18. USB kapcsolat

Írta: Kalmár Gábor

Lektorálta: Szabó Ádám, Lágler Gergely

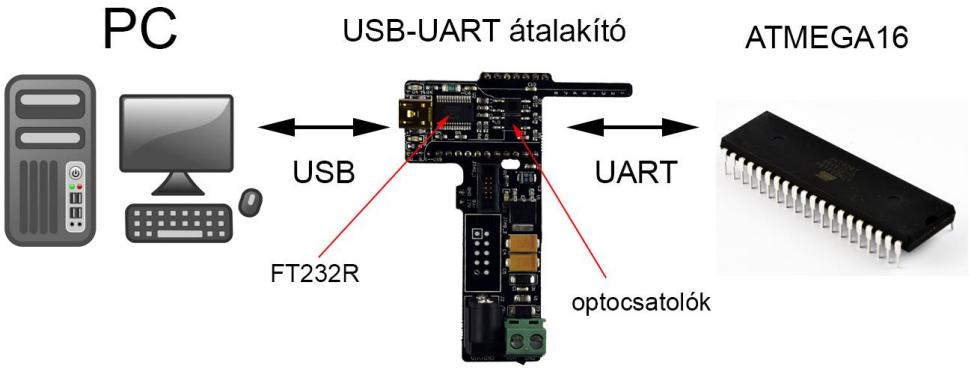
BEVEZETŐ

A tananyagrész végére képesek leszünk számítógépünkről vezérelni a mikrokontrollert egy előre megírt program segítségével.

A tisztelt olvasó megismerkedhet a mikrokontroller és a PC közötti soros porti kommunikációval. A taglalt mikrokontrollerünk nem rendelkezik valódi USB stack-el, azaz olyan hardver komponenssel, amellyel hatékonyan lehet kommunikálni USB buszon.

Található benne viszont soros kommunikációra alkalmas interfész, azaz UART. Ennek a perifériának a segítségével soros COM porton keresztül tudunk csatlakozni a számítógéphez. A probléma az, hogy a mai modern PC-ken vagy laptopokon már nem, vagy nagyon ritkán találkozhatunk ezzel a perifériával. USB viszont szinte kivétel nélkül jelen van minden eszközön.

A PC oldali USB-t és a mikrokontroller oldali UART-ot jelen esetben egy speciális integrált áramköri elem köti össze. Ez az IC az FTDI FT232R USB-UART átalakító. Ezzel a megoldással mintegy soros porton tudunk kommunikálni, hiszen az IC PC oldali szoftvere megoldja, hogy soros portnak látszódjon az FTDI UART ki és bemenete. Így az USB fizikai rétegét teljesen elrejti a szemünk elől, nem is kell vele foglalkoznunk.



***1. ábra** - Kommunikáció a PC és a mikrovezérlő között*

UART

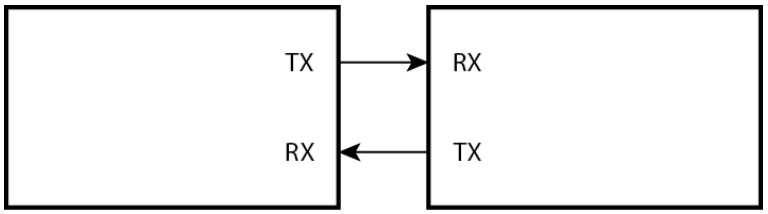
Az UART mozaikszó az angol “universal asynchronous receiver/transmitter” kezdőbetűiből áll. Magyarul annyit tesz, mint “univerzális aszinkron fogadó és küldő”. Aszinkron kommunikáció alatt azt értjük, hogy a küldő és a vevő nincs szinkronban, nincs közös órajel, ami meghatározná a kommunikáció ütemét. A kommunikáció két jelvezetéken folyik:

- Rx (vétel)
- Tx (adás)

Ezt a két vezetéket nevezhetjük adatbusznak vagy csak egyszerűen busznak. Buszállapot alatt a későbbiekben ezeknek a jelvezetékeknek a feszültségszintjeit kell érteni.

Feszültségszintet valamihez képest lehet mérni. Jelen esetben a föld, azaz GND és a jelvezeték közti potenciálkülönbséget értjük feszültségszint alatt.

Mindkét jelvezetéken folyhat adás és vétel is. Rx –ről és Tx-ről csak egy adott chip kontextusán belül beszélhetünk, hiszen ami az egyiknek Tx az a másiknak Rx. Ebből következik, hogy két UART kompatibilis eszközt úgy kötünk össze, hogy az egyik Rx lábát kötjük a másik Tx lábára és fordítva.



***2. ábra** - Két eszköz összekapcsolása UART-on keresztül*

Az aszinkronitás miatt a vevőnek tudnia kell, hogy az adó mikor kezdi el az adást és mikor fejezi azt be, illetve milyen sebességgel küld. A kezdésre és a befejezésre kitüntettet busz állapotokat határozott meg a szabvány.

"FRAME" FORMÁTUM

Az adatkeret, azaz "frame" magán az adaton kívül a szinkronizációs bitekből (start és stop bitek) és az opcionális paritás bitből áll. A felépítés a következő:

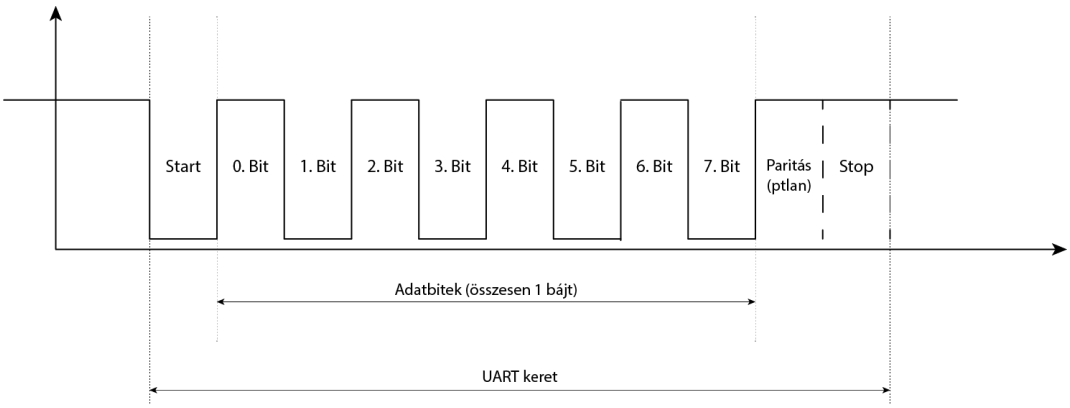
- Start bit: logikai 0 jelszint egy bitidőnyire
- Adat bitek: 5-9 adat bit, normál esetben 8
- Opcionális paritás bit: jelentése páros vagy páratlan
- Stop bit(ek): 1 vagy 2 stop bit logikai 1 jelszinttel

A busz alapállapota a logikai 1 jelszint. A kommunikáció a start bittel kezdődik. Ezt az alapértelmezettől eltérő 0 jelszint reprezentálja. Ezután következik az adat, majd legvégül a stop bit zárja a kommunikációt.

***3. ábra** - UART adatkeret felépítése*

A helyes működéshez szükséges, hogy mind az adó, mind a vevő „megállapodjanak" ugyanabban a felépítésben, azaz frame formátumban és ugyanabban a sebességben, azaz a baud rate-ben. Erre azért van szükség, hogy a vevő akkor mintavételezzen egy bitet, amikor azt az adó valóban küldi is.

A leggyakrabban használt formátum 1 start bitet, 8 adat bitet, és 1 stop bitet használ paritás bit nélkül. A következő ábra egy adatkeret felépítését szemlélteti 1 start bittel, 8 adat bittel, 1 páratlan paritás bittel és egy stop bittel:



***4. ábra** - UART üzenet felépítése*

PARITÁS

A küldendő adatbitek közül az egyesek száma 0 és 8 között lehetséges. A páros paritás azt jelenti, hogy minden olyan bájthoz, amelyben az 1 adatbitek párosan vannak (0, 2, 4, 6 vagy 8 darab), a paritásbit 0 lesz. Ha páratlan számú egyes szerepel az adatbitek között, akkor a paritásbit 1. Azaz úgy egészül ki a paritásbittel az adat, hogy páros számú lesz az egyesek száma az egész frame-ben a start és stop biteket nem számolva. Páratlan paritás esetében a paritásbit értéke akkor 0, ha az 1 adatbitek száma páratlan. 1-es értéket pedig akkor vesz fel ha az 1 adatbitek száma páros, tehát pont fordított a helyzet a páros paritáshoz képest.

Tegyük fel, hogy a vett adat: 01100101. Páros paritása 0, mert 4 db 1-est tartalmaz, ami páros szám. Páratlan paritása 1, mert a 4 nem páratlan.

A paritásbit hibadetektálásra szolgál. Ezt a bitet mind a küldő, mind a vevő legenerálja. Ha vételi oldalon a frame-ben kapott paritásbit nem egyezik meg a helyileg generálttal, akkor az adatbithibát jelent. Ezzel a módszerrel páratlan számú bit hibáját vagyunk képesek detektálni. Képzeljük el, hogy egyszerre 2 bit is átfordul. Ebben az esetben a paritás, azaz az egyesek és nullák számossága nem változik így ez a hiba sajnos felderítetlen marad. Ha már 3 bit is sérül, akkor újra látszik a hiba. Tipikusan 1 bit hibáját hivatott kimutatni a paritásbit. Több bit hibáját is lehetséges természetesen észlelni 100%-osan, de ez már nem az UART fizikai rétegének a feladata.

BAUD RATE

A baud rate határozza meg a kommunikáció sebességét, mértékegysége a szimbólum/szekundum. Egy szimbólum elvileg több mint két állapotot is fel tud venni és így több információt is képes lehet hordozni, mint egy bit. Szimbólum lehet például az ABC egy betűje is. A magyar ABC-ben 40 betű van ezért a szimbólum jelen esetben 40 állapotot vehet fel. Ahhoz, hogy 40-ig el tudjunk számolni binárisan, azaz le tudjunk kódolni egy betűt, 6 bitre van szükségünk. Ebben az esetben 1 baud/sec = 6 bit/sec. UART esetében a szimbólum nem más, mint egy bit, ezért a baud/sec és a bit/sec ebben az esetben megegyezik.

$$1 \text{ baud/sec} = 1 \text{ bit/sec}$$

A baud rate-nek szabványos értékei vannak UART esetén. 2400 bit/sec-től indul és akár 1Mbit/s-ig is elmehet, ha a kommunikációban résztvevő eszközök támogatják. A teljesség igénye nélkül néhány szokásos érték: 2400, 4800, 9600, 14400, 19200. Észrevehetjük, hogy az értékek duplázódnak vagy 1.5-

szereződnek. A teljes lista a támogatott értékekkel megtalálható az ATmega16A adatlap ide vonatkozó fejezetében (165. oldal).

