Beágyazott és Ambiens Rendszerek

4. gyakorlat tematikája

Az előző gyakorlat során megnéztük, hogyan lehet egy UART perifériát egyszerű módon használni: inicializáció, karakterek írása, karakterek olvasása (blokkoló jelleggel). Ezen gyakorlat során pedig megnézzük, hogyan lehet egy UART perifériát kicsit hatékonyabban (nem blokkoló karakter fogadás, megszakításkezelés) illetve kényelmesebben (UART – stdio összekötés) működtetni.

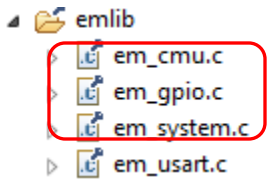
1. UART inicializálás

Az UART inicializálása az előző gyakorlaton megismert módon történik. Az alábbiakban kivonatolva megtalálható az inicializáláshoz szükséges minden adat. Akinek meg van a régi kódja, használhatja azt is.

A következő header fájlokat kell include-olni:

```
#include "em_cmu.h"
#include "em_usart.h"
#include "em_gpio.h"
```

Hozzá kell adni a projekthez az alábbi c nyelvű fájlokat:



A fájlok elérési útvonala:

[telepítés helye]\SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\platform\emlib\src\

Inicializálni kell az UART-ot:

```
        // Enable clock for GPIO
        CMU->HFPERCLKEN0 |= CMU_HFPERCLKEN0_GPIO;

        // Set PF7 to high
        GPIO_PinModeSet(gpioPortF, 7, gpioModePushPull, 1);

        // Configure UART0
        // (Now use the "emlib" functions whenever possible.)

        // Enable clock for UART0
        CMU_ClockEnable(cmuClock_UART0, true);


        // Initialize UART0 (115200 Baud, 8N1 frame format)

        // To initialize the UART0, we need a structure to hold
        // configuration data. It is a good practice to initialize it with
        // default values, then set individual parameters only where needed.
```

```
        USART_InitAsync_TypeDef UART0_init = USART_INITASYNC_DEFAULT;

        USART_InitAsync(UART0, &UART0_init);
        // USART0: see in efm32ggf1024.h

        // Set TX (PE0) and RX (PE1) pins as push-pull output and input resp.
        // DOUT for TX is 1, as it is the idle state for UART communication
        GPIO_PinModeSet(gpioPortE, 0, gpioModePushPull, 1);
        // DOUT for RX is 0, as DOUT can enable a glitch filter for inputs,
        // and we are fine without such a filter
        GPIO_PinModeSet(gpioPortE, 1, gpioModeInput, 0);

        // Use PE0 as TX and PE1 as RX (Location 1, see datasheet (not refman))
               // Enable both RX and TX for routing
        UART0->ROUTE |= UART_ROUTE_LOCATION_LOC1;
               // Select "Location 1" as the routing configuration
        UART0->ROUTE |= UART_ROUTE_TXPEN | UART_ROUTE_RXPEN;
```

Az előző gyakorlathoz képest annyi a különbség, hogy az inicializációs paramétereket a `USART_INITASYNC_DEFAULT` előre definiált tömbbel inicializáltuk, amely történetesen pontosan az általunk kívánt beállításokat tartalmazza, így nem kell az egyes paramétereket külön-külön beállítani.

2. Nem blokkoló karakter fogadás

Az USART_Rx() függvény blokkol (addig nem tér vissza, amíg nem sikerül venni egy karaktert):

```
uint8_t USART_Rx(USART_TypeDef *usart)
{
  while (!(usart->STATUS & USART_STATUS_RXDATAV)) {
  }

  return (uint8_t)usart->RXDATA;
}
```

Látható, hogy addig maradunk egy while ciklusban, amíg a paraméterül átadott UART/USART periféria STATUS regiszterében RXDATAV bit[1] be nem billen. Ha ez megtörtént, akkor a vett karaktert tartalmazó RXDATA (RX Buffer Data Register) tartalmával tér vissza a függvény.

Ez annál az egyszerű programnál, amit az előző gyakorlaton írtunk, nem okozott problémát, mert a főciklus nem csinált mást, mint várt egy karakterre, és ha jött egy, akkor vissza is küldte azt („echo”). Belátható azonban, hogy ha bármi más, hasznos feladata is lenne a főciklusnak, akkor nem lenne célravezető blokkoló módon várakozni bejövő karakterekre.

A blokkolást kikerülhetjük, ha a státusz regisztert mi magunk ellenőrizzük, és csak akkor hívjuk meg az USART_Rx() függvényt, ha van bejövő karakter. A státusz regisztert direkt módon is el tudjuk érni, de használhatjuk az emlib USART API alábbi függvényét is:

```
__STATIC_INLINE uint32_t USART_StatusGet(USART_TypeDef *usart)
{
  return usart->STATUS;
}
```

[1] “RX Data Valid: Set when data is available in the receive buffer. Cleared when the receive buffer is empty.” *EFM32GG Reference Manual*

Ekkor a főciklusunk „echo” funkcióért felelős kódrészlete pl. az alábbi módon nézhet ki:

```
  if (USART_StatusGet(UART0) & USART_STATUS_RXDATAV) {
    USART_Tx(UART0, USART_Rx(UART0));
  }
```

