equal to
                                      0x00000FFF */

}SDADC_AINStructTypeDef;
```

DAC

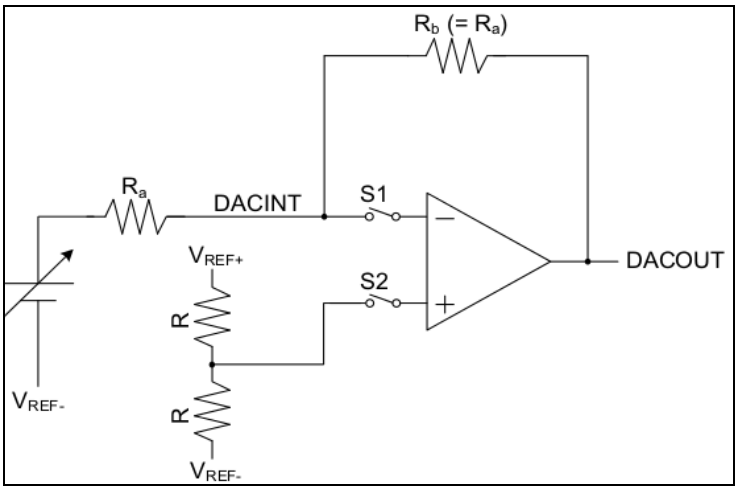
A digitális-analóg átalakító (DAC) analóg feszültségértékek adott határok között történő előállítását teszi lehetővé adott felbontással. Az STM32F407 processzorban 2 db DAC található, amelyek az alábbi tulajdonságokkal rendelkeznek:

- konfigurálható 8 vagy 12 bites felbontás
- bemenő feszültség referencia (V_{REF+})
- zaj és háromszögjel generálási képesség
- DMA-s vezérlési lehetőség
- külső trigger konverzió indításához



Ábra 12 – Az STM32F407 mikrokontrollerben található DAC felépítése.

Az STM32F4 Discovery boardon a DAC referenciája (V_{REF+}) 3 V-ra van kötve, ezért 0-3 V a teljes kimeneti tartomány, 732.6 μV felbontás mellett. Az egyes konverziók indítása történhet szoftveresen, külső esemény (EXTI_9) vagy timer kimeneti trigger (TIMx_TRGO) hatására. A DAC mindkét csatornáján bekapcsolható egy-egy kimenő buffer (beépített műveleti erősítő), ezek csökkentik a kimeneti impedanciát és lehetővé teszik külső terhelésként jelentkező áramkörök közvetlen meghajtását további (külső) műveleti erősítő használata nélkül. A DAC helyettesítő kapcsolása a lenti ábrán látható.



Ábra 13 – Az STM32F407 mikrokontroller beépített DAC perifériájának helyettesítő kapcsolása (az AN4566 AppNote alapján)

Programozása (stm32f4xx_dac.[h/c])

Beállító struktúra:

```
typedef struct
{
  uint32_t DAC_Trigger;                      /*!< Specifies the external trigger for the selected
                                                 DAC channel.
                                                This parameter can be a value of @ref
                                                 DAC_trigger_selection */

  uint32_t DAC_WaveGeneration;                /*!< Specifies whether DAC channel noise waves or
                                                   triangle waves
                                               are generated, or whether no wave is generated.
                                               This parameter can be a value of @ref
                                                 DAC_wave_generation */

  uint32_t  DAC_LFSRUnmask_TriangleAmplitude;  /*!< Specifies the LFSR mask for noise wave
                                                    generation or
                                               the maximum amplitude triangle generation for
                                               the DAC channel.
                                               This parameter can be a value of @ref
                                               DAC_lfsrunmask_triangleamplitude */

  uint32_t DAC_OutputBuffer;                 /*!< Specifies whether the DAC channel output buffer
                                                 is enabled or disabled.
                                               This parameter can be a value of @ref
                                                 DAC_output_buffer */
}DAC_InitTypeDef;
```

Főbb vezérlő függvények:

```
// adott csatorna inicializálása
void DAC_Init(uint32_t DAC_Channel, DAC_InitTypeDef* DAC_InitStruct);

// beállító struktúra alapértelmezésbe állítása
void DAC_StructInit(DAC_InitTypeDef* DAC_InitStruct);

// adott DAC csatorna aktiválása vagy leállítása
void DAC_Cmd(uint32_t DAC_Channel, FunctionalState NewState);

// adott DAC csatorna szoftveres triggerelésének engedélyezése vagy letiltása
void DAC_SoftwareTriggerCmd(uint32_t DAC_Channel, FunctionalState NewState);

// a DAC1 csatorna adatregiszterének (így a kimeneti feszültség értékének) beállítása
void DAC_SetChannel1Data(uint32_t DAC_Align, uint16_t Data);

// a DAC2 csatorna adatregiszterének (így a kimeneti feszültség értékének) beállítása
void DAC_SetChannel2Data(uint32_t DAC_Align, uint16_t Data);

// az adott DAC csatorna legutolsó kimenő értékének lekérdezése
uint16_t DAC_GetDataOutputValue(uint32_t DAC_Channel);
```

Felkészülés a gyakorlatra

- [Opcionális, de ajánlott] Ac6 System Workbench telepítése innen: www.openstm32.org/
  - Eclipse alapú ingyenes és korlátozások nélküli környezet, regisztrációt igényel
  - a telepítés után egy alap projekt létrehozása és annak szerkezeti áttekintése
- STM32F407 Reference Manual **Timer** (elég egy), **ADC**, **DAC** és **DMA** fejezetei
- Az STM32F4 Discovery board StdPeriph könyvtárából az alábbi fájlok áttekintése és készségszintű ismerete (a gyakorlaton ne azt kelljen keresgélni, hogy pl. az ADC_InitTypeDef mire jó és milyen mezői vannak):
  - **STM32F4-Discovery_FW_V1.1.0/Libraries/STM32F4xx_StdPeriph_Driver/[inc|src]**
    - stm32f4xx_gpio.[h|c]
    - stm32f4xx_adc.[h|c]
    - stm32f4xx_dac.[h|c]
    - stm32f4xx_tim.[h|c]
    - stm32f4xx_dma.[h|c]

Referenciák

- http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf
- http://www.st.com/web/en/resource/technical/document/reference_manual/DM00041563.pdf
- http://www.st.com/web/en/resource/technical/document/application_note/CD00258017.pdf
- http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/application_note/DM00129215.pdf

Programok készítése

A program készítéséről általában annyit kell tudni, hogy a fejlesztő környezetben (IDE) a megfelelő gomb megnyomására elkészül az új változata a programnak, és már ki is lehet próbálni. A fejlesztő környezet biztosítja, hogy csak a szükséges feladatok hajtódjanak végre a végrehajtható kód létrehozása során. A környezet ezt a feladatot az egyes alkotórészek keletkezési idejének alapján tudja megoldani, azaz ha valamely program keletkezésénél frissebb alkotórészt talál a project leírásban, akkor a szükséges feladatok végrehajtásával frissíti a programot. Ilyen leírás és kezelő program pl.: a „make”.
A fejlesztő környezetekben létrehozott project file automatikusan létrehozza az összefüggéseket és továbbiakban annak megfelelően működik.
A programok létrehozásánál általában valamilyen szabályrendszer szerinti leírásmóddal megszerkesztett text file-t kell készíteni ez a forrás nyelvű program. Különböző formai követelmények terjedtek el a forrással kapcsolatban. Ilyen pl.: Pascal C, vagy C++. Ezek az említett nyelvek magas szintű programozási nyelvek. Azért nevezik magas szintűnek, mert a futtatásuknál használt processzortól független utasításkészletet tartalmaznak.

Magas szintű programnyelvek jellemzői

A magas szintű nem azt jelenti, hogy a nyelv magasabb rendű lenne az alacsony szintű társaihoz képest – ennek inkább az ellenkezője lehet igaz a számítógép működéséről való ismeretek mélysége tekintetében. A magas szintű elnevezés sokkal inkább arra utal, hogy nagyobb mértékű az elvonatkoztatás (absztrakció) a gépi kódtól. A magas szintű nyelvekben a regiszterek, memóriacímek és a vermek helyett változókkal, tömbökkel és összetett (komplex) aritmetikai vagy logikai kifejezésekkel lehet dolgozni. Ezen kívül nincsenek bennük olyan műveleti kódok, amelyek közvetlenül gépi kódra lehetne fordítani, mint az alacsony szintű (pl. assembly) nyelvekben. Ezen kívül jelen lehetnek bennük karakterlánc (string) kezelő rutinok, objektumorientált nyelvi funkciók és file input/output.
Általánosságban elmondható, hogy míg a magas szintű programok az összetett programok írását egyszerűbbé teszik, addig az alacsony szintű nyelveken hatékonyabb kódot írhatunk. A magas szintű nyelvekben az összetett elemeket fel lehet bontani egyszerűbb, de még mindig meglehetősen komplex elemekre, amelyekhez a nyelvben találunk absztrakt eszközöket, és ez megkíméli attól a programozót, hogy mindig újra „feltalálja a spanyolviaszt”. Emiatt az olyan kódokat, amelyeknek különösen nagy sebességgel és hatékonysággal kell futniuk, gyakran akkor is alacsony szintű nyelven írják, ha egy magasabb szintű nyelven könnyebb lenne.
Ennek ellenére az újabb mikroprocesszor-architektúrák egyre nagyobb bonyolultsága/ összetettsége miatt a magas szintű nyelvekhez íródott, jól tervezett fordítók gyakran hatékonyabb programokat hoznak létre alacsony szintű nyelveken írt társaikhoz képest.
Általánosságban elmondható, hogy a magas szintű nyelvek forrása tartalmaz

- Fogalom meghatározásokat (definiciókat), melyben többnyire adatokat, műveleteket határozunk meg, melyeket a továbbiakban használni szeretnénk.
- Végrehajtandó részeket, melyeket a fordítás után a ténylegesen futó programban végrehajtható utasításokat hoznak létre.
- Kapcsolódási pontokat a külvilághoz, azaz olyan részeket melyek meghatározzák a más programforrások által használható adatokat vagy eljárásokat.

Példa magas szintű programnyelvre:
C nyelvű forrás

```c
// Kapcsolat a külvilággal
#include <stdio.h>

// Deklaráció:
int i;
float a;
// Végrehajtható kód
int main(int argc, char ** argv)
{
  printf(„Hello word”);
}
```

1. ábra Mintaprogram „C” programozási nyelven

Basic nyelvű forrás

```
' Allow easy reference to the System namespace classes.
Imports System

' This module houses the application's entry point.
Public Module modmain
   ' Main is the application's entry point.
   Sub Main()
     ' Write text to the console.
     Console.WriteLine ("Hello World using Visual Basic!")
   End Sub
End Module
```

2. ábra Mintaprogram „Basic” programozási nyelven

Alacsony szintű programnyelv jellemzői

Alacsony szintű programozás során közvetlenül a számításokat végző processzor hardware utasításait kezeljük, azaz azokat írjuk be a programunkba. Történeti okokból eleinte ezeket a kódokat kézzel papíron állították elő és azokat írták be a program memóriába.

Első generációs nyelv

Az első generációs programnyelv (1GL) a processzor gépi kódja. Ez az egyetlen olyan nyelv, amelyet a CPU közvetlenül képes értelmezni. Manapság szinte soha senki nem ír programot közvetlenül gépi kódban, mivel ekkor nemcsak hogy számos olyan részletre kellene odafigyelni, melyeket egy magasabb szintű nyelv automatikusan kezelne, valamint minden egyes utasításhoz számkódokat kellene megjegyezni vagy kikeresni egy listából.
Példa: 8080 processzor által végrehajtható kódra

```
3E 00  <- tegyen az 'A' regiszterbe 0 értéket
CD 105 <- Hívja meg a 105 címen levő lejárást
Beírni pedig a következőt kell
1000-t kell írni a cím mezőbe majd a következő kódokat az adatmezőbe
majd az adatmezőbe 3E-t és enter-t kell nyomni a cím növeléséhez
majd az adatmezőbe 00-t és enter-t kell nyomni a cím növeléséhez
majd az adatmezőbe CD-t és enter-t kell nyomni a cím növeléséhez
majd az adatmezőbe 05-t és enter-t kell nyomni a cím növeléséhez
majd az adatmezőbe 01-t és enter-t kell nyomni a cím növeléséhez és így tovább …
A kódok beírását követ a futtatás
```

3. ábra Első generációs gépikódú programozás

Ebben a környezetben a program készítője a készítés közben számol ki minden szükséges adatot címzést, és helyettesít be minden szükséges utasítás kódot. Ez igen fárasztó és könnyen elrontható feladat. A hibázás lehetőségének csökkentésére vezették be a fordító programot, mely képes kezelni az egyszerűbb címek kiszámítását és az utasításkódok behelyettesítését.

Második generációs nyelv

A második generációs nyelvek (2GL) a különböző assembly nyelvek. E nyelvek utasításkészlete a processzortól függő utasításokat tartalmaz. Azért nevezik őket második generációsnak, mert jóllehet nem a CPU saját nyelve, a programozónak mégis ismernie kell a mikroprocesszor egyedi architektúráját (pl.: a regisztereket és az utasításokat). A gépi kóddal

leírható programokat assembly nyelven a programozó számára sokkal olvashatóbb alakban lehet leírni. A különbség csak az írásmódban van a gépi kódhoz képest, illetve az assembler fordító átvehet néhány adminisztrációs feladatot a programozótól, például a kezdőcím allokációját és a program címkéinek innen számított címeit. Az assembly által használt szavakat, aminek a processzor egy-egy utasítása megfeleltethető, *mnemonikoknak* nevezzük.
Példa: 8080-as processzor mnemonikus megjelenésére

```
        leírt kód     komment
        ORG     2000H
Valtozo:DS      2     ; „Valtozo” nevű 2 byte-os terület létrehozása
        ORG     1000H ; elhelyezés számláló értéke legyen 1000 hexa
        MOV     A,00H ; 'A' regiszterbe tegyen 0 értéket
        CALL    105H  ; hívja meg a 105H címen található eljárást
                      ; ehhez az utasításhoz tartozik stack kezelés melyet a
                      ; processzor HW-e támogat
```

4. ábra Második generációs Assembly programozás

Ebben a környezetben jelent meg a kommentezési lehetőség nagyban segítve a programok utólagos olvashatóságát. További lehetőség, hogy bizonyos helyeket memória címeket un. címkékkel lehetett ellátni, és később hivatkozni ezekre, így támogatja a fordító a címzési hibák elkerülését.
Ebben a környezetben került elő, azaz igény, hogy több mint egy fordítási egység, azaz forrás file alkalmazása esetén valahogy hivatkozni kell a másik egységben meghatározott változóra, azaz annak címére, vagy máshol megadott eljárásra. A probléma megoldására vezették be a program összefűző (linker) fogalmat, ill. eljárást.
Assembley minta ARM processorhoz

```
    THUMB               ; utasítja a fordítót a THUMB utasításkészlet kezelésére
    AREA |.data|,DATA,ALIGN=2
msg                     ; Az msg egy címke azaz ennek hatására ismert a „Hello, ARM!”
                        ; kezdocíme
    DCB     "Hello, ARM!\n"    ; Ascii kódolással lehelyezett szöveg
len equ .-msg           ; A len egy assembly címke melynek értéke az msg string hossza
                        ; ezt az cimke helyének és a msg helyének különbséges adja
    AREA |.text|,CODE,READONLY,ALIGN=2              ; Programterületre való elhelyezést jelent
                        ; utasítja a fordítót, hogy végrahajtanto területre
                        ; tegye a következoket
    EXPORT _start       ; utasítja a fordítót, hogy a _start cimkét tegye fordításon kívül is
                        ; elérhetové
    ENTRY               ; a linkernek szól, azt jelenti, hogy a program futása
                        ; a _start cimkén indul. Az operációs rendszer a betöltés után ide
                        ; adja vezérlést
_start                  ; létrehozza a _start cimkét
;   syscall write(int fd, const void *buf, size_t count) ez komment
;   magasszintu utasításnak megfelelo asm kód következik
    mov     r0, #1      ; fd -> stdout azaz az r0 regiszterbe „1”-et kell írni
    ldr     r1, msg     ; buf -> msg a kivitelre használt buffer (msg) kezdocíme
    mov     r2, #len    ; count -> len(msg) a kivitel hossza
    mov     r7, #4      ; write a 4-es rendszerhívás (syscall #4) tehát
                        ; az r7-be 4-et kell írni
    swi     #0          ; syscall végrehajtása