Kitekintés

Az USB azaz az Uneversal Serial Bus a számítógépeken a legelterjedtebb csatlakozó felület. Létrejöttének célja az volt, hogy egységesítse a perifériák csatlakozásának módját, illetve kiváltsa az elavult szabványokat. Jelenleg a perifériák igen nagy hányada USB szabvány szerint működik. Ezek lehetnek billentyűzetek, egerek, nyomtatók, telefonok, stb. Topológiája jellemzően egy host port-ból és egy kliensből áll, de HUB segítségével egy host port több klienst is ki tud szolgálni. Host többnyire egy PC lehet, kliens pedig egy periféria eszkösz például egér. Jellemzője a viszonylag nagy adatátviteli sebesség, ami 2.0-ás szabvány esetén 480MBit/s. Az újabb szabványok (3.0 és 3.1) még ennél is nagyobb 5, illetve 10Gbit/s sebességet kínálnak.

FEJLESZTŐ PANEL ÉS FTDI FT232R

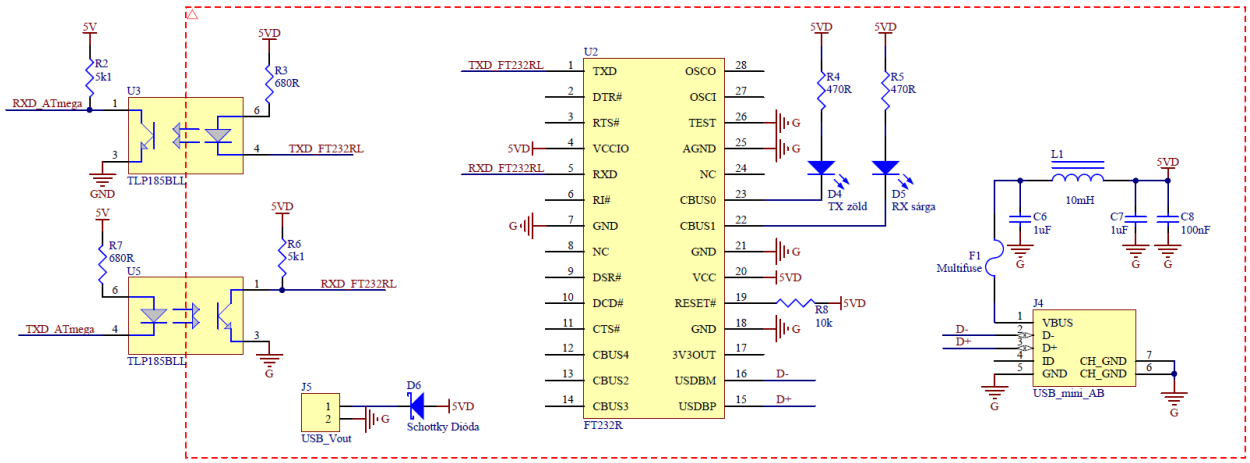
A példák bemutatásához egy olyan komplex hardvert használok, ami a következő komponensekből áll:

- Dugasztáp 12V kimenettel
- Breadboard
- ATmega16A mikrokontroller (MCU), ami a breadboard-ra van ültetve
- FTDI adapter, amin megtalálható a tápellátást biztosító 5V-os feszültségkonverter és az FT232R típusjelzésű IC. Ez utóbbi az UART és USB közötti kapcsolattartásról gondoskodik.

Nagyvonalakban a következő blokk diagram szemlélteti az elrendezést:

5. ábra *- Kommunikáció felépítése a számítógép és a mikrovezérlő között*

A kapcsolási rajz FTDI chippel kapcsolatos része a következő ábrán látható:

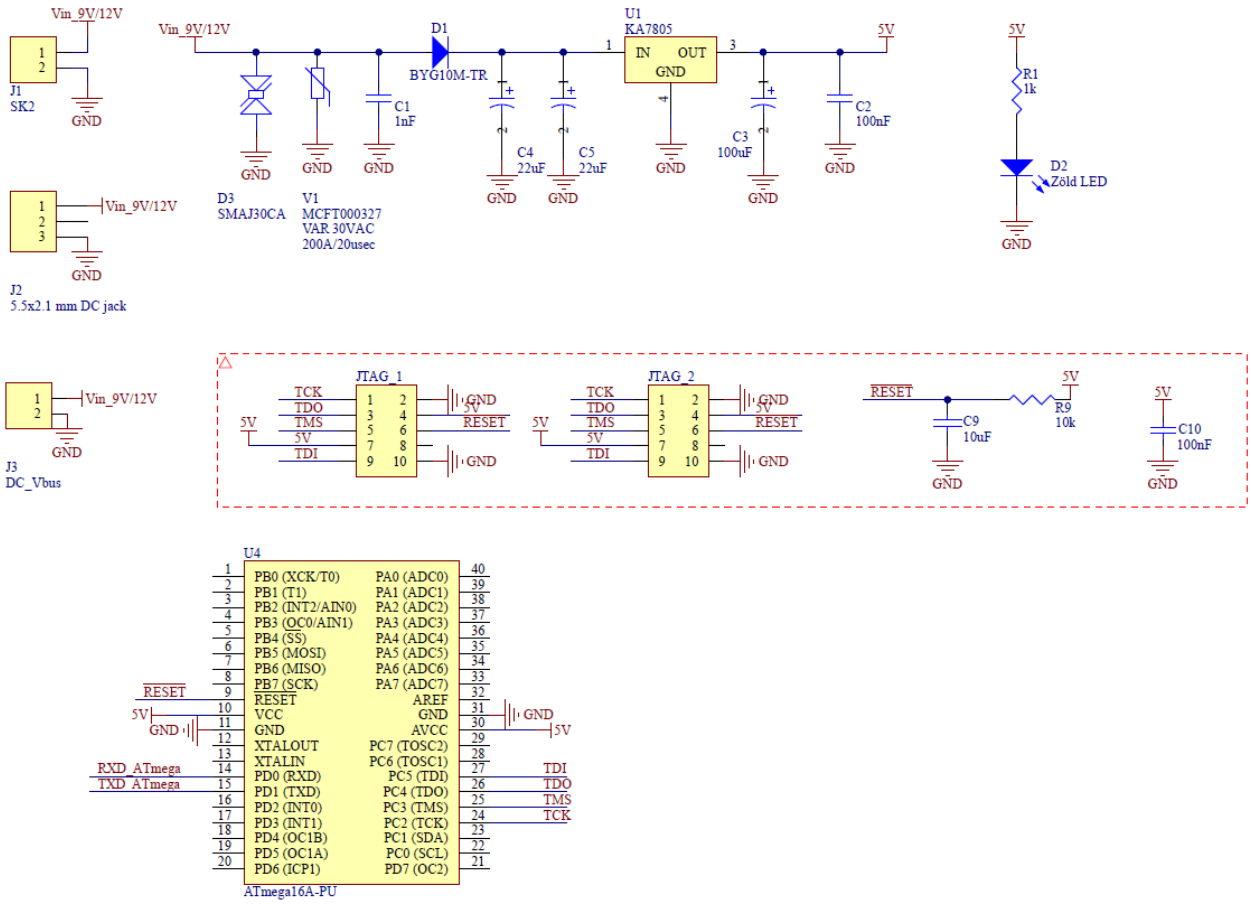


6. ábra *- FTDI chip kapcsolási rajza*

Jobbról balra haladva:

- USB_mini_AB: D+ és D- a két adatvezeték. VBUS az USB 5V-os tápvonala. GND a föld.
- F1: Pozitív hőmérséklet együtthatójú öngyógyuló biztosító (resetable fuse).
- C6, C7, C8, L1: A tápvonal stabilitásáért felelős pufferelő és szűrő elemek.
- FT232R: A fentebb említett USB-UART konverzióért felelős integrált áramkör.
- TLP185BLL: Optocsatoló. Feladata a galvanikus leválasztás a mikrokontroller és az USB között. Erre egyrészt azért van szükség, mert a mikrokontrollernek és az FTDI IC-nek nem közös a tápforrása. Másrészt így elkerülhető, hogy az USB-n keresztül bármilyen zavar tönkre tegye a mikrokontrollert és fordítva.

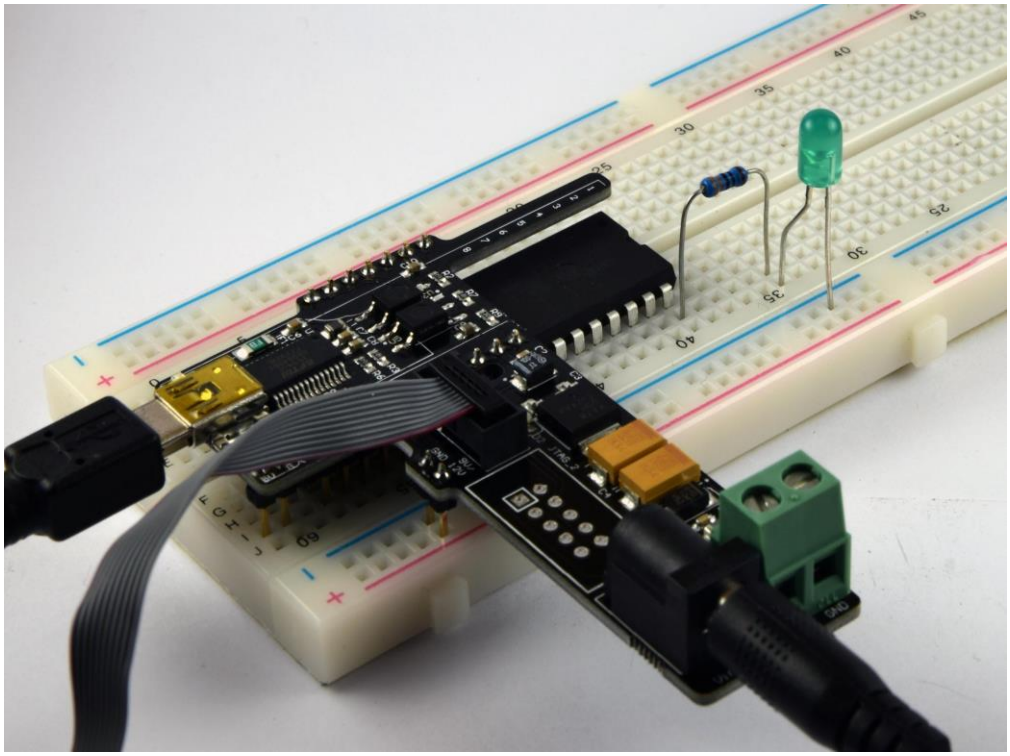
A kapcsolási rajz MCU része és a tápellátásért felelős rész a következő ábrán látható:



7. ábra *- Mikrovezérlő és tápellátásának kapcsolási rajza*

A kapcsolás felső részén a tápellátásért felelős áramkör található 7805-ös lineáris feszültség stabilizátor áramkör köré építve. Alatta a csatlakozók áramköri rajzai láthatók a bekeretezett részen. Legalul magának a mikrokontrollernek a bekötése látható.

Fizikailag egy nyomtatott huzalozású lemezen foglal helyet az egész hardver egység.



***8. ábra** - Adapter NyÁK, mikrovezérlő és az USB kapcsolata*

A továbbiakban próbapanel alatt a fehér breadboardot értem. FTDI adapter alatt pedig a fekete nyomtatott huzalozású lemezt, amit a próbapanelre lehet ráilleszteni.

