Mikrovezérlők programozása

Nagy Gergely

BME EET

2012. április 4.

A mikrovezérlők programozása

- Áttekintjük a a **mikrovezérlők programozásának alapvető elemeit:**
  - **üzemmódok beállítása, a vezérlő konfigurálása,**
  - **megszakítások kezelése,**
  - **állapotgépek létrehozása.**
- A példák az Atmel cég AVR Atmega8 mikrovezérlőjére íródtak, de a **bemutatott technikák álatalánosan érvényesek.**
- A programok **C nyelven** íródtak – a legtöbb vezérlőhöz ingyenesen hozzáférhető egy C nyelvű fejlesztőkörnyezet, amely nagyban leegyszerűsíti a munkát.
- Az assembly nyelvet csak **nagyon kiélezett helyzetekben** alkalmazzák:
  - kevés memória,
  - szigorú időzítési feltételek.

Bitműveletek I.

- A mikrovezérlők programozása során egészen biztos, hogy **szükség van rájuk:**
  - `~` bit-szintű NEM operátor
  - `&` bit-szintű ÉS operátor
  - `|` bit-szintű VAGY operátor
  - `^` bit-szintű KIZÁRÓ-VAGY (XOR) operátor
  - `«` biteltoló (shift) operátor (balra, MSB felé – 0 értékű biteket tölt be az LSB felől)
- Ezek eszközök a **változók bitjeinek manipulásához:**
  - egy bit/bitek **értékének a megállapítása,**
  - egy bit/bitek **beállítása 1-be,**
  - egy bit/bitek **nullázása.**
- **Minden bitmanipuláló művelet alapja:**
  1. **maszk előállítása:** biztosítja, hogy az adott logikai művelet csak a kérdéses bitekre legyen hatással,
  2. **logikai művelet:** elvégzi a szükséges bitmanipulációt.

Bitműveletek II. – Bit értékének a megállapítása

- A **bitvizsgálat alapművelete: ÉS**
  - **Ha egy ismeretlen bitet 1-gyel ÉS kapcsolatba hozunk, akkor az eredmény pontosan a kérdéses bit lesz.**
  - **Ha egy bitet 0-val hozunk ÉS kapcsolatba, az eredmény 0 lesz.**
- **A feladat tehát:** a változónk kérdéses bitjének/bitjeinek a helyére 1-est, a többi helyre 0-t kell írni a maszkba.
- Így az eredmény:
  - 0, ha a kérdéses bit(ek) 0 volt(ak),
  - $\sum 2^i$, ha a kérdéses bit(ek) 1-es értű(ek) volt(ak).
- **Példa:** vizsgáljuk a `szam` változó 3. bitjét. A maszk a 3. bitjére 1-est kell írni – a **biteltoló operátort használjuk:**

```
int szam = 132, maszk, eredmeny;
maszk = 1 << 2;
eredmeny = szam & maszk;

if (eredmeny == 0) {...} // 3. bit egyenlő 0-val
```

Bitműveletek III. – Bit beállítása 1-be

- A **bitbeállítás alapművelete: VAGY**
  - **Ha egy ismeretlen bitet 0-val VAGY kapcsolatba hozunk, akkor az eredmény pontosan a kérdéses bit lesz.**
  - **Ha egy bitet 1-gyel hozunk VAGY kapcsolatba, az eredmény 1 lesz.**
- **A feladat tehát:** a változónk kérdéses bitjének/bitjeinek a helyére 1-est, a többi helyre 0-t kell írni a maszkba, majd a VAGY művelet eredményét vissza kell írni a változóba.
- Így csak azok a bitek változnak, amelyek helyén a maszkban 1 áll.
- **Példa:** állítsuk a `szam` változó 5. bitjét 1-be:

```
int szam = 40, maszk;
maszk = 1 << 4;

eredmeny = eredmeny | maszk;
```