;   syscall exit(int status) kilépés a programból
    mov     r0, #0      ; status -> 0 azaz az r0 regiszterbe 0-t kell írni
    mov     r7, #1      ; exit az 1-es rendszerhívás (syscall #1) tehát
                        ; az r7 regiszterbe 1-et kell írni
    swi     #0          ; syscall végrehajtása
    end
```

5. ábra Mintaprogram ARM Assembly programozási nyelven

Ebben az egyszerű programrészletben is látszik, hogy szükség van a más file-kban megadott erőforrások kezelésére.
Általánosságban elmondható, hogy több forrásfile felhasználásával készült program fordítása esetén az egyes források lefordítását követően szükség van egy összefűzési folyamatra. Ahhoz viszont, hogy az összefűzés működhessen, szükség van bizonyos adatok és táblázatok létrehozására a fordítási folyamat során.

Fordítás és összefűzés folyamata

A fordító programok által biztosított eredmény az object kód. Az object kód tartalmazza azokat az információkat melyek a forrás file-ból és a fordítás folyamatából származnak.
A fordítási folyamat során keletkezik néhány információs táblázat. Ezen táblázatok tartalmazzák a fordítás eredményét.

- Meghatározott deklarált változók elhelyezését. Szokás a változókat csoportosítva kezelni aszerint, hogy kell-e kezdeti értéket adni, vagy 0 kezdeti értéket kell adni.
- A táblázatot, ami a létrehozott és mások számára is elérhető változók nevét és címét tartalmazza.
- A táblázatot, amely a más fordítási egységben létrehozott és ott nyilvánosan elérhetővé tett változóra való hivatkozást tartalmazza.
- Meghatározott eljárások felfordított kódját tartalmazza. Beszéljük meg!
- A táblázatot melyben az eljárások neve és címe van feltüntetve, ami más fordítási egységek számára is nyilvánosan elérhető.
- A táblázatot melyben a más fordítási egységekből használt eljárások vannak feltüntetve.
- A magas szintű programozási nyelven írt forrás fordításánál a programnyelv fordítója előre meghatároz és library-ban elkészít olyan eljárásokat, melyeket a magasszintű nyelv használ. Így szükséges a fordítás eredményét képező állományban rögzíteni az alkalmazott library nevét.

A következő feladat az egyes fordítási termékek összefűzése, azaz a linkelés. A linker beolvassa az összes felhasználó által megadott forrás fordítási eredményeit. Összegzi és behelyettesíti a nyilvánossá tett és máshol keresett változók és eljárások táblázatát. Szükség esetén a rendelkezésére álló library-k felhasználásával további hivatkozások feloldása is megtörténik.

Fordítási lista

Az assembly mintaprogram fordítási listája.
Az első részben a fordítás tényleges konstansokat, azaz programot és szövegkonstansokat tartalmazó része látható.
Az oszlopok jelentése:

1. A forrásfile sorának sorszáma
2. Az elhelyzés számláló értéke (memória címhez hasonló érték, hexadecimális formában jelenik meg).
3. Az adott címen elhelyezett érték, ez vagy konstans mint pl a „Hello, ARM” szöveg elemei, vagy utasításkód és immediate operandus.
4. Az eredeti forrás adott sora. Ebből keletkezett a fordítási érték, ami a 3. sorban látható.

```
ARM Macro Assembler    Page 1

    1 00000000                  THUMB               ; utasítja a fordítót a THUMB utasításkészlet kezelésére
    2 00000000                  AREA    |.data|,DATA,ALIGN=2
    3 00000000          msg                         ; Az msg egy címke azaz ennek hatására ismert a „Hello, ARM!”
    4 00000000                                      ; kezdocíme
    5 00000000 48 65 6C
               6C 6F 2C
               20 41 52
               4D 21 0A          DCB    "Hello, ARM!\n" ; Ascii kódolással lehelyezett szöveg
    6 0000000C 0000000C
                        len     equ     .-msg       ; A len egy assembly címke melynek értéke az msg string hossza
    7 0000000C                                              ; ezt az cimke helyének és a msg helyének különbséges adja
    8 0000000C                  AREA    |.text|,CODE,READONLY,ALIGN=2 ; Programterületre való elhelyezést jelent
    9 00000000                                      ; utasítja a fordítót, hogy végrahajtanto területre
   10 00000000          ; tegye a következoket
   11 00000000                  EXPORT  _start      ; utasítja a fordítót, hogy a _start cimkét tegye fordításon kívül is
   12 00000000                                      ; elérhetové
   13 00000000                  ENTRY               ; a linkernek szól, azt jelenti, hogy a program futása
   14 00000000                                      ; a _start cimkén indul. Az operációs rendszer a betöltés után ide
   15 00000000                                      ; adja vezérlést
   16 00000000          _start                      ; létrehozza a _start cimkét
   17 00000000                                      ;   syscall write(int fd, const void *buf, size_t count) ez komment
   18 00000000                                      ;   magasszintu utasításnak megfelelo asm kód következik
   19 00000000 F04F 0001        mov     r0, #1      ; fd -> stdout azaz az r0 regiszterbe „1”-et kell írni
   20 00000004 F85F 1004        ldr     r1, msg     ; buf -> msg a kivitelre használt buffer (msg) kezdocíme
   21 00000008 F04F 020C        mov     r2, #len    ; count -> len(msg) a kivitel hossza
   22 0000000C F04F 0704        mov     r7, #4      ; write a 4-es rendszerhívás (syscall #4) tehát
   23 00000010                                      ; az r7-be 4-et kell írni
   24 00000010 DF00             swi     #0          ; syscall végrehajtása
   25 00000012
   26 00000012          ;   syscall exit(int status) kilépés a programból
   27 00000012 F04F 0000        mov     r0, #0      ; status -> 0 azaz az r0 regiszterbe 0-t kell írni
   28 00000016 F04F 0701        mov     r7, #1      ; exit az 1-es rendszerhívás (syscall #1) tehát
   29 0000001A                                      ; az r7 regiszterbe 1-et kell írni
   30 0000001A DF00             swi     #0          ; syscall végrehajtása
   31 0000001C                  end
```

6. ábra Fordítási lista ARM Assembly fordítást követően

Egy adminisztratív üzenet következik, ami a fordító paraméterezését mutatja meg. Ezt a sort a fejlesztőkörnyezet állította össze. A következőkben összefoglalót találunk arról, hogy az egyes területeken definiált értékek mekkorák, hogy lettek meghatározva és hol lettek felhasználva.

```
Command Line: --debug --xref --width=132 --cpu=Cortex-M4.fp --apcs=interwork --depend=.\testasm.d -o.\testasm.o -IG:\Arm_tárgy\RTE -
IC:\Keil_v5\ARM\PACK\Keil\STM32F3xx_DFP\1.0.1\Device\Include --list=.\testasm.lst TESTasm.s
```

```
ARM Macro Assembler    Page 1 Alphabetic symbol ordering
Relocatable symbols

.data 00000000

Symbol: .data
   Definitions
      At line 2 in file TESTasm.s
   Uses
      None
Comment: .data unused

msg 00000000

Symbol: msg
   Definitions
      At line 3 in file TESTasm.s
   Uses
      At line 6 in file TESTasm.s
      At line 20 in file TESTasm.s

2 symbols
```

```
ARM Macro Assembler    Page 1 Alphabetic symbol ordering
Relocatable symbols

.text 00000000

Symbol: .text
   Definitions
      At line 8 in file TESTasm.s
   Uses
      None
Comment: .text unused

_start 00000000

Symbol: _start
   Definitions
      At line 16 in file TESTasm.s
   Uses
      At line 11 in file TESTasm.s
Comment: _start used once

2 symbols

ARM Macro Assembler    Page 1 Alphabetic symbol ordering
Absolute symbols

len 0000000C

Symbol: len
   Definitions
      At line 6 in file TESTasm.s
   Uses
      At line 21 in file TESTasm.s
Comment: len used once
1 symbol
```

7. ábra Fordítási lista összefoglaló táblázatai ARM Assembly fordítás eredménye

A C mintakód fordítási listája

```
; generated by Component: ARM Compiler 5.04 update 1 (build 49) Tool: ArmCC [5040049]
; commandline ArmCC [--list --debug -c --asm --interleave -o.\testc.o --asm_dir=.\ --list_dir=.\
--depend=.\testc.d          --cpu=Cortex-M4.fp          --apcs=interwork          -O0          -
Ic:\Keil_v5\ARM\Pack\ARM\CMSIS\3.20.4\CMSIS\Include            -IG:\Arm_tárgy\RTE            -
IC:\Keil_v5\ARM\PACK\Keil\STM32F3xx_DFP\1.0.1\Device\Include     -D_RTE_     -DSTM32F37x     --
omf_browse=.\testc.crf TESTC.c]
                        THUMB

                        AREA ||.text||, CODE, READONLY, ALIGN=2

                main PROC
;;;6      // Végrehajtható kód
;;;7      int main(void)
000000  b510               PUSH     {r4,lr}
;;;8      {
;;;9        printf("Hello word");
000002  a002               ADR      r0,|L1.12|
000004  f7fffffe           BL       __2printf
;;;10     }
000008  2000               MOVS     r0,#0
00000a  bd10               POP      {r4,pc}
                        ENDP

                |L1.12|
00000c  48656c6c           DCB      "Hello word",0
000010  6f20776f
000014  726400
000017  00                 DCB      0

                __ARM_use_no_argv EQU 0
```

8. ábra Fordítási lista ARM C fordítás eredménye I

A C mintakód fordítási listája, ha adatterületetet is tartalmaz

```
; generated by Component: ARM Compiler 5.04 update 1 (build 49) Tool: ArmCC [5040049]
; commandline ArmCC [--list --debug -c --asm --interleave -o.\testc.o --asm_dir=.\ --list_dir=.\
--depend=.\testc.d          --cpu=Cortex-M4.fp          --apcs=interwork          -O0          -
Ic:\Keil_v5\ARM\Pack\ARM\CMSIS\3.20.4\CMSIS\Include            -IG:\Arm_tárgy\RTE            -
IC:\Keil_v5\ARM\PACK\Keil\STM32F3xx_DFP\1.0.1\Device\Include     -D_RTE_     -DSTM32F37x     --
omf_browse=.\testc.crf TESTC.c]
                        THUMB

                        AREA ||.text||, CODE, READONLY, ALIGN=2

                main PROC
;;;6      // Végrehajtható kód
;;;7      int main(int argc, char ** argv)
000000  b570               PUSH     {r4-r6,lr}
;;;8      {
000002  4604               MOV      r4,r0
000004  460d               MOV      r5,r1
;;;9        printf("Hello word");
000006  a002               ADR      r0,|L1.16|
000008  f7fffffe           BL       __2printf
;;;10     }
00000c  2000               MOVS     r0,#0
00000e  bd70               POP      {r4-r6,pc}
                        ENDP

                |L1.16|
000010  48656c6c           DCB      "Hello word",0
000014  6f20776f
000018  726400
00001b  00                 DCB      0

                        AREA ||.bss||, DATA, NOINIT, ALIGN=2

                a
                        %        400

                __ARM_use_no_argv EQU 0
```

9. ábra Fordítási lista ARM C fordítás eredménye II

A linker ellenőrzi, hogy minden általa kezelt eljárás és változó címe egyszer és csak egyszer lett definiálva a táblázatokban.
Linker üzenet lehet az

- *Unresolved external,* azaz ha nem találta meg a hivatkozás feloldását
- *Multiple definition,* azaz több mint egy alkalommal határozta meg a cím értékét a linker számára.

Mindkét eset azt jelenti, hogy a linker nem tudja létrehozni az eredmény állományt.

Linkelés eredménye

A linkelés eredménye általában az operációs rendszer (pl.: windows, android) által kezelt formátumban állítja elő az eredményt. Ez az eredmény az executable file közvetlenül végrehajtásra még nem alkalmas, mivel operációs rendszer a program indításakor határozza meg a betöltési címet. A betöltési cím ismeretében az exe file-ben a linker által előállított program kód teljes egészét a memóriába tölti és elvégzi a tényleges címek módosítását annak érdekében, hogy futtatható legyen a kód.
A vázolt műveletek után az operációs rendszer a programunk kezdőcímére adja a vezérlést, ami a tényleges indítás. Továbbiakban a programunk az operációs rendszer szabályai szerint működik.
Alapvetően más a linker feladata abban az esetben, ha az eredmény kód egy operációs rendszer nélküli mikrokontrolleren fog futni, mivel ebben az esetben a keletkezett végrehajtható állomány minden címet pontosan tartalmaz, azaz a linkernek meg kell adni a tényleges betöltési címet.
Az első részben a map file tartalmazza (10. ábra Map file összegző része) az összefűzés során felhasznált objekt file-okban meghatározott mindenki által használható címkéket, Ez általában a startup kód referenciával kezdődik, ezután következnek az általunk írt programok, majd a library és más betöltött objetek tartalma.
Bizonyos linkerek képesek arra, hogy a nem hivatkozott, de betöltött programrészleteket kitöröljék. A minta szerint egy két területet érintett.
A következő részben a local, azaz a fordítási egységben használt szimbólumok majd a global azaz mindenki számára nyilvános szimbólumok vannak felsorolva.
Ezt a rész követi a ténylegesen létrehozott betölthető image elrendezése.
A következőkben pedig az összesítések olvashatók.
Végül összesítés olvasható a felhasznált területekről:

- Code + Read-Only adatok A programterület és a program konstansok összessége
- Read-Write és a Zero inicializált adatok. Ez a terület a RAM-ba kerül, de a Read-Write adatok kezdeti értékeit külön a program indítása előtt be kell tölteni (ezt a startup elvégzi).
- Teljes igényelt ROM terület ami a Code + Read-Only adatok + Inicializálási értékek az Read-Write területre.

```
Product: MDK-ARM Standard 5.10
Component: ARM Compiler 5.04 update 1 (build 49)
Tool: armlink [5040049]

==============================================================================

Section Cross References

    startup_stm32f37x.o(STACK) refers (Special) to heapauxi.o(.text) for __use_two_region_memory
    startup_stm32f37x.o(HEAP)  refers (Special) to heapauxi.o(.text) for __use_two_region_memory
    startup_stm32f37x.o(RESET) refers (Special) to heapauxi.o(.text) for __use_two_region_memory
    startup_stm32f37x.o(RESET) refers to startup_stm32f37x.o(STACK) for __initial_sp
    startup_stm32f37x.o(RESET) refers to startup_stm32f37x.o(.text) for Reset_Handler
    startup_stm32f37x.o(.text) refers (Special) to heapauxi.o(.text) for __use_two_region_memory
    startup_stm32f37x.o(.text) refers to system_stm32f37x.o(.text) for SystemInit
    startup_stm32f37x.o(.text) refers to __main.o(!!!main) for __main
    startup_stm32f37x.o(.text) refers to startup_stm32f37x.o(HEAP) for Heap_Mem
    startup_stm32f37x.o(.text) refers to startup_stm32f37x.o(STACK) for Stack_Mem
    system_stm32f37x.o(.text)  refers to system_stm32f37x.o(.data) for SystemCoreClock

    __main.o(!!!main) refers to rtentry.o(.ARM.Collect$$rtentry$$00000000) for __rt_entry

    libinit.o(.ARM.Collect$$libinit$$00000000) refers (Special) to libinit2.o(.ARM.Collect$$libinit$$0000002C) for __rt_lib_init_alloca_1
    libinit.o(.ARM.Collect$$libinit$$00000000) refers (Special) to libinit2.o(.ARM.Collect$$libinit$$0000002A) for __rt_lib_init_argv_1
…
==============================================================================
```

```
Removing Unused input sections from the image.

    Removing system_stm32f37x.o(.rev16_text), (4 bytes).
    Removing system_stm32f37x.o(.revsh_text), (4 bytes).

2 unused section(s) (total 8 bytes) removed from the image.

