Crystal Clear Electronics

The development of the Crystal Clear Electronics curriculum was supported by the European Commission in the framework of the Erasmus + programme in connection with the “Developing an innovative electronics curriculum for school education” project under “2018-1-HU01-KA201-047718” project number.

The project was implemented by an international partnership of the following 5 institutions:

- Xtalin Engineering Ltd. – Budapest
- ELTE Bolyai János Practice Primary and Secondary Grammar School – Szombathely
- Bolyai Farkas High School – Târgu Mureș
- Selye János High School – Komárno
- Pro Ratio Foundation working in cooperation with Madách Imre High School – Šamorín

Contact http://crystalclearelectronics.eu/en/
info@kristalytisztaelektronika.hu

18 - USB connection

Written by Gábor Kalmár

English translation by Xtalin Engineering Ltd.

Revised by Ádám Szabó, Gergely Lágler

INTRODUCTION

At the end of this chapter, you will be able to control the microcontroller from your computer using the provided code.

In this chapter you can learn about serial communication between the microcontroller and the PC. The microcontroller we use does not have a proper USB stack, which is the name of the hardware elements making efficient communication over USB possible.

Instead, there is an interface for serial communication, which is the UART. With this peripheral it is possible to connect to a serial COM port of the PC. The problem is that on modern PC-s and laptops these serial ports are very rare, but almost all of them have USB.

The PC side USB, and the microcontroller side UART is connected by a special integrated circuit, called the FTDI FT232R USB-UART converter. With this solution it is almost like communicating with a real serial port, because the PC software emulates the behaviour of a serial port over USB, hiding the USB implementation from our eyes. We don't even have to think about it.

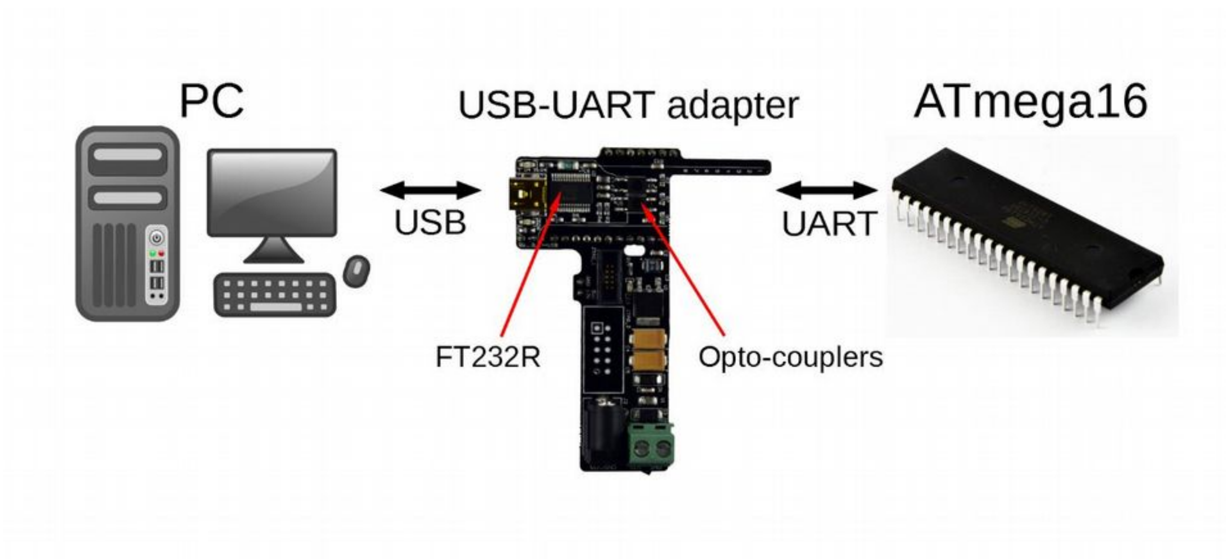


Figure 1 – Communication between the PC and the microcontroller

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

UART

The UART acronym stands for "Universal Asynchronous Receiver / Transmitter". Asynchronous communication means that the sender and the receiver are not synchronized, there is no common clock signal, which would determine the timing of the communication. Communication happens over two signal lines:

- Rx (receive)
- Tx (transmit)

These two lines are called the data bus or simply bus. When we talk about bus status later, it will mean the voltage levels of these two signal lines.

Voltage can be measured only relative to something. In this case voltage level means the potential difference between earth (GND) and the signal line.

Both signal lines can transmit and receive data. Rx (receive) and Tx (transmit) only means something in context of a chip, because the Tx line of one device is the Rx line of the other device. This also means that we connect two UART-compatible devices by connecting the Rx pin of a device to the Tx pin of the other one and vice versa.

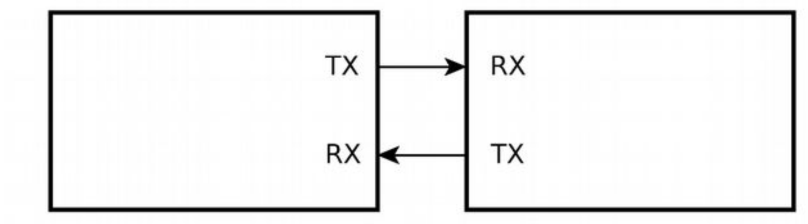


Figure 2 – Connecting two devices through UART

Because the communication is asynchronous, the receiver must know when does the transmitter start transmission, and when does it end. The standard describes special states for beginning and ending a transmission.

STRUCTURE OF AN UART FRAME

The data frame consists of – on top of the data itself - the synchronization bits (start and stop bits) and an optional parity bit. The structure looks like this:

- Start bit: logical level 0 for the duration of one bit
- Data bits: 5-9 data bits, usually 8
- Parity bit (optional): it can be odd or even parity
- Stop bit(s): logic level 1 for the duration of 1 or 2 bits

The idle state of the bus is the logical 1 level. The communication starts with the start bit, represented by logical 0, changing the bus state from idle. Then the data bits are transmitted, and finally the stop bit closes the communication, and the bus returns to idle.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

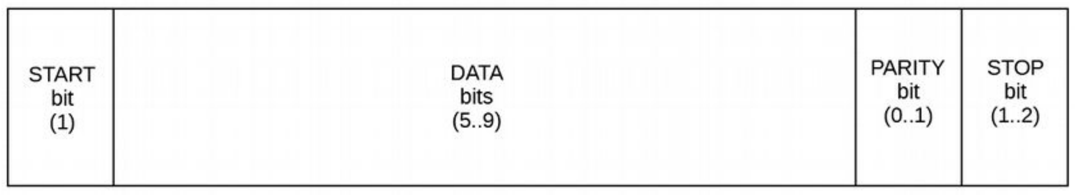


Figure 3 – Structure of an UART frame

For correct operation both the transmitter and the receiver have to interpret the bits on the bus the same way, or in other words “agree” on the same frame structure, and the speed of the communication called the baud rate. The latter is necessary, so the receiver samples a bit the same time the transmitter actually transmits it.

The most common format uses 1 start bit, 8 data bits, and 1 stop bit without a parity bit. The following figure illustrates the structure of a data frame with 1 start bit, 8 data bits, 1 odd parity bit, and 1 stop bit:

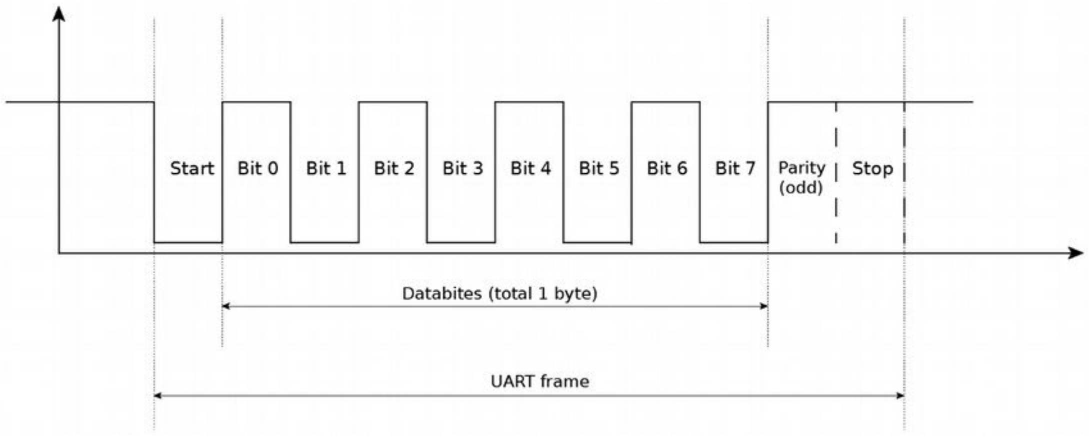


Figure 4 – Bits of an UART message

Parity

In any 8-bit binary number the number of 1 digits is between 0 and 8. An even parity means that for every byte where the number of 1 digits is even (0, 2, 4, 6, or 8) the parity bit stays 0, while if the number of 1 digits is odd (1, 3, 5, or 7) the parity bit becomes 1. Generally, in case of an even parity, the parity bit will be such, so that the combined number of “1” digits in the data bits and the parity bit will be even. When using an odd parity, the reverse is true, so the parity bit will become 1 to make the combined number of “1” digits in the data bits and the parity bit odd.

Suppose that the received data is 01100101. The even parity of this message is 0, because there are 4 digits of 1, and the odd parity is 1, because 4 is not an odd number.