Megjegyzés: mondhatjuk, hogy nem túl hatékony dolog egy regiszter direkt kiolvasása helyett egy függvény meghívása, ami ugyanúgy csak kiolvassa az adott regisztert, majd a kiolvasott értékkel tér vissza. Ennek kiküszöbölésére hivatott a függvény deklarációjában szereplő `__STATIC_INLINE` makró, amelynek az alábbi a definíciója (`cmsis_gcc.h`):

```
  #define __STATIC_INLINE static inline
```

Az inline kulcsszó arra kéri a fordítót, hogy függvény hívás helyett a függvény törzsét szúrja be arra a pontra, ahol eredetileg meghívtuk volna. Így a kecske is jól lakik, és a káposzta is megmarad: áttekinthető kódot tudunk írni teljesítmény vesztés nélkül. (Az inline kulcsszóról annyit érdemes még megjegyezni, hogy ez csak „kéri” – és nem „kötelezi” – a fordítót arra, hogy inline módon kezelje a függvény hívást. Továbbá e kulcsszó hiányában is dönthet úgy a fordító, hogy inline módon kezel egy függvényt.)

További kérdések merülhetnek fel bennünk:

1. Miért tesszük a függvény törzsét a header fájlba (mikor azt a C fájlba szokás)?
2. Miért kell még a static kulcsszó is?
3. Miért kell a kulcsszavakat egy makró mögé rejteni?

Mivel az ezen kérdésekre adandó válaszok már nem képezik szorosan a gyakorlat anyagát, a Függelékben találhatóak az érdeklődőbbek számára.

Ha nem szeretnénk, hogy a karakterek fogadása feltartsa a programunkat, egy másik alternatíva az, ha készítünk egy nem blokkoló függvényt karakterek vételére, pl.:

```
int USART_RxNonblocking(USART_TypeDef *usart)
{
  if (usart->STATUS & USART_STATUS_RXDATAV) {
      return (int)(usart->RXDATA);
  } else {
      return -1;
  }
}
```

Ha van vett karakter (a leggyakoribb, 8 bites UART keret formátum esetén 0...255 tartományban), akkor azzal térünk vissza, ellenkező esetben egy, az érvényes értékkészleten kívüli értékkel (-1). (Emiatt a visszatérési érték már nem uint8_t, hanem int.)

Megjegyzés: több kódolási szabvány[2] sem javasolja, hogy egy függvényből több helyen is visszatérjünk. A magyarázat az, hogy így a kód kevésbé átlátható, és egy korai visszatéréssel esetleg olyan részeit is kihagyjuk a függvénynek, amit nem szerettünk volna. A javasolt megoldás az, hogy csak egy ponton, a függvény végén térjünk vissza. A mi függvényünk elég kicsi ahhoz, hogy e nélkül is áttekinthető, ám nem haszontalan gyakorolni a biztonságosabb programozást megcélzó módszereket:

[2] Pl. a biztonság kritikus rendszerek számára készített MISRA C legújabb (2012) ajánlásai között a 15.5-ös szabály mondja ki, hogy egy függvényből csak egy ponton (a végén) javasolt visszatérni.

```
int USART_RxNonblocking(USART_TypeDef *usart)
{
  int retVal = -1;

  if (usart->STATUS & USART_STATUS_RXDATAV) {
        retVal = (int)(usart->RXDATA);
  }

  return retVal;
}
```

A főciklus vonatkozó kódrészlete pedig valami hasonló lehet:

```
  int ch;
  ch = USART_RxNonblocking(UART0);
  if (ch != -1) {
      USART_Tx(UART0, ch);
  }
```

3. Megszakításkezelés

Az előző pontban részletezett módszerek megoldást kínálnak arra, hogy miképp lehet a főciklust nem fenntartva fogadni karaktereket. Az előző gyakorlat egyszerű példájában más probléma nem merülhet fel, de beláthatjuk, hogy ha a főciklus csinál más hasznosat is, és adott esetben ezt hosszabb ideig végzi, minthogy ismét lekérdezze, hogy van-e új érkezett karakter, akkor adatvesztés léphet fel.

Ez utóbbi problémára megoldást nyújthatnak a megszakítások. A megszakítások tetszőleges kód futását meg tudják szakítani, és a kezelésükre hivatott kódrészlet így el tudja végezni az időkritikus feladatokat (a mi példánknál maradva egy karakter kiolvasását).

Ha egy perifériát megszakítások segítségével szeretnénk kiszolgálni, akkor általánosságban az alábbi lépések szükségesek:

1. Fel kell konfigurálni a kérdéses perifériát, hogy kérjen megszakítást egy adott esemény bekövetkeztekor
2. A megszakításkérés kiszolgálását engedélyezni kell a processzor oldalán
3. A megszakítás vektor tábla megfelelő sorába el kell helyezni a megszakítást kiszolgáló rutinra (ISR[3]) mutató vektort
4. Meg kell írni a megszakítás kezelő rutint
    a. El kell menteni a processzor kontextusát
    b. Ki kell deríteni a megszakítás kérés pontos okát
    c. El kell végezni az adott periféria által kért megszakítás kiszolgálását
    d. Nyugtázni kell az adott periféria által kért megszakítást
    e. Vissza kell állítani a processzor kontextusát
    f. Vissza kell térni a megszakítás kezelő rutinból

A fenti lépések általánosak, nem mindig van szükség mindegyikre. Most nézzük meg ezeket egyesével az általunk használt (ARM Cortex-M3 magú) mikrovezérlő általunk használt (UART0) perifériájára!