==============================================================================
```

```
Image Symbol Table

    Local Symbols

    Symbol Name                              Value     Ov Type        Size  Object(Section)

    ../clib/angel/boardlib.s                 0x00000000   Number         0  boardinit2.o ABSOLUTE
…
    TestC.c                                  0x00000000   Number         0  testc.o ABSOLUTE
    dc.s                                     0x00000000   Number         0  dc.o ABSOLUTE
    RESET                                    0x08000000   Section      392  startup_stm32f37x.o(RESET)
    !!!main                                  0x08000188   Section        8  __main.o(!!!main)
    !!!scatter                               0x08000190   Section       52  __scatter.o(!!!scatter)
```

```
    !!handler_copy                  0x080001c4   Section       26  __scatter_copy.o(!!handler_copy)
    !!handler_zi                    0x080001e0   Section       28  __scatter_zi.o(!!handler_zi)
…
    .text                           0x08000226   Section        0  testc.o(.text)
    .text                           0x0800022c   Section       64  startup_stm32f37x.o(.text)
    $v0                             0x0800022c   Number         0  startup_stm32f37x.o(.text)
    .text                           0x0800026c   Section        0  system_stm32f37x.o(.text)
    SetSysClock                     0x0800026d   Thumb Code   188  system_stm32f37x.o(.text)
    .text                           0x08000444   Section        0  heapauxi.o(.text)
    .text                           0x0800044a   Section       74  sys_stackheap_outer.o(.text)
    .text                           0x08000494   Section        0  exit.o(.text)
    .text                           0x080004a0   Section        8  libspace.o(.text)
    .text                           0x080004a8   Section        0  sys_exit.o(.text)
    .text                           0x080004b4   Section        2  use_no_semi.o(.text)
    .text                           0x080004b6   Section        0  indicate_semi.o(.text)
    x$fpl$fpinit                    0x080004b6   Section       10  fpinit.o(x$fpl$fpinit)
    $v0                             0x080004b6   Number         0  fpinit.o(x$fpl$fpinit)
    .data                           0x20000000   Section       20  system_stm32f37x.o(.data)
    .bss                            0x20000014   Section       96  libspace.o(.bss)
    HEAP                            0x20000078   Section      512  startup_stm32f37x.o(HEAP)
    Heap_Mem                        0x20000078   Data         512  startup_stm32f37x.o(HEAP)
    STACK                           0x20000278   Section     1024  startup_stm32f37x.o(STACK)
    Stack_Mem                       0x20000278   Data        1024  startup_stm32f37x.o(STACK)
    __initial_sp                    0x20000678   Data           0  startup_stm32f37x.o(STACK)
```

```
    Global Symbols

    Symbol Name                     Value     Ov Type        Size  Object(Section)


BuildAttributes$$THM_ISAv4$E$P$D$K$B$S$7EM$VFPi3$EXTD16$VFPS$VFMA$PE$A:L22UL41UL21$X:L11$S22US41US21$IEEE1$IW$USESV6$~STKCKD$USESV7$~SHL$OSPACE$R
OPI$EBA8$UX$STANDARDLIB$REQ8$PRES8$EABIv2 0x00000000   Number         0  anon$$obj.o ABSOLUTE
    __ARM_use_no_argv               0x00000000   Number         0  testc.o ABSOLUTE
    __ARM_exceptions_init            - Undefined Weak Reference
…
    __Vectors_Size                  0x00000188   Number         0  startup_stm32f37x.o ABSOLUTE
    __Vectors                       0x08000000   Data           4  startup_stm32f37x.o(RESET)
    __Vectors_End                   0x08000188   Data           0  startup_stm32f37x.o(RESET)
    __main                          0x08000189   Thumb Code     8  __main.o(!!!main)
    __scatterload                   0x08000191   Thumb Code     0  __scatter.o(!!!scatter)
…
    Region$$Table$$Base             0x080004c0   Number         0  anon$$obj.o(Region$$Table)
    Region$$Table$$Limit            0x080004e0   Number         0  anon$$obj.o(Region$$Table)
    SystemCoreClock                 0x20000000   Data           4  system_stm32f37x.o(.data)
    AHBPrescTable                   0x20000004   Data          16  system_stm32f37x.o(.data)
    __libspace_start                0x20000014   Data          96  libspace.o(.bss)
    __temporary_stack_top$libspace  0x20000074   Data           0  libspace.o(.bss)
```

```
Memory Map of the image

  Image Entry point : 0x08000189

  Load Region LR_IROM1 (Base: 0x08000000, Size: 0x000004f4, Max: 0x00040000, ABSOLUTE)

    Execution Region ER_IROM1 (Base: 0x08000000, Size: 0x000004e0, Max: 0x00040000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x08000000   0x00000188   Data   RO          14    RESET               startup_stm32f37x.o
    0x08000188   0x00000008   Code   RO          69  * !!!main             c_w.l(__main.o)
    0x08000190   0x00000034   Code   RO         223    !!!scatter          c_w.l(__scatter.o)
    0x080001c4   0x0000001a   Code   RO         225    !!handler_copy      c_w.l(__scatter_copy.o)
    0x080001de   0x00000002   PAD
    0x080001e0   0x0000001c   Code   RO         227    !!handler_zi        c_w.l(__scatter_zi.o)
    0x080001fc   0x00000002   Code   RO          96    .ARM.Collect$$libinit$$00000000  c_w.l(libinit.o)
…


    Execution Region RW_IRAM1 (Base: 0x20000000, Size: 0x00000678, Max: 0x00008000, ABSOLUTE)

    Base Addr    Size         Type   Attr      Idx    E Section Name        Object

    0x20000000   0x00000014   Data   RW          23    .data               system_stm32f37x.o
    0x20000014   0x00000060   Zero   RW          98    .bss                c_w.l(libspace.o)
    0x20000074   0x00000004   PAD
    0x20000078   0x00000200   Zero   RW          13    HEAP                startup_stm32f37x.o
    0x20000278   0x00000400   Zero   RW          12    STACK               startup_stm32f37x.o
```

```
Image component sizes


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Object Name

        64         26        392          0       1536        852   startup_stm32f37x.o
       472         36          0         20          0       4611   system_stm32f37x.o
         4          0          0          0          0       1831   testc.o

    ----------------------------------------------------------------------
       542         62        424         20       1536       7294   Object Totals
         0          0         32          0          0          0   (incl. Generated)
         2          0          0          0          0          0   (incl. Padding)
    ----------------------------------------------------------------------
      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Member Name

         8          0          0          0          0         68   __main.o
        52          8          0          0          0          0   __scatter.o
        26          0          0          0          0          0   __scatter_copy.o
```

```
        28           0           0           0           0           0   __scatter_zi.o
        12           0           0           0           0          72   exit.o
         6           0           0           0           0         152   heapauxi.o
         0           0           0           0           0           0   indicate_semi.o
         2           0           0           0           0           0   libinit.o
         6           0           0           0           0           0   libinit2.o
         2           0           0           0           0           0   libshutdown.o
         2           0           0           0           0           0   libshutdown2.o
         8           4           0           0          96          68   libspace.o
         0           0           0           0           0           0   rtentry.o
        12           0           0           0           0           0   rtentry2.o
         6           0           0           0           0           0   rtentry4.o
         2           0           0           0           0           0   rtexit.o
        10           0           0           0           0           0   rtexit2.o
        12           4           0           0           0          68   sys_exit.o
        74           0           0           0           0          80   sys_stackheap_outer.o
         2           0           0           0           0          68   use_no_semi.o
        10           0           0           0           0         116   fpinit.o

    ----------------------------------------------------------------------
       282          16           0           0         100         692   Library Totals
         2           0           0           0           4           0   (incl. Padding)

    ----------------------------------------------------------------------

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug   Library Name

       270          16           0           0          96         576   c_w.l
        10           0           0           0           0         116   fz_wm.l

    ----------------------------------------------------------------------
       282          16           0           0         100         692   Library Totals
    ----------------------------------------------------------------------
==============================================================================

      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

       824          78         424          20        1636        7830   Grand Totals
       824          78         424          20        1636        7830   ELF Image Totals
       824          78         424          20           0           0   ROM Totals
```

```
    Total RO  Size (Code + RO Data)                 1248 (   1.22kB)
    Total RW  Size (RW Data + ZI Data)              1656 (   1.62kB)
    Total ROM Size (Code + RO Data + RW Data)       1268 (   1.24kB)

==============================================================================
```

10. ábra Map file összegző része

„C” programozási nyelv

A „C” programozási nyelvvel kapcsolatos általános tudnivalókról a nyelv tervezői által írt könyvből lehet tájékozódni.
http://vili.pmmf.hu/portal/hu/c/document_library/get_file?uuid=85b19ab7-2954-4376-9ddf-7ed57c18f80a&ei=2m6aVOmQCeWvygPgy4KoDA&usg=AFQjCNHgeK5QZUBkWz5xv30G5Ug32rDLsA&bvm=bv.82001339,d.bGQ

A továbbiakban „C/C++” programozási nyelv azon részeit tárgyalom, ami kicsit túlmutat a szokásos programozási feladatokon. A forrásnyelvi program fordítása általában több egymásutáni folyamatból áll, ezt nevezik több menetű fordításnak. A fordítási folyamat egy előfeldolgozási (preprocesszor) résszel kezdődik. Ennek feladata a teljes forrás összeállítása a fordító számára. A preprocesszor működése különböző módokon befolyásolható. A preprocesszor feladata a forrás file előkészítése a fordítónak. Ekkor elő kell készíteni a konstansok behelyettesítését, össze kell állítani a használt és nem használt sorokat, ez utóbbit ki kell hagyni a kimenetből.

C/C++ Előfeldolgozó direktívák

Az előfeldolgozás folyamata vezérelhető, befolyásolható különböző direktívák segítségével. Ebben a fejezetben a teljesség igénye nélkül néhány fontosabb direktívát mutatok be.

#include

Az egyik legegyszerűbb direktíva a #include. Valószínűleg mindenki találkozott már vele és tudja, hogy segítségével a forrásprogramhoz adhatunk előre meghatározott kódrészletet, mely egy file-ban található, és a file nevét kell megadni. Formailag két különböző módon

#define/#undefine

Ez egy egyszerű szöveg helyettesítő eljárás amely a #define utáni szóközökkel elválasztott szövegrész(eket) megkeresi a forrás állományban és a #define diraktíva szövegrésze utáni szóköz és sorvég jellel határolt résszel helyettesíti.
pl.:
#define VALAMI 2
Ezen #define után a VALAMI szövegrész helyére 2-t helyettesít a preprocesszor.
A #undefine egy adott behelyettesítési utasítást hatályon kívül helyez.
Bonyolultabb eset amikor a #define direktívát valami függvény szerű megnevezés követ.
pl.: #define VALAMI(I) …
Ilyen körülmények között a behelyettesítés során a formális paraméter (I) értékét is belyettesíti.

#error/#warning

A fordítót hibaüzenet, vagy figyelmeztető üzenet generálására szólítja fel.

#line

Megadja a forrássor sorszámát, jól használható a #error vagy #warning sorokban.

#if/#elif/#else/#endif/#ifdef/#ifndef

Feltételes fordítást jelent, azaz ha az #if mögött található kifejezés igaz értékű, akkor a következő #if –en belüli utasításokat hozzá teszi a fordítandókhoz. Természetesen létezik az #if-nek #else ága valamint #elif azaz else if ág is megvalósítható. A struktúra vége #endif. További lehetőség, az #ifdef ami egy fordító által kezelt #define meglétét figyeli, ill ennek

ellentéte a #ifndef. Ez utóbbi struktúrának is van #else ága és minden körülményben #endif zárja le a struktúrát.

#pragma

A pragma után a fordító parancssori meghívásakor kiadható opciókat adhatunk meg. Egyik érdekes lehetőség a #pragma pack(1), ami a #pragma szakasz hatálya alatt biztosítja, hogy a változók, adat struktúrák byte folytonosan legyenek elhelyezve.

C/C++ Előfeldolgozó operátotok

operátor

A (#) operátor konvertálja a macro paraméterét stringé.

```
// stringizer.cpp
#include <stdio.h>
#define stringer( x ) printf_s( #x "\n" )
int main() {
   printf_s( "In quotes in the printf function call\n" "\n" );
   stringer( In quotes in the printf function call );
   printf_s( "\"In quotes when printed to the screen\"\n" "\n" );
   stringer( "In quotes when printed to the screen" );
   printf_s( "\"This: \\\" prints an escaped double quote\"" "\n" );
   stringer( "This: \"  prints an escaped double quote" );
}
```

#@ operátor

Karakterré alakítja a macro paraméterét

```
#define makechar(x) #@x

a = makechar(b)

eredmény:
a= 'b';
```

operátor

Továbbítja a bemenő paraméter és kiegészíti vele a makro belsejében a szöveget.

```
#define paster( n ) printf_s( "token" #n " = %d", token##n )
int token9 = 9;

paster(9);
eredmény
printf_s( "token" "9" " = %d", token9 );
printf_s( "token9 = %d", token9 );
```

C/C++ Adatstrukturák (A C programozás nyelv B.W. Kernigen- D.M. Ritchie)

struct

A struct kulcsszó a struktúra deklarációját vezeti be, amely nem más, mint egy kapcsos zárójelek közé zárt deklarációlista. A struct kulcsszót struktúracímke követheti (mint pl. az előbb a date). Ez egy név, amely megnevezi az adott típusú struktúrát, és a továbbiakban rövidítésként használható a részletes deklaráció helyett. A struktúrában előforduló elemeket, ill. változatokat tagoknak nevezzük. Valamely struktúratagnak, ill. -elemnek és egy közönséges (vagyis nem tag) változónak lehet ugyanaz a neve: ebből nem származik zavar, mivel ezek a szövegkörnyezet alapján mindig megkülönböztethetők. Az már természetesen stílus kérdése, hogy az ember ugyanazokat a neveket csak szorosan összetartozó objektumok esetében használja. A tagok listáját lezáró jobboldali zárójelet a változólista követheti ugyanúgy, mint minden alaptípusnál.

```
struct date {
    int day;
    int month;
    int year;
    int yearday;
    char mon_name [4];
}
```

Ez idáig köztudott és ismert. ami kevésbé ismert, hogy lehetséges egy vagy több adatag alkalmazásnál meghatározni annak hosszát bitben az un. bitfielt allokáció alkalmazásával. Ez **`int x:10;`** formában jelenik meg. Ez a lehetőség biztosítja, hogy a processzorban elhelyezkedő regiszterek tartalmát bitenként adjuk meg. pl.:

```
struct date {
    int day:5;
    int month:4;
    int yearday:3;
    int dummy:4;
    int year;
    char mon_name [4];
}
```

Ez a deklaráció összesen 2 byte-on helyezi el a nap, hónap és hét napja adattagot, és még 4 bit hely marad is.

union

A union olyan változó, amely (különböző időpontokban) különféle típusú és méretű objektumokat tartalmazhat oly módon, hogy a fordító ügyel a méretre és illeszkedésre vonatkozó követelmények teljesülésére. A unionok lehetővé teszik, hogy ugyanazon a tárterületen különbözőfajta adatokkal dolgozzunk anélkül, hogy a programban gépfüggő információt kellene elhelyeznünk. Példánkat ismét a fordító szimbólumtáblájából véve tegyük fel, hogy állandóink int-ek, float-ok vagy karaktermutatók lehetnek. Valamely adott állandó értékét a megfelelő típusú változóban kell tárolnunk, ugyanakkor a táblakezelés szempontjából a legkényelmesebb, ha az érték ugyanannyi tárterületet foglal el és ugyanazon a helyen tárolódik, a típusától függetlenül. Ez a union használatának célja - olyan változót létrehozni, amely megengedett módon több típus bármelyikét tartalmazhatja. A mezőkhöz hasonlóan a szintaxis a struktúrákon alapul.