The parity bit is used for error detection. The bit is generated on both transmitter and receiver side, and if the receiver gets a different result compared to what is included in the message, it signals a receive error. With this method we can detect a fault where an odd number of bits are received incorrectly. If you think about what happens if two bits get flipped during a transmission error, you can easily see that the parity will remain the same, so the error goes undetected. If a third bit is also flipped, the error appears again. Typically, parity bits are designed to detect single bit errors. There are ways of course to detect other errors with a greater reliability, but that is not the job of the UART physical layer.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Baud rate

Baud rate determines the communication speed, its unit of measurement is symbols per second. The baud rate and the bitrate can differ in some cases, because a symbol can have more than just two states, so it can carry more information than a single bit. For example, the letters of the alphabet can be symbols, and there are 26 letters in the English alphabet, so such a symbol can have 26 different states. To encode 26 different symbols in binary we need 5 bits. Here a baud rate of 1 symbol / second would equal to a bitrate of 6 bits/s (bps). In UART communication symbols are bits, so the baud rate and the bitrate is the same.

$$1 \text{ baud} \;=\; 1 \text{ symbol} / \text{second} \;=\; 1 \text{ bit/s}$$

Baud rates cannot be arbitrary but have standard values in UART communication starting from 2400 bits/s up to 1 Mbit/s if the devices are fast enough. A few usual values are: 2400 bps, 4800 bps, 9600 bps, 11400 bps, 19200 bps, and 115200 bps. You can see that the values are approximately 1.5-2 times bigger than the previous ones. A full list with all the supported values can be found in the datasheet of ATmega16A on page 165.

USB

USB stand for Universal Serial Bus, and it is the most common interface on computers today. It was created to unify the way peripherals were connected to a computer and to replace outdates standards. Today a very large percentage of peripherals are connected through USB, such as keyboards, mice, printers phones, etc.

The usual topology consists of a host and a client, but with a hub one host can serve multiple clients. The host is usually a PC, and the client is usually a device such as a mouse. It has a relatively high data rate of 480 Mbit/s when using USB 2.0, and an even higher 5, or 10 Gbit/s data rate when using USB 3.0 or 3.1.

Development board and FTDI FT232R

To illustrate the examples, I use a complex hardware consisting of the following components:

- Power supply with a 12 V output
- Breadboard
- ATmega16A microcontroller (MCU), seated on the breadboard
- FTDI adapter, which contains voltage converter for 5V supply and FT232R IC. The latter makes communication between UART and USB possible.

The following block diagram illustrates the layout:

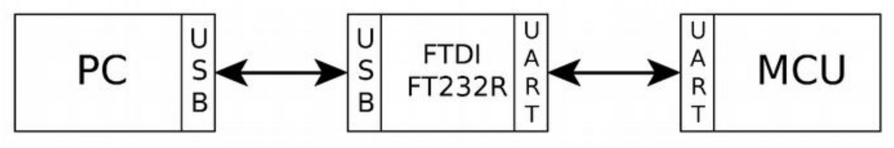


Figure 5 – Communication between the computer and microcontroller

Erasmus+
This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The schematic using FTDI chip is shown in the following figure:

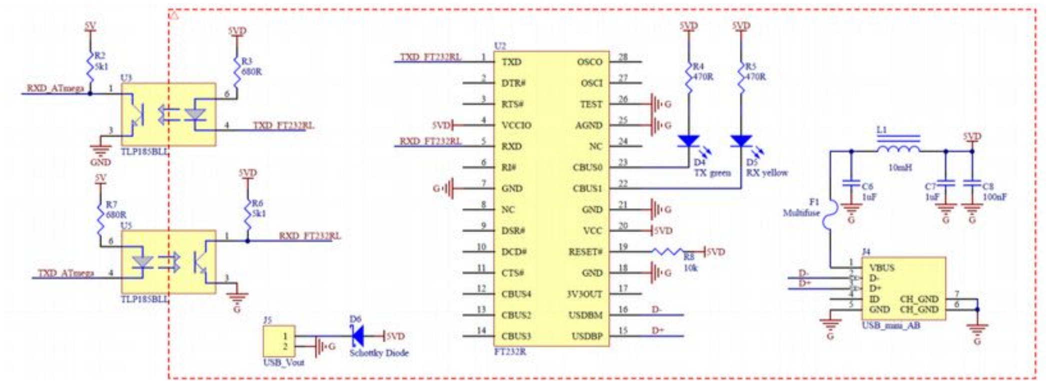


Figure 6 – Schematic of FTDI adapter section

From right to left:

- USB_mini_AB: D+ and D- are two data lines. VBUS is the USB 5V power supply. GND is the ground.
- F1: Resetable fuse with a positive temperature coefficient.
- C6, C7, C8, L1: Bypass capacitors and filters for stability of the supply voltage.
- FT232R: An integrated circuit for the USB-UART conversion.
- TLP185BLL: Optocoupler. Its task is to implement the galvanic isolation between the microcontroller and the USB. On one hand, it is necessary because the microcontroller and the FTDI IC do not share a common supply voltage. On the other hand, this way any noise via USB damaging the microcontroller or the PC can be avoided.

The MCU part and the power supply part of the schematic are shown in the following figure:

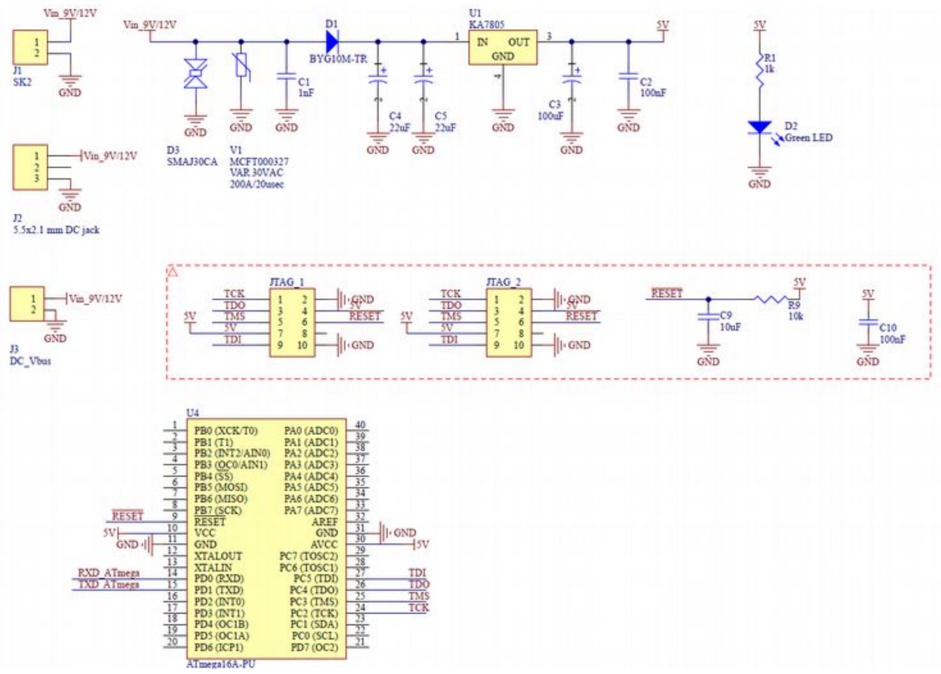


Figure 7 – Schematic part using the microcontroller and its power supply

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

At the top of the schematic, there is the power supply circuit built around the 7805 linear voltage regulator. Below, the connectors are shown in the boxed part. At the bottom, the connection of the microcontroller itself.

The entire hardware unit is placed on a printed circuit board (PCB).

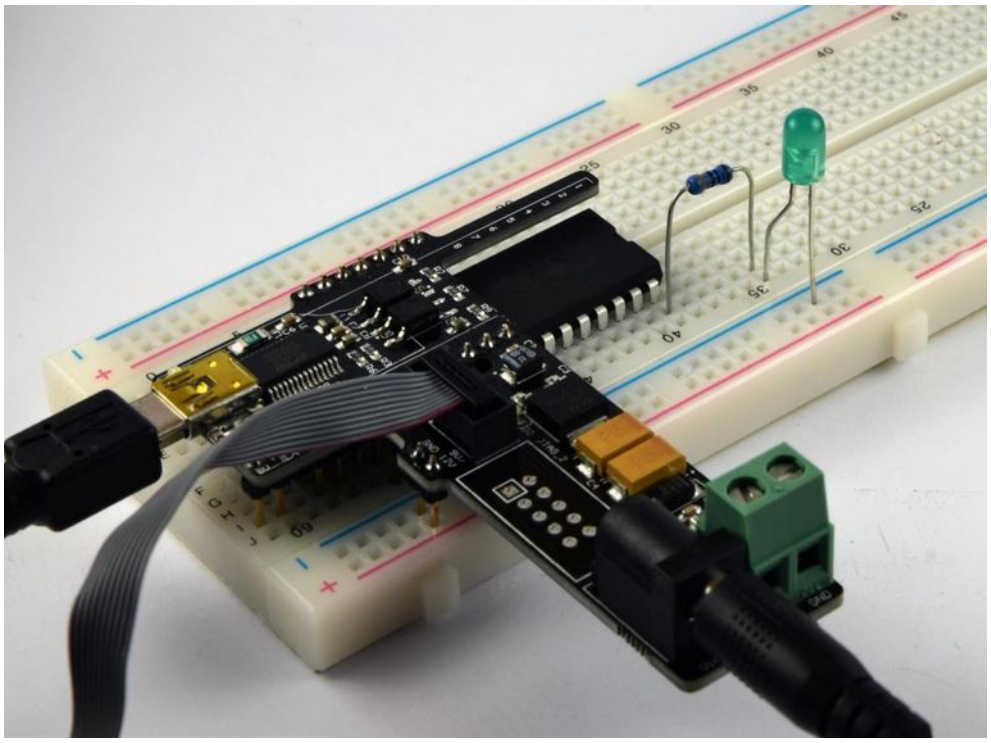


Figure 8 – Adapter PCB, microcontroller and USB connection

From now on “FTDI adapter” means the black PCB that can be connected to the breadboard over the microcontroller.