Bitműveletek IV. – Bit beállítása 0-ba

- A **bit törlésének alapművelete: ÉS**.
- **A feladat:** a kérdéses bitet ÉS kapcsolatba kell hozni 0-val, a többit 1-gyel és az eredményt vissza kell írni a változóba.
- Csakhogy a biteltoló operátor mindig 0-t helyez be, ezért a trükk: **a szükséges bitminta invertáltját állítjuk elő, majd negálunk.**
- **Példa:** állítsuk a `szam` változó 2. bitjét 0-ba:

```
int szam = 18, maszk;
maszk = ~(1 << 1);

eredmeny = eredmeny & maszk;
```

Egy egyszerű program felépítése

```
#include <...>                  // 1.

SIGNAL (SIG_...) {...}          // 2.

void  init (void)  {...}        // 3.

int  main() {
     init ();                   // 4.

     while (1)  {...}           // 5.

     return 0;                  // 6.
}
```

1. A szükséges **fejléc-állományok beszerkesztése.**
2. **Megszakítás-kezelő rutin(ok).**
3. **Inicializáló függvény.**
4. Az inicializáció.
5. **Végtelen ciklus(!)** – a munka a kikapcsolásig tart (egyszerű esetekben).
6. A soha végre nem hajtott return a fordító kedvéért.

Az inicializáló függvény I.

- Az inicializáció során állítjuk be a **mikrovezérlő és a beépített moduljainak üzemmódjait és paramétereit.**
- Ez **konfigurációs regiszterek megfelelő bitjeinek beállítását** jelenti.
- Például az Atmega8-ban a B port irányát (bemenet (0) / kimenet (1)) a DDRB (Data DiRection of port B) regiszterrel állítjuk be.
- Az alábbi kódrészlet a teljes B portot kimenetnek konfigurálja.

```
DDRB= 0xFF;
```

Az inicializáló függvény II.

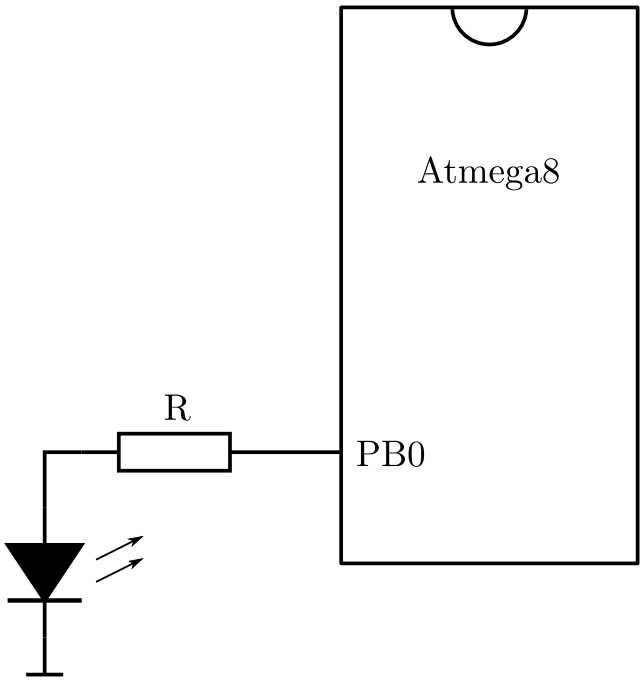
- Bizonyos modulok a **beállítás pillanatában működésbe lépnek** és ha engedélyezzük, akkor **megszakításokat küldhetenek.**
- Ezért a **megszakításokat az inicializáció idejére le szokták tiltani:**

```
void  init (void) {
    cli ();
    ...
    sei ();
}
```

- A használt C függvények nevei az általuk hívott gépi utasítások neveivel egyeznek meg:
  - **cli:** *clear interrupt bit* – az általános megszakítás-engedélyező bit 0-ba állítása
  - **sei:** *set interrupt bit* – az általános megszakítás-engedélyező bit 1-be állítása

Mikrovezérlős "Helló világ": LED villogtatás I.