```
union u_tag {
   int ival;
   float fval;
   char *pval;
} uval;
```

Az uval változó elég nagy lesz ahhoz, hogy a három típus közül a legnagyobbat is tartalmazhassa, függetlenül attól a géptől, amelyen a fordítás történik - a programkód független a hardver jellemzőitől. E típusok bármelyike hozzárendelhető uval-hoz, majd kifejezésekben használható mindaddig, amíg a használat következetes: a visszanyert típus a legutoljára tárolt típussal kell, hogy megegyezzen. A programozó feladata annak követése, hogy éppen mit tárolt a union-ban; ha valamit adott típusként tárolunk és más típusúként olvassuk ki, akkor az eredmények gépfüggőek lesznek.

„C/C++” program elhelyezkedése a memóriában

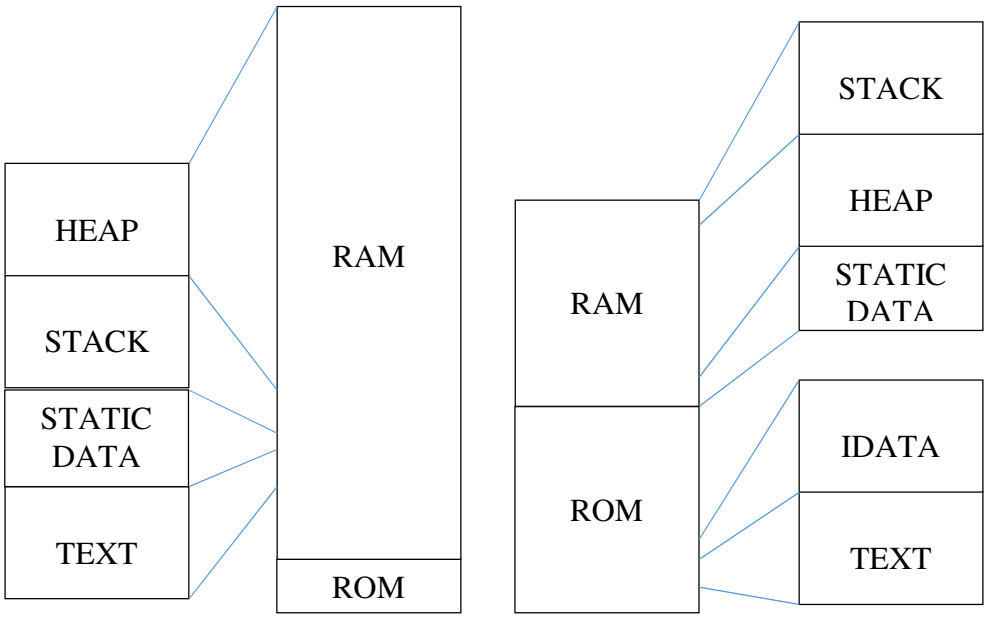
Operációs rendszer a program indításakor a háttértárból érkező programot elhelyezi a memóriában. Egymás mellé kerülnek a futtatható kód, az inicializált memória terület. Továbbiakban a betöltéskor helyet hagy az inicializálatlan változóknak és a Stack-nek. A program elindulása után szükség lehet további memóriaterület foglalására. Ezeket a foglalásokat az operációs rendszer memória managementje valósítja meg. Így az operációs rendszer feladata az allokált területek felszabadítás és újra lefoglalás valamint bővítése és szűkítése is.

ROM és RAM kezelés

A mikrokontrollerekben a memória kiosztást befolyásolja az a tény, hogy az elkészült program nem egy operációs rendszer felügyelet mellet kerül a memóriába egy háttértár eszközről. A mikrokontroller alapvetően kétféle memória típussal rendelkezik, melyből az egyik megőrzi az adatokat akkor is energiamentes, azaz kikapcsolt állapotba került az eszköz, ez az un. ROM (Read Only Memory).

ROM-ban kell elhelyezni a futtatható kódot és az inicializált memória kezdeti értékeit. Mivel a mikrokontroller bekapcsolásakor az írható olvasható RAM (Random Access Memory) terület tetszőleges értékre áll be, így annak inicializálására szükség van. Az inicializáló programrészlet a main() program futtatása előtt hajtódik végre. Az inicializáló program egyrészt átmásolja a meghatározott kezdeti értékeket a nekik kijelölt helyre, másrész a további memória területet null értékkel tölti fel. Ez utóbbi nem minden esetben történik meg.

Általában nem lesz inicializálva a stack. A stack-et általában a RAM terület végére helyezik el, és a stack-et változtató programrészek ellenőrzik, hogy nem létek-e ki a rendelkezésre álló területből. Ha túl sok adatot teszünk a stack-ba akkor kapjuk a STACK OVERFLOW üzenetet, ha meddig túl sok adatot vesszük ki belőle kapjuk a STACK UNDERFLOW üzenetet. Ezeket az üzeneteket a felhasználói programnak kell kezelnie.



1. ábra Program elhelyezése memóriában

A stack-hoz hasonlóan foglal helyet az un heap. Ez a terület szolgál arra, hogy az allokáció során kezeljük a kéréseket, és felszabadításokat. Az allokáció kezelésére a mikrokontrolleres clib-ben a megfelelő eljárások rendelkezésre állnak. A memória allokációs hibák kezelésére,

kijelzésére a felhasználói programnak kell felkészülni a hibát NULL pointer visszaadásával jelzi a könyvtári függvény. Ha a felhasználó program nem kezeli helyesen az allokációkat és NULL pointerrel címez, akkor memória kezelési hiba lép fel.
A program betöltése a memóriába operációs rendszert tartalmazó gépek esetében a felügyelő rendszer feladata. Az operációs rendszer biztosítja a teljes program folyamatos elhelyezését és a futtatásra alkalmas kód címzéseinek beállítását. Lényeges szempont, hogy a heap területet kezelését, azaz annak pillanatról pillanatra változó méretét és a különböző programok számára biztosított helyek menedzselését a rendszer végzi és azok egy közös területen helyezkednek el. (1. ábra Program elhelyezése memóriában)
A memória területek kezelését részben a programozónak kell megcsinálni akkor, ha a program egy mikrokontrolleren operációs rendszer felügyelete nélkül fut. A mikrokontroller használatánál általában a két bekapcsolás között nem, változó részeket a ROM/FLASH területen helyezzük el és a futás közben használt adatokat tesszük a RAM területre. Mivel a Heap kezelő általában növekvő irányban kezeli a memóriát a stack pedig visszafelé töltődik, így a maximális kihasználhatóságot ezek egymásfölé helyezése valósítja meg. (1. ábra Program elhelyezése memóriában)
„C/C++" programban elhelyezett változók és azok helyfoglalása.
Lehetséges un. globál változó létrehozása:

```
int i;
int j = 5;

void main()
{
…
}
```

2. ábra Minden programrészből elérhető global változó

Az 2. ábra megmutatja, hogy global változót hogyan lehet létrehozni. Ezeket a global változókat a fordító program a static adatterületre helyezi el, de mivel a „j" kap kezdeti értéket azt (5) az inicializáló adatok között is feltünteti a fordító. A leírtaknak és a C/C++ nyelvi definiciónak megfelelően a fordító program nem biztosítja, nem is kell neki biztosítani, hogy egymás mellett helyezkedjenek el az adatok. Az ábra szerinti esetben két független szegmensben fognak helyet foglalni a deklarált global változók.
A global változóhoz hasonlóan foglalnak helyet a memóriában a static változók. (3. ábra Korlátozottan elérhető static változó)

```
void main()
{
static int i;
static int j = 5;
…
}
```

3. ábra Korlátozottan elérhető static változó

A static változók elhelyezésével kapcsolatban is hasonló a helyzet, mint a global változóknál. A különbség csupán a változó nevek felhasználási körében van, ami a fordítás során eldől, így nem szükséges a memóriabeli elhelyezésben különbséget tenni.

```
void main()
{
int i;
int j = 5;
…
}
```

4. ábra Függvényen belül elérhető változó

A következő lehetőség, hogy egy függvényen belül csak a függvény által használható változókat hozunk létre. Ezekkel a változókkal kapcsolatban a nyelvi definició azt határozza meg, hogy csak addig vannak elérhető állapotban, ameddig a függvény létezik és fut. A nyelvi definició alapján ezeket a változókat a stack területen helyezi el a fordító. (4. ábra Függvényen belül elérhető változó).
A függvény hívása után és a végrehajtásának megkezdése előtt inicializálni kell a változót, ezt az egyszerű esetben, mint pl.: j = 5; egy konstans beírásával valósítja meg a fordító, ha viszont olyan a helyzet, hogy sok adatot, hosszú adatsort kell definiálni akkor az idata megfelelő részéből másolja ki a változóba a szükséges inicializálási paramétereket. Ez a folyamat igen hosszú is lehet, így nem célszerű a használata.

Stack kezelés C/C++ nyelvben

Mint az az előző részben láttuk a local változókat a „C" fordító a stack-ben helyezi el. Ez a tény korlátozza a lokál változók méretét, hiszen azok összesen nem foglalhatnak el nagyobb helyet, mint ami a stack-ben rendelkezésre áll. Tovább csökkenti a stack ideiglenes tárolásra történő használatát az is, hogy a függvények hívásával kapcsolatos adatok is a stack-be kerülnek.

```
void myfv(int k)
{
int i;
int j = 5;
…
}
```

5. ábra Formális paraméter felhasználása függvényben

A program gyorsítása érdekében a fordító a kisszámú paraméter átadása esetén un. regiszter átadást használ. A regiszter átadás azt jelenti, hogy a függvény hívásakor a formális paramétereket a processzor regiszterében helyezi el. Mivel nagyon korlátozott a processzorban a közvetlen elérhetőséget biztosító regiszterek száma, így nagyon korlátozott a gyors paraméter átadás száma is. Egyetlen paraméter átadása esetén az R0 regisztert fogja használni a fordító.

```
void myfv(int a,int b,int c, int d, int e, int f, int g, int h, int i)
{

}

int main(void)
{
   myfv(1,2,3,4,5,6,7,8,9);

}
```

6. ábra Formális paraméter felhasználása függvényben

```
     7: int main(void)
     8: {
0x080001A6 B500      PUSH          {lr}             // save link register
0x080001A8 B085      SUB           sp,sp,#0x14      // 20 byte helyet csinál a stack-ben
     9:    myfv(1,2,3,4,5,6,7,8,9);
    10:
0x080001AA 2009      MOVS          r0,#0x09         // 9. paraméter
0x080001AC 2108      MOVS          r1,#0x08         // 8. paraméter
0x080001AE 2207      MOVS          r2,#0x07         // 7. paraméter
0x080001B0 2306      MOVS          r3,#0x06         // 6. paraméter
0x080001B2 E9CD3201  STRD          r3,r2,[sp,#0x04] // elhelyezi a stack-ben
0x080001B6 E9CD1003  STRD          r1,r0,[sp,#0x0C]
0x080001BA 2005      MOVS          r0,#0x05
0x080001BC 2304      MOVS          r3,#0x04
0x080001BE 2203      MOVS          r2,#0x03
0x080001C0 2102      MOVS          r1,#0x02
0x080001C2 9000      STR           r0,[sp,#0x00]
0x080001C4 2001      MOVS          r0,#0x01
0x080001C6 F7FFFFE9  BL.W          myfv (0x0800019C)
    11: }
0x080001CA 2000      MOVS          r0,#0x00
```