INSTALLING THE FTDI DRIVER

PREREQUISITES

Installing the driver under windows 7 or later does not require any special preparation. When the USB-UART converter chip is first connected to the PC, the correct driver will be downloaded and installed automatically. The required steps are the following:

1. Connect the FTDI adapter to the breadboard.
2. Power the circuit using an external power supply through the adapter. This can be done either with a compatible power supply through the coaxial power jack connector (the inner pin is the positive, and the outer pin is the negative) or through the terminal block. In both cases, the input voltage should be between 7-25 V. The rated output current of the power supply should be at least 500 mA.
3. Connect the PC and the FTDI Adapter with a USB 2.0 compatible cable

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Installation process

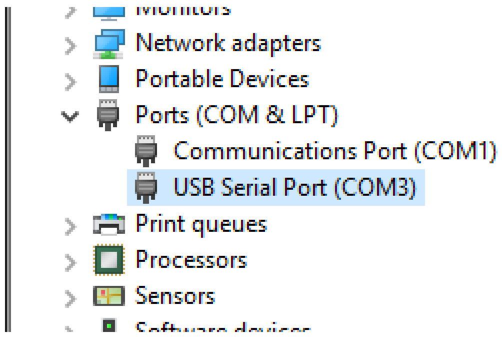
If we have done everything well so far, the driver will be installed automatically.

Verifying the installation

Once the installation is complete, we should check the followings in our computer's device manager:

1. A new virtual COM port has been created
2. The FTDI chip is recognized by the PC as "USB serial converter"

The device manager should show this:

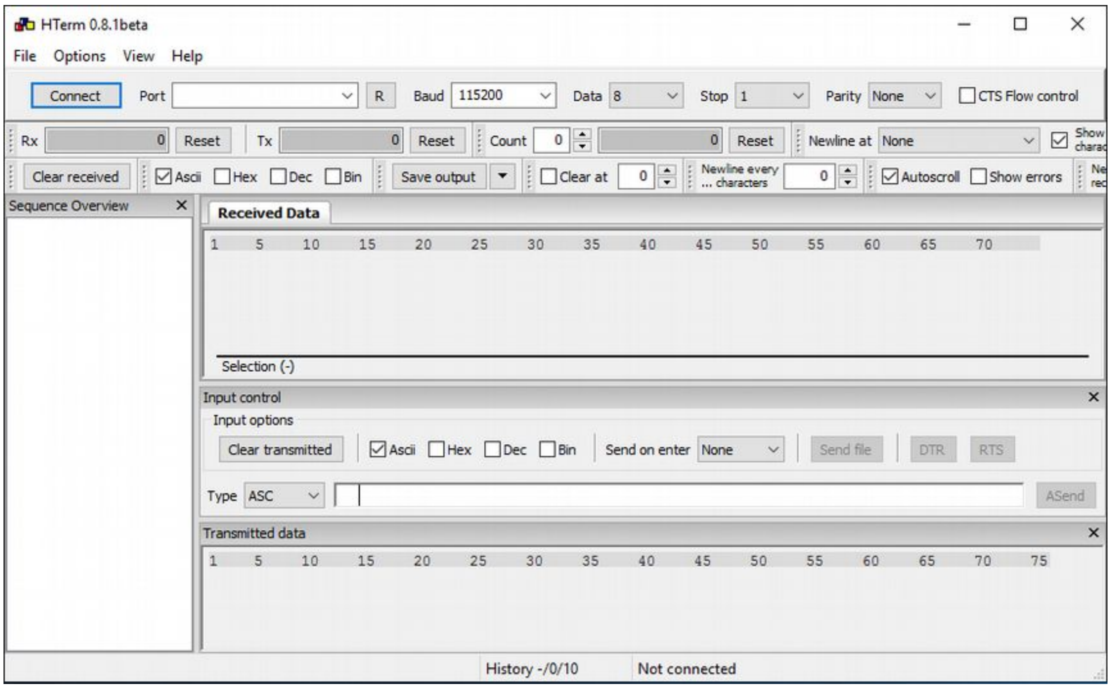


Network adapters
Portable Devices
Ports (COM & LPT)
Communications Port (COM1)
USB Serial Port (COM3)
Print queues
Processors
Sensors

Figure 9 – COM ports in the device manager

HTERM

We have several options for sending and receiving data on the PC side. One of them is the terminal emulator software called HTERM. The application is capable of sending and receiving data over a serial port. The user interface looks like this:



HTerm 0.8.1beta

File Options View Help

Connect | Port | R | Baud 115200 | Data 8 | Stop 1 | Parity None | CTS Flow control

Rx 0 Reset | Tx 0 Reset | Count 0 | 0 Reset | Newline at None | Show charac

Clear received | ☑ Ascii ☐ Hex ☐ Dec ☐ Bin | Save output | ☐ Clear at 0 | Newline every ... characters 0 | ☑ Autoscroll ☐ Show errors

Sequence Overview

Received Data

Selection (-)

Input control

Input options

Clear transmitted | ☑ Ascii ☐ Hex ☐ Dec ☐ Bin | Send on enter None | Send file | DTR | RTS

Type ASC | ASend

Transmitted data

History -/0/10 | Not connected

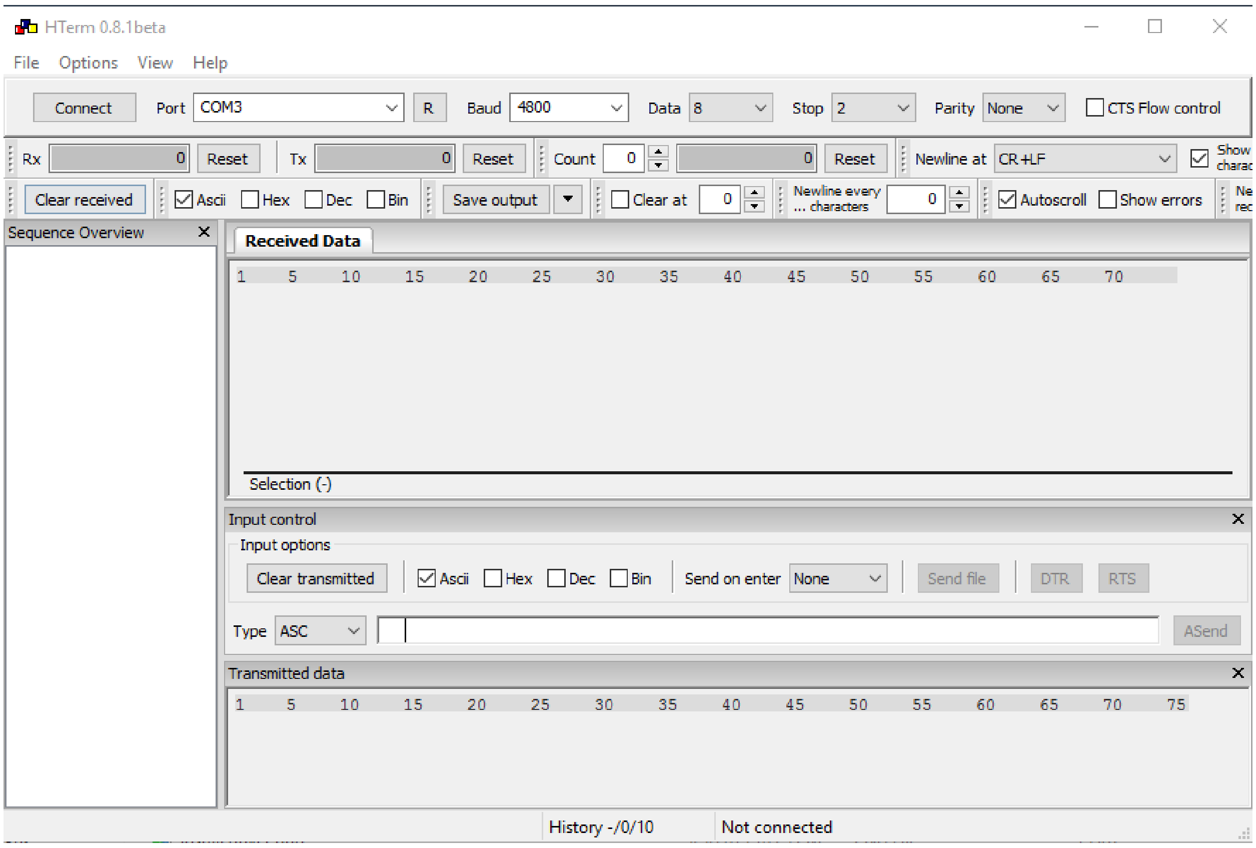
Figure 10 – HTerm main screen

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Using it is very simple. The appropriate virtual COM port has to be selected at the top under “Port”. As shown in the “Verifying the Installation” section, we can find out the port number assigned to the FTDI chip in the device manager! COM3 has been assigned for me, so I will use that in the examples. This number may vary from computer to computer, so you should check it for yourself!

Set the baud rate to 4800. At this speed the communication will still be error-free. At higher speeds, due to the isolation of data lines by optocouplers, the signals would become so distorted that they could not be decoded without error. The 8 data bits remain the default setting, the number of stop bits should be two. I don't use parity, so it stays on the default "None".

The "Newline at" field may also be interesting. I set this value to CR+LF, which is used on windows systems, we will talk more about its meaning later. The point is, if a line feed character arrives, HTerm also displays the text by inserting a line feed. Let's see how the settings should look:


HTerm 0.8.1beta

Port COM3 Baud 4800 Data 8 Stop 2 Parity None ☐ CTS Flow control

Rx 0 Tx 0 Count 0 0 Newline at CR+LF ☑ Show charac

☑ Ascii ☐ Hex ☐ Dec ☐ Bin ☐ Clear at 0 Newline every ... characters 0 ☑ Autoscroll ☐ Show errors

Input options: ☑ Ascii ☐ Hex ☐ Dec ☐ Bin Send on enter None

Type ASC

History -/0/10 Not connected

Figure 11 – HTerm after setup

Before pressing connect and start communicating with our microcontroller we need to write a program for the microcontroller that implements communication on that side. In the next section we will do just that.