[3] Interrupt Service Routine

3.1. A megszakítás kérés felkonfigurálása

Az EFM32 Giant Gecko mikrovezérlők UART/USART perifériái több lehetséges esemény hatására is kérhetnek megszakítást. Ezeket két nagy csoportra bonthatjuk: adással illetve vétellel kapcsolatos megszakítások.

Adással kapcsolatos megszakítások[4]:

- TXC: TX Complete
- TXBL: TX Buffer Level
- TXOF: TX Overflow
- CCF: Collision Check Fail

Vétellel kapcsolatos megszakítások[5]:

- RXDATAV: RX Data Valid
- RXFULL: RX Buffer Full
- RXOF: RX Overflow
- RXUF: RX Underflow
- PERR: Parity Error
- FERR: Framing Error
- MPAF: Multi-Processor Address Frame
- SSM: Slave-Select In Master Mode

Tekintve, hogy a karakterek fogadását kívánjuk megszakításokkal megoldani, számunkra a vétellel kapcsolatos interruptok az érdekesek. MPAF és SSM olyan üzemmódokhoz tartoznak, amiket nem használunk. RXOF, RXUF, PERR és FERR különböző hibák esetén következnek be, minket viszont a sikeresen vett karakterek érdekelnek most elsősorban. Két szóba jöhető megszakítás maradt: RXDATAV és RXFULL. A kettő között azért van különbség, mert az EMF32GG mikrovezérlőkben található UART/USART perifériáknál a vételi buffer két elemű (FIFO jelleggel). RXDATAV akkor igaz, ha van legalább egy, sikeresen vett karakter a bufferben. RXFULL pedig akkor, ha ez a buffer tele van (azaz már kettő karakter is van benne).

> *Megjegyzés*: ha egy harmadik karakter is érkezik, az még nem okoz adatvesztést. A vett karakterek ugyanis bitenként érkeznek, így először egy shift regiszterbe lépkednek be. Onnan továbbítódnak a vételi FIFO-ba (ha van benne hely). Ha nincs, akkor az utolsó marad a shift regiszterben. Így tehát három vett karaktert el tud tárolni a periféria. Ha viszont érkezik egy negyedik is, akkor az már adatvesztést okoz: a jelenleg a shift regiszterben lévő harmadik karaktert felül fogja írni.

Mi alapvetően azonnal szeretnénk egy megszakítást, ha érkezik egy érvényes karakter, ezért az RXDATAV megszakítást fogjuk beállítani. Ehhez az emlib USART_IntEnable() függvényét tudjuk használni!

[4] EFM32_GG-RM.pdf: 17.5.16 - USARTn_IF - Interrupt Flag Register (488. old - 2016-04-28 - Giant Gecko Family - d0053_Rev1.20)
[5] EFM32_GG-RM.pdf: 17.5.16 - USARTn_IF - Interrupt Flag Register (488. old - 2016-04-28 - Giant Gecko Family - d0053_Rev1.20)

Első paramétere a kérdéses periféria: UART0 (nem USART0). A második pedig az engedélyezni kívánt megszakítás, amit az USART_IF_<megszakítás_rövid_neve> define-okkal (efm32gg_usart.h – elérése: em_device.h → efm32gg990f1024.h → efm32gg_usart.h) tudjuk kiválasztani (egyszerre akár többet is, ha összeVAGYoljuk őket).

3.2. A megszakítás kérés engedélyezése

Az előző feladatpontban egyelőre csak azt oldottuk meg, hogy az UART0 periféria kérjen megszakítást, ha van új (még ki nem olvasott) adat a vételi buffereiben. Ez azonban nem jelenti azt, hogy a processzor ki is fogja szolgálni ezt a megszakítást! Ehhez a processzor megszakítás vezérlőjében (NVIC[6]) engedélyezni is kell az adott megszakítás kiszolgálását!

Láttuk, hogy egy adott periféria számos esemény bekövetkeztekor kérhet megszakítást. Azt is tudjuk, hogy bizonyos típusú perifériákból általában több is van egy mikrovezérlőben, továbbá igen sok fajta periféria lehetséges. Mindent összevetve túl sok megszakításkérés lenne. Ezen a Cortex-M típusú processzorok spórolnak picit, és sokszor összevonnak bizonyos megszakításokat. Pl. az UART/USART perifériák felől a processzor már csak kettő megszakításkérést lát: egy vételit és egy adással kapcsolatosat. Azt már a megszakítás kezelő rutinnak kell kiderítenie, hogy ha több forrása is lehetett a megszakításkérésnek, akkor mi volt az egészen pontosan. Mivel mi csak egy vétellel kapcsolatos megszakítást konfiguráltunk fel, ezért ezzel nem kell majd foglalkoznunk.