```
0x080001CC B005      ADD           sp,sp,#0x14
0x080001CE BD00      POP           {pc}
```

Fejlesztő környezet

Manapság a fejlesztőkörnyezet egy integrált fejlesztő környezet (angolul IDE azaz *Integrated Development Environment*). A számítógép-programozást jelentősen megkönnyítő program, részben automatizált programkészítést tesz lehetővé. Az integrált fejlesztői környezetnek alapvető szerepe van a gyors alkalmazásfejlesztésben.

Az IDE-k rendszerint tartalmaznak egy szövegszerkesztőt a program forráskódjának szerkesztésére, egy fordítóprogramot vagy értelmezőt, fordításautomatizálási eszközöket, valamint nyomkövetési, grafikusfelület-szerkesztési és verziókezelési lehetőségeket sok egyéb mellett. A komolyabbakhoz például kiegészítők tömege érhető el, amelyek a teljes rendszerfejlesztés egyéb fázisaiban, például dokumentálás, projectmenedzsment stb. nyújtanak nagy segítséget.

ARM Cortex-M magokat tartalmazó mikrokontrollerek esetében is több különböző fejlesztő környezet szerezhető be a munka megkönnyítésére. Léteznek általános célú környezetek, melyek kiegészíthetők a szükséges egységekkel. Ezek legnagyobb hátránya, hogy pontos ismereteket igényel a telepítésük során, mind az alkalmazott processzorról annak debug funkcióiról a debug interface kezeléséről valamint magáról az IDE-ről. Elvileg léteznek pontos leírások ezek telepítéséről, de a sok különböző verziójú driver nem minden esetben alkalmazható egy környezetben így nagyon nehézkes a kezelésük.

További lehetőség a processzorgyártók által összerakott fejlesztő környezet, mely már sokkal megbízhatóbb működést biztosít, de csak egyféle gyártó által forgalmazott processzorok családjára működnek.

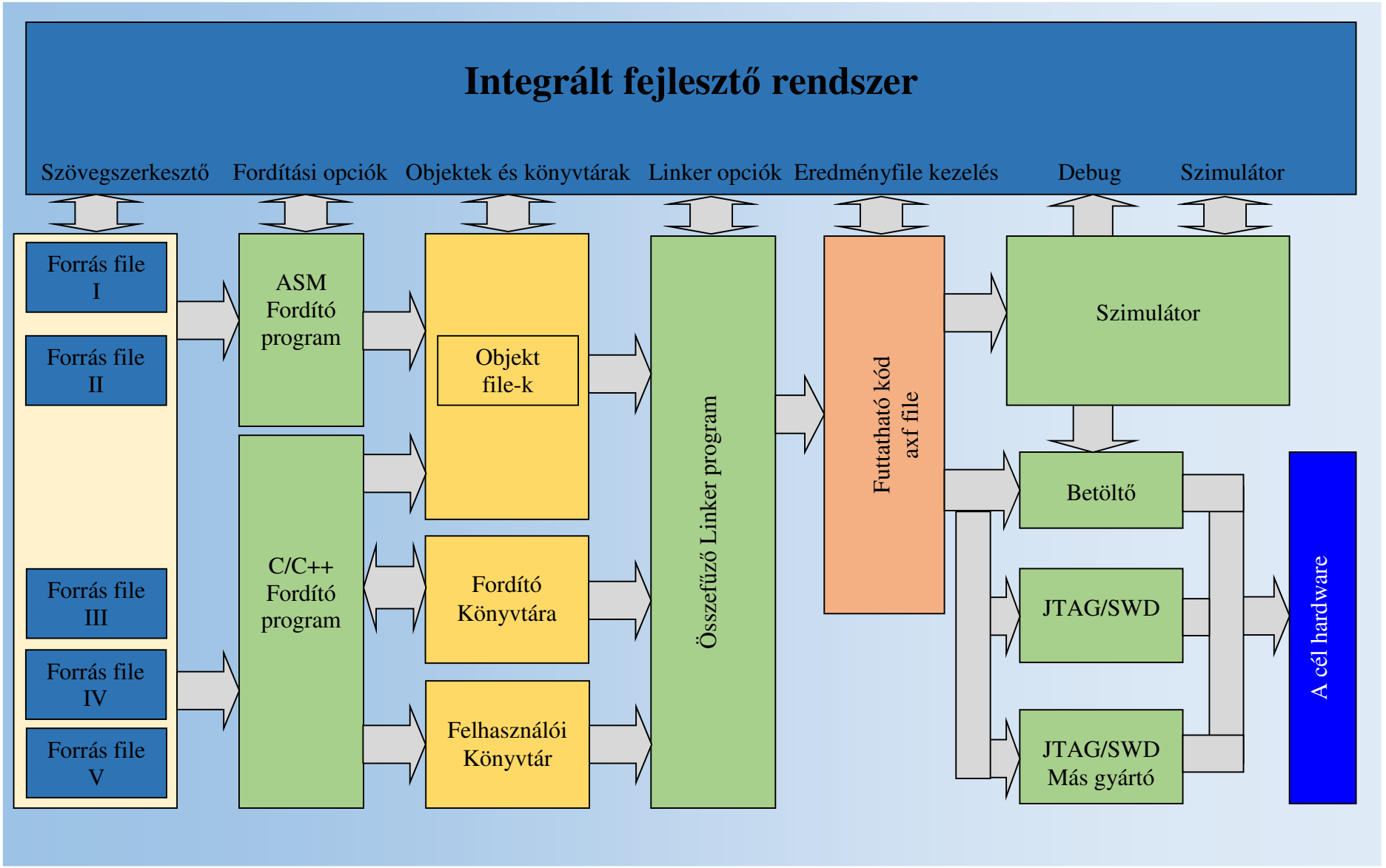
Lehetőség van általános fejlesztő környezetek beszerzésére vagy korlátozott körülmények közötti használatára. Egy fejlesztőkörnyezettel szemben támasztott követelmények megértéséhez annak felépítését kell megfigyelni (1. ábra Fejlesztőkörnyezet felépítése).

A fejlesztőkörnyezetnek kezelni kell tudni a forrásállományokat. Ez általában egy kontext szenzitív editor beépítését jelenti, így segítve a fejlesztő munkáját.

A következő nagyon fontos rész, hogy tudnia, kell kezelni és konfigurálni egy vagy több fordítóprogramot. Manapság általában ez a GCC-nek valamely változatát jelenti. Lényeges szempont, hogy a gyakrabban használt opciókat grafikus felületről lehessen beállítani, és ezen beállítások alapján vezérelje a fordítás folyamatát. Lehessen a segítségével felhasználói könyvtárat létrehozni, amit más projektben fel lehet használni. Hasonlóan tudja kezelni a fordítóhoz tartozó könyvtárat is, mivel előfordulhat, hogy annak részeit vagy egészét újra létre kell hozni. A fordító programokkal kompatibilis linker programmal kapcsolatban is hasonló beállítási vezérlési és feltételeket kell megoldania.

Az elkészül futtatható kóddal kapcsolatban is van további funkciója egy jól használható fejlesztőkörnyezetnek. Jó dolog, ha rendelkezik valamilyen szimulátorral, ami alkalmas a futtatható kód valamilyen szintű tesztelésére. Ez a funkció nagyban tudja segíteni a numerikus számítási hibák megkeresését és kijavítását, azonban általában nem vagy csak korlátozottan lehet a szimulátorokkal a perifériákat kezelni.

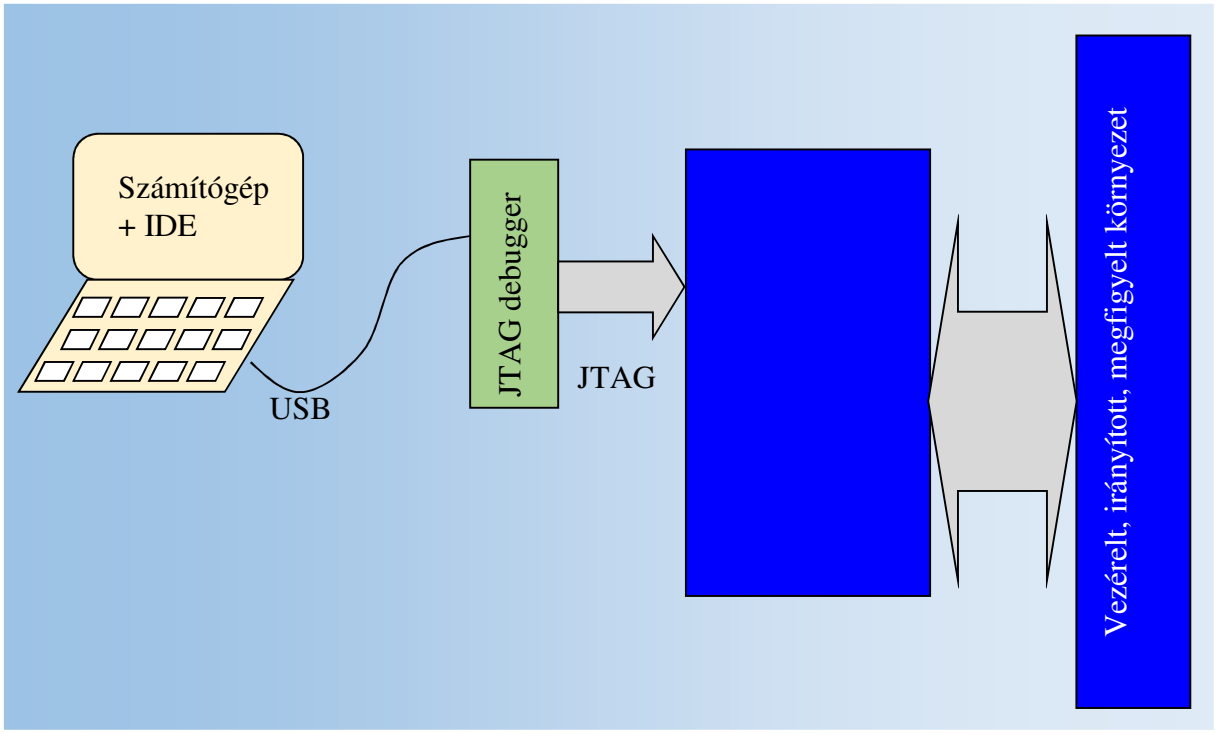
A perifériák működőképességét és a teljes program használhatóságát a debug interface-en keresztül lehet kezelni egy fejlesztő környezettel. Jó, ha a fejlesztő környezet segít a program nyomkövetés nélküli letöltésében valamint a nyomkövethetőség biztosításában. Debug interface-t általában a fejlesztő környezet szállítója is biztosít, de jó, ha a processzor gyártója által adott debug interface-t is tudja támogatni. Ez utóbbi általában jobban, pontosabban tudja kezelni az adott processzort.



1. ábra Fejlesztőkörnyezet felépítése

Program készítő környezet

Nyomkövetés ARM processzorokon



2. ábra Debug környezet felépítése

Az önálló vezérlési, irányítási és megfigyelési feladatok ellátására alkalmas processzoros rendszereket általában nem lehet a saját maga által kezelt perifériák segítségével debuggolni. (2. ábra Debug környezet felépítése). A vezérlési feladatok során pl. egy mosógép funkcióit kell vezérelni, ezek egyike sem alkalmas arra, hogy a még nem működő program futásával kapcsolatban további adatokhoz jussunk. A futás közben keletkező adatok megfigyelésére alkalmas a JTAG interface. Ez az interface lehetőséget biztosít a processzor belső perifériáinak és műveletvégzéseinek megfigyelésére.
Az ilyen kialakítások esetében lehetőség van egy adott pillanatban a processzor futásának megszakítására, ekkor a regiszterek perifériák adatait lehet megfigyelni. A lényeg a program futásának megszakítása, azaz mintegy megállítjuk az időt és körülnézünk a program által kezelt tartalmak között. Sajnos az ilyen módon történő idő megállítás sok esetben nem nyújt megoldást. Gondoljunk csak egy több processzorból álló rendszerre melyben az egyik processzor megáll és a többi pedig tovább fut. Ilyen körülmények között a megfigyelést úgy kell elvégezni, hogy a vizsgált processzor futásának minimális módosításával juthassunk hozzá a kívánt adatokhoz.
A JTAG interface ilyen esetben is tud segíteni.
SWD-vel ki kell egészíteni

SWD / JTAG Connectors and Pinout

(http://www.support.code-red-tech.com/CodeRedWiki/HardwareDebugConnections)

JTAG was the traditional mechanism for debug connections for ARM7/9 parts, but with the Cortex-M family, ARM introduced the Serial Wire Debug (SWD) Interface. SWD is

designed to reduce the pin count required for debug from the 5 used by JTAG (including GND) down to 3. In addition, one of the pins freed up by this can be used for Single Wire Viewing (SWV), which is a low cost tracing technology (which is used by the "Red Trace" functionality within Red Suite).

The SWD/SWV pins are overlaid on top of the JTAG pins as follows:

JTAG Mode	SWD Mode	Signal	Notes
TCK	SWCLK	Clock into the core	Use 10K-100K Ohm pull-down resistor to GND
TDI	-	JTAG Test Data Input	Use 10K-100K Ohm pull-up resistor to VCC
TDO	SWV	JTAG Test Data Output / SWV trace data output	Use 10K-100K Ohm pull-up resistor to VCC
TMS	SWDIO	JTAG Test Mode Select / SWD data in/out	Use 10K-100K Ohm pull-up resistor to VCC
GND	GND	-	-

Usually, MCUs do not include pull-up or pull-down resistors on JTAG/SWD pins. Resistors should be added externally onto the board as detailed above. You may use resistors between 10K and 100K for these signals. This will prevent the signals from floating when they are not connected to anything.

Note that Cortex-M0 does not support SWV trace.

Other signals to note

- RESET
  - Connect this pin to the (active low) reset input of the target MCU
  - We would strongly recommend also including RESET in addition to SWDIO, CLK and GND. For debugging some MCUs, such as NXP LPC11xx, RESET is essential.
- ISP
  - Most NXP MCU's have an ISP pin which (when pulled low) can be used to cause the MCU to enter a bootloader on reset.
  - For example on LPC17xx this is P2.10 and on LPC11xx and LPC13xx it is P0.1.
  - Always ensure that you have a 10K to 100K Ohm pull up resistor on the ISP pin, otherwise you are unlikely to be able to make a successful debug connection.
- RTCK, DBGSEL
  - Some NXP LPC2000 devices have special pins that enable the JTAG interface. For example, on the NXP LPC2129 the signal RTCK must be driven low during RESET to enable the JTAG interface. You may want to add jumpers to your hardware to accomplish this.

Switching between JTAG and SWD modes of debug

Some Cortex-M based MCUs support both SWD and JTAG, others support only SWD (such as NXP LPC11xx and LPC13xx). Where both are supported, there are special sequences defined to switch from JTAG mode (default) to SWD mode (and vice versa) that can sent to the core. This switch sequence uses TMS (SWDIO), and this line is connected for any SWD/JTAG connection

Normally where both modes are supported, Red Suite will default to using SWD mode. However this can be modified by editing the launch configuration for a project.

Logic Levels and Ground

Red Probe+ attempts to adjust logic levels based on the voltage it sees on Vtref referenced to whatever GND it has to work with. The voltage at Vtref is coming from your hardware, thus you need a good GND, shared with your target hardware.

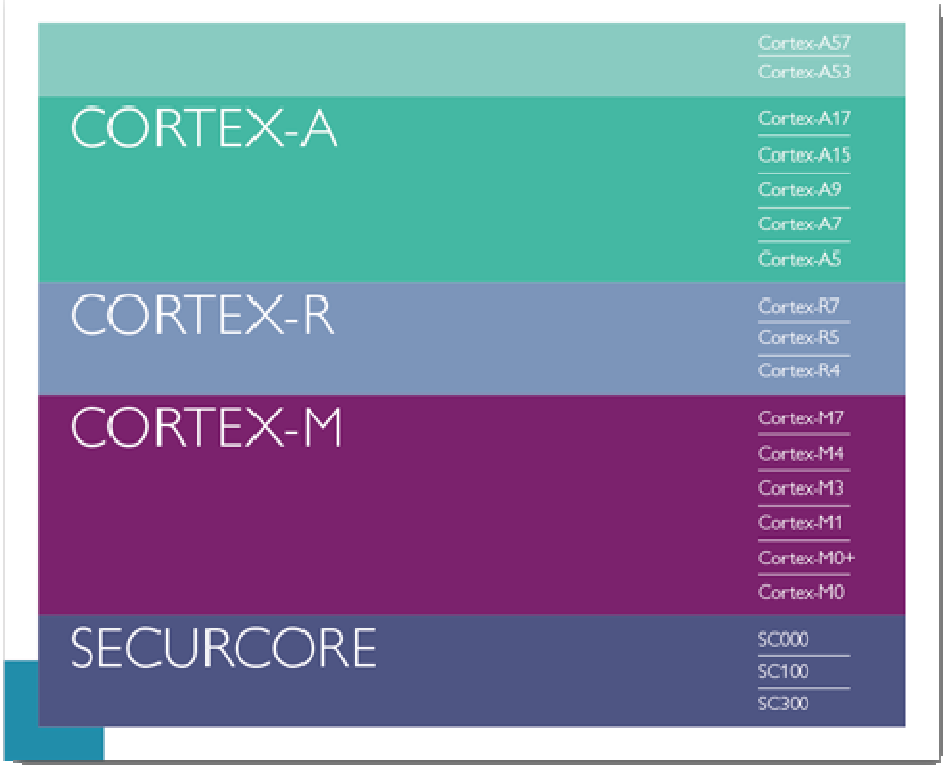
Red Probe+ (and LPC-Link) can be killed (like most USB devices) by excessive over current through ground of the probe and back through the PC used for debugging. The usual cause of this is that your target has it's own PSU and has a ground differential slightly different from your debug PC. Please do not rely on Red Probe+ / LPC-Link to ground your PC to the same potential as your target.

Power

If you have designed your debug circuit according to the specification, you should also check that sufficient power is being supplied to your target. If you are using a USB port on your PC to power your target, make sure that your PC is able to supply the required power over USB - many PC USB ports do not meet USB power requirements.

ARM processzorok felépítése

Az ARM processzorok több családra bontható közösséget alkotnak.
Az Cortex-A sorozatú processzorok, ill. az azokból felépülő mikrokontrollerek a high-end kategóriájú, nagy teljesítményű 32 bites eszközök. Az új fejlesztésű Cortex-A57 és Cortex-A53 már kombinált 32/64 bites felépítésük miatt a következő generáció alapját képezik. Különösen alkalmasak mobil, hálózati eszköz, szerver termékek létrehozására.



1. ábra ARM Cortex sorozatú processzorok családfája (www.arm.com)

A Cortex-A sorozat többi tagja jelenleg is hasonló alkalmazások készítésében nyújt segítséget. Ezekben az eszközökben jellemző, hogy a processzort már UNIX/LINUX operációs rendszerrel látják el és annak felügyelete mellett készítik az alkalmazást.
A Cortex-R sorozat processzorai beágyazott real time rendszerek kialakítására alkalmas eszközök, jellemzőjük az alacsony fogyasztás.
A Cortex-M sorozat processzorai árérzékeny alkalmazások esetében ajánlottak, nagy sebességű és megbízhatóságú fejlesztések általános processzoraként. Használatukkal beágyazott feladatok széles köre valósítható meg (például inteligens szenzorok vagy alacsony fogyasztású távadatfeldolgozó eszközök).
Az SC sorozat az igen alacsony számítási igényű rendszerekben, mint a biztonsági kártyák, vagy érzékelők területén lehet hatékony.

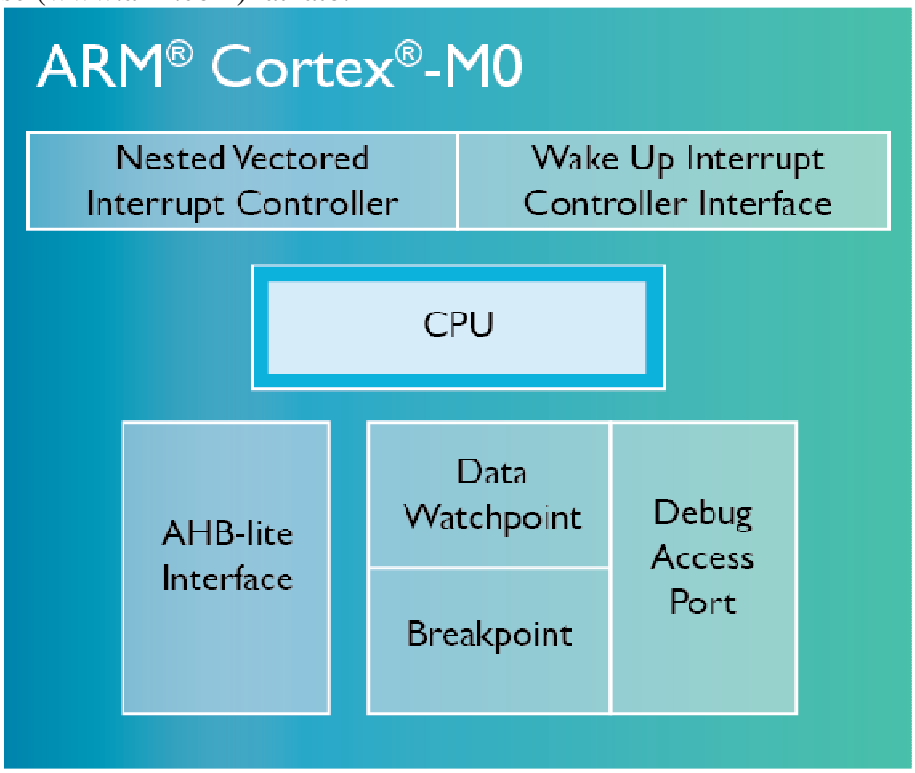
Cortex-M sorozatú ARM processzorok

ARM CORTEX-M sorozatú processzorok többféle felhasználására és kialakításban készülnek. A sorozat skálázható, azaz mindenféle méretű feladat elvégzésére található benne megfelelő elem, sok gyártó készít a különböző magok felhasználásával többkevesebb perifériát tartalmazó megvalósítást. A sorozat közös jellemzője, hogy beágyazott rendszerekhez készül, és kis fogyasztású megoldások készítésére alkalmas. Létezik egy szabványosított szoftver

keretrendszer is, mely megkönnyíti és gyorsítja a munkát, ill. szabadon hozzáférhető DSP library, mely sok szokásos feladat megoldására nyújt lehetőséget, mint pl.: szűrő, szabályozó.

ARM Cortex M0

A család legkisebb tagja. Ennek megfelelően a minimális igényeket támasztó feladatok megvalósítására lehet használni ezeket a processzorokat. A felépítésük a 2. ábra ARM Cortex-M0 felépítése (www.arm.com) látható.



2. ábra ARM Cortex-M0 felépítése (www.arm.com)

A vektoros megszakítás vezérlő lehetővé teszi a különböző helyekről érkező kérések, események priorozált kiszolgálását. A rendszer tartalmaz egy ébredés vezérlő egységet mely megteremti annak lehetőségét, hogy csak a külső események lekezelésének idejére kelljen teljes működést biztosítani a rendszer számára. Hasonló feladatot lát el az AHB-lite interface, melyen keresztül megvalósítható a perifériák ki és bekapcsolása. Kis méretű és lábszámú eszközök esetében különösen fontos a program fejlesztési fázisában, hogy az egyszerű nyomkövetés biztosítható legyen, mivel az ilyen eszközök esetében általában nincs lehetőség kijelző vagy más adatellenőrző eszköz bekötésére. A Data Watchpoint, Breakpoint, Debug access port kezelése biztosítja ezeket a szükséges nyomkövetési lehetőségeket.

Az ARM Cortex-M0 tulajdonságai (www.arm.com)

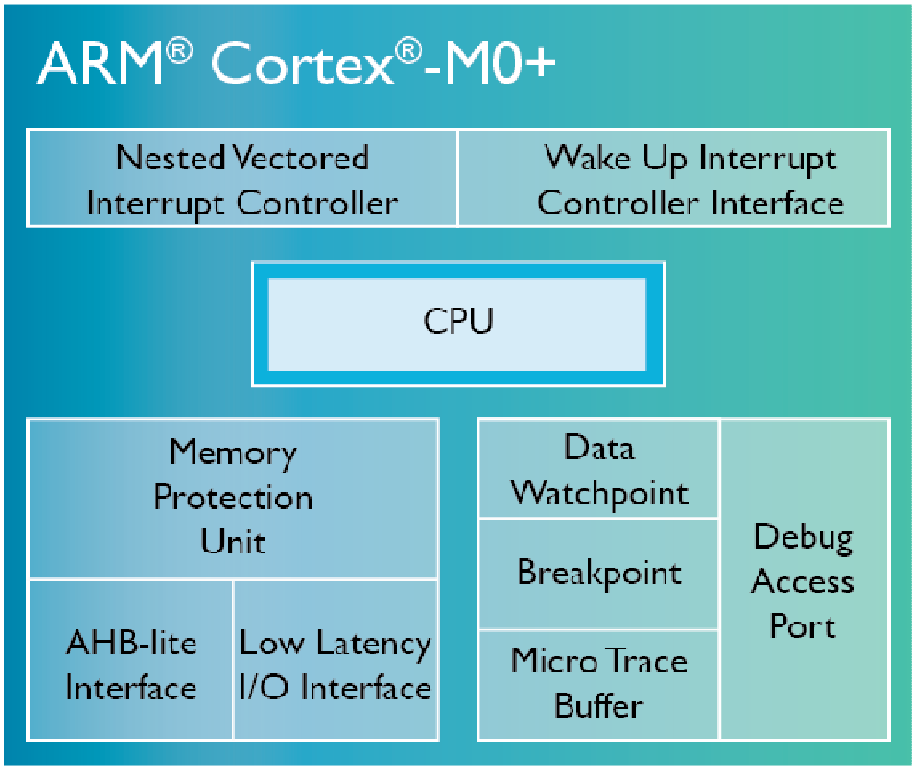
ARM Cortex-M0 Features	
ISA Support	Thumb® / Thumb-2 subset
Pipeline	3-stage
Performance Efficiency	2.33 CoreMarks/MHz*
Performance Efficiency	0.87 / 1.02 / 1.27 DMIPS/MHz**
Interrupts	Non-maskable Interrupt (NMI) + 1 to 32 physical interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability Sleep & Deep Sleep Signals Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Bit banding region can be implemented with Cortex-M System Design Kit
Enhanced Instructions	Hardware single-cycle (32x32) multiply option
Debug	Optional JTAG or Serial-Wire Debug Ports. Up to 4 Breakpoints and 2 Watchpoints

Az ARM Cortex-M0 teljesítőképessége (www.eembc.org)

	STM32F051C8
Type of Platform	Hardware/Production Silicon
Processor Information	
Processor Name	STM32F051C8
Processor Operating Frequency	24
Number of Processor Cores	1
Memory configuration	Internal Flash
Link to Processor Data Sheet (PDF)	http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1574/LN7
Software Environment	
Compiler name and version	IAR 6.60
Compile Flags	--endian=little --cpu=Cortex-M0 --fpu=None ---Ohs --use_c++_inline --no_size_constraints
Operating System Name and Version	
Parallel Execution	-
Benchmark Scores	
CoreMark/MHz	**2.33**
CoreMark	**56.05**
CoreMark/Core	**56.05**

ARM Cortex M0+

A Cortex család M0-ás tagjának kisebb kiegészítésével jutunk az M0+ taghoz. Fejlesztettek a memoria védelmén, valamint az energiafogyasztás csökkentése érdekében az I/O interface-eket alakították át.



3. ábra Cortex-M0+ felépítése (www.arm.com)

Ez a család még hatékonyabban használja ki a rendelkezésre álló energiát. A jellemző fogyasztás értéke mindössze 9,8µW/MHz (minimális konfigurácó esetén), mindemellett a számítási teljesítmény figyelemre méltóan magas.