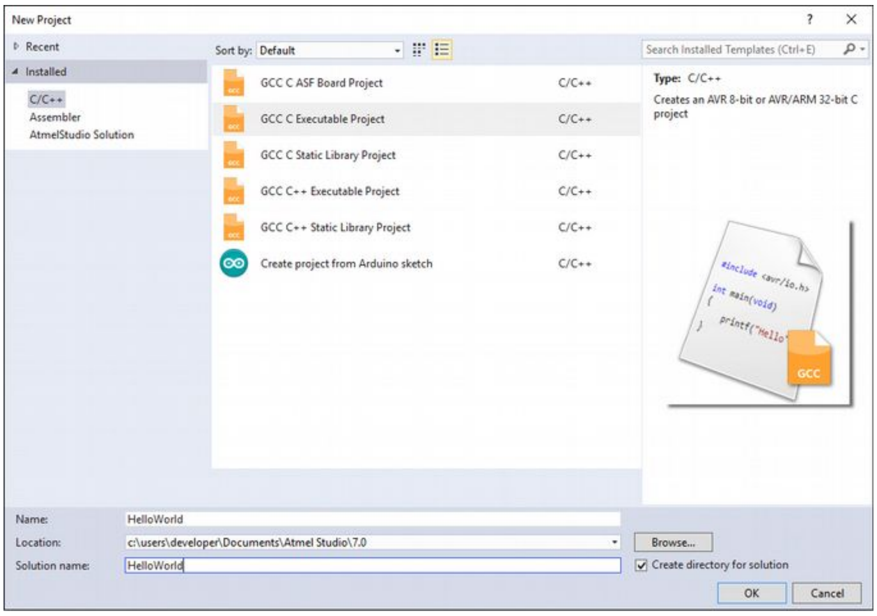
Erasmus+
This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Hello World!

By “Hello World!”, we mean a program that does nothing but prints “Hello World!” First we set up the ATmega16A microcontroller or "initialize" it with technical jargon and then we send out a string on the serial port: "Hello World!"

Creating the project

First, create a new project in Atmel Studio called HelloWorld:


New Project

Installed: C/C++ | Sort by: Default

Template	Language
GCC C ASF Board Project	C/C++
GCC C Executable Project	C/C++
GCC C Static Library Project	C/C++
GCC C++ Executable Project	C/C++
GCC C++ Static Library Project	C/C++
Create project from Arduino sketch	C/C++

Type: C/C++

Creates an AVR 8-bit or AVR/ARM 32-bit C project

Name: HelloWorld

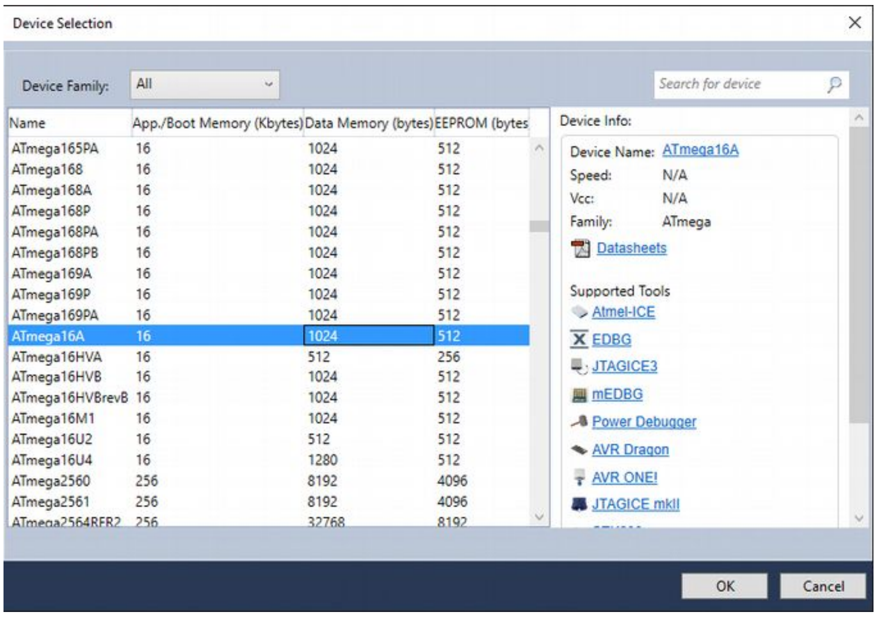
Location: c:\users\developer\Documents\Atmel Studio\7.0

Solution name: HelloWorld

☑ Create directory for solution

Figure 12 – New project wizard of Atmel Studio

At device selection, choose ATmega16A.


Device Selection

Device Family: All

Name	App./Boot Memory (Kbytes)	Data Memory (bytes)	EEPROM (bytes
ATmega165PA	16	1024	512
ATmega168	16	1024	512
ATmega168A	16	1024	512
ATmega168P	16	1024	512
ATmega168PA	16	1024	512
ATmega168PB	16	1024	512
ATmega169A	16	1024	512
ATmega169P	16	1024	512
ATmega169PA	16	1024	512
ATmega16A	16	1024	512
ATmega16HVA	16	512	256
ATmega16HVB	16	1024	512
ATmega16HVBrevB	16	1024	512
ATmega16M1	16	1024	512
ATmega16U2	16	512	512
ATmega16U4	16	1280	512
ATmega2560	256	8192	4096
ATmega2561	256	8192	4096
ATmega2564RFR2	256	32768	8192

Device Info:

Device Name: ATmega16A

Speed: N/A

Vcc: N/A

Family: ATmega

Datasheets

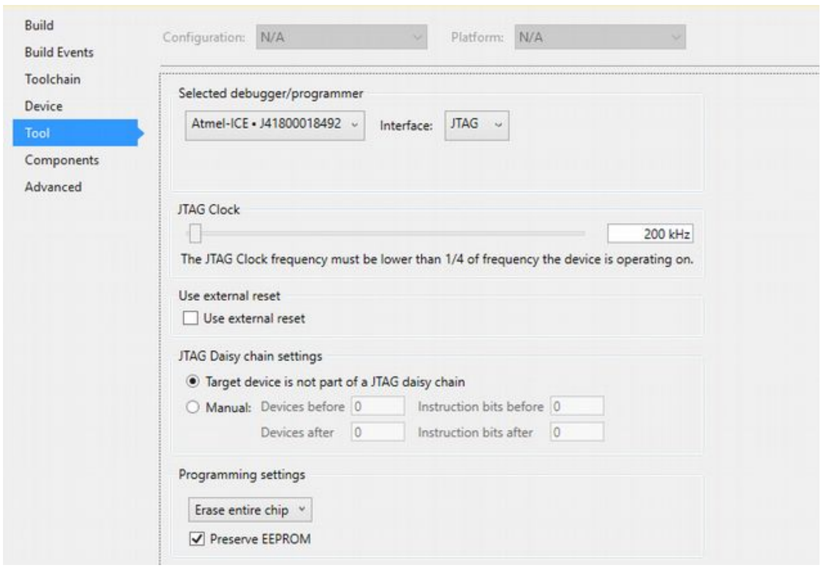
Supported Tools: Atmel-ICE, EDBG, JTAGICE3, mEDBG, Power Debugger, AVR Dragon, AVR ONE!, JTAGICE mkII

Figure 13 – Device selection in Atmel Studio

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Finally, press OK. Then, set the programmer type under project properties (right click on the project and select "properties"). I use the Atmel-ICE debugger, so this will be included in the example. Let's see the correct settings for the Atmel-ICE debugger:


Build
Build Events
Toolchain
Device
Tool
Components
Advanced

Configuration: N/A Platform: N/A

Selected debugger/programmer
Atmel-ICE • J41800018492 Interface: JTAG

JTAG Clock
200 kHz
The JTAG Clock frequency must be lower than 1/4 of frequency the device is operating on.

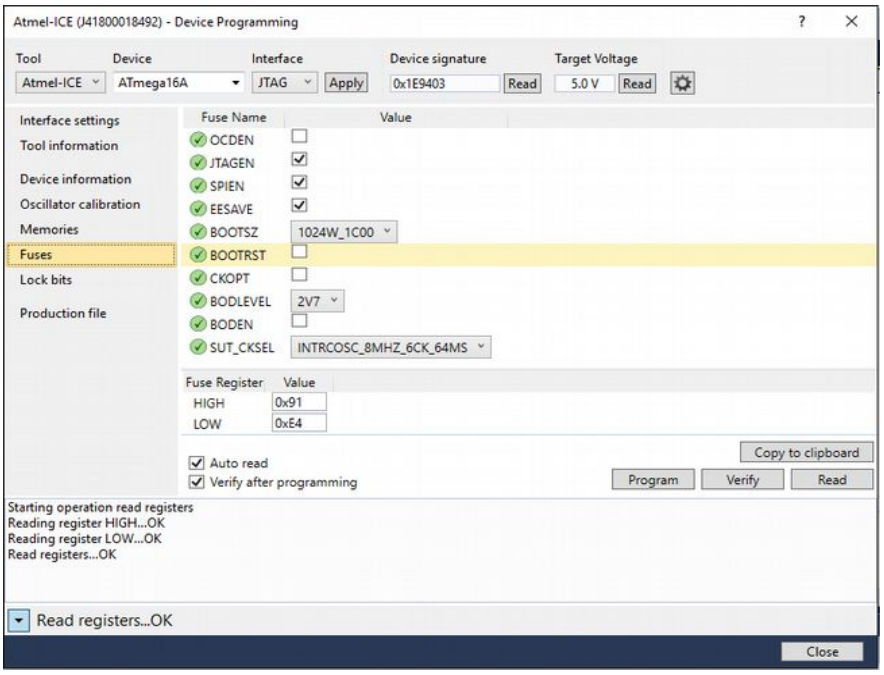
Use external reset
☐ Use external reset

JTAG Daisy chain settings
● Target device is not part of a JTAG daisy chain
○ Manual: Devices before 0 Instruction bits before 0
Devices after 0 Instruction bits after 0