- **Feladat:** az egyik digitális portra (port B 0. bitjére) kötött LED-et adott frekvencián villogtassuk.



- A **LED elé előtét-ellenállást illik tenni**, hiszen a logikai 1-nek megfelelő 5 V-os feszültségnél nagyon nagy áram folyna.
- Valójában a mikrovezérlők lábai áramkorlátozottak, de ettől még a LED áramát korlátozni kell – általában 1-2 mA-re.

Mikrovezérlős "Helló világ": LED villogtatás II.

- **Konfigurációs feladatok:**
  1. az adott lábat kimenetté kell tenni,
  2. elő kell állítani a négyszögjelet.
- Utóbbihoz **számlálót** használunk:
  - az mikrovezérlő órajelét szabályozható értékkel előosztja,
  - megszakítást tud adni egy adott érték elérésekor / túlcsorduláskor.
- Mivel a számláló nagyon sokat használt modul a vezérlésben, általában több is van belőle.
- Az Atmega8-ban a 0-s számláló egy egyszerű 8-bites időzítő.

Mikrovezérlős "Helló világ": LED villogtatás III.

- **Négy konfigurációs regiszter kapcsolódik hozzá:**
  - **TCCR0 – Timer/Counter Control Register 0:** üzemmód választás
  - **TCNT0 – Timer/Counter Register:** a számláló aktuális értéke
  - **TIMSK – Timer/counter Interrupt Mask Register:** megszakítások engedélyezése
  - **TIFR – Timer/counter Interrupt Flag Register:** megszakítás flagek
- Számunkra a TCCR0 és a TIMSK lesznek fontosak.

Mikrovezérlős "Helló világ": LED villogtatás IV.

A TCCR0 0-2 bitjeinek (CS0-CS2) jelentése:

CS02	CS02	CS02	Jelentés
0	0	0	A számláló inaktív.
0	0	1	Nincs előosztás.
0	1	0	8-as előosztás.
0	1	1	64-es előosztás.
1	0	0	256-os előosztás.
1	0	1	1024-es előosztás.
1	1	0	Külső órajel (T0 láb), ↓ él
1	1	1	Külső órajel (T0 láb), ↑ él

- Mivel a legalacsonyabb belső órajel 1 MHz, a legnagyobb előosztást érdemes használni, így a bitek értéke: 101 lesz, ami hexadecimálisan 0x05.

Mikrovezérlős "Helló világ": LED villogtatás V.

- **Megszakításokat is szeretnénk kapni**, ehhez a TIMSK regiszter legalsó bitjét kell beállítani:
  - **TOIE0**: Timer/Counter0 Overflow Interrupt Enable
- Ennek hatására, **amikor a számláló túlcsordul** (a 255 érték elérése után):
  - a TIFR regiszter 0. bitje 1-es értéket kap (TOV0 – Timer/Counter0 Overflow Flag),
  - megszakítást kapunk: (SIG_OVERFLOW0).
- **A villogás frekvenciája így:**
  - az 1 MHz-es órajelet 1024-gyel előosztjuk,
  - a számláló még 256-al oszt, hiszen minden 256. órajelnél kapunk megszakítást,
  - tehát 1 MHz / 1024 / 256 = $\sim$2 Hz.

Mikrovezérlős "Helló világ": LED villogtatás VI.

Az **inicializáló kód** tehát:

```
void init (void) {
    cli ();

    /* Portok */
    DDRB = 0xFF;     // PORTB kimenet

    /* Timer0 */
    TCCR0 = 0x05;    // 1/1024 az előosztás
    sbi(TIMSK ,0);   // Overflow0 int engedélyezve

    sei ();
}
```

Mikrovezérlős "Helló világ": LED villogtatás VII.