Az ARM Cortex-M0+ tulajdonságai (www.arm.com)

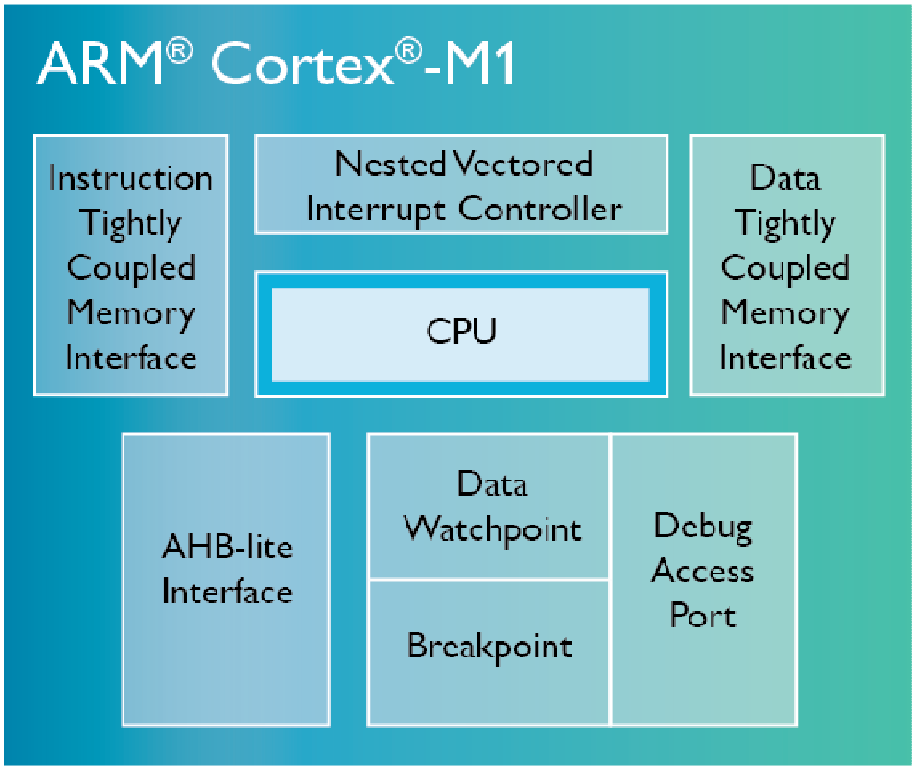
ARM Cortex-M0+ Features	
ISA Support	Thumb® / Thumb-2 subset
Pipeline	2 stage
Performance Efficiency	2.46 CoreMarks/MHz*
Performance Efficiency	0.95 / 1.11 / 1.36 DMIPS/MHz**
Memory Protection	Optional 8 region MPU with sub regions and background region
Interrupts	Non-maskable Interrupt (NMI) + 1 to 32 physical interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability Sleep & Deep Sleep Signals Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Bit banding region can be implemented with Cortex-M System Design Kit
Enhanced Instructions	Hardware single-cycle (32x32) multiply option
Debug	Optional JTAG or Serial-Wire Debug Ports Up to 4 Breakpoints and 2 Watchpoints
Trace	Optional Micro Trace Buffer

Az ARM Cortex-M0+ teljesítőképessége (www.eembc.org)

	Atmel SAM D20
Type of Platform	Hardware/Production Silicon
Processor Information	
Processor Name	Atmel SAM D20
Processor Operating Frequency	24 MHz
Number of Processor Cores	1
Memory configuration	Cache enabled; code running from internal flash
Link to Processor Data Sheet (PDF)	http://www.atmel.com/Images/Atmel-42129-SAM-D20_Datasheet.pdf
Software Environment	
Compiler name and version	IAR-EWARM-6.60
Compile Flags	--endian=little --cpu=Cortex-M0+ --no_size_constraints --const_in_rodata --fpu=None –Ohs
Operating System Name and Version	
Parallel Execution	-
Benchmark Scores	
CoreMark/MHz	**2.46**
CoreMark	**59.04**
CoreMark/Core	**59.04**

ARM Cortex M1

A Cortex M1 processzor az ARM Cortex család azon tagja ami FPGA-n könnyen megvalósítható. A teljesség igénye miatt került bele ebbe az összefoglalóba. A felépítése a 4. ábra ARM Cortex-M1 felépítése (www.arm.com) látható

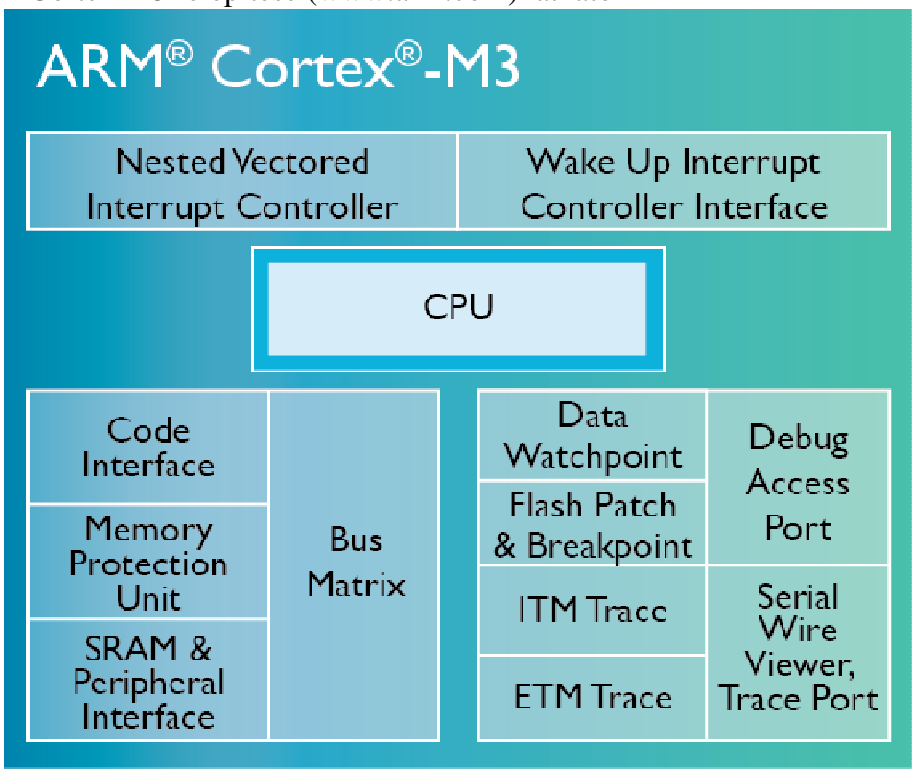


4. ábra ARM Cortex-M1 felépítése (www.arm.com)

A felhasználásnak megfelelően az utasítás és adattároló memória kialakításán változtattak.

ARM Cortex M3

Az M3-as már egy jelentősebben kiépített és nagyobb memóriával rendelkező processzor. Sok általános beágyazott feladat megoldására alkalmas, valódi 32 bites felépítés mellett. Felépítése 5. ábra ARM Cortex M3 felépítése (www.arm.com) látható



5. ábra ARM Cortex M3 felépítése (www.arm.com)

Jelentősen módosították a memória kezelő egységet, melyben finomítottak a védelmi rendszeren és megvalósították a külső memória csatlakoztatási lehetőséget. A programok fejlesztőire is gondoltak, amikor a Debug port lehetőségeit is jelentősen bővítették. Új lehetőség az SWD (Serial-Wire Debug) használata, valamint a trace funkció bővítése.

Az ARM Cortex-M3 tulajdonságai (www.arm.com)

ARM Cortex-M3 Features	
ISA Support	Thumb® / Thumb-2
Pipeline	3-stage
Performance Efficiency	3.32 CoreMark/MHz*
Performance Efficiency	1.25 / 1.50 / 1.89 DMIPS/MHz**
Memory Protection	Optional 8 region MPU with sub regions and background region
Interrupts	Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels
Wake-up Interrupt Controller	Up to 240 Wake-up Interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Integrated Instructions & Bit Banding
Enhanced Instructions	Hardware Divide (2-12 Cycles), Single-Cycle (32x32) Multiply, Saturated Math Support.
Debug	Optional JTAG & Serial-Wire Debug Ports. Up to 8 Breakpoints and 4 Watchpoints.
Trace	Optional Instruction Trace (ETM), Data Trace (DWT), and Instrumentation Trace (ITM)

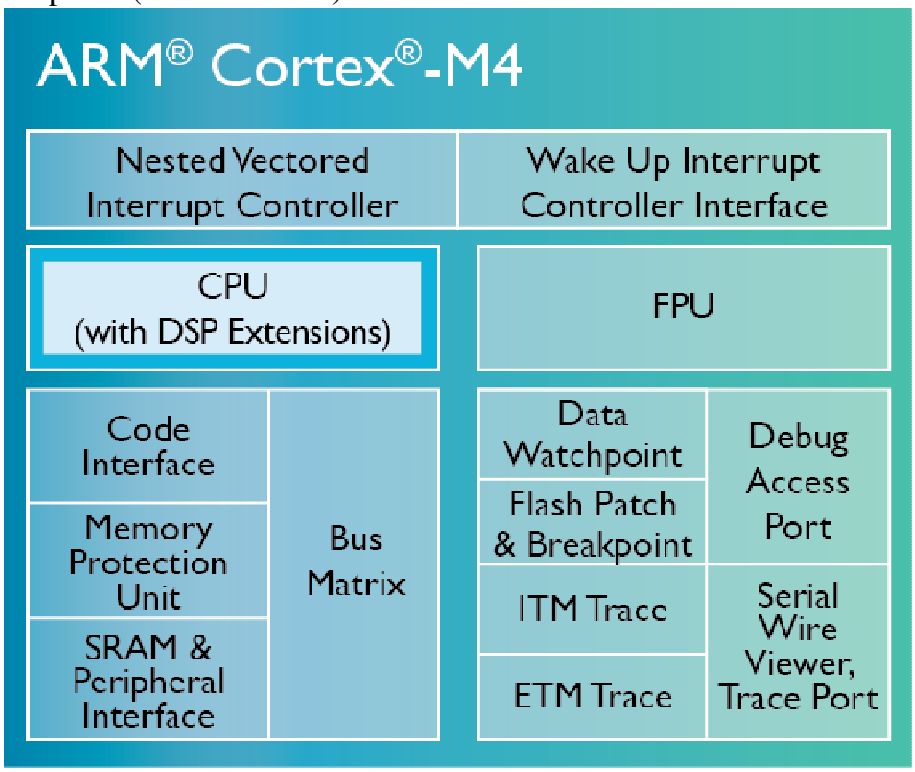
Az ARM Cortex-M3 teljesítőképessége (www.eembc.org)

	Atmel SAM3SD8CAU
Type of Platform	Hardware/Production Silicon
Processor Information	
Processor Name	Atmel SAM3SD8CAU
Processor Operating Frequency	25
Number of Processor Cores	1
Memory configuration	Code running from Flash; 0 wait states
Link to Processor Data Sheet (PDF)	http://www.atmel.com/Images/doc11090.pdf
Software Environment	
Compiler name and version	IAR-EWARM-6.50

Compile Flags	--endian=little --cpu=Corte0x-M3 --no_size_constraints --const_in_rodata --fpu=None –Ohs
Operating System Name and Version	
Parallel Execution	-
Benchmark Scores	
CoreMark/MHz	**3.32**
CoreMark	**82.97**
CoreMark/Core	**82.97**

ARM Cortex M4

A Cortex M4 az M3-as bővítésével jött létre. Az új elem egy lebegőpontos processzor (FPU), mely nagyban segíti a gyors kódok megvalósítását. Az M4 kontroller felépítésa az 6. ábra ARM Cotrec-M4 felépítésa (www.arm.com) látható.



6. ábra ARM Cotrec-M4 felépítésa (www.arm.com)

Az M4-es processzor könnyebb kihasználása érdekében elkészítettek néhány jelfeldolgozó algoritmust, melyek a CMSIS szabadom elérhető és használható könyvtárban kaptak helyet.

Az ARM Cortex-M4 tulajdonságai (www.arm.com)

ARM Cortex-M4 Features	
ISA Support	Thumb® / Thumb-2
DSP Extensions	Single cycle 16/32-bit MAC Single cycle dual 16-bit MAC 8/16-bit SIMD arithmetic Hardware Divide (2-12 Cycles)

ARM Cortex-M4 Features	
Floating Point Unit	Single precision floating point unit IEEE 754 compliant
Pipeline	3-stage + branch speculation
Performance Efficiency	3.40 CoreMark/MHz*
Performance Efficiency	Without FPU: 1.25 / 1.52 / 1.91 DMIPS/MHz** With FPU: 1.27 / 1.55 / 1.95 DMIPS/MHz**
Memory Protection	Optional 8 region MPU with sub regions and background region
Interrupts	Non-maskable Interrupt (NMI) + 1 to 240 physical interrupts
Interrupt Priority Levels	8 to 256 priority levels
Wake-up Interrupt Controller	Up to 240 Wake-up Interrupts
Sleep Modes	Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep & Deep Sleep Signals. Optional Retention Mode with ARM Power Management Kit
Bit Manipulation	Integrated Instructions & Bit Banding
Debug	Optional JTAG & Serial-Wire Debug Ports. Up to 8 Breakpoints and 4 Watchpoints.
Trace	Optional Instruction Trace (ETM), Data Trace (DWT), and Instrumentation Trace (ITM)

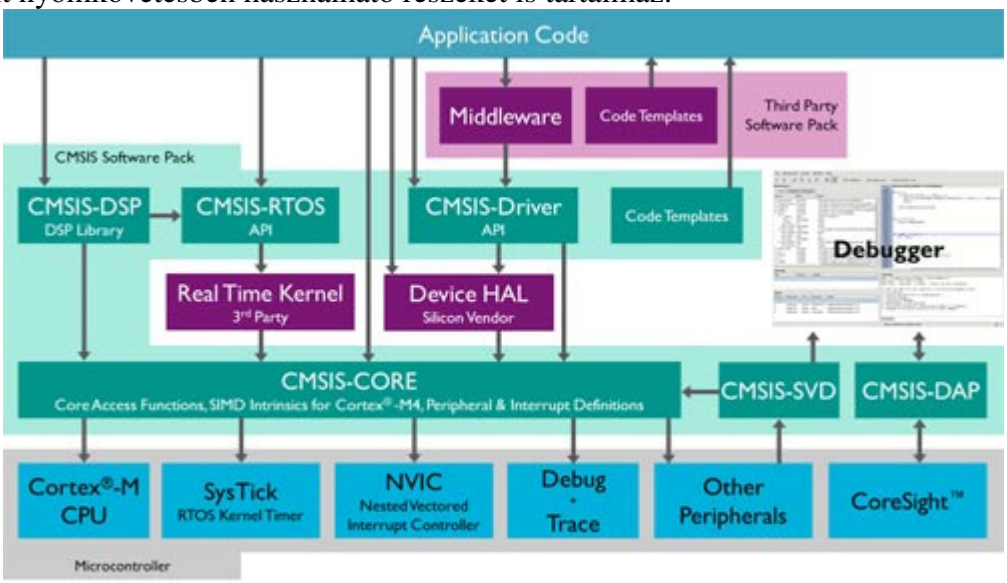
Az ARM Cortex-M4 teljesítőképessége (www.eembc.org)

	Freescale Kinetis K70 90nm
Type of Platform	Hardware/Production Silicon
Processor Information	
Processor Name	Freescale Kinetis K70 90nm
Processor Operating Frequency	150
Number of Processor Cores	1
Memory configuration	Code in internal Flash - Data in internal RAM
Link to Processor Data Sheet (PDF)	http://cache.freescale.com/files/32bit/doc/prod_brief/K70PB.pdf?fpsp=1
Software Environment	
Compiler name and version	IAR v6.50
Compile Flags	--endian=little --cpu=Cortex-M4 -e --fpu=None -Ohs --use_c++_inline --no_size_constraints
Operating System Name and Version	
Parallel Execution	-

Benchmark Scores	
CoreMark/MHz	3.40
CoreMark	510.02
CoreMark/Core	510.02

CMSIS Cortex Microcontroller Software Interface Standard

Az ARM Cortex mikrokontroller szoftware interface szabvány (CMSIS) egy gyártó független harware absztrakciós réteg a Cortex-M sorozatú kontrollerekhez. A CMSIS előre megírt és felhasználható programrészleteket, valamint nyomkövetésben használható részeket is tartalmaz.



7. ábra CMSIS felépítése (www.arm.com)

A CMSIS alkotóelemei:

- CMSIS-CORE: alkalmazási réteg a kümönböző Cortex-M processzorok és perifériáik számára. Azonosan használható csatoló felületeket biztosít a Cortex-M0, Cortex-M3, Cortex-M4, SC000, SC300 eszközök kezeléséhez.
- CMSIS-DRIVER: általánosan használható illesztőprogramokat tartalmaz a middleware-ek számára. Operációs rendszertől függetlenül kezelhetővé teszi a mikrokontroller perifériáit, így segíti a file rendszer vagy kommunkációs feladatok egyszerű megdását.
- CMSIS-DSP: egy Digital Signal Processing library, mely különböző adattípusok alkalmazásával ad megoldást számos gyakorlati életben előforduló problémára. A DSP függvények rendelkezésre állnal q7, q15, q31 valamint lebegőpontos számábrázolással is.
- CMIS-RTOS API: egy közös applikációs interface valós idejű operációs rendszerek könnyű implementálása érdekében.
- CMSIS-PACK: egy szoftware csomagok készítését támogató rendszer, mely XML formátum felhasználásával írja le a csomag elemeit. Tartlmazza a forrásokat, könyvtárakat és a dokumentációkat.
- CMSIS-SVD: System View Description a priférániknak. Egy XML nyelvű leírás mely a periférák használatát tartja nyilván, és segíti a hibakeresést.

- CMSIS-DAP: Debug hozzáférési lehetőség. Szabványosított firmware a debug eszközök által biztosított lehetőségek egységes kihasználására.

Önálló feladat III.

1.) Az interneten található dokumentációk felhasználásával mutassa meg, hogy különböző gyártók az egyes processzorokat milyen felépítéssel és perifériákkal hozzák forgalomba.
2.) Az összegűjtött adatok felhasználásával csoportosítsa a priférikat.

Feladat beadási határidő a következő heti óra.

Timerek az ARM Cortex-M4 mikrokontroller családban

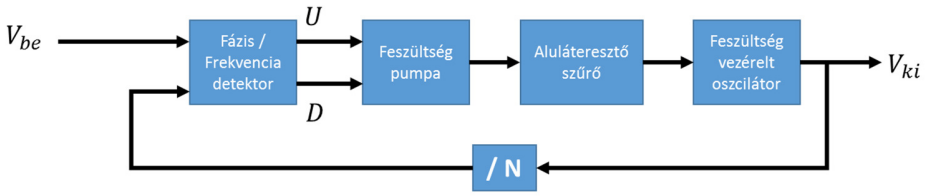
1. A mikrokontrollerekben használt Timer modulokról általánosságban.

A Timer modulok szerves részét alkotják bármely mikrokontroller családnak. Minden mikrokontroller rendelkezik egy központi időzítővel mely biztosít egy stabil órajelet (MHz –es tartományban, ~1% pontossággal) a véges állapotú állapotgép működéshez.

1. ábra Kvarckristályok

A forrása a timer moduloknak lehet egy külső referencia (1. ábra) vagy a belső rendszer óra, minden esetben a működéshez használt vezérlő jel áthalad egy PLL (phase-locked loop) –en egy szabályozókörön mely előállít egy a bemeneti jel fázisához rögzített kimeneti jelet, emellett a frekvenciákat is rögzíti. A PLL nyomon tudja követni a bemeneti és a kimeneti frekvenciát, ami többszöröse is lehet a bemenőnek ezek a tulajdonságok, melyek alkalmassá teszik vezérlő órajelek előállítására, demodulációra és frekvencia szintetizálásra. A PLL (2. ábra) megvalósításánál legyen az analóg vagy digitális ugyan azt a négy alapelemet használjuk:

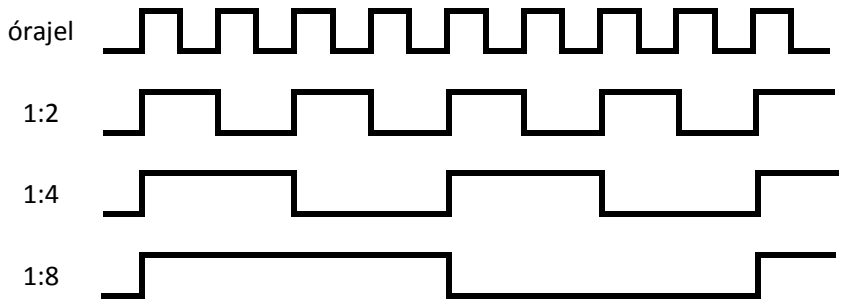
- Fázisdetektor
- alul áteresztő szűrő
- Változtatható frekvenciájú oszcillátor
- visszacsatoló tag mely egy osztót is tartalmazhat



2. ábra PLL blokdiagramja

Általánosságban a timer modulokról elmondható, hogy rendelkeznek a következő tulajdonságokkal: prescaler (előosztó), postscaler (utóosztó), számláló üzemmód stb. Bármely alkalmazás a nagytól a kicsiig rendelkezik a rendszer timer mellet nagy gyakorisággal egy másodlagos timerrel is melynek a forrása lehet a központi vagy egy külső órajel és alacsonyabb frekvencián üzemeltetünk. Az így előállított timerrel általában számláló műveleteket hajtunk vagy speciális időzítési feladatokat végre. Általában ezeket a modulokat beállathatjuk úgy, hogy egy interruptot kiváltódjon a konfiguráció feltételeinek teljesülésekor mely a timer modulhoz tartozó számláló túlcsordulásakor szokott bekövetkezni.

A következő fontos dolog, amiről beszélni kell az, hogy mekkora a timer modulhoz tartozó számláló regiszter mérete ez rendszerint 8, 16- bit. Ez a méret határozza meg a maximális számlálás mértékét. 8-bit esetén ez $2^8 - 1 = 255$, 16- bit esetén pedig $2^{16} - 1 = 65535$. Mennél nagyobb a bit szám annál tovább tud számolni mielőtt, kiváltódik az interrupt. A timer modulok számolás szempontjából háromféle lehet: *up-counter* nullától számolunk felfelé egy fix N-ig, egy adott N-től számolunk a maximális méretig, *dowen-counter* leszámolunk egy adott N-től nulláig. A timer modul időzítését a prescaler és postscaler értékekkel tudjuk tovább finomítani mely értékek rögzítettek 1:2, 1:4, 1:8 stb. Az esetleges prescaler regiszterben beállított értékek azt határozzák, meg hogy hány vezérlő órajel szükséges hogy timer regiszter egy jel pulzust észleljen, ezt láthatjuk a következő ábrán.



1. táblázat méretezési arány

Ezzel a módszerrel lényegében a timer regiszter értéke szorzódik fel egy N prescaleel. A postscaler hasonlóan viselkedik, mint a presceler de itt az érték azt határozza, meg hogy hányszor kell a regiszternek átfordulnia, hogy egy interrupt kiváltódjon ekkor pedig olyan, mint ha egy M postscalel értékel, szorzódna fel a regiszter értéke.

Legyen T az a számláló érték, ami után a timer modul kivált egy interruptot, amikor mind a presaler és postscaler használva van. Up-count esetén: $T = N \times [timer\ regiszter\ érték] \times M$, down-counter estén: $T = N \times [(timer\ regiszter\ érték) - timer\ regiszter\ maximális\ értéke] \times M$.

A középiskolában tanultak alapján mind tudjuk, hogy:

„**Azt a mennyiséget, amely megmutatja a periodikus mozgás egységnyi idő alatt bekövetkező ismétlődéseinek a számát, frekvenciának nevezzük**” $\boldsymbol{f = \frac{1}{T}; [\frac{1}{s} = Hz]}$

Legyen adott egy $20MHz$ –es órajel a fenti definíció alapján ez azt jelenti, hogy 20×10^6 négyszög jel megy végbe egy másodperc alatt és így egy teljes négyszögjel lefutásához pedig $\frac{1}{20\ MHz} = 0.05 \times 10^{-6} s$ szükséges.

Vegyük figyelembe azt, hogy nem minden mikrokontroller használja közvetlenül a rendszer órajelet, valamilyen méretezést alkalmaznak a felhasználáskor emiatt bármely timer modul használatánál az IC adatlapjának áttanulmányozása szükséges, hogy ezeket az arányokat tudjuk.

2. Clock és startup

Minden egyes elindulás vagy újra indulás esetén a belső 16 Mhz –es oszcillátort használja, az eszköz mivel ez az alap értelmezet processzor óra. Ezt követően az adott alkalmazás választja ki a rendszer órát vagy a belsőről vagy a külső kvarc kristályról (1. ábra) működtetjük, ami 4-26 MHz lehet. Külső forrás használása esetén figyelni tudjuk, hogy hibázik-e és ha bekövetkezik, egy hiba vissza tudunk kapcsolni a belső órára és egy szoftveres megszakítás váltódik ki (ha a konfigurálás során ezt beállítottuk). Ez az óra lesz a PLL bemenete és így tudjuk előállítani akár a maximális 168 MHz-es órajelet. Sok prescaler lehetővé teszi, hogy konfiguráljuk a három AHB buszt (max 168Mhz) a nagy sebességű APB (APB2 , max 84MHz) és az alacsony sebességű APB (APB1, max 42 MHz) buszt. Az ARM Cortex-M4 processzor család rendelkeznek egy dedikált PLL -vel az I^2S modulnak (PLLI2S) mely lehetővé teszi a magas minőségű hangmodul működtetését. Így előtudunk állítani minden szükséges mintavételezési frekvenciát 8- 192 kHz –ig.