A mikrovezérlő adatlapjának tanúsága szerint a processzorhoz tehát az alábbi két megszakítás vonal ér el: UART0_RX és UART0_TX. A megszakítás vezérlőben az egyes interrupt forrásokat engedélyezni az NVIC_EnableIRQ() függvénnyel lehet (core_cm3.h, ami áttételesen device.h-ból már be van hivatkozva). Nézzük meg, hogy milyen típusú paramétert vár a függvény, és válasszuk ki a megfelelő értéket (figyelem, itt se keverjük össze az UART0 perifériát az USART0-val).

Az egyes NVIC IT vonalak elnevezése az efm32gg990f1024.h fájlban találhatók.

Általában nem okoz problémát, de jó gyakorlat, ha a megszakítások engedélyezése előtt töröljük a hozzá tartozó megszakítás flag-et, hogy az esetlegesen bennragadt megszakítások ne jussanak érvényre.

3.3. A megszakítás vektor tábla kitöltése

A megszakítás vektor tábla a programmemória elején található. Minden megszakításhoz tartozik egy bejegyzés ebben a táblázatban. Az itt lévő vektor mondja meg, hogy hol van az adott megszakítást kiszolgáló rutin.

Sok architektúrán vektor alatt ugró utasításokat kell érteni, mert a processzor utasításokat vár ebben a táblázatban. Cortex architektúra esetén ide memória címeket kell tenni, mert a processzor nem végrehajtandó utasításként kezeli őket, hanem fixen ugrik, a kérdés csak az, hogy hova.

A megszakítás vektor táblát a startup kód (CMSIS/EFM32GG/startup_gcc_efm32gg.s) előre definiáltan tartalmazza (lásd __Vectors: címke alatt). Minden megszakításhoz rendel egy címet (<megszakítás neve>_IRQHandler alakban). Ezen címek mindegyikének később (a fájl végén) egy makró (def_irq_handler) segítségével ad értéket, mégpedig egy alapértelmezetten használni kívánt rutin címeként: Default_Handler.

[6] Nested Vectored Interrupt Controller

Az alapértelmezett kiszolgáló rutin nem tartalmaz mást, mint egy ugró utasítást (b, mint branch). Az ugráshoz megadott cím a „.”, ami az ARM assembly-ben az éppen aktuális utasítás címeként használható. Azaz kapunk egy végtelen ciklust. Ez arra jó, hogy ha bármilyen oknál fogva érvényre jutna egy olyan megszakítás, amire a mi kódunk nincs fölkészülve, akkor se legyen teljesen kontrollálatlan a végrehajtás.

Rendben, de mit kell tennünk, ha azt szeretnénk, hogy egy általunk megírt ISR hívódjon meg? Ha jobban megnézzük a def_irq_handler makrót, akkor láthatjuk, hogy minden egyes hozzárendeléshez egy „.weak” kulcsszót is hozzáfűz. Ez azt teszi, hogy ezen hozzárendeléseket felül lehet definiálni. Ha tehát írunk egy függvényt <megszakítás neve>_IRQHandler néven, akkor a linker annak címét fogja a vektor tábla megfelelő helyére beszúrni, és nem az alapértelmezett rutinét.

Látjuk tehát, hogy a megszakítás vektor tábla kitöltésével nekünk nem kell foglalkoznunk, azt készen kapjuk a startup kóddal. Arra kell csak ügyelnünk, hogy jól nevezzük majd el a saját ISR-ünket.

3.4. A megszakítás kezelő rutin megírása

A megszakítás kezelő rutin a Cortex mikrovezérlők esetében hagyományos C nyelvű függvénnyel implementálható.

> *Megjegyzés*:
>
> Ez elsőre nem tűnik nagy dolognak, de egyszerűbb architektúrákon nem ez a helyzet. Az ISR-ben található utasítások ugyanis ugyanúgy használják a processzor regisztereit, mint a háttérben futó kód, amit épp megszakított. Ez elrontaná a működést, ha nem gondoskodna arról az ISR, hogy az általa használt regisztereket lementse valahova a futása elején, majd a végén visszaállítsa azokat (erre a célra az adatmemóriát, egész pontosan a vermet szokták használni).
>
> Továbbá van olyan processzor, ami speciális return utasítással rendelkezik ISR-ek számára (mert pl. ISR-be lépéskor tilt minden további megszakítást, amit ezzel a speciális return utasítással lehet ismét engedélyezi).
>
> Egy normál C nyelvű függvény azonban nem menteget regisztereket, és nem is használ speciális return utasítást. Olyan architektúrákon, ahol erre szükség van, C nyelvi kiterjesztéseket kell alkalmazni. Ezen kiterjesztések ráadásul fordító függőek.
>
> A Cortex magú mikrovezérlők egy másik regiszter készletet használnak, ha interrupt módba kerülnek. Ilyen módon nincs szükség a regiszterek lementésére, visszaállítására. Valamint nincs semmi olyan sem, amiért speciális return utasítás kellene.

3.4.1. Definiáljuk a megszakítás kezelő rutin függvényét

Nézzük tehát meg, hogy milyen nevű rutin tartozik a startup kódban az UART0_RX megszakításhoz (a startup kód a CMSIS/EFM32GG/startup_gcc_efm32gg.s fájlban található)! Ezzel a névvel hozzunk létre egy függvényt a saját kódunkban (mind a visszatérési értéke, mind a paramétere void, továbbá itt se keverjük össze az USART0 perifériával az UART0-t)!