Programming settings
Erase entire chip
☑ Preserve EEPROM

Figure 14 –Debugger options in Atmel Studio

When that is done, it is good to check a few important settings that the microcontroller stores in the form of fuse bits. These settings are permanent and should be handled with care as you can adjust values such as processor frequency or enable individual debuggers. Open the Fuses window under Tools-> Device Programming:


Atmel-ICE (J41800018492) - Device Programming

Tool	Device	Interface	Device signature	Target Voltage
Atmel-ICE	ATmega16A	JTAG	0x1E9403	5.0 V

Interface settings
Tool information
Device information
Oscillator calibration
Memories
Fuses
Lock bits
Production file

Fuse Name	Value
OCDEN	☐
JTAGEN	☑
SPIEN	☑
EESAVE	☑
BOOTSZ	1024W_1C00
BOOTRST	☐
CKOPT	☐
BODLEVEL	2V7
BODEN	☐
SUT_CKSEL	INTRCOSC_8MHZ_6CK_64MS

Fuse Register	Value
HIGH	0x91
LOW	0xE4

☑ Auto read
☑ Verify after programming

```
Starting operation read registers
Reading register HIGH...OK
Reading register LOW...OK
Read registers...OK
```

Read registers...OK

Figure 15 – Device programming, Fuses

If the settings are the same as in the picture above, we are happy. If not, adjust them and press the program button to apply.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

INITIALIZING PORTS

After the settings, you can start writing the program.

To get started, we get a minimal, almost empty main.c file with the following content generated by the development environment:

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

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

This chapter already assumes that the reader has a basic knowledge of the C language. The code snippet must be clear.

The source code above does nothing, so first we need to add the initialization part. When a microcontroller starts, it has an initial state. Therefore, the first step in any embedded program is initialization. This is a step, where we set up the necessary parameters, and prepare the ground for running our program. In case of AVR microcontrollers, initially all port pins are configured as inputs, and are logically low, additionally peripherals are disabled by default.

The unused pins should still be set as inputs and to a logical high level. This prevents them from "floating". For more information, see paragraph 12.2.4 of datasheet ATmega16A. Connect a status LED to the very first pin on port A. When the device is in the initialized state, the LED will light up. Let's see the code:

```
int main(void)
{
    /* Enabling PORTA pull-up resistors on the inputs, PA0 output should have logic
     * 1 level (5V), this pin is the status LED, other pins configured as input */

    PORTA = 0xFF;
    DDRA = 0x01;

    /* Enabling PORTB pull-up resistors, and all pins configured as inputs */
    PORTB = 0xFF;
    DDRB = 0x00;

    /* Enabling PORTC pull-up resistors, and all pins configured as inputs */
    PORTC = 0xFF;
    DDRC = 0x00;

    /* Enabling PORTD pull-up resistors, and all pins configured as inputs */
    PORTD = 0xFF;
    DDRD = 0x00;

    while (1)
    {
    }
}
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

If the above code is executed with the "Start Without Debugging" button or the "Ctrl+Alt+F5" key combination, then the status LED should light up. We are pleased to note that the initialization of the I/O pins was successful.

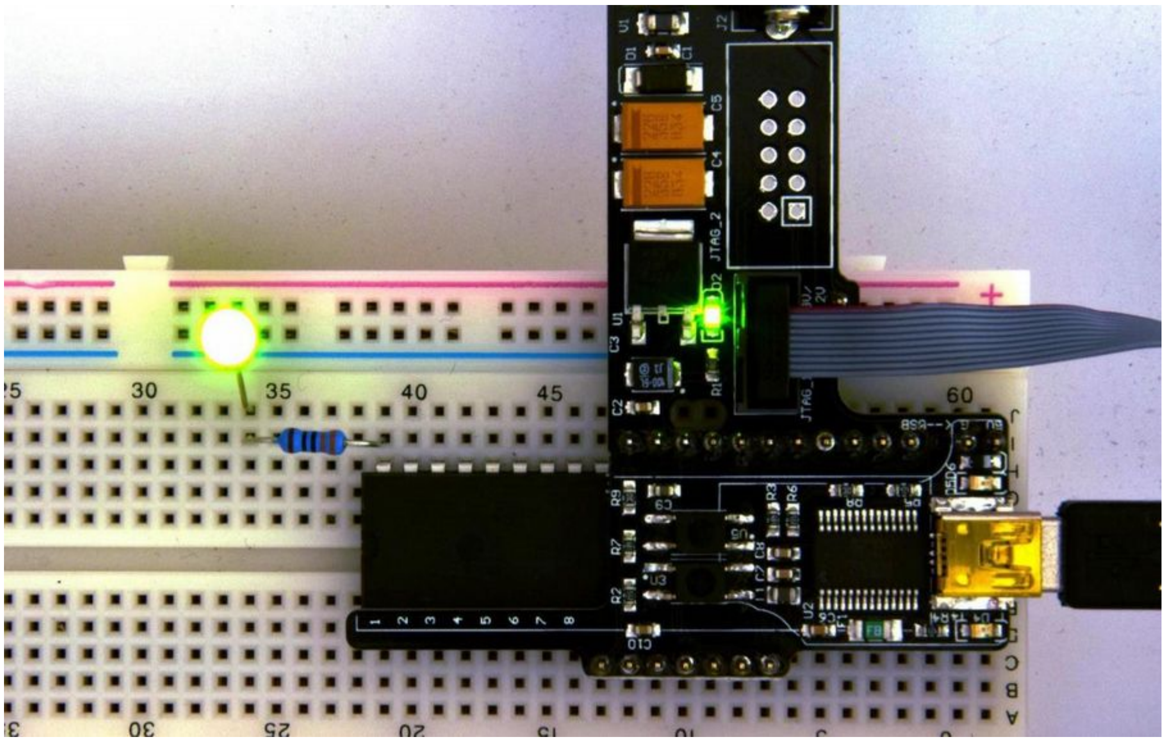


Figure 16 – Executing sample code on the microcontroller

INITIALIZING THE UART

The next step is to configure the UART module for 8 data bits and 2 stop bits as we mentioned in the HTerm section. Fortunately, the microcontroller manufacturer also provides sample codes in the ATmega16A datasheet for each functionality. In the USART chapter on page 146, the following code is provided for initialization:

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

All we have to do is copy it into the source file of our HelloWorld program before the main function and call the `USART_Init()` function with the correct value after I/O init. The correct value can be seen on page 165 of the same datasheet.

The UBRR value for 4800 baud with an 8 MHz clock frequency is 103. The UBRRH and UBRRL registers are responsible for adjusting the baud rate (see ATmega16A datasheet 19.11.5), changing the UCSRB

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

(see ATmega16A datasheet 19.11.3) we can enable the transmitter and receiver, and UCSRC (see ATmega16A datasheet 19.11.4) is responsible for setting the frame format. The state of our code after inserting:

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

void UARTInit( unsigned int ubrr)
{
    /* Baud rate settings */
    UBRRH = (unsigned char)(ubrr>>8);
    UBRRL = (unsigned char)ubrr;
    /* Enabling transmitter and receiver */
    UCSRB = (1<<RXEN)|(1<<TXEN);
    /* Frame format settings: 8data, 2stop bits */
    UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}

int main(void)
{
    /* Enabling PORTA pull-up resistors on the inputs
     * PA0 output should have logic 1 level (5V),
     * on this pin is the status LED
     * The other pins should be configured as inputs */
    PORTA = 0xFF;
    DDRA = 0x01;

    /* Enabling PORTB pull-up resistors, and all pins configured as inputs */
    PORTB = 0xFF;
    DDRB = 0x00;

    /* Enabling PORTC pull-up resistors, and all pins configured as inputs */
    PORTC = 0xFF;
    DDRC = 0x00;

    /* Enabling PORTD pull-up resistors, and all pins configured as inputs */
    PORTD = 0xFF;
    DDRD = 0x00;

    UARTInit(103);

    while(1)
    {
        /* Infinite loop */
    }
    /* We never get here, but for completeness we put a return at the end */
    return 0;
}
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

TRANSMITTING OVER UART

The UART has already been initialized but we cannot send any messages yet. Fortunately, the datasheet also contains sample code, which explains how to send something. This can be found on page 147 under 19.6.1:

```
void UARTTransmit(unsigned char data )
{
      /* We wait until the buffer of the transmitter gets empty. */
      while ( !( UCSRA & (1<<UDRE)) )
      {
             /* Empty loop. We do nothing else, just wait. */
      }
      /* We put the data into the buffer.
       * The sending is solved by the microcontroller’s HW. */
      UDR = data;
}
```

The code waits for the transmitter to complete any ongoing transmissions and then it puts the data into the UDR register, from where it is sent out by the UART hardware module with the parameters, which are already set.

Data refers to an 8-bit, i.e. 1-byte value. A value from 0 to 255 can be put in an unsigned byte. Let's send the value 42 to the PC! The code will be changed as follows:

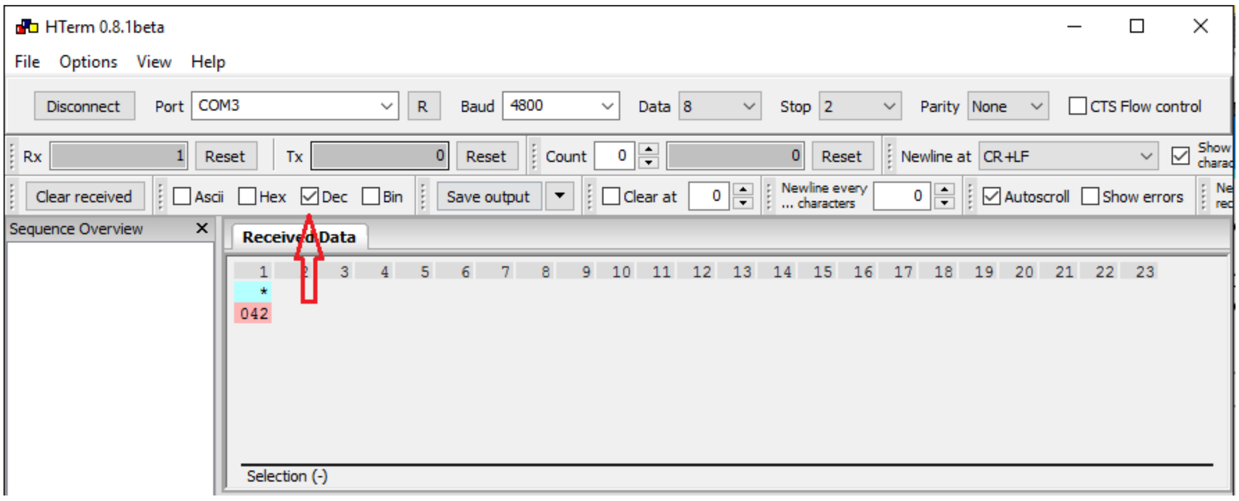
```
#include <avr/io.h>