```
/* ################################    System Clock Config  ######################################### */

  static void SystemClock_Config(void)
  {
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_OscInitTypeDef RCC_OscInitStruct;

    /* Enable Power Control clock */
    __PWR_CLK_ENABLE();

    /* The voltage scaling allows optimizing the power consumption when the device is
       clocked below the maximum system frequency, to update the voltage scaling value
       regarding system frequency refer to product datasheet.  */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /* Enable HSE Oscillator and activate PLL with HSE as source */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 8;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    if(HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
      Error_Handler();
    }

    /* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2
       clocks dividers */
    RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK | RCC_CLOCKTYPE_PCLK1 |
  RCC_CLOCKTYPE_PCLK2);
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
    if(HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
    {
      Error_Handler();
    }
  }
```

1. kód

```
/* ################################    System Clock update  ######################################### */
  void SystemCoreClockUpdate(void)
  {
    uint32_t tmp = 0, pllvco = 0, pllp = 2, pllsource = 0, pllm = 2;

    /* Get SYSCLK source -------------------------------------------------------*/
    tmp = RCC->CFGR & RCC_CFGR_SWS;

    switch (tmp)
    {
    case 0x00:  /* HSI used as system clock source */
    SystemCoreClock = HSI_VALUE;
    break;
    case 0x04:  /* HSE used as system clock source */
    SystemCoreClock = HSE_VALUE;
    break;
    case 0x08:  /* PLL used as system clock source */