FTDI DRIVER TELEPÍTÉSE

ELŐFELTÉTELEK

A driver telepítés windows 7 vagy újabb operációs rendszer alatt különösebb előkészítést nem igényel. Az USB-UART átalakító IC-t, ha először észleli a PC, akkor automatikusan megtalálja és feltelepíti a megfelelő meghajtó programot. A következő előzetes lépések szükségesek:

1. FTDI adapter csatlakoztatása a próbapanelre
2. Próbaáramkör feszültség alá helyezése a külső táp segítségével az adapteren keresztül. Ez történhet egyrészt kompatibilis dugasztáppal a kör alakú csatlakozón keresztül (a belső tüske a +, a külső rész a -) vagy a sorkapcson keresztül megfelelő polaritással. Mind a két esetben a tápegység feszültsége 7-25V közé essen. A tápegység kimenő áramának ajánlott terhelhetősége legalább 500mA legyen.
3. PC és FTDI adapter összekötése USB 2.0 kábel segítségével

TELEPÍTÉS MENETE

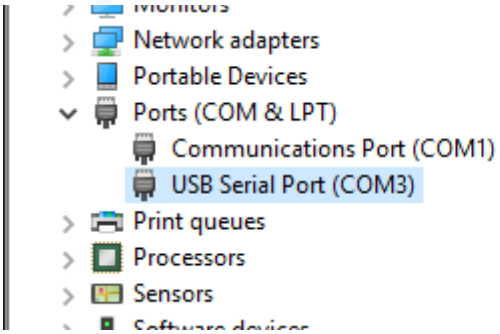
Ha eddig mindent jól csináltunk, akkor a driver automatikusan települ.

TELEPÍTÉS ELLENŐRZÉSE

Amint a telepítés befejeződött a következőket kell leellenőriznünk számítógépünk eszközkezelőjében:

1. Létrejött egy új virtuális COM port
2. A PC felismerte az FTDI chip-et, mint „USB serial converter”

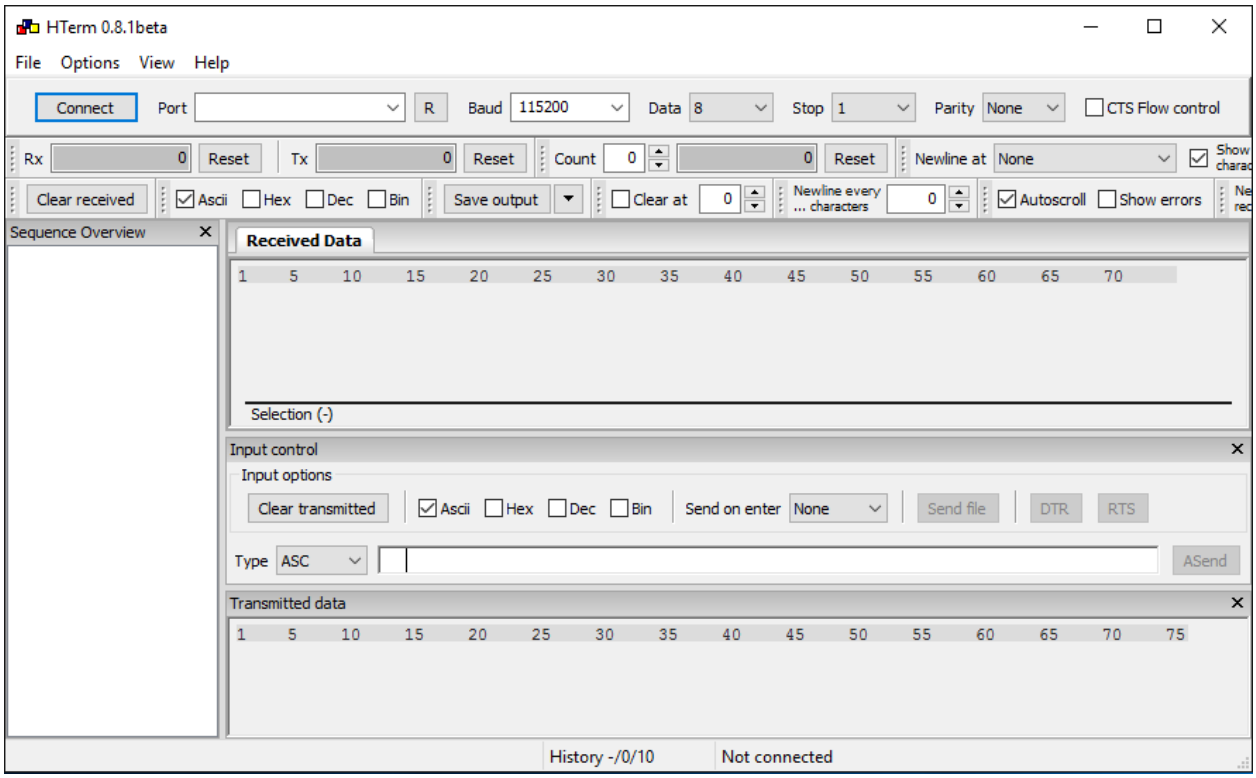
Az eszközkezelőnkben ez a következőképpen kell, hogy megjelenjen:



***9. ábra** – Soros port driver az eszközkezelőben*

HTERM

Adatok küldésére és fogadására a PC oldalon különböző lehetőségeink vannak. Egyik ezek közül a HTERM nevű terminál program. Az alkalmazás képes adatok küldésére és fogadására. Felhasználói felülete a következőképpen néz ki:

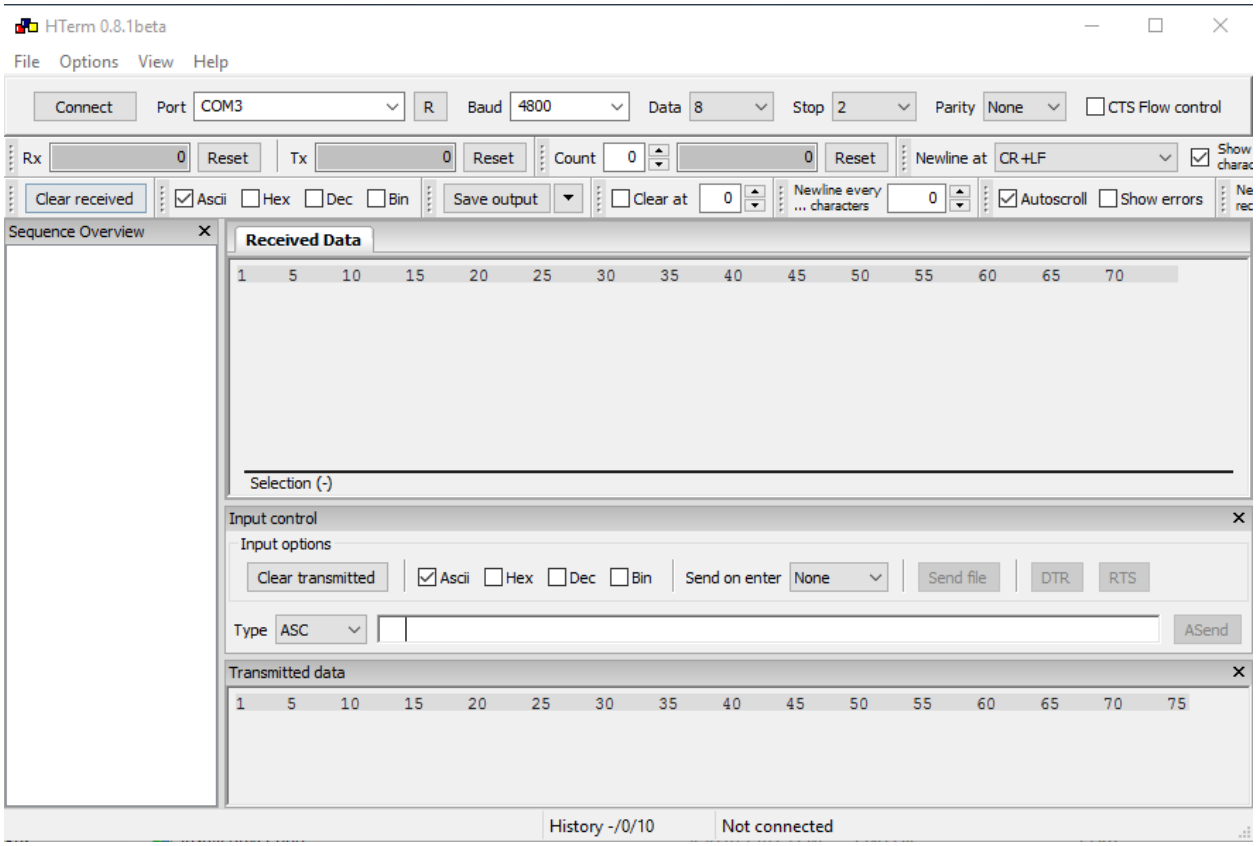


***10. ábra** – Hterm kezelőfelület az alapbeállításokkal*

Használata nagyon egyszerű. A portnál ki kell választani a megfelelő virtuális COM portot. A telepítés ellenőrzése részben látott módon az eszközkezelőben győződhetünk meg az FTDI chiphez rendelt port számáról! Nálam éppen a COM3 lett hozzárendelve, ezért én a példákban ezt fogom megadni. Ez a szám gépről gépre változhat, ezért ellenőrzése feltétlenül szükséges!

Baud rate-nek 4800-at állítsunk be. Ezen a sebességen még hibamentes a kommunikáció. Magasabb sebességen az adatvezetékek optocsatolóval való leválasztása miatt már annyira torzulna a jelek alakja, hogy hibamentesen nem lehetne őket dekódolni. A 8 adatbit, mint alapértelmezett beállítás marad, a stopbitek száma legyen kettő. Paritást nem használok, ezért marad az alapértelmezett “None”-on.

Érdekes lehet még a “Newline at” mező. Ennek az értékét a windows rendszereken használatos CR+LF-re állítottam. Jelentéséről később több szó esik. A lényeg, hogy ha sortörés karakter érkezik, akkor a HTerm is úgy jeleníti meg a szöveget, hogy beszúr egy sortörést. Nézzük meg, hogy kell a beállításoknak kinéznie:



***11. ábra** – Szükséges beállítások a HTerm-en*

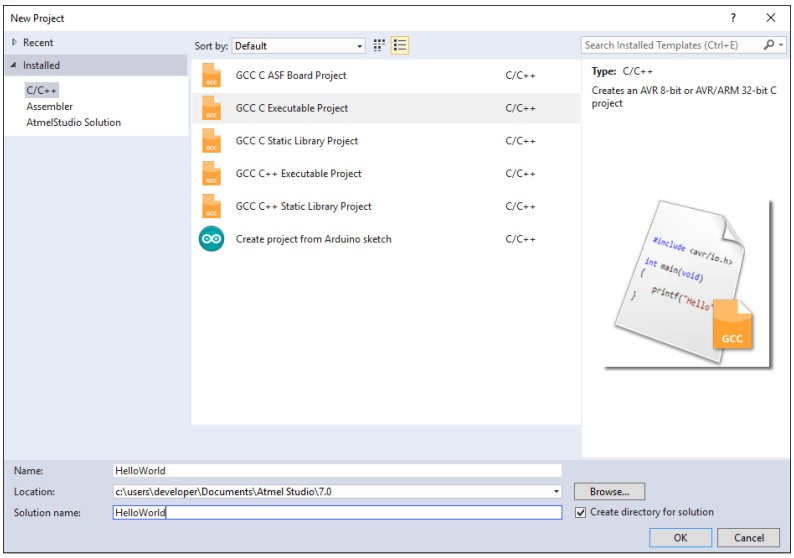
Mielőtt rányomnánk a connect feliratú gombra és elkezdenénk kommunikálni a mikrokontrollerünkkel egy programot kell írnunk, ami megvalósítja a mikrovezérlő oldali kommunikációt. A következő alpontban egy ilyen egyszerű programot fogok implementálni.