void UARTInit( unsigned int ubrr)
{
    /* Baud rate settings */
    UBRRH = (unsigned char)(ubrr>>8);
    UBRRL = (unsigned char)ubrr;
    /* Enabling transmitter and receiver */
    UCSRB = (1<<RXEN)|(1<<TXEN);
    /* Frame format settings: 8data, 2stop bit */
    UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}


void UARTTransmit( unsigned char data )
{
    /* We wait until the buffer of the transmitter gets empty */
    while ( !( UCSRA & (1<<UDRE)) )
      {
             /* Empty loop. We do nothing else, just wait. */
      }
    /* We put the data into the buffer.
     * The sending is solved by the microcontroller’s HW. */
    UDR = data;
}
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

```c
int main(void)
{
    /* Enabling PORTA pull-up resistors on the inputs
     * PA0 output should have logic 1 level (5V),
     * on this pin is the status LED
     * The other pins should be configured as inputs */
    PORTA = 0xFF;
    DDRA = 0x01;

    /* Enabling PORTB pull-up resistors, and all pins configured as inputs */
    PORTB = 0xFF;
    DDRB = 0x00;

    /* Enabling PORTC pull-up resistors, and all pins configured as inputs */
    PORTC = 0xFF;
    DDRC = 0x00;

    /* Enabling PORTD pull-up resistors, and all pins configured as inputs */
    PORTD = 0xFF;
    DDRD = 0x00;

    UARTInit(103);

    UARTTransmit(42);

    while(1)
    {
        /* Infinite loop */
    }
   /* We never get here, but for completeness we put a return at the end */
    return 0;
}
```

Set up HTerm to wait for the data in decimal format and press "connect". Let's run the code on the microcontroller. The result, if we did everything well, speaks for itself:



HTerm 0.8.1beta

Port COM3, Baud 4800, Data 8, Stop 2, Parity None, CTS Flow control ☐

Rx 1, Tx 0, Count 0, Newline at CR+LF

Ascii ☐, Hex ☐, Dec ✓, Bin ☐, Clear at 0, Newline every ... characters 0, Autoscroll ✓, Show errors ☐

Received Data

```
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22  23
  *
042
```

Selection (-)

Figure 17 – Receiving data in HTerm

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

SENDING A CHARACTER

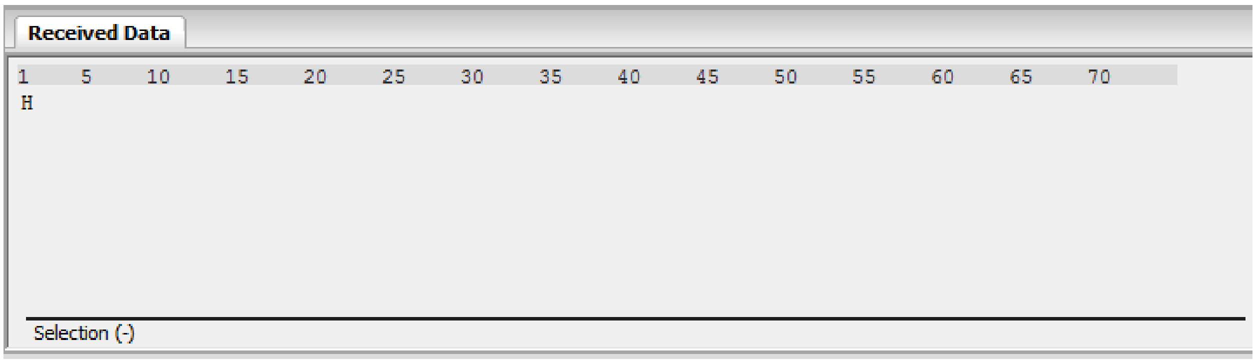
Let’s move on and send the character capital H to the PC.

There are several ways to encode characters. The simplest of these is the ASCII character table. It encodes the letters of the English alphabet in 1 byte and a lot of other punctuation marks and characters as well.

The character table is available on the Internet in many forms. I wouldn't insert it here now. Without giving a complete list, it is enough to know that the code of the capital H is 72. Let's see how our code changes:

```
...
UARTTransmit(72); /* It is equivalent with this: UARTTransmit(‘H’); */
...
```

We need to change the encoding of incoming data in HTerm from decimal to ASCII and delete any data we have received so far. After running our modified program, we should get the following:



```
1    5    10   15   20   25   30   35   40   45   50   55   60   65   70
H
```

Figure 18 – Receiving a character in HTerm

So, indeed we sent the letter H.

SENDING STRINGS

The next thing to do is, is to send the entire "Hello World!" string. To do this, we will need a function that receives a C string, i.e. an array of characters, as a parameter and calls the `USART_Transmit()` for each character one by one. We should name the function `USART_TransmitString()`. It will receive a char type array and will not have a return value, so the declaration looks like this:

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

In C, character arrays are so-called “null terminated strings”. This means that the last element is the NULL character, which has an ASCII code of 0, but it is written as ’\0’.

Let's look at the C representation of the string "Hello World!":

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

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The size of the character array will be 12, but the size of the array itself will be 13! In C, it is very important that arrays are indexed from 0!

Let's see the implementation of the function:

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

Using a for loop, we move through the elements of the array until we reach the end, represented by the null character ('\ 0').

How to Find out the Length of a String

The length of a C string can also be determined by the `strlen` function in the standard library, so our code would change as follows:

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

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

Or we could interpret the received array as a pointer and then write:

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

These solutions are equally good, but they differ in speed. Let's analyze both of them!

The third is the fastest solution because we do not index an array. Because if I write that `string[1]`, it is the same as `*(string+1)`. Semantically, an array is nothing more, but a pointer to element 0 of an array. So, by increasing the value of the pointer, we move through the array. This saves us 12 additions compared to the first solution.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The slowest is the second solution, where we use `strlen` to get the size of the array. This is slower than the first one, because the `strlen` inside is nothing more than a for loop that goes through the array counting the characters until it finds the null character.

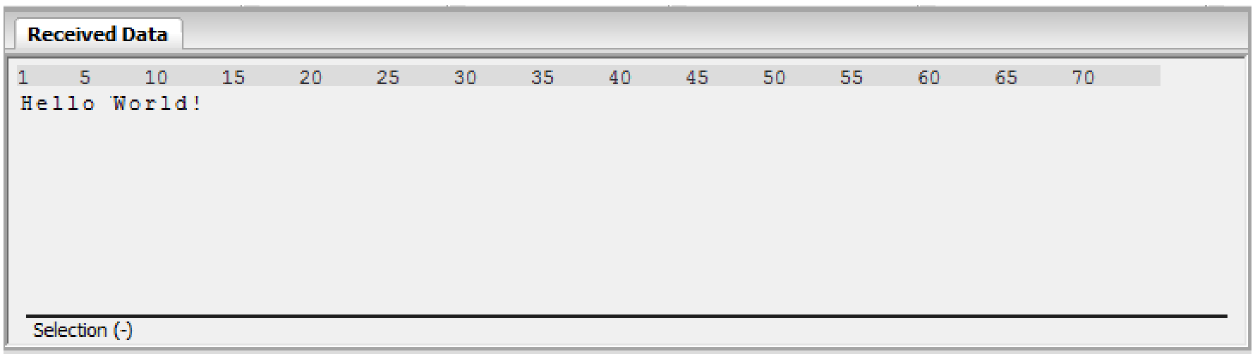
The slowest would be if we would write this:

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

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

What's the problem with this solution? Well, that's a for loop embedded in a for loop! Even worse, because it's a function call as well, which means that it has a cost! The `strlen` runs not once, but as many times as the length of the string! For arrays with small size this is not a problem, but for larger arrays we are already spending a lot of resources on unnecessary iterations!

Insert our newly implemented function into the source file and call it from the main function with the "Hello World!" argument! The source file in its final form is available on the website.

If we did everything well, the code should run, and the following should appear in HTerm:



```
Hello World!
```

Figure 19 – Hello World in HTerm

In the next section, I will implement a more complex functionality with more sophisticated sending and receiving modes.

LED SCANNER LIGHT STRIP

In the previous section we were sending data from the microcontroller to the PC. Now we are going to control 8 LEDs from the PC.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

HARDWARE

I connected the anodes of the LEDs to one pin each on the same port and the cathodes of the LEDs to ground via a resistor.

The LED is a non-linear component that can only be driven by a current source. In this example, the resistor acts as a current source by limiting the current. The voltage drop on an LED is around 1.7V when open meaning, that current is flowing through it. At 1 mA, it is just barely lit and at 10 mA it lights with full brightness. The supply voltage is 5 V and the voltage drop is 1.7 V, which means that 10 mA should cause a 3.3 V drop on the resistor. So, the resistance can be calculated by the well-known formula R=U/I: 330 Ohm.