3.4.2. Olvassuk ki és küldjük vissza a vett karaktert

Mivel az UART0 perifériának csak egy vételi típusú megszakítás kérését (RX Data Valid) konfiguráltuk fel, biztosak lehetünk benne, hogy ha idáig jutottunk, akkor van egy új (és hiba mentes) karakter az UART0 vételi bufferében. Olvassuk ki!

Kiolvasásra használható a már megismert USART_Rx(), azonban célszerűbb az USART_RxDataGet(). Az előbbi ugyanis a kiolvasás előtt addig vár, ameddig a periféria nem jelzi egy megfelelő bit bebillentésével, hogy érkezett egy új karakter. Ez felesleges, ha már ISR-ünkben vagyunk, hisz az épp akkor hívódik meg, ha jött karakter.

A kiolvasott karaktert küldjük vissza, ezzel ISR-ből valósítjuk meg az „echo” funkciót.

3.4.3. Nyugtázzuk a megszakítást

Sok megszakítás esetében pusztán az a tény, hogy bekerültünk az ISR-be, nem nyugtázza az adott interruptot. Azaz az ISR-ből való kilépés után ismét fog egy ugyanolyan megszakítás keletkezni. Az interrupt kérést törölni pl. az USART_IntClear() függvény segítségével lehet.

Az általunk használ megszakítás viszont olyan, hogy ha kiolvassuk a vett karaktert, akkor az egyben nyugtázásnak is megfelel, így explicit módon az interrupt nyugtázásától most eltekinthetünk.

3.5. A megszakítás kezelő rutin módosítása

Mivel egy megszakítás kezelő rutin tetszőleges háttérben futó kódot képes megszakítani, olyan rövidnek célszerű megírni, amilyennek csak lehet. Nem „illik” benne várakozni, és célszerű minden olyat kiszervezni belőle, ami nem időkritikus, különben a háttérben futó kódot elképzelhető, hogy zavaró mértékben fogjuk fenntartani.

A mi példánkban elég rövid az ISR, ezzel együtt is mondhatjuk, hogy a karakter visszaküldése már nem időkritikus feladat. Továbbá ha megnézzük az USART_Tx() függvényt, akkor láthatjuk, hogy ez is blokkol. Ha éppen van adás, addig nem írja be az új adatot az UART/UASRT periféria adó bufferébe, amíg annak nincs vége:

```
void USART_Tx(USART_TypeDef *usart, uint8_t data)
{
  /* Check that transmit buffer is empty */
  while (!(usart->STATUS & USART_STATUS_TXBL)) {
  }
  usart->TXDATA = (uint32_t)data;
}
```

Ez vélhetően ritkábban és rövidebb ideig okoz késleltetést, mint a hasonló várakozás USART_Rx() esetében, de ettől még késleltetheti az interrupt gyors kiszolgálását.

3.5.1. A karakter visszaküldésének kivétele az ISR-ből

Az ISR-ben az azonnali visszaküldés helyett olvassuk ki a vett karaktert egy globális változóba. A háttérben futó kódot valahogy értesíteni kell az új karakter érkeztéről. Ezt szintén egy globális változón keresztül tudjuk megtenni

3.5.2. Fordítói optimalizációk hatása

Próbáljuk ki, hogy mi történik, ha optimalizált módon fordítunk (ehhez a „Release” típusú build kell az eddigi „Debug” helyett).

Jó eséllyel nem fog működni a programunk. Ennek az az oka, hogy az optimalizációk hatására a fordító kb. a következőképpen gondolkodik. „*Az a változó, amin keresztül az ISR jelez a főciklusnak, alapértelmezetten false értékű, és ezt követően sehol nem íródik át true-ra. Hisz ugyan az ISR függvényében van egy ilyen sor, de az ISR-t, mint függvényt, senki nem hívja meg, így nem is fog futni. Tehát a jelzésre használt változó végig false értékű. Így viszont a főciklusban a feltételhez kötött teljes kód kiszedhető, hisz úgyse fog soha lefutni, és feleslegesen minek foglalja a helyet a program memóriában.*”

A probléma megoldható, ha a jelzésre használt változó definíciójába beírjuk a „volatile” kulcsszót is. Ez angolul „illékonyat” jelent. Azt mondjuk meg vele a fordítónak, hogy a megjelölt változó mögötti memória tartalma bármikor megváltozhat, így ne próbáljon meg optimalizálni, hanem minden hozzáféréskor ténylegesen olvassa ki az adott változót.

Megjegyzés: nagy valószínűséggel a probléma csak a jelzésre használt változót érinti. A karakter átadására használt változó esetén nincs ilyen gond. Jóllehet, a fenti logika szerint ott is lehetne egy picit spórolni azzal, ha nem végeznénk el a változó kiolvasását minden ciklusban, hanem a kezdeti értékét feltételeznénk mindig. Talán azért, mert ez már nem jelentene nagy előnyt, talán más okból kifolyólag, de itt nem optimalizál a fordító. Mindenesetre, ha így is van, ezt a változót is jelöljük meg a „volatile” kulcsszóval!