HELLO WORLD!

Helló Világ alatt azt a programot értjük, ami csak annyit csinál, hogy kiírja a következőt: Helló Világ! Az ATmega16A mikrokontrollert először beállítjuk, vagy szakmai zsargonnal „felinicializáljuk”, majd kiküldünk róla egy karaktersorozatot: „Hello World!”

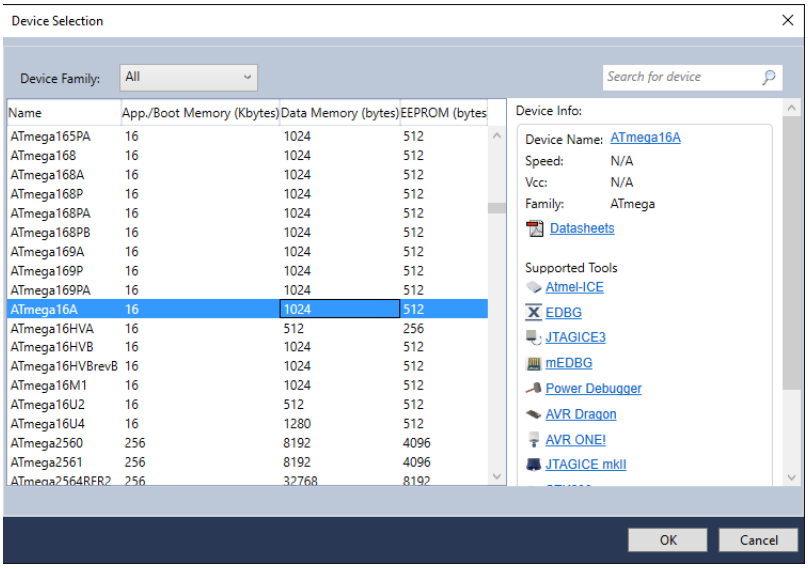
PROJEKT LÉTREHOZÁSA

Először hozzunk létre egy új *GCC C Executable Project*-et az Atmel Studio-ban HelloWorld névvel:



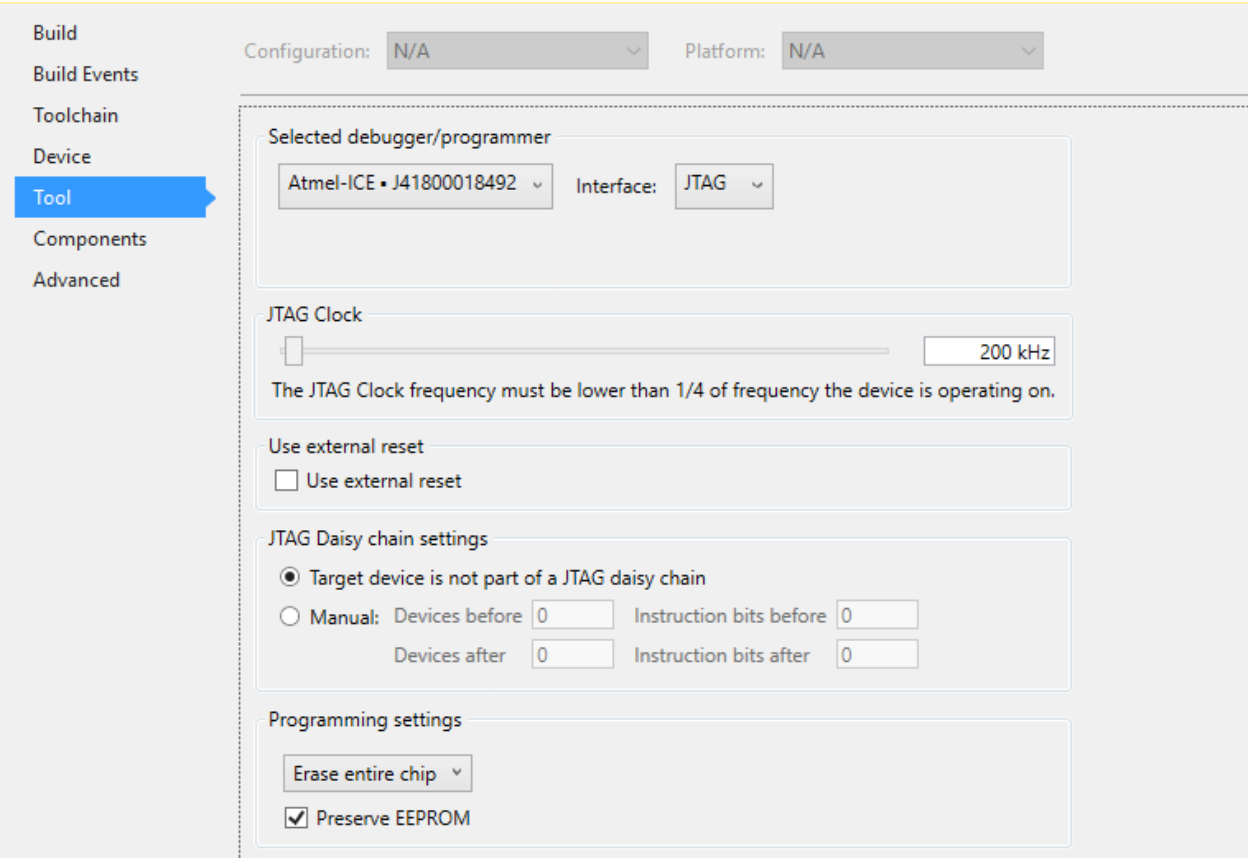
12. ábra *- Projekt létrehozása az Atmel Studio-ban*

Az eszközválasztásnál válasszuk az ATmega16A-t.



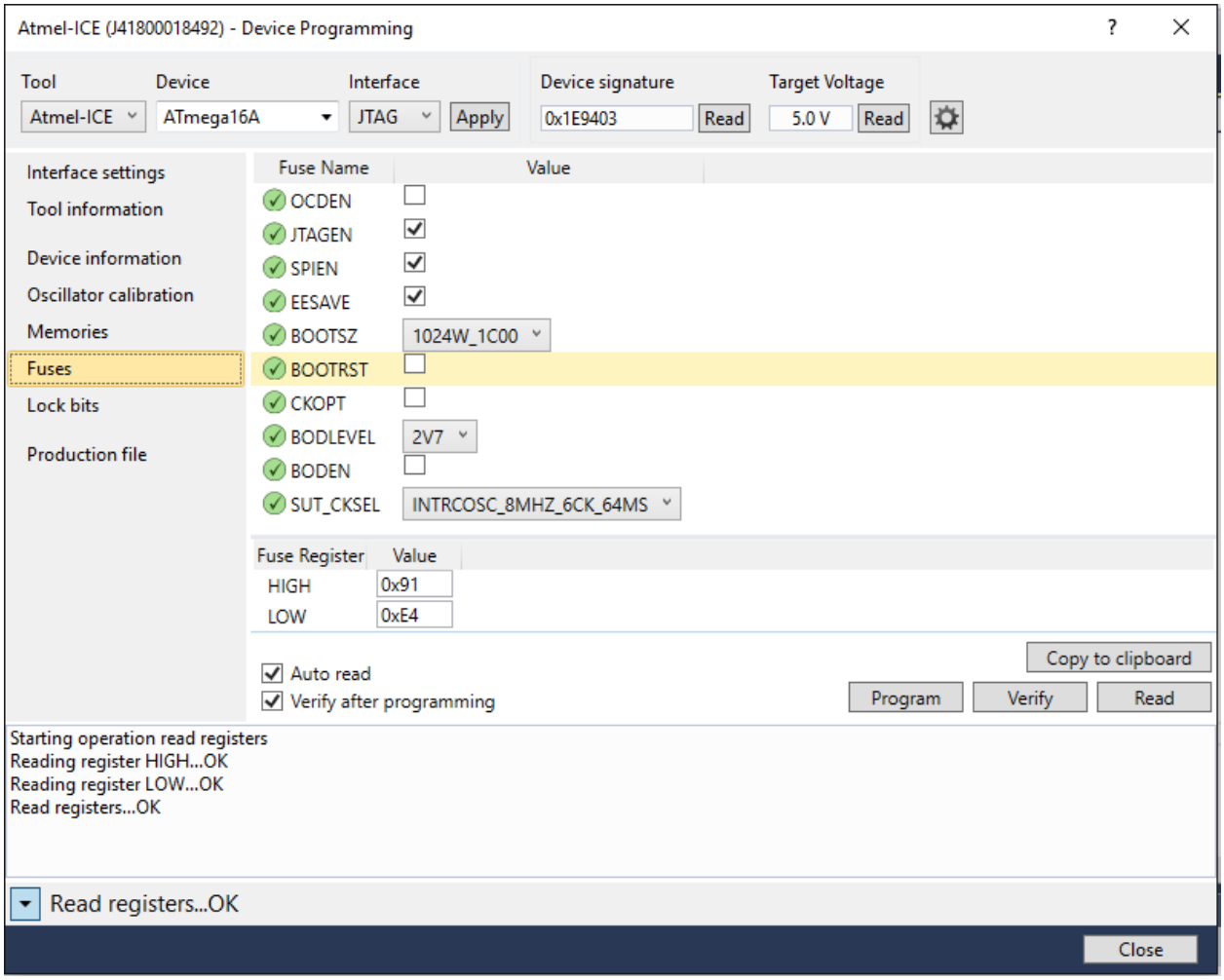
13. ábra *- Mikrovezérlő kiválasztása az Atmel Studio-ban*

Majd nyomjunk az OK gombra. Ezután állítsuk be a projekt tulajdonságai alatt (jobb gomb a projekten, majd “properties”) a programozónk típusát. Én az Atmel-ICE debuggert használom, ezért a példában is ez fog szerepelni. Lássuk a helyes beállításokat Atmel-ICE debugger esetében:



***14. ábra** - Programozó eszköz kiválasztása*

Ezek után nem árt leellenőriznünk még pár fontos beállítást, amiket a mikrokontroller úgynevezett fuse bitek formájában tárol. Ezek a beállítások maradandóak és óvatosan kell velük bánni, hiszen olyan értékeket lehet átállítani, mint a processzor frekvenciája vagy az egyes debuggerek engedélyezése. Nyissuk meg a Tools->Device Programming alól a Fuses ablakot:



15. ábra *- Mikrovezérlő paramétereinek beállítása*

Ha a beállítások megegyeznek, akkor örülünk. Ha nem, akkor állítsuk át őket, majd nyomjunk a program gombra az érvényre jutáshoz.

PORTOK INICIALIZÁLÁSA

A beállítások után elkezdhetjük megírni a konkrét programot.