If only one resistor is used for all the LEDs, only one LED can be lit at a time, because otherwise, the brightness would decrease depending on the number of LEDs on. The more LEDs were lit at a time, the less current would flow through each LED. The circuit looks like this:

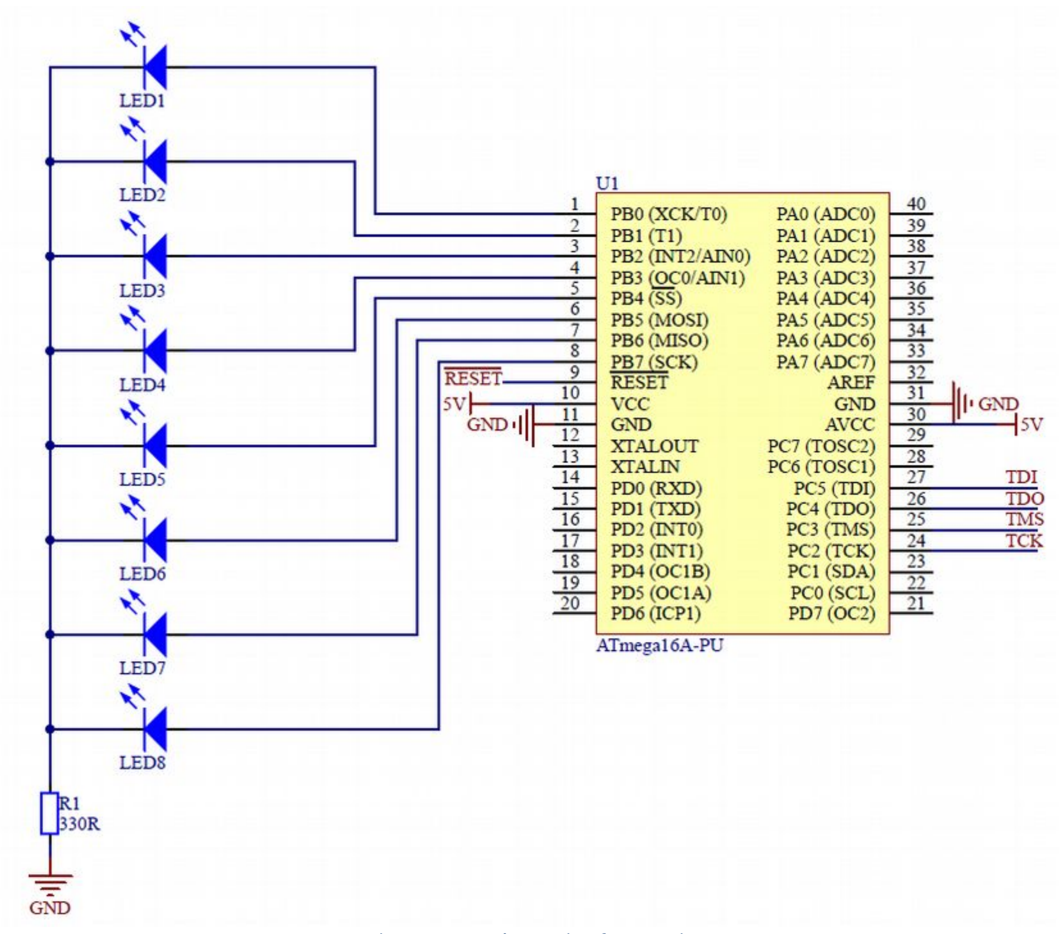


Figure 20 – Schematic of LED strip

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

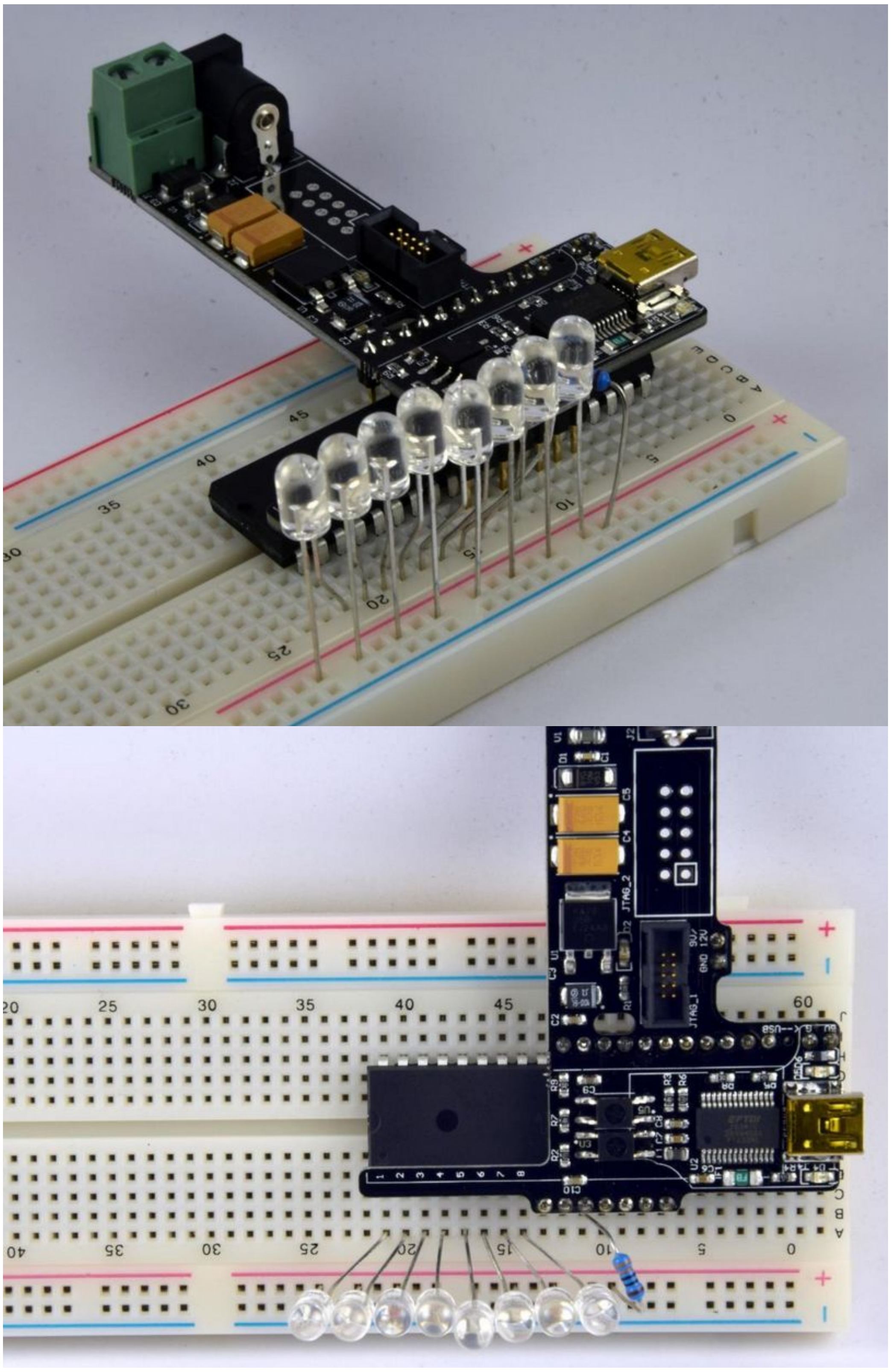


Figure 21 –-The assembled circuit of LED light strip

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

IMPLEMENTATION

The point of a scanner LED strip is to move an illuminating LED to the left or right with the appropriate command. If the commands are sent often enough in succession, we get an LED running light.

An attentive reader may have noticed that all the LEDs are connected to port B. This is no coincidence. The reason for this is fairly simple: it is easier to write a program if you only need to control one port.

In the next section, I will discuss the structure and operation of the code. I also wrote down my thoughts while putting together this little program. In the C language, in which I have coded so far, the most important part is the data and the operations executed on it. A bit simplified: data = variable, operation = function.

I followed the bottom-up way of software development, that is, I was building code from the bottom. This kind of thinking can be formulated as follows: I have some data, on which I need to execute some operations. So, I will need a function that does … and another one, that does …

When all the components are ready, all you have to do is "putting them together". Let's see the "data" and the "must have" operations:

- Data:
  - PORTB: an 8-bit variable representing port B
  - UCSRA: Output or input 8-bit data belonging to UART
- Operations, which means, we must have a function that …
  - Can do the Initialization
  - Moves the light to the left
  - Moves the light to the right
  - Can receive data
  - Can send data

Initialization

We have to set 2 things after start:

1. We have to configure the I/O pins as output to which the LEDs are connected.
2. The UART must be turned on and the correct settings should be applied.

Shifting to the left

The following code snippet, which is also a full function, is designed to implement shifting to the left. If we want the LED on a given pin to be off, we need to write binary 0, and if we want the LED to be on, we need to write binary 1 in the PORTB register to the correct position. There is only one LED on at a time, so all possible combinations on the PORTB are the following, represented in binary, hexadecimal, and decimal numeral system:

0000 0001 = 0x01 = 1

0000 0010 = 0x02 = 2

0000 0100 = 0x04 = 4

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

0000 1000 = 0x08 = 8

0001 0000 = 0x10 = 16

0010 0000 = 0x20 = 32

0100 0000 = 0x40 = 64

1000 0000 = 0x80 = 128

The task of the function is to shift the only one in the number to the left. For this, in language C we can use the binary left shift operator, i.e. "<<". This operator shifts the integer to the left and it brings in zeros from the right. We can use this operator every time except when the state is 1000 0000. This is the leftmost state when the cycle restarts and the next value will be the initial value 0000 0001.