Az **megszakítást kezelő kódrészlet:**

```
/* TIMER0 overflow */
SIGNAL (SIG_OVERFLOW0) {
   (PINB & 0x01) ? cbi(PORTB, 0) : sbi(PORTB, 0);
}
```

- A **megszakításkezelő rutinok** szintaktikája: `SIGNAL (...)`.
- Egy **port értékének a beolvasása:** `PINx` olvasása, ahol *x* a port neve.
- **Port írása:** a `PORTx` írásával (*x* a port neve).
- A `cbi()` és `sbi()` függvények, amelyek **bitek beállításában segítenek:**
  - **cbi(reg, n) – (clear bit):** a `reg` *n.* bitjét törli.
  - **sbi(reg, n) – (set bit):** a `reg` *n.* bitjét állítja 1-re.

Mikrovezérlős "Helló világ": LED villogtatás VIII.

A **főprogram:**

```
int main(void) {
    init ();

    //Főciklus
    while (1) {
    }

    return 0;
}
```

- Az inicializáció után **csak egy végtelen ciklus:**
  - a **számlálás és megszakítás-generálás automatikusan** történik,
  - **minden feladat a megszakításrutinban** lett megfogalmazva.
- Ez jól jelzi a **sok beépített áramköri modul előnyét: maguktól működnek – kevés programkóddal elkészíthető egy bonyolult vezérlés.**

Mikrovezérlős "Helló világ": LED villogtatás IX.

- Az előző példában **az előállítható legkisebb frekvenciájú négyszögjelet használtuk.**
- Mit tehetünk, ha **ennél is kisebb frekvenciára van szükségünk?**
- Például **programozottan tovább oszthatjuk a frekvenciát:**
  - a **megszakításrutinban csak egy változó értékét növeljük,**
  - a **főciklusban villogtatunk.**

Mikrovezérlős "Helló világ": LED villogtatás X.

```
volatile unsigned char szamolo = 0;

SIGNAL (SIG_OVERFLOW0) {
    szamolo++;
}
...
int main(void) {
    init ();

    while (1) {
        if (szamolo == 10) {
            (PINB & 0x01) ? cbi(PORTB, 0) : sbi(PORTB, 0);
            szamolo = 0;
        }
    }
}
```

Mit jelent a `volatile` kulcsszó és miért szükséges a `szamolo` deklarálásakor?

Mikrovezérlős "Helló világ": LED villogtatás XI.

- A fordító **kioptimalizál minden olyan változót**, amelynek
  - sosem változtatjuk a kódban az értékét,
  - az értékét változtató kódrészlet sosem fut le.
- Az előző kódban a `szamlalo` értéke 0-tól különböző **értéket csak a megszakítás rutinban kap.**
- A megszakításrutint a **kódban sehol sem hívjuk – ezt a "hardver hívja".**
- Ezt a fordító nem tudja, ezért **konstans 0-val helyettesíti a változót.**
- A `volatile` kulcsszó azt jelzi a fordító számára, hogy az **adott változóval kapcsolatosan nem szabad optimalizációt végeznie.**

A sorosport használata I.

- A sorosport már tűnik el a számítógépekről, de az **iparban még használják**, illetve vannak IC-k, amelyekhez RS-232 protokollon keresztül kell kapcsolódni.
- Maga az **adatküldés és fogadás** nagyon egyszerű:
  - **Küldés:** UDR regiszter írása,
  - **Fogadás:** UDR regiszter olvasása.

  UDR: USART I/O Data Register (USART: Universal Synchronous and Asynchronous Serial Receiver/Transmitter).
- A **konfigurációja összetettebb:**
  - be kell állítani a keretformátumot,
  - engedélyezni kell a küldő és fogadó modulokat,
  - be kell állítani a baud rate regisztereket, amelyek alapján előáll a megfelelő időzítés.