Kezdésnek kapunk egy minimális majdnem üres main.c fájlt a következő tartalommal, amit a fejlesztőkörnyezet generált számunkra:

```
#include <avr/io.h>

int main(void)
{
	/* Replace with your application code */
	while (1)
	{
	}
}
```

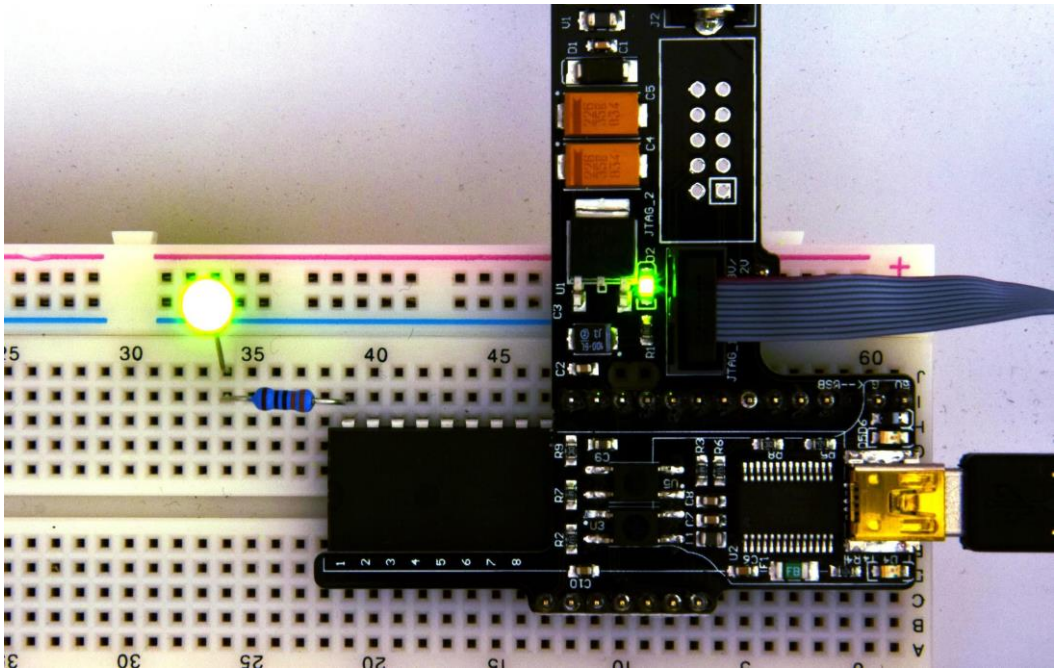
Ez a tananyag már feltételezi, hogy az olvasó rendelkezik alapszintű C nyelvismerettel. A kódrészletnek világosnak kell lennie.

A fenti forrás ebben a formában semmit nem csinál, ezért először ki kell egészítenünk az inicializációs résszel. Amikor egy mikrokontroller elindul, akkor először alapállapotba kerül. Éppen ezért minden beágyazott program első lépése az inicializáció. Ez az a lépés, amikor beállítjuk a szükséges paramétereket, előkészítjük a terepet programunk futtatásához. AVR mikrokontrollerek esetében alapesetben minden portláb bemenet és logikailag alacsony szinten van, illetve a perifériák tiltva vannak.

A nem használt portlábakat továbbra is bemenetként állítsuk be, ill. logikailag magas jelszintre. Ez megakadályozza, hogy "lebegjenek". További információért lásd: ATmega16A adatlap 12.2.4 bekezdés. Az A porton a legelső lábra kössünk egy státusz LED-et. Ha az eszköz inicializált állapotba kerül, akkor a LED kigyullad. Lássuk a kódot:

```
int main(void)
{
	/* PORTA felhúzóellenállások bekapcsolása a bemeneteken
	 * PA0 kimenet legyen logikai 1 jelszinttel(5V),
	 * ezen a lábon van státusz LED
	 * A többi pin legyen bemenet */
	PORTA = 0xFF;
	DDRA = 0x01;

	/* PORTB felhúzóellenállások bekapcsolása és minden pin bemenet */
	PORTB = 0xFF;
	DDRB = 0x00;

	/* PORTC felhúzóellenállások bekapcsolása és minden pin bemenet */
	PORTC = 0xFF;
	DDRC = 0x00;

	/* PORTD felhúzóellenállások bekapcsolása és minden pin bemenet */
	PORTD = 0xFF;
	DDRD = 0x00;

	while (1)
	{
	}
}
```

Ha a fenti kódot lefuttatjuk a "Start Without Debugging" gomb vagy a "Ctrl+Alt+F5" billentyűkombinációval, akkor az állapot LED kigyulladását kell tapasztalnunk. Örömmel konstatálhatjuk tehát, hogy az I/O lábak inicializációja sikeres volt.



***16. ábra** - Példakód futása a mikrovezérlőn*

UART INICIALIZÁLÁSA

A következő dolgunk az UART modul felkonfigurálása a HTerm alpontban említett 8 adatbitre, és 2 stopbitre. Szerencsére a mikrokontroller gyártója az ATmega16A adatlapjában példakódokat is közöl az egyes funkcionalitásokhoz. Az USART fejezetben a 146-os oldalon közli a következő kódot az inicializációhoz:

```
void USART_Init( unsigned int ubrr)
{
      /* Set baud rate */
      UBRRH = (unsigned char)(ubrr>>8);
      UBRRL = (unsigned char)ubrr;
      /* Enable receiver and transmitter */
      UCSRB = (1<<RXEN)|(1<<TXEN);
      /* Set frame format: 8data, 2stop bit */
      UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}
```

Nincs más dolgunk, mint beszúrni a HelloWorld programunk forrásfájljába a main függvény elé és meghívni a fenti függvényt a helyes értékkel az I/O init után. A helyes értéket ugyanezen adatlap 165. oldaláról puskázhatjuk ki.

A 4800 baudhoz tartozó UBRR érték 8MHz-es processzorfrekvencia mellet 103. Az UBRRH és UBRRL regiszterek a baud rate beállításáért felelősek (lásd ATmega16A adatlap 19.11.5), az UCSRB (lásd ATmega16A adatlap 19.11.3) módosításával kapcsolhatjuk be az adót és a vevőt, illetve az UCSRC (lásd ATmega16A adatlap 19.11.4) felelős a frame formátum beállításáért. A kódunk állapota a beillesztés után:

```
#include <avr/io.h>

void UARTInit( unsigned int ubrr)
{
	/* Baud rate beállítása */
	UBRRH = (unsigned char)(ubrr>>8);
	UBRRL = (unsigned char)ubrr;
	/* Adó és vevő engedélyezése */
	UCSRB = (1<<RXEN)|(1<<TXEN);
	/* Frame formátum beállítása: 8data, 2stop bit */
	UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}

int main(void)
{
	/* PORTA felhúzóellenállások bekapcsolása a bemeneteken
	 * PA0 kimenet legyen logikai 1 jelszinttel(5V),
	 * ezen a lábon van státusz LED
	 * A többi pin legyen bemenet */
	PORTA = 0xFF;
	DDRA = 0x01;

	/* PORTB felhúzóellenállások bekapcsolása és minden pin bemenet */
	PORTB = 0xFF;
	DDRB = 0x00;

	/* PORTC felhúzóellenállások bekapcsolása és minden pin bemenet */
	PORTC = 0xFF;
	DDRC = 0x00;

	/* PORTD felhúzóellenállások bekapcsolása és minden pin bemenet */
	PORTD = 0xFF;
	DDRD = 0x00;

	UARTInit(103);

	while(1)
	{
		/* Végtelen ciklus */
	}

	/* Ide sosem jutunk el, de szintaktikailag illik ide tenni */
	return 0;
}
```

UART KÜLDÉS

Az UART-ot már inicializáltuk, de küldeni még nem tudunk. Szerencsére az adatlap tartalmaz küldésre vonatkozó példakódot is. Ez a 147-es oldalon a 19.6.1 alatt található:

```
void UARTTransmit( unsigned char data )
{
	/* Addig várunk, amíg az adó buffer ki nem ürül */
	while ( !( UCSRA & (1<<UDRE)) )
	{
		/* Üres ciklus. Nem csinálunk mást csak várunk. */
	}
	/* Betesszük az adatot a bufferbe.
	 * A küldést a mikrokontroller HW megoldja. */
	UDR = data;
}
```

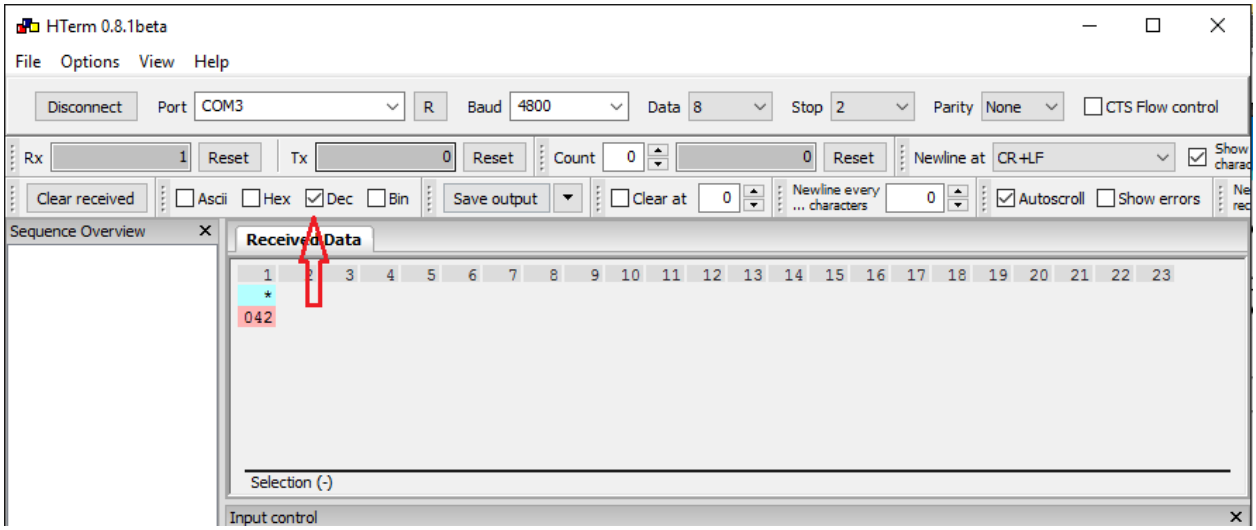
A kód megvárja, hogy az adó befejezze az esetleges küldést, majd behelyezi az adatot az UDR regiszterbe, ahonnan a hardveres UART modul küldi ki a már beállított paraméterekkel.