3.5.3. A karakter buffer növelése – kiegészítő feladat

Miután az ISR-ből kivettük a karakterek visszaküldését, nem maradt benne felesleges kód. Mondhatjuk azonban, hogy ennek van viszont némi ára. Ha a főciklus mást is csinál, mint az új karakterek visszaküldése, előfordulhat, hogy több megszakítás is le fog futni, mire a főciklus ismét megnézi, hogy van-e új fogadott karakter. Így tehát hiába olvasta ki az ISR az éppen aktuális új karaktert azonnal, a funkció szempontjából elveszett.

Erre megoldást nyújthat, hogy ha a globális változónk, mint karakter buffer, több karakter eltárolására is képes lenne. A buffer méretének elegendően nagynak kell lennie ahhoz, hogy a rövid idő alatt jött karakterek feltételezett maximális száma mellett se legyen adatvesztés. (Ha üzemszerűen jönnek gyakrabban a karakterek, mint ahogy fel tudjuk dolgozni őket, akkor természetesen nincs akkora buffer, ami ezt ki tudná védeni.)

Ezt a feladatot nem kötelező megoldani, elég csak átgondolni.

4. Energiahatékony működés

Nem túl hatékony dolog egy főciklusban folytonosan arra várakozni, hogy bebillenjen egy változó. A processzorokat általában el lehet altatni valamilyen mélységig, amiből általában megszakítások hatására fel tudnak ébredni.

A Giant Gecko mikrovezérlők esetén is több ilyen mód van. A legegyszerűbb esetben csak magát a CPU-t tesszük el aludni (nem kap órajelet), de a többi periféria üzemel. Ezt EM1-nek nevezik.

Nézzük meg először, hogy mennyit fogyaszt jelenleg a panel!

Ezt követően módosítsuk a kódot úgy, hogy a főciklusban léptessük be a processzort az EM1 módba! Az energiatakarékos üzemmódok közti váltásért a mikrovezérlő EMU (Energy Management Unit) perifériája

felel. Így az *emlib* forrás fájlok közül az „em_emu.c”-re lesz szükségünk. Ez valószínűleg alapértelmezetten is része a projektnek. Ha nem, tegyük be!

A használandó függvény az EMU_EnterEM1(). Ahhoz természetesen, hogy ezt meg tudjuk rendesen hívni, hivatkozzuk be az „em_emu.h” fejléc fájl! Nézzük meg, sikerült-e csökkenteni a fogyasztást.

Megjegyzés: tekintettel arra, hogy most csak egy megszakítást használunk, felébredés után biztosak lehetünk abban, hogy érkezett egy karakter. Így akár a jelzésre használ változó és annak kezelése mellőzhető.

5. Az Stdio használata (UART0)

Az UART0 perifériát eddig direkt módon használtuk az USART_Rx() és USART_Tx() függvények segítségével. Azonban hordozhatóbb kódot kapnánk, ha helyettük a standard getchar(), putchar() függvényeket tudnánk használni. Nem beszélve pl. a printf() nyújtotta kényelemről.

Szerencsére a C nyelv standard I/O-ja átirányítható. A magas szintű, általunk használható rutinok alacsonyabb szintű függvényeket használnak, amik olyan alap funkció ellátására képesek, mint egy karakter küldése, fogadása, stb. Ha ezen függvényeket úgy valósítjuk meg, hogy pl. egy UART perifériát használjanak, akkor végeredményben pl. egy printf() is a soros portra fog írni.

5.1. A szükséges fájlok hozzáadása

Ahhoz, hogy működjön a standard I/O átirányítása, két fájlra van szükségünk: retargetio.c és retarget<periféria>.c (a mi esetünkben retargetserial.c). A retargetio.c egy nagyon vékony szoftver réteget definiál, amiben azon részeket tették bele, amik közösek, akármelyik konkrét perifériára is akarjuk átirányítani a standard be- ill. kimenetet. A másik pedig a választott perifériára implementált kódrészleteket tartalmazza.

Adjuk hozzá tehát a projekthez (pl. egy „drivers” almappa alá) a következő fájlokat[7]:

- retargetio.c
- retargetserial.c

A fájlok elérési útja:

[telepítési könyvtár]\SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\hardware\kit\common\drivers\

Megjegyzések:

- Ha belenézünk a fájlokba, látszik, hogy a karakterek fogadása interrupt segítségével történik. Így tehát a retargetserial.c használata esetén mi magunk már nem használhatjuk a kiválasztott U(S)ART periféria Rx Data Valid megszakítását.
- Továbbá érdemes észrevenni, hogy a getchar() akkor is vissza fog térni, ha nincs fogatott karakter, csak épp -1 a visszatérési érték (ergo nem blokkol ez a függvényhívás).

[7] A most használt virtuális gépben itt vannak:
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\hardware\kit\common\drivers

A fenti fájlok igényelnek más egyéb fájlokat. Attól függően, hogy milyen projekthez adtuk őket hozzá, több-kevesebb egyéb fájlt is hozzá kell tennünk.