    /* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N
    SYSCLK = PLL_VCO / PLL_P
    */
    pllsource = (RCC->PLLCFGR & RCC_PLLCFGR_PLLSRC) >> 22;
    pllm = RCC->PLLCFGR & RCC_PLLCFGR_PLLM;

    if (pllsource != 0)
    {
    /* HSE used as PLL clock source */
    pllvco = (HSE_VALUE / pllm) * ((RCC->PLLCFGR & RCC_PLLCFGR_PLLN) >> 6);
    }
    else
    {
    /* HSI used as PLL clock source */
    pllvco = (HSI_VALUE / pllm) * ((RCC->PLLCFGR & RCC_PLLCFGR_PLLN) >> 6);
    }

    pllp = (((RCC->PLLCFGR & RCC_PLLCFGR_PLLP) >>16) + 1 ) *2;
    SystemCoreClock = pllvco/pllp;
    break;
    default:
    SystemCoreClock = HSI_VALUE;
    break;
    }
    /* Compute HCLK frequency --------------------------------------------------*/
    /* Get HCLK prescaler */
    tmp = AHBPrescTable[((RCC->CFGR & RCC_CFGR_HPRE) >> 4)];
    /* HCLK frequency */
    SystemCoreClock >>= tmp;
  }
```

2. kód

3. System timer

Az ARM Cortex-M4 processzor család egy 24-bites system timerrel rendelkezik ez alapértelmezettben egy dowen-counter a kezdő értéket a `SYST_RVR` regiszterben tudunk megadni. A számlálás akkor kezdődik el ha `SYST_CSR` regiszter engedélyező bitje be van állítva és akkor áll le miután ezt a bitet töröltük.

a. System timer (SysTick) konfigurálása és a regiszterei

A Systick kivétel egy olyan kivétel amit a rendszer időzítő vált ki amikor eléri a nullát. Ez a kivétel szoftveresen is kiváltható. A felhasználok számára a CMSIS kompatibilis eszközillesztő biztosít egy egyszerű függvény hívást(2. kód), ami konfigurálja a SysTicket. A 2. táblázat mutatja azt a négy

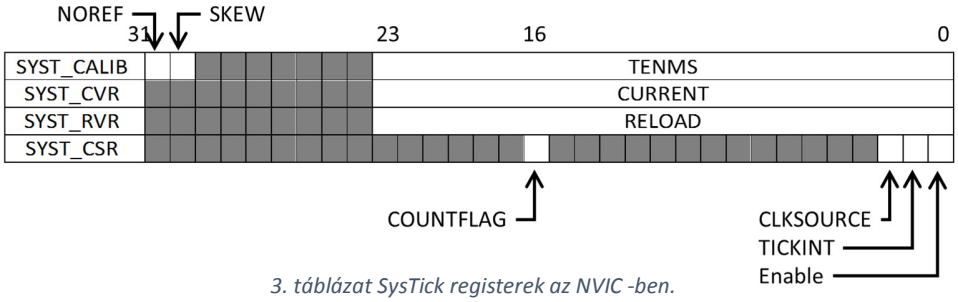
regisztert mellyel a system timert tetszés szerint konfigurálható a felhasználó és az alkalmazás igénye szerint.

	name	register	type	address
a	SysTick Control and Status	SYST_CSR	RW	0xE000E010
b	SysTick Reload Value Register	SYST_RVR	RW	0xE000E014
c	SysTick Current Value Register	SYST_CVR	RW	0xE000E018
d	SysTick Calibration Value Register	SYST_CALIB	RO	0xE000E01C

2. táblázat System timer konfigurációs regiszterek

```
/* ##################################    SysTick function  ######################################### */
/** \ingroup  CMSIS_Core_FunctionInterface
\defgroup CMSIS_Core_SysTickFunctions SysTick Functions
\brief      Functions that configure the System.
@{
*/

#if (__Vendor_SysTickConfig == 0)

/** \brief  System Tick Configuration

The function initializes the System Timer and its interrupt, and starts the System Tick Timer.
Counter is in free running mode to generate periodic interrupts.

\param [in]  ticks  Number of ticks between two interrupts.

\return          0  Function succeeded.
\return          1  Function failed.

\note     When the variable <b>__Vendor_SysTickConfig</b> is set to 1, then the
function <b>SysTick_Config</b> is not included. In this case, the file <b><i>device</i>.h</b>
must contain a vendor-specific implementation of this function.

*/
__STATIC_INLINE uint32_t SysTick_Config(uint32_t ticks)
{
if ((ticks - 1) > SysTick_LOAD_RELOAD_Msk)  return (1);      /* Reload value impossible */

SysTick->LOAD  = ticks - 1;                                  /* set reload register */
NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1);  /* set Priority for Systick Interrupt */
SysTick->VAL   = 0;                                          /* Load the SysTick Counter Value */
SysTick->CTRL  = SysTick_CTRL_CLKSOURCE_Msk |
                 SysTick_CTRL_TICKINT_Msk   |
                 SysTick_CTRL_ENABLE_Msk;                    /* Enable SysTick IRQ and SysTick Timer */
return (0);                                                  /* Function successful */
}

#endif
```

3. kód A SysTick konfigurációs függvény (forrás: core_cm4.h)



3. táblázat SysTick registerek az NVIC -ben.

i. SysTick kontrol és státusz regiszter

A `SYST_CSR` regiszter lehetővé teszi a SysTick használatát. A regiszter a `0x00000000`, vagy a `0x00000004` alapértelmezett értékre áll, ha nem állítunk be neki referencia órajelet.

`[16]` –bit `COUNTFLAG` egyet add vissza, ha a legutóbbi lekérdezés óta leszámolt nulláig.

`[2]` –bit `CLKSOURCE` a vezérlő órajel forrását jelzi: 0 – külső forrás, 1 – belsőforrás.

`[1]` –bit `TICKINT` beállítja a kivételkérelmet: 0 – kivétel nem váltódik ki a leszámlálás végén, 1 – a kivétel kiváltódik a leszámlálás végén.

`[0]` –bit `ENABLE` bekapcsolja a számlálót.

Miután az `ENABLE` értéke 1 lett, a számláló betölti a `RELOAD` értékét a `SYST_RVR` regiszterből és elszámol nulláig. Miután elérte a nullát a `COUNTFLAG` értéke 1 lesz, és adott esetben átállítja a SysTicket attól függően, hogy a `TICKINT` mi az értéke. Ezt követően újra betöltődik a `RELOAD` értéke és újra kezdődik a számláló.

ii. SysTick betöltési érték regiszter

A SYST_RVR regiszter határozza meg a visszaszámlálás kezdőértékét, amit a `SYST_CVR` regiszterbe betölt.

`[23:0]` –bit `RELOAD` az az érték, amit a visszaszámlálás végén betöltődik a `SYST_CVR` regiszterbe, hogy ha a számláló engedélyezve van.

A `RELOAD` értékének meghatározása: a `RELOAD` `0x00000001-0x00FFFFFF` tartományból bármekkora értéket felvehet. Nulla kezdőérték is lehetséges, de semmilyen hatása nem lesz mivel a SysTick kivétel kiváltódik és a `COUNTFLAG` is aktív, amikor 1 –ről 0 –ra váltunk. Egy N periódusú órajel előállításához a `RELOAD` értéknek N-1 et kel megadni.

iii. SysTick aktuális érték regiszter

A `SYST_CVR` tartalmazza a SysTick számláló aktuális értékét.

`[23:0]` –bit `CURRENT` kiolvasása vissza adja a számláló aktuális értékét bármilyen érték írása kinullázza a regisztert és a `SYST_CSR COUNTFLAG` -et is kinullázza.

iv. SysTick kalibrációs értékregiszter

`[31]` –bit `NOREF`, 0 – referenciaóra biztosítva, 1 – nincs referencia óra biztosítva, ha az eszköz nem biztosít referencia órát akkor a SYST_CSR.CLKSOURCE bitet egynek kel venni és figyelmen kívül kell hagyni az írását.

`[30]` –bit `SKEW`, 0 – TENMS értéke pontos, 1 – TENMS értéke pontatlan vagy nem adott, a pontatlan TENMS értéke befolyásolja a SysTick szoftveres valós idejű óraként való használatát (Real Time Clock - RTC).

`[23:0]` –bit `TENMS`, 10ms időzítéshez szükséges `RELOAD` érték, ha ez az érték nulla akkor a kalibrációs érték ismeretlen.

4. Felhasználói timerek

Az STM32F4xx család tartalmaz két advanced-control timert, nyolc általános felhasználású timer, két alap timer és két watchdog timer. Bármely timer számlálója befagyasztható debug üzemmódban.

Timer típus	Timer	felbontás	számlálás módja	prescaler	DMA	komparátor csatornák	Max interfész óra [MHz]	Max timer óra [MHz]
Advanced	TIM1,TIM8	16 –bit	fel, le, fel/le	1 - 2^{16}	✓	4	84	168
GP	TIM2,TIM5	32 –bit	fel, le, fel/le	1 - 2^{16}	✓	4	42	84
	TIM3,TIM4	16 –bit	fel, le, fel/le	1 - 2^{16}	✘	4	42	84
	TIM9	16 –bit	fel	1 - 2^{16}	✘	2	84	168
	TIM10,TIM11	16 –bit	fel	1 - 2^{16}	✘	1	84	168
	TIM12	16 –bit	fel	1 - 2^{16}	✘	2	42	84
	TIM12,TIM14	16 –bit	fel	1 - 2^{16}	✘	1	42	84
Base	TIM6,TIM7	16 –bit	fel	1 - 2^{16}	✓	0	42	84

```
/* ################################    Timer Init ########################################### */

HAL_StatusTypeDef HAL_InitTick (uint32_t TickPriority)
{
    RCC_ClkInitTypeDef sClokConfig;
    uint32_t uwTimclock, uwAPB1Prescaler = 0;
    uint32_t uwPrescalerValue = 0;
    uint32_t pFLatency;
  /* Configure the TIM6 IRQ priority */
          HAL_NVIC_SetPriority(TIM6_DAC_IRQn, TickPriority ,0);
  /* Get clock configuration */
          HAL_RCC_GetClockConfig(&sClokConfig, &pFLatency);
  /* Get APB1 prescaler */
   uwAPB1Prescaler = sClokConfig.APB1CLKDivider;
  /* Compute TIM6 clock */
  if (uwAPB1Prescaler == 0)
  {
    uwTimclock = HAL_RCC_GetPCLK1Freq();
  }
  else
  {
    uwTimclock = 2*HAL_RCC_GetPCLK1Freq();
  }
  /* Compute the prescaler value to have TIM6 counter clock equal to 1MHz */
  uwPrescalerValue = (uint32_t) ((uwTimclock / 1000000) - 1);
  /* Initialize TIM6 */
  TimHandle.Instance = TIM6;
  /* Initialize TIMx peripheral as follow:
       + Period = [(TIM6CLK/1000) - 1]. to have a (1/1000) s time base.
       + Prescaler = (uwTimclock/1000000 - 1) to have a 1MHz counter clock.
       + ClockDivision = 0
       + Counter direction = Up
  */
  TimHandle.Init.Period = (1000000 / 1000) - 1;
  TimHandle.Init.Prescaler = uwPrescalerValue;
  TimHandle.Init.ClockDivision = 0;
  TimHandle.Init.CounterMode = TIM_COUNTERMODE_UP;
  if(HAL_TIM_Base_Init(&TimHandle) != HAL_OK)
  {
    /* Initialization Error */
    Error_Handler();
  } /* Start the TIM time Base generation in interrupt mode */
          if(HAL_TIM_Base_Start_IT(&TimHandle) != HAL_OK)
  {
    /* Starting Error */
    Error_Handler();
  }
  /* Return function status */
  return HAL_OK;
}
```

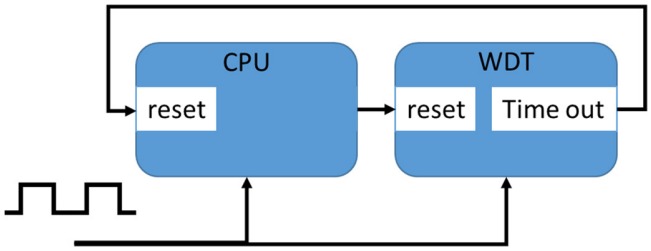
a. 16 és 32 bites timerek
b. Időzítő üzemmód
c. PWM üzemmód
d. Esemény számláló üzemmód

5. Valós idejű óra (Real Time Clock – RTC)
   Feladata, hogy emberi módon értelmezze az idő mérését és nem egyszerű négyszögjelek formájában. A működéshez biztosít a processzor egy belső 32,768 kHz órát. Az RTC perifériák működtetéséhez a CMSIS driver könyvtárban rendelkezésre állnak a megfelelő függvények. A processzor rendelkezik egy backup regisztert az RTC számára, ami biztosítja, hogy újraindulást követően vagy ellem üzemmódban (Vbat) az óra tovább működjön. Az RTC képes a processzort felébreszteni bármely power módból ezt az EXTI vonalra kötött interrupt teszi lehetővé. A könyvtár biztosít időkonvertáló függvényt (az eltelt másodperceket 1970. 01. 01 00:00:00 óta).

```
/* ################################    RTC Init ################################################### */
/** @brief  Initializes the RTC peripheral
* @param hrtc: pointer to a RTC_HandleTypeDef structure that contains the configuration information for RTC.
  * @retval HAL status
  */
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc)
{/* Check the RTC peripheral state */
  if(hrtc == NULL)
  {return HAL_ERROR;}
/* Check the parameters */
  assert_param(IS_RTC_HOUR_FORMAT(hrtc->Init.HourFormat));
  assert_param(IS_RTC_ASYNCH_PREDIV(hrtc->Init.AsynchPrediv));
  assert_param(IS_RTC_SYNCH_PREDIV(hrtc->Init.SynchPrediv));
  assert_param (IS_RTC_OUTPUT(hrtc->Init.OutPut));
  assert_param (IS_RTC_OUTPUT_POL(hrtc->Init.OutPutPolarity));
  assert_param(IS_RTC_OUTPUT_TYPE(hrtc->Init.OutPutType));

  if(hrtc->State == HAL_RTC_STATE_RESET)
  {/* Initialize RTC MSP */
    HAL_RTC_MspInit(hrtc);
  }/* Set RTC state */
  hrtc->State = HAL_RTC_STATE_BUSY;
  /* Disable the write protection for RTC registers */
  __HAL_RTC_WRITEPROTECTION_DISABLE(hrtc);
  /* Set Initialization mode */
  if(RTC_EnterInitMode(hrtc) != HAL_OK)
  {/* Enable the write protection for RTC registers */
    __HAL_RTC_WRITEPROTECTION_ENABLE(hrtc);
    /* Set RTC state */
    hrtc->State = HAL_RTC_STATE_ERROR;
    return HAL_ERROR;
  }
  else
  {/* Clear RTC_CR FMT, OSEL and POL Bits */
    hrtc->Instance->CR &= ((uint32_t)~(RTC_CR_FMT | RTC_CR_OSEL | RTC_CR_POL));
    /* Set RTC_CR register */
    hrtc->Instance->CR |= (uint32_t)(hrtc->Init.HourFormat | hrtc->Init.OutPut | hrtc->Init.OutPutPolarity);
    /* Configure the RTC PRER */
    hrtc->Instance->PRER = (uint32_t)(hrtc->Init.SynchPrediv);
    hrtc->Instance->PRER |= (uint32_t)(hrtc->Init.AsynchPrediv << 16);
    /* Exit Initialization mode */
    hrtc->Instance->ISR &= (uint32_t)~RTC_ISR_INIT;
    hrtc->Instance->TAFCR &= (uint32_t)~RTC_TAFCR_ALARMOUTTYPE;
    hrtc->Instance->TAFCR |= (uint32_t)(hrtc->Init.OutPutType);
    /* Enable the write protection for RTC registers */
    __HAL_RTC_WRITEPROTECTION_ENABLE(hrtc);
    /* Set RTC state */
    hrtc->State = HAL_RTC_STATE_READY;
    return HAL_OK;
  }}
```

6. Watchdog (WDT)

A Watchdog (3. ábra) a timer modulok családjába tartozik a processzor stabil üzemeltetéséhez nélkülözhetetlen. Olyan esetekben játszik szerepet, amikor kivétel váltódik ki, amit kódból nem lehet kezelni vagy visszaállítani (p.l.: nullával való osztás vagy malloc() hiba). A WDT –nak két fő fajtája van a belső és a külső, belső az ahol a



3. ábra A Watchdog tipikus összeállítása

cpu –val össze van integrálva egy szilíciumon található meg, külső, amit a tokozott cpu mellet kell elhelyezni. Az STM32F4 Cortex-M4 processzorokban két független WDT található, az egyik 12 –bites számlálóval, 8 –bites prescaler –rel és 32kHz –es független belső órával rendelkezik, ez biztosítja azt, hogy ha minden tönkre ment a CPU –ban a watchdog akkor is tudjon tüzelni. A másik pedig egy 7 –bites Window Watchdog (WWDT) amit a központi órajel vezérel és rendelkezik egy korai jelző mechanizmussal, amely kiváltódik, még mielőtt újraindulna az eszköz.