Adat alatt 8bites, azaz 1 bájtos értéket értünk. Egy előjel nélküli bájtban 0-255-ig terjedő értéket lehet betenni. Küldjük el a PC-nek a 42-őt! A kód az alábbiak szerint fog módosulni:

```
#include <avr/io.h>

void UARTInit( unsigned int ubrr)
{
	/* Baud rate beállítása */
	UBRRH = (unsigned char)(ubrr>>8);
	UBRRL = (unsigned char)ubrr;
	/* Adó és vevő engedélyezése */
	UCSRB = (1<<RXEN)|(1<<TXEN);
	/* Frame formátum beállítása: 8data, 2stop bit */
	UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}

void UARTTransmit( unsigned char data )
{
	/* Addig várunk, amíg az adó buffer ki nem ürül */
	while ( !( UCSRA & (1<<UDRE)) )
	{
		/* Üres ciklus. Nem csinálunk mást csak várunk. */
	}
	/* Betesszük az adatot a bufferbe.
	 * A küldést a mikrokontroller HW megoldja. */
	UDR = data;
}

int main(void)
{
	/* PORTA felhúzóellenállások bekapcsolása a bemeneteken
	 * PA0 kimenet legyen logikai 1 jelszinttel(5V),
```

```
         * ezen a lábon van státusz LED
         * A többi pin legyen bemenet */
        PORTA = 0xFF;
        DDRA = 0x01;

        /* PORTB felhúzóellenállások bekapcsolása és minden pin bemenet */
        PORTB = 0xFF;
        DDRB = 0x00;

        /* PORTC felhúzóellenállások bekapcsolása és minden pin bemenet */
        PORTC = 0xFF;
        DDRC = 0x00;

        /* PORTD felhúzóellenállások bekapcsolása és minden pin bemenet */
        PORTD = 0xFF;
        DDRD = 0x00;

        UARTInit(103);

        UARTTransmit(42);

        while(1)
        {
              /* Végtelen ciklus */
        }

        /* Ide sosem jutunk el, de szintaktikailag illik ide tenni */
        return 0;
}
```

A HTerm-ben állítsuk be, hogy decimális formában várjuk az adatot és nyomjunk a “connect” gombra! A mikrokontrolleren pedig futtassuk a kódot! Az eredmény, ha mindent jól csináltunk magáért beszél:



17. ábra *- Decimális formátum beállítása a HTerm-ben*

KARAKTER KÜLDÉSE

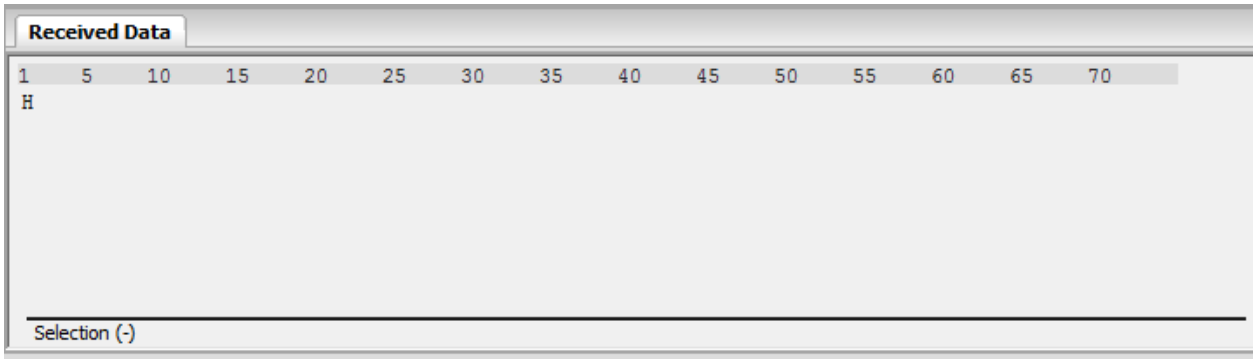
Lépjünk tovább és küldjük ki a nagy H betűt a PC felé.

A karakterek kódolására többféle eljárás használható. Ezek közül a legegyszerűbb az ASCII karaktertábla. Ez 1 bájton kódolja az angol ABC betűit és még egy csomó más írásjelet és karaktert.

A karaktertábla sok formában fent van az interneten. Ide most nem szúrnám be. A teljesség igénye nélkül elég ha azt tudjuk, hogy a nagy H kódja 72. Lássuk hogyan módosul a kódunk:

```
...
UARTTransmit(72); /* Egyenértékű ezzel: UARTTransmit('H'); */
...
```

Át kell állítanunk a HTerm-ben a bejövő adatok kódolását decimálisról ASCII-re, valamint törölnünk kell az eddig beérkezett adatokat. A módosított programunk lefuttatása után a következőt kell kapnunk:



***18. ábra** – A soros porton fogadott adat egy „H”betű*

Tehát valóban a H betűt sikerült elküldenünk.

STRING KÜLDÉSE

A következő dolog, hogy már az egész „Hello World!” mondatot elküldjük. Ehhez szükségünk lesz egy függvényre, ami egy C stringet, azaz karaktertömböt fogad paraméterként és sorban egyesével meghívja az egyes karakterekre az előzőleg megismert `USART_Transmit()` függvényünket. A függvény neve legyen `USART_TransmitString()`. Egy char típusú tömböt fog megkapni és nem lesz visszatérési értéke, tehát a deklaráció így néz ki:

```
void USART_TransmitString(char string[]);
```

C-ben a karaktertömbök úgynevezett “nul terminated string”-ek. Ez azt jelenti, hogy az utolsó elem a NUL karakter, aminek ASCII kódja is 0, de így is szokás írni: ’\0’.

Nézzük meg hogyan néz ki a „Hello World!” string C reprezentációja:

H	e	l	l	o		W	o	r	l	d	!	’\0’
0	1	2	3	4	5	6	7	8	9	10	11	12

A karaktertömb mérete 12 lesz, de magának a tömbnek a mérete 13! C-ben nagyon fontos, hogy a tömbök indexelése 0-tól kezdődik!

Lássuk a függvény implementációját:

```
void USART_TransmitString(char string[])
{
	unsigned char iterator;
	for(iterator = 0; string[iterator] != '\0'; iterator++)
	{
		USART_Transmit(string[iterator]);
	}
}
```

Egy for ciklus segítségével végig lépkedünk a tömb elemein, amíg el nem érünk a tömb végére, amit a nul karakter (’\0’) jelez.

Kitekintés

Egy C string hosszát a könyvtári strlen fügvénnyel is meghatározhatjuk, így a kódunk az alábbiak szerint módosulna:

```
void USART_TransmitString(char string[])
{
	unsigned char iterator;
	unsigned char stringLength = strlen(string);

	for (iterator = 0; iterator < stringLength; iterator++)
	{
		USART_Transmit(string[iterator]);
	}
}
```

Vagy pointerként is értelmezhetnénk a megkapott tömböt és akkor a következőt írhatnánk:

```
void USART_TransmitString(char * string)
{
	while(*string != '\0')
	{
		USART_Transmit(*string);
		string++;
	}
}
```

Ezek a megoldások mind egyformán jók, de sebességben eltérnek. Elemezzük ki az egyes megoldásokat!

A leggyorsabb a harmadik megoldás, azért mert nem indexelünk benne tömböt. Ugyanis ha azt írom, hogy string[1] az ugyanaz, mintha azt írnám, hogy *(string + 1). Tudni illik a tömb szemantikailag nem más, mint egy pointer a tömb nulladik elemére. Tehát a pointer értékét növelve lépkedünk végig a tömbön. Így megspóroltunk az első megoldáshoz képest 12db összeadást.

A leglassabb a második, ahol strlen-t használunk a tömb méretének megismeréséhez. Ez azért lassabb az elsőnél, mert az strlen belül nem más, mint egy for ciklus, ami végigmegy a tömbön addig, amíg meg nem találja a nul karaktert.

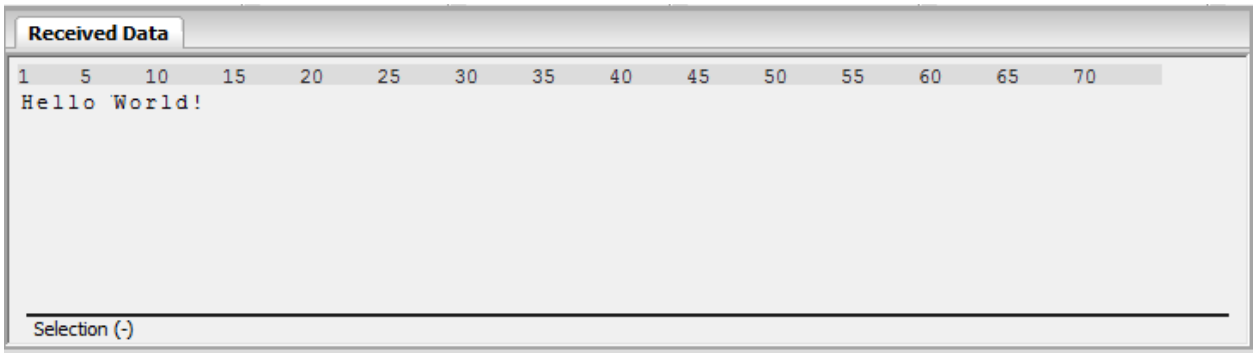
A leglassabb az lenne ha ezt írnánk:

```
void SendStringOnUART(char string[])
{
	unsigned char iterator;

	for (iterator = 0; iterator < strlen(string); iterator++)
	{
		USART_Transmit(string[iterator]);
	}
}
```

Mi ezzel a probléma? Hát az, hogy ez egy for ciklusba ágyazott for ciklus! Sőt még rosszabb, hiszen ez egy függvényhívás is egyben, aminek költsége van! Az strlen nem egyszer fut le, hanem annyiszor amekkora a string hossza! Kis elemszámnál ez nem probléma, de méretes tömböknél már komoly teljesítményt vesztünk a felesleges iterációkkal!

Szúrjuk be a forrásfájlunkba a frissen implementált függvényünket és hívjuk meg a main függvényből "Hello World!" argumentummal! A forrásfájl végleges formában elérhető a honlapról.

Ha mindent jól csináltunk, akkor a kód lefutása a következőt kell eredményezze a HTerm-ben:



***19. ábra** – A soros porton fogadott adat „Hello World!"*

A következő pontban egy összetettebb funkcionalitást valósítok meg szofisztikáltabb küldési és fogadási módokkal.