Első körben a „retargetserial.c”-nek triviális függése az alábbi, mivel a soros portra irányítunk át. Ha ez nem lenne benne a projektben, adjuk hozzá (pl. az „emlib” mappa alá):

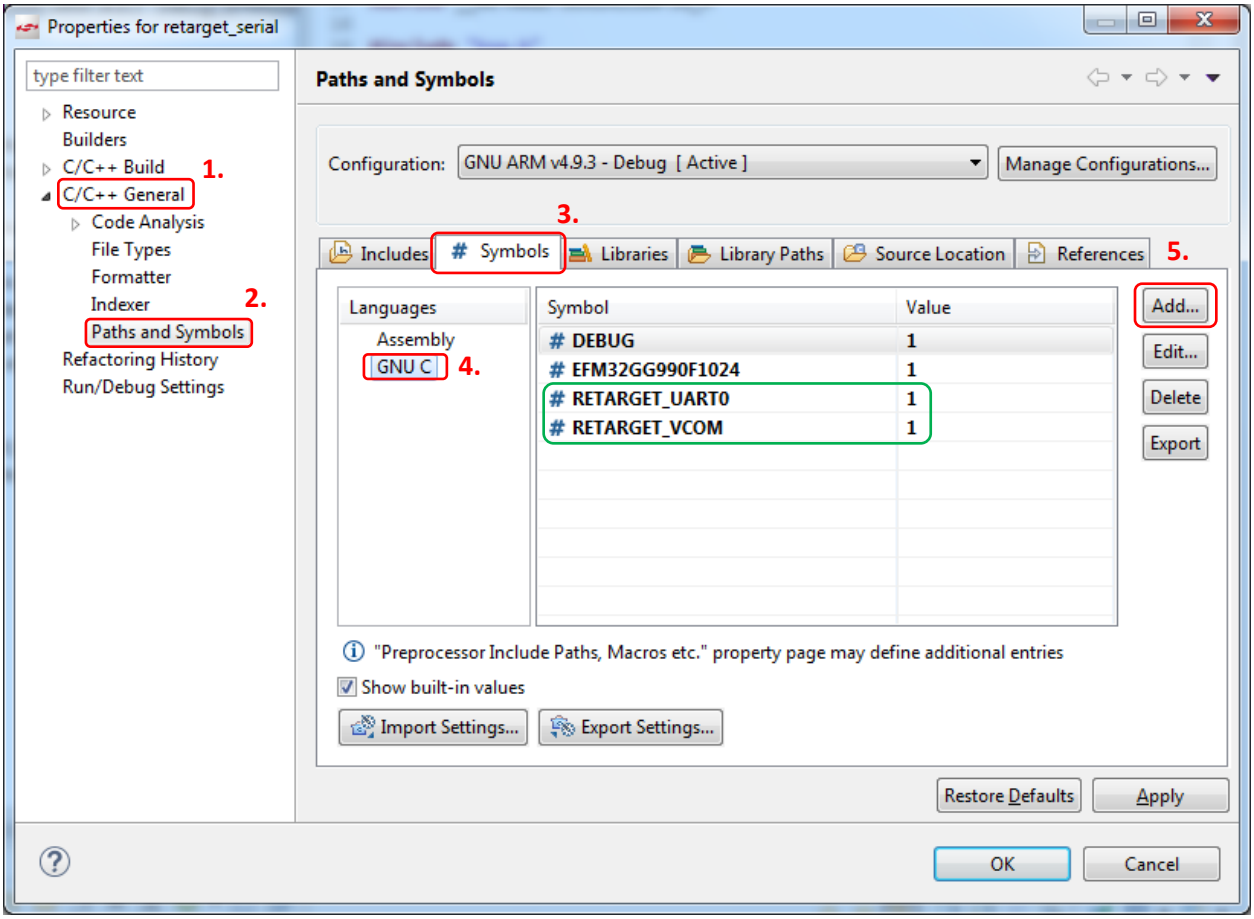
- em_usart.c

Mivel a soros porti perifériák meghajtásához kell órajel, a lábaikat megfelelően ki kell vezetni, továbbá a beállításuk során az emlib használ egyéb alap függvényeket, az alábbi három fájl és a hozzá tartozó header fájlok is kellenek (ha már nem tartalmazná a projekt):

- **em_core.c** (ez eddig nem volt a projekt része)
- em_cmu.c
- em_gpio.c

5.2. A használni kívánt soros port kiválasztása

Ezt preprocesszor direktívákkal tudjuk megtenni, amiket a projekt számára előre definiálnunk kell (ekkor a fordítás során parancssori paraméterként kerülnek átadásra). Két érték kell: RETARGET_UART0, hogy az UART0-t használjuk, és RETARGET_VCOM, hogy engedélyezzük is az UART0 kivezetését a Board Controlleren keresztül az USB portra mint virtuális soros port. A define-ok értéke legyen 1-1. Ezt az alábbi helyen tudjuk megtenni (Project / Properties):



5.3. Használata

Hivatkozzuk be először is az alábbi kettő fejlécet:

- stdio.h
- retargetserial.h

Ezt követően hívjuk meg az alábbit:

- RETARGET_SerialInit()

Opcionálisan az alábbit is meghívhatjuk:

- RETARGET_SerialCrLf(true)

Ennek ha igaz értéket adunk paraméterül, akkor bármelyik sorvég karaktert is küldjük ki (\r (carriage return, kocsi vissza) vagy \n (new line, line feed, új sor, sor emelés)) automatikusan beszúrja a másikat. Így kicsit kényelmesebb a printf() használata.