The accomplished task is the following: if it is not the final state, move the light to the left by one. If it is the final state, PORTB should contain the initial value again. In code form, it looks like this:

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

I could have written in this way as well:

```
void RunningLightLeft(void)
{
	if(PORTB != 0x80)
	{
		PORTB <<= 1; // It equals with this: PORTB = PORTB << 1;
	}
	else
	{
		PORTB = 0x01;
	}
}
```

Shifting to the right

Shifting to the right is analogous to shifting to the left, except that here the right final state, i.e. 0x01 should be treated differently. Let's see the code:

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

Receiving data

The USART data register is calledUDR (USART I/O Data Register). We need to read the incoming data from here. We can find out when there is incoming data to be read from another register,   UCSRA (USART Control and Status Register A). More precisely, bit 7 of the UCSRA register (which can be referred to as RXC (RX Complete)) tells us whether data has been received or not. Let's see how this can be implemented in the form of code:

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

```
unsigned char USART_Receive( void )
{
      /* We are waiting for incoming data */
      while ( !(UCSRA & (1<<RXC)) );
      /* We return with the received data */
      return UDR;
}
```

The code above is a sample code copied from page 150 of the ATmega16A datasheet. We can simplify it by knowing that AVR Libc, i.e. the subset of the standard C library released by Atmel, contains a macro with that we can wait until a bit is set: `loop_until_bit_is_set (sfr, bit)`

```
char uart_getchar(FILE *stream) {
      loop_until_bit_is_set(UCSRA, RXC);  /* We are waiting,
                                              * for data to receive. */
      return UDR;
}
```

We can make things even easier by using the FDEV_SETUP_STREAM macro. This allows you to set up a temporary storage to use the C standard `getchar()` function. The buffer will be of `FILE` type. The exact usage of the macro is as follows:

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

All we have to do is redirect stdin to our newly created buffer:

```
stdin = &uart_input;
```

Scanning a character from UART happens as follows:

```
char character = getchar();
```

Sending data

You can also use the UDR register for data transmitting. Transmission is also done per character, just like reception. You cannot write to this register at any time during the transmission, as a previous transmission may be pending, or a byte of data may have been received. The UDRE (Usart Data Register Empty) bit of the UCSRA register indicates if the buffer is free and sending can take place. Formulated as a C code, we get the following:

```
void uart_putchar(char c, FILE *stream) {
      /* If a line feed character is detected, then a \r must be inserted,
       * before in order to make it windows compatible */
      if( c == '\n')
      {
            loop_until_bit_is_set(UCSRA, UDRE);
            UDR = '\r';
      }

      loop_until_bit_is_set(UCSRA, UDRE);
      UDR = c;
}
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The code above needs some explanation. If we already use *stdin*, it is logical to use *stdout* as well. In this way we can send information with `printf()` and `puts()` using UART as usual. To redirect *stdout* we need to use the windows format (\r\n) of the line feed character (\n). Therefore, if we describe the following:

```
puts("This is a whole line with a newline at the end.\n");
```

then at the end of this string there will actually be not only one \n but \r\n. The next thing we need to do is to create a temporary repository here as well, what we can pass to *stdout*:

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

The redirection:

```
stdout = &uart_output;
```

There are two ways to send a series of punctuation marks. Without formatting, use `puts()` or with formatting use `printf()`. For example, let's say we want to send out a string that says "Hello world!". This can be done as follows:

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

If we want to print something more complicated, we need to use `printf()`. Suppose a sensor is connected to the microcontroller that measures the temperature outside. This data is stored in a variable called "temperature" and we want to send out this information using UART. The code is as follows:

```
printf("The temperature is: %d \n",temperature);
```

Newline on different systems

The line feed character is different on different operating systems. Under UNIX/LINUX it is only \n, while under windows it is \r\n. [1]

`Printf` can not only format a number into a given text, but also a string and much more as well. [2]

Main function

All we have to do is put the previous parts together in the main function:

```
/* Macro definitions */
#define  LED_CODE 0x55
#define  LEFT     0x3C
#define  RIGHT    0x3E

int main(void)
{
      /* Buffer variable for the received data */
      char buffer;
      /* Initialization of inputs and outputs */
      IOInit();
      /* Set valid initial value for PORTB */
      PORTB = 0x01;
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

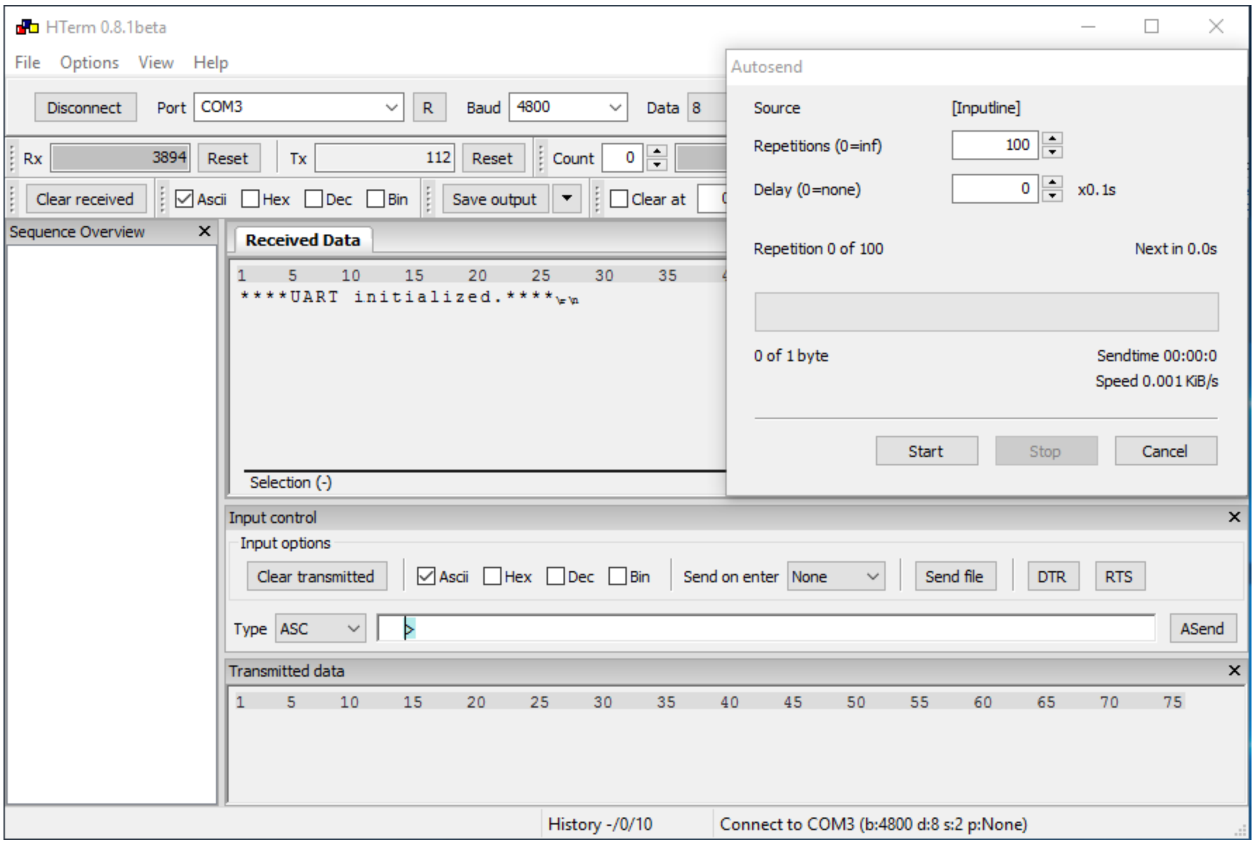
```
        /* Initialization of UART, 103=4800 baud,
         * page 165 of ATmega16A datasheet */
        UARTInit(103);

        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;
        /* Testing */
        puts("****UART initialized.****");
              /* Infinite loop */
        while (1)
        {
              /* I put the received data into the buffer */
              buffer = getchar();
               /* Processing the received data with switch-case structure:
                * The structure allows the variable to be compared to
                * a list of data.
                * Each data value corresponds to one case.
                * /
              switch(buffer)
              {
                    case LED_CODE:
                    {
                          puts("LED flip command has been received");
                          PORTA = ~PORTA;
                          break;
                    }
                    case LEFT:
                    {
                          puts("Light to the left command has been received");
                          RunningLightLeft();
                          break;
                    }
                    case RIGHT:
                    {
                          puts("Light to the right command has been received");
                          RunningLightRight();
                          break;
                    }
                    default:
                    {
                          printf("Unknown command: %c \n", buffer );
                          break;
                    }
              }
        }
        return 0;
}
```

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Usage

The '>' character causes the running light to move to the right and the '<' character to the left. It is also possible to change the state of the status LED. In this case, we need to send the hexadecimal 55. Sending left or right commands many times in quick succession gives you a real running light. This can be achieved with the autosend function of HTerm, where, for example, we can also set repetition (e.g. 100 times) as follows:



Port COM3 Baud 4800 Data 8
Rx 3894 Tx 112 Count 0
☑ Ascii ☐ Hex ☐ Dec ☐ Bin ☐ Clear at

```
****UART initialized.****\r\n
```

Autosend
Source [Inputline]
Repetitions (0=inf) 100
Delay (0=none) 0 x0.1s
Repetition 0 of 100 Next in 0.0s
0 of 1 byte Sendtime 00:00:0
Speed 0.001 KiB/s

☑ Ascii ☐ Hex ☐ Dec ☐ Bin Send on enter None
Type ASC >

History -/0/10 Connect to COM3 (b:4800 d:8 s:2 p:None)

Figure 22 --Sending commands from HTerm

SUMMARY

This chapter of the curriculum introduced you the concept of the UART peripheral and its usage through an example. With the acquired knowledge, many new tasks and challenges can be easily solved. Keep in mind, however, that UART is typically a very simple form of communication, so it has its own limitations. Both in speed and robustness. Before choosing this peripheral for communication purposes, it is worth considering whether it satisfies the requirements, or not.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).