A sorosport használata II.

```
void init (void) {
    cli ();

    /* A soros port incializ álása */
    UBRRH = 0;
    UBRRL = 25; //1 MHz, 2400 Baud
    UCSRB = (1 << RXEN) | (1 << TXEN) | (1 << RXCIE);
    UCSRC = (1 << URSEL) | (1 << USBS) | (1 << UCSZ1)
            | (1 << UCSZ0);

    sei ();
}
```

- **RXEN:** Receive enable
- **TXEN:** Transmit enable
- **RXCIE:** Receive interrupt enable
- **URSEL:** Register select
- **USBS:** Stop bits
- **UCSZ12-0:** Data bits (011: 8 bits)

A sorosport használata III.

A **megszakítása rutin és a főprogram:**

```
enum {SRX};
volatile unsigned char flags = 0x00;

SIGNAL (SIG_UART_RECV) {
    soros_adat = UDR; /* Kiolvassuk a fogadott adatot */
    set_flag(flags, SRX); /* Beállítjuk az "adat érkezett" bitet */
}

int main(void) {
    init ();
    while (1) {
        if (check_flag(flags, SRX)) {
            clr_flag(flags, SRX);
            /* Megvárjuk, hogy a soros adó puffer üres legyen */
            while (!(UCSRA & (1 << UDRE)));
            UDR = soros_adat + 1;
}}}
```

A sorosport használata IV.

- Az előző programrészletben **flageket használtunk az események lekezelésekor.**
- Ennek oka: a megszakításkezelő rutinokat érdemes rövidre fogni, mert véges számú mélységben tudja lekövetni a vezérlő a megszakításhívásokat.
- Így az összetett feladatok áthelyezhetőek a főprogramba.
- A felhasznált **"segédfüggvények":**

```
#define set_flag(flags, flag) ((flags) |= (1<<flag))
#define clr_flag(flags, flag) ((flags) &= (~(1<<flag)))
#define check_flag(flags, flag) ((flags) & (1<<flag))
```

Ezek megvalósítják a korábban látott bitmanipuláló műveleteket.

A beépített A/D átalakító használata I.

- A beépített átalakító **10 bites felbontásra** képes.
- A **konverziós idő** 13-260 μs.
- 6 db **multiplexelt bemeneti csatorna** van.
- A teljes 0–V_{CC} tartományt képes fogadni.
- Van **beépített, 2,56 V-os referencia,** illetve választható **külső referencia mód.**
- Választható **egyszeri vagy folyamatosan futó konverziós mód.**
- Beállítható **megszakítás a konverzió befejeztekor.**
- **Külön "alvó mód" a konverzió idejére.**

A beépített A/D átalakító használata II.

- A **átalakítás eredménye az ADCL, ADCH regiszterpárosba kerül.**
- Az **érték és a bemeneti feszültség** kapcsolata:

$$ADC = \frac{V_{in} \cdot 1024}{V_{ref}}$$

- A jó minőségű konverzió érdekében mind az analóg referencia, mind az analóg tápbemenetet érdemes szűrni.

A beépített A/D átalakító használata III.

Az **eredmény elhelyezése a regiszterpárban kétféle lehet:**

ADLAR =0

Bit	15	14	13	12	11	10	9	8	
							ADC9	**ADC8**	**ADCH**
	ADC7	**ADC6**	**ADC5**	**ADC4**	**ADC3**	**ADC2**	**ADC1**	**ADC0**	**ADCL**
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR =1

Bit	15	14	13	12	11	10	9	8	
	ADC9	**ADC8**	**ADC7**	**ADC6**	**ADC5**	**ADC4**	**ADC3**	**ADC2**	**ADCH**
	ADC1	**ADC0**							**ADCL**
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

1. Akkor hasznos, ha mind a 10 bitre szükségünk van
   `adc = ADCH << 8 | ADCL;`
2. Ha csak a felsó nyolc bitet használjuk
   `adc = ADCH;`