Ezek után az alábbinak már működnie kell:

- printf("Hello!\n");

6. Az LCD használata - Kiegészítő

6.1. Gyári függvények segítségével

A panelen található LCD használata az SDK részét képező függvényeknek hála igen egyszerű. A szükséges fájl a „segmentlcd.c”.[8] Ezt adjuk hozzá a projekthez (pl. egy „drivers” almappába). A benne található függvények deklarációi pedig a „segmentlcd.h” headerben vannak, így ezt hivatkozzuk be.

Mivel az LCD vezérlésére a mikrovezérlőben található LCD kontroller perifériát használja a „segmentlcd.c”, ezért szükséges még az „em_lcd.c”[9] fájl hozzáadása is a projekthez.

Az LCD-t használat előtt inicializálni kell, erre a „SegmentLCD_Init()” használható. Egyetlen paramétere az erősebb meghajtást engedélyezi. Alapvetően e nélkül is jók vagyunk, így legyen „fasle” (ha valakinél nagyon gyenge az LCD képe, ki lehet próbálni „true” megadásával, hátha jobb lesz tőle).

Az LCD-re írást kipróbálhatjuk pl. úgy, hogy a főciklusunkat kiegészítjük a vett karakter ASCII kódjának kiírásával (SegmentLCD_Number()). A többi függvény használata „self explanatory”.

6.2. Szegmensenkénti meghajtás

A házi feladatokhoz szükséges, hogy az LCD alsó felén elhelyezkedő, hét darab alfanumerikus karakter kiírására alkalmas kijelzőt szegmensenként is tudjuk vezérelni. Ehhez a gyári SDK nem ad támogatást. Azt nekünk kellett kiegészíteni, módosítani. Ehhez van egy demo projekt (displaySegmentField). Ezt be lehet

[8] A most használt virtuális gépen itt található:
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\hardware\kit\common\drivers
[9] Az emlibhez tartozó fájlok pedig itt:
C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v1.1\platform\emlib\src

importálni, és meg lehet nézni, hogyan kell használni. Mivel viszonylag egyértelmű, ezért alapvetően ez is „self explanatory”

Függelék: __STAIC_INLINE

Ezen függelékben a következő kérdésekre kapunk választ:

1. **Miért tesszük a függvény törzsét a header fájlba (mikor azt a C fájlba szokás)?**

   Normál esetben fordítás során egy függvény hívás helyére bekerül az adott architektúra szubrutin hívásért felelős assembly utasítása (pl. call). Ennek egy operandusa szokott lenni, egy memória cím a meghívni szándékolt függvény kódjának az elejére. Mivel a függvény lehet, hogy egy másik C fájlban kerül definiálásra, ezen a ponton konkrét címet nem tud a fordító, csak egy címkét helyez el. Aztán ha a meghívni szándékozott függvénynek otthont adó C fájlt is lefordítjuk, akkor már meglesz az ő kódja. Miután minden forrás fájlt lefordítottunk, következik a linkelés fázisa. A linker felel azért, hogy a fordító által lefordított kódrészleteket (object fájlok) összeillessze, és a bennük lévő címkéket feloldja immáron tényleges memória címekre.

   Ha viszont inline módon akarjuk használni a függvényt, akkor már a fordítás során szükség van a kódjára (hisz nem egy call utasítást kell elhelyezni, hanem a függvény törzsét kell beszúrni). Ez viszont azt jelenti, hogy a fordítónak már ismernie kell a kérdéses függvény törzsét. Az alapvetően nem járható út, hogy csak valami olyasmit fordítanánk, hogy inline call függvény, mert a függvény törzse ugye csak egy másik forrás fájlban lesz meg, ergo a linkerre várna, hogy ezt feloldja, a linker viszont nem fordító. Állítólag vannak olyan fordítók, ahol a linker egy ilyen esetben újra meghívná a fordítót, hogy segítse ki, de most ettől tekintsünk el. Kell tehát a függvény törzse. Ez a legegyszerűbben pedig úgy tehető meg, hogy nem egy adott modulhoz tartozó C, hanem a H fájlba tesszük. Ezért vannak az inline függvények törzsei is a fejléc fájlokban.

2. **Miért kell még a static kulcsszó is?**

   Ha viszont egy fejléc fájlban szerepel a függvény a törzsével együtt, akkor – ha több forrás fájl is behivatkozza az adott fejlécet – akkor több forrás fájlban is benne lesz a függvény definíciója (az pedig alapesetben nem megengedett, hisz a linker később honnan tudná, hogy egy adott nevű függvény akkor tulajdonképpen melyik is). A static kulcsszó itt kerül képbe. Az ugyanis lekorlátozza a függvény láthatóságát az alapértelmezett globálisról lokálisra (azaz csak az adott forrás fájlra). Ezért tehát a static is.

3. **Miért kell a kulcsszavakat egy makró mögé rejteni?**
   A CMSIS szabvány fordítófüggetlen akar lenni. Ezért ha valahol felmerül a gyanú, hogy egy adott funkció esetleg fordítófüggő lehet, akkor ott azt egy makró mögé rejti. Így ha egy olyan fordítót használnánk, ahol ahhoz, hogy egy függvény static és inline legyen valamilyen oknál fogva nem pont ezek a kulcsszavak kellenének, akkor is működni fog (egy alternatív makróval).