LED FUTÓFÉNY

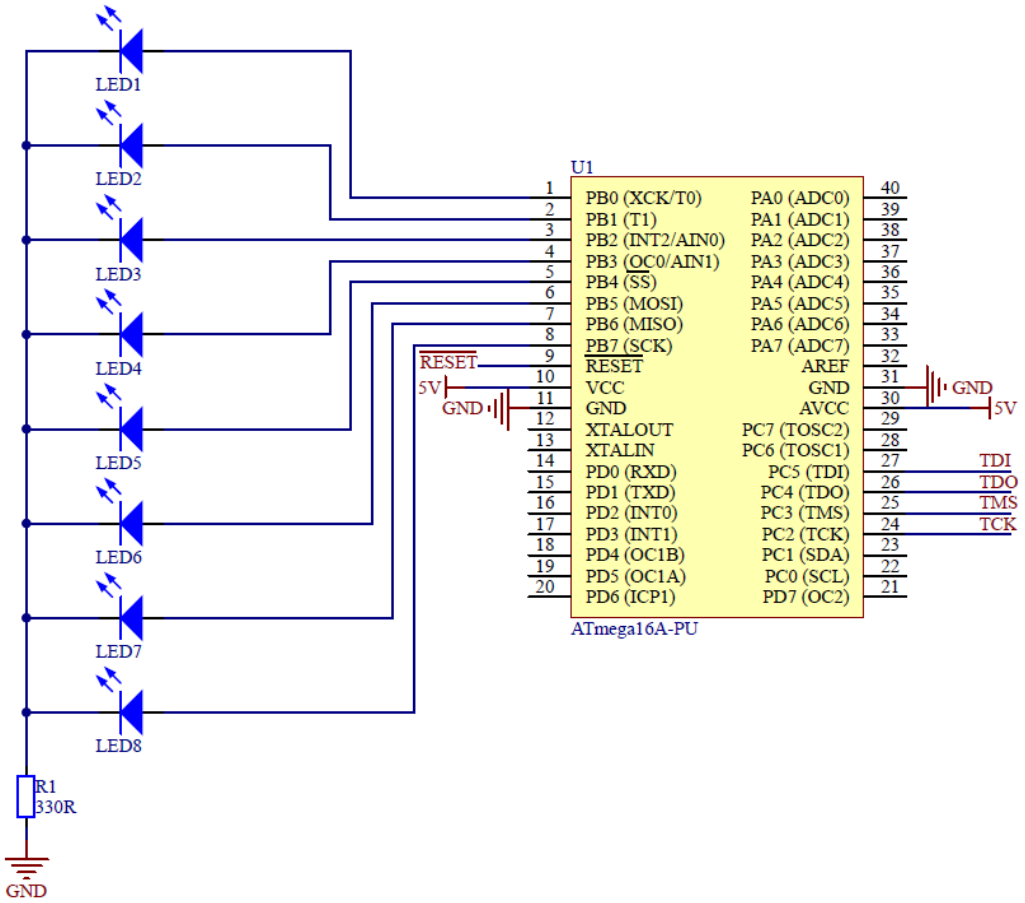
Az előző részben eljutottunk oda, hogy adatokat küldtünk a mikrokontrollerről. Most 8 LED-ből álló futófényt fogunk vezérelni.

HARDVER

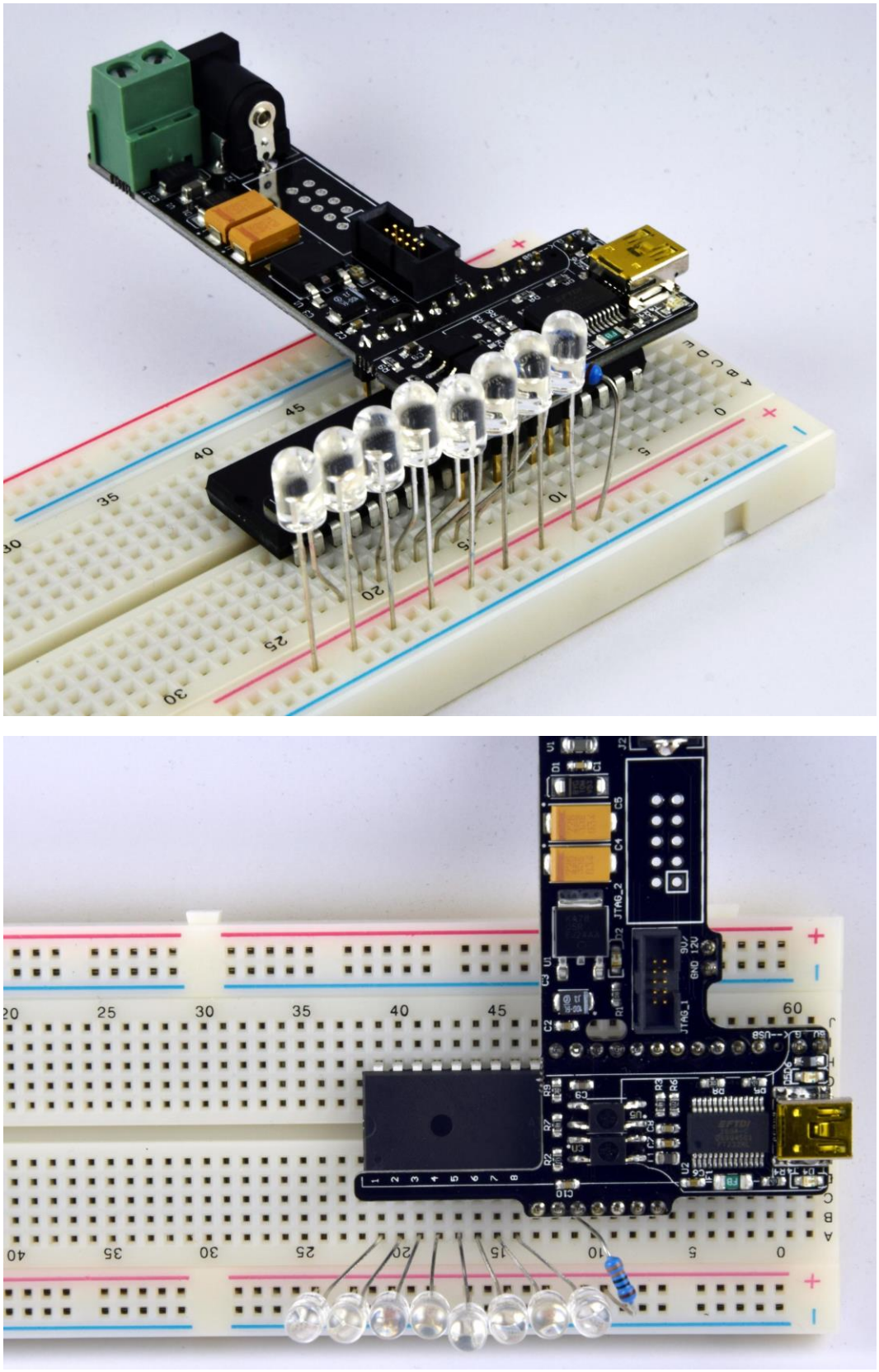
A LED-ek anódjait egy-egy portlábra kötöttem és a katódjaikat pedig közösítettem és egy ellenálláson keresztül a földre kapcsoltam..

A LED egy nemlineáris áramköri elem, amit áramgenerátorral szabad csak meghajtani. Jelen példában az áramgenerátor szerepét az ellenállás tölti be. Egy LED-en kb 1.7V esik ha már kinyitott, azaz áram folyik rajta keresztül. 1mA-től már éppen világít, és 10mA-rel már kielégítően működik. A tápfeszültség 5V, a feszültségesés 1.7V, azaz 3.3V-on kell 10mA-nek átfolynia. Tehát az ellenállás értékét a méltán népszerű R=U/I képlettel kapjuk: 330 Ohm.

Ha csak egy ellenállást használunk az összes LED-hez, akkor egyszerre csak egy LED világíthat, hiszen ha nem így lenne, akkor a fényerősség gyengülne a bekapcsolt LED-ek számának függvényében. Minél több LED világítana egyszerre annál kevesebb áram folyna át az egyes LED-eken. A kapcsolás a következőképpen néz ki:



20. ábra *- LED futófény kapcsolási rajza*



21. ábra *- LED futófény megépítve*

IMPLEMENTÁCIÓ

A futófény lényege, hogy egy világító LED-et a megfelelő paranccsal balra vagy jobbra tolunk el. Ha a parancsokat kellően sokszor egymás után elküldjük, akkor egy LED-es futófényt kapunk.

A szemfüles olvasó észrevehette, hogy az összes LED a B portra van kötve. Ez nem véletlen. A dolognak meglehetősen prózai oka van: egyszerűbb programot lehet írni ha csak egy portot kell vezérelni.

A következőkben a kód felépítését és működését fogom tárgyalni. A gondolatmenetemet is leírom, miközben ezt a kis programot összeraktam. A C nyelvben, amiben eddig is kódoltam, a főszereplő az adat és a rajta végzett műveletek. Kicsit leegyszerűsítve: adat=változó, művelet=függvény.

Én most a „bottom-up" metodikát követtem, azaz alulról felfelé építkeztem. Ezt a fajta gondolkodást a következőképpen lehet megfogalmazni: Vannak ezek az adataim, amiken ezeket meg ezeket a műveleteket kell végrehajtanom. Akkor kell majd olyan függvény, hogy ... és majd kell egy olyan is, hogy…

Amikor az összes alkotóelem kész van, akkor pedig csak össze kell „legózni" a kódot. Lássuk az "adatokat" és a "biztosan kell" típusú műveleteket:

- Adatok:
    - PORTB: „B" portot reprezentáló 8 bites változó.
    - UCSRA: UART-hoz tartozó ki vagy bemeneti 8 bites adat
- Műveletek, azaz biztosan kell olyan függvény, ami…
    - Inicializál
    - Balra lépteti a futófényt
    - Jobbra lépteti a futófényt
    - Adatot fogad
    - Adatot küld

Inicializáció

Két dolgot kell indulás után beállítanunk:

1. Azon I/O lábakat kimenetre kell állítanunk, amelyekre a LED-ek vannak kötve.
2. Az UART-ot be kell kapcsolnunk és a helyes beállításokat kell alkalmaznunk

Balra léptetés

A következő kódrészlet, ami egyben egy teljes függvény is, a balra léptetést hivatott megvalósítani. Ha azt szeretnénk, hogy az adott port lábon a LED ne világítson, akkor bináris 0-át, ha pedig azt, hogy világítson, akkor bináris 1-et kell beírnunk a PORTB regiszterbe a megfelelő pozícióba. Egyszerre mindig csak egy LED világít, így a PORTB-n az összes lehetséges kombináció a következő kettes-, tizenhatos-, ill. tízes számrendszerben reprezentálva:

0000 0001 = 0x01 = 1

0000 0010 = 0x02 = 2

0000 0100 = 0x04 = 4

0000 1000 = 0x08 = 8

0001 0000 = 0x10 = 16

0010 0000 = 0x20 = 32

0100 0000 = 0x40 = 64

1000 0000 = 0x80 = 128

A függvény feladata, hogy azt az egy szem egyest tolja el egyel balra. Erre a C nyelvben a bináris balra tolás operátort, azaz a „<<”-t használhatjuk. Ez az operátor az egész számot egyel eltolja balra és behoz egy 0-át jobbról. Bármilyen értékre meghívhatjuk ezt az operátort kivéve egyet: 1000 0000. Ez a bal oldali végállapot, amikor a ciklus újraindul és a következő érték a kezdeti lesz, azaz 0000 0001.

A megvalósított feladat a következő: ha nem végállapot van, akkor léptessük egyel balra a fényt. Ha végállapot van akkor újra a kezdő értéket vegye fel PORTB. Kód formában ez a következőképpen néz ki:

```
void RunningLightLeft(void)
{
	PORTB = (PORTB != 0x80) ? (PORTB << 1) : (0x01);
}
```

Ezt így is írhattam volna:

```
void RunningLightLeft(void)
{
	if(PORTB != 0x80)
	{
		PORTB <<= 1; // Megegyezik ezzel: PORTB = PORTB << 1;
	}
	else
	{
		PORTB = 0x01;
	}
}
```

Jobbra léptetés

A jobbra léptetés a balra léptetés analógiájára történik azzal a különbséggel, hogy itt a jobb szélső végállapotot, azaz 0x01-et kell kitüntetett módon kezelnünk. Lássuk a kódot:

```
void RunningLightRight(void)
{
	PORTB = (PORTB != 0x01) ? (PORTB >> 1) : (0x80);
}
```

Adat fogadása

Az USART adat regisztere UDR (USART I/O Data Register) néven érhető el. Innen kell kiolvasnunk a beérkező adatot. Azt, hogy ez mikor történik meg, egy másik regiszterből, az UCSRA-ból (USART Control and Status Register A) tudhatjuk meg. Egészen pontosan az előző regiszternek a 7. bitje (amire RXC-ként (RX Complete) hivatkozhatunk) mondja meg, hogy érkezett-e adat vagy sem. Nézzük, hogy ez hogyan implementálható kód formájában:

```
unsigned char USART_Receive( void )
{
	/* Várunk az adatokra */
	while ( !(UCSRA & (1<<RXC)) );
	/* Visszatérünk a vett adatokkal */
	return UDR;
}
```

A fenti kód az ATmega16A adatlapjának 150. oldaláról kimásolt példakód. Egyszerűsíthetünk ha tudjuk, hogy az AVR Libc, azaz Atmel által kiadott standard C könyvtárak részhalmaza tartalmaz egy olyan makrót, amivel addig tudunk várakozni, amíg egy bit be nem „áll”: loop_until_bit_is_set(sfr,bit)

```
char uart_getchar(FILE *stream)
{
	loop_until_bit_is_set(UCSRA, RXC);  /* Addig várunk,
	                                     * amíg nem érkezik adat. */
	return UDR;
}
```

Tovább könnyíthetjük a dolgunkat, ha használjuk az FDEV_SETUP_STREAM makrót. Ennek a segítségével beállíthatunk egy ideiglenes tárolót, amivel használható lesz a C stdin getchar() függvény. A buffer FILE típusú lesz. A makró pontos használata pedig a következő:

```
FILE uart_input = FDEV_SETUP_STREAM(NULL, uart_getchar, _FDEV_SETUP_READ);
```

Nem maradt más dolgunk, mint a stdin-t átirányítani az újonnan létrehozott bufferünkre:

```
stdin = &uart_input;
```

A következőkben egy karakter beolvasása UART-ról a következőképpen történik:

```
char karakter = getchar();
```

Adat küldése

Adat küldésére is az UDR regisztert használhatjuk. A küldés is karakterenként történik, úgy mint a vétel. A küldés során nem írhatunk bármikor ebbe a regiszterbe hiszen lehet, hogy még egy előző küldés várakozik vagy éppen bejött egy bájtnyi adat. Az UCSRA regiszter UDRE (Usart Data Register Empty) bitje indikálja ha szabad a buffer és mehet a küldés. C kódként megfoglmazva a következőt kapjuk:

```
void uart_putchar(char c, FILE *stream) {
	/* Ha sor vege karakter jon, akkor be kell szurni egy \r-t
	 * hogy windows kompatibilis legyen*/
	if( c == '\n')
	{
		loop_until_bit_is_set(UCSRA, UDRE);
		UDR = '\r';
	}
	loop_until_bit_is_set(UCSRA, UDRE);
	UDR = c;
}
```

A fenti kód némi magyarázatra szorul. Ha már a stdin-t használjuk, akkor logikus, hogy használjuk a stdout-ot is. Így a jól megszokott módon printf-el és puts-al küldhetünk információt UART-on. Ahhoz, hogy a stdout-ot át tudjuk irányítani a sor vége karakternek (\n) a windowsos formátumát (\r\n) kell használnunk. Ezért, ha leírjuk majd a következőt:

```
puts("Ez egy teljes sor, aminek a vegen sortores van.\n");
```

akkor ennek a karaktersorozatnak a végén valójában nem csak egy \n lesz, hanem \r\n. A következő dolgunk, hogy itt is létrehozunk egy ideiglenes tárolót, amit átadhatunk a stdout-nak:

```
FILE uart_output = FDEV_SETUP_STREAM(uart_putchar, NULL, _FDEV_SETUP_WRITE);
```

Az átirányítás:

```
stdout = &uart_output;
```

A következőkben írásjelek sorozatának küldése kétféleképpen lesz lehetséges. Formázás nélkül a puts() segítségével, ill. formázással a printf() segítségével. Például tegyük fel, hogy ki akarjuk küldeni azt a karaktersorozatot, hogy: „Hello vilag!”. Ezt a következőképpen tehetjük meg:

```
puts("Hello vilag!");
```

Ha valami bonyolultabbat szeretnénk kiíratni, akkor a printf-et kell használnunk. Tegyük fel, hogy egy érzékelő van kötve a mikrokontrollerre, ami a külső hőmérsékletet méri. Ezt az adatot a „temperature” nevű változóban tároljuk és ki akarjuk küldeni uart-on ezt az információt. A kód a következő:

```
printf("A homerseklet a kovetkezo: %d \n",temperature);
```

Kitekintés

A sor vége karakter eltérő a különböző operációs rendszereken. UNIX/LINUX alatt csak \n, míg windows-on \r\n. [1]

A printf nem csak számot tud beformázni adott szövegbe, hanem karaktersorozatot és még sok mást is. [2]

Main függvény

Nincs más dolgunk, mint az eddigiek összerakása a main függvényben:

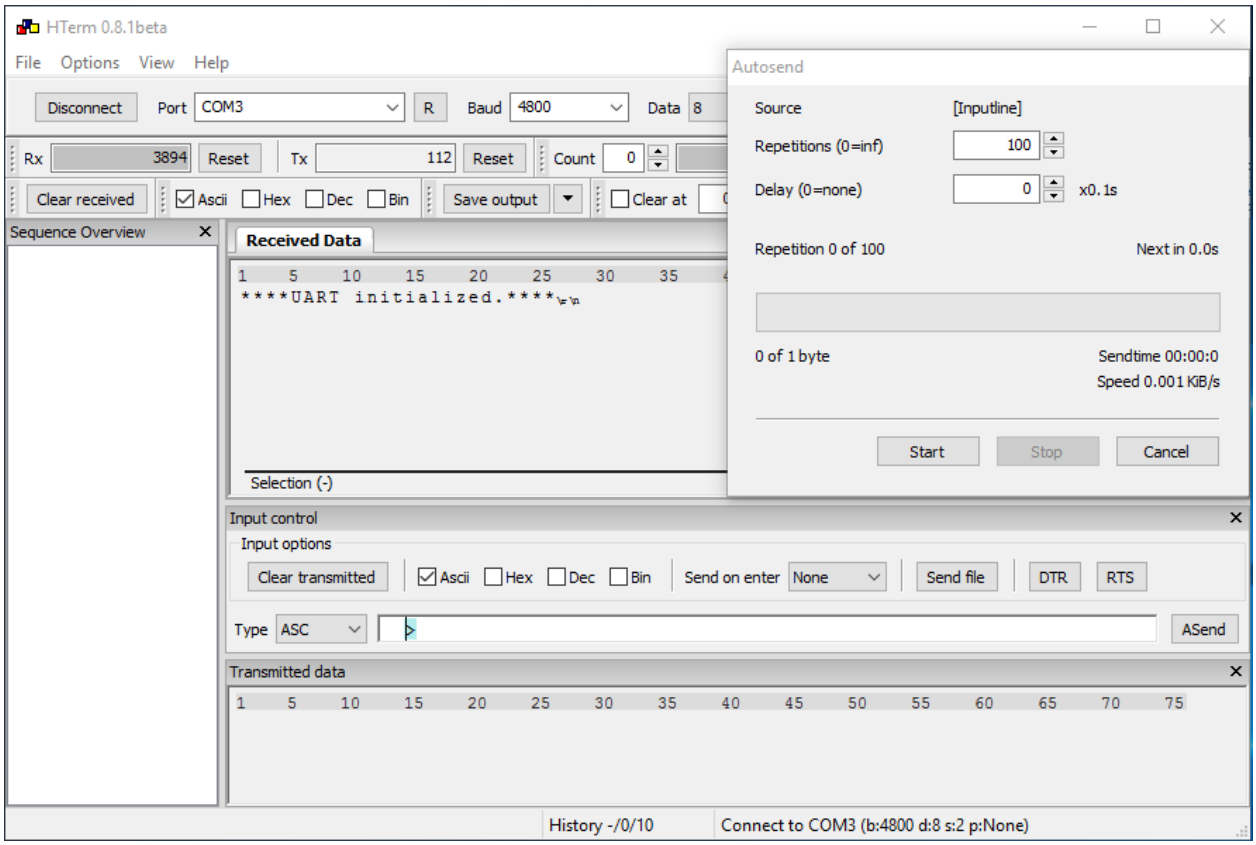
```
/* Makró definíciók */
#define  LED_CODE 0x55
#define  LEFT      0x3C
#define  RIGHT    0x3E

int main(void)
{
      /* Buffer változó a vett adat számára */
      char buffer;

      /* Ki- és bemenetek inicializálása */
      IOInit();
```

```
	/* Valid kezdőérték beállítása PORTB-nek */
	PORTB = 0x01;

	/* UART inicializálása, 103=4800 baud,
	 * 165.oldal ATmega16A adatlap */
	UARTInit(103);

	FILE uart_output =
FDEV_SETUP_STREAM(UARTPutchar,NULL,_FDEV_SETUP_WRITE);
	FILE uart_input =
FDEV_SETUP_STREAM(NULL,UARTGetchar,_FDEV_SETUP_READ);
	stdout = &uart_output;
	stdin  = &uart_input;

	/* Tesztelés */
	puts("****UART initialized.****");

	/* Végtelen ciklus */
	while (1)
	{
		/* A vett adatot betöltöm a bufferbe */
		buffer = getchar();

		/* Beérkező adat feldolgozása switch-case szerkezettel:
		 * A szerkezet lehetővé teszi, hogy a vizsgálandó változót
		 * egy listányi adattal vessük össze.
		 * Minden egyes adat érték egy case-nek felel meg.
		 */
		switch(buffer)
		{
			case LED_CODE:
			{
				puts("Led billentes erkezett");
				PORTA = ~PORTA;
				break;
			}
			case LEFT:
			{
				puts("Futofeny balra parancs erkezett");
				RunningLightLeft();
				break;
			}
			case RIGHT:
			{
				puts("Futofeny jobbra parancs erkezett");
				RunningLightRight();
				break;
			}
			default:
			{
				printf("Ismeretlen parancs: %c \n", buffer );
```

```
                        break;
                    }
                }
            }
        return 0;
}
```

Használat

A ‘>’ karakter hatására a futófény jobbra lép, a ’<’ hatására pedig balra. Lehetőség van a státusz LED állapotának megváltoztatására is. Ebben esetben a hexadecimális 55-öt kell elküldenünk. Ha gyorsan egymás után sokszor küldjük a balra vagy jobbra parancsokat, akkor valódi futófényt kapunk. Ezt a HTerm autosend funkciójával érhetjük el, ahol pl. 100 ismétlést is beállíthatunk az alábbiak szerint:



22. ábra *- LED futófény vezérlése a HTerm-ből*

ÖSSZEFOGLALÁS

A tananyagrész megismertette az olvasóval az UART periféria működési elvét és annak használatát egy konkrét példán keresztül. A megszerzett tudás segítségével megannyi új feladat és kihívás válik könnyen megoldhatóvá. Ne feledjük el azonban, hogy az UART tipikusan egy egyszerű kommunikációs forma, így megvannak a maga limitációi is. Mind sebességben, mind robosztusságban. Mielőtt ezt a perifériát választjuk kommunikációs célokra, érdemes végiggondolnunk kielégíti-e a szükségleteinket.