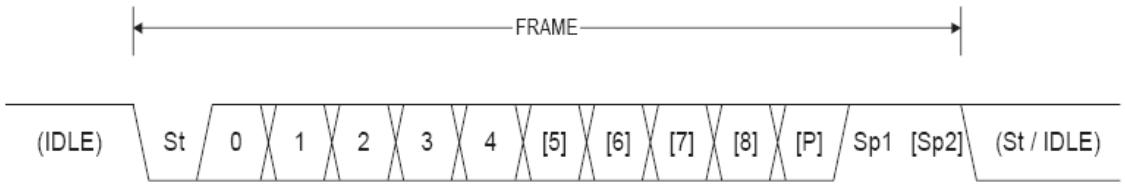
Exercise 2: Serial Communication

1 UART

A *Universal Asynchronous Receiver/Transmitter* (UART) is a piece of hardware that translates between parallel and serial data. UARTs are usually used in conjunction with other communication standards such as *EIA RS-232* (like in our case) to perform serial communication.

The UART takes a byte of data and transmits the individual bits in a sequential fashion, along with some extra information to mark the beginning and end of the message. Such a message is also called a UART *frame*. At the destination, a second UART reassembles the bits into complete bytes. Although a simple UART could also be implemented manually by toggling a bunch of I/O pins at the right frequency, it is much easier to use the built-in UART module of a microcontroller. Such a unit provides register that receive the byte to be transmitted or are set to the value of incoming UART frames. This allows the UART communication to happen in parallel to normal program execution. ATmega168 has one built-in UART module available.

In asynchronous mode, a UART frame consists of the bits depicted in figure 1.



St **Start bit:** Always 1 bit with the opposite polarity of the data line's idle state (i.e., low).
(n) **Data bits:** 5 to 9 bits, least significant bit first.
P **Parity bits:** optional (0 or 1 bit), odd or even parity mode supported.
St **Stop bits:** 1, 1.5, or 2 bits.
IDLE **Idle time:** No transfers on the communication line. An idle line must be high.

Figure 1: UART Frame Format

Some microcontrollers also support synchronous transmission mode (USART), in which the clock is recovered separately from the data stream and no start/stop bits are used. This improves the efficiency of transmission on the channel. ATmega168 support it, but we will only use the asynchronous transmission (normal) mode in this lab course for reasons of simplicity.

In normal mode, the UART settings (number of data bits, parity, number of stop bits) must be agreed between the communicating devices (in our case, between host PC and microcontroller) before communication starts. At the target side, those settings are configured in special registers. Another key parameter that needs to be set before communication is the *baud rate* (transmission speed). There are a set of predefined baud rate values that we can choose from.

Exercise 2.1

a) Which pins serve for receiving data (`RXD`) and transmitting data (`TXD`) in ATmega168, respectively? Connect the two-wire cable so that the corresponding pins and the UART connector (`RS232 SPARE`) are connected.

b) What is the difference between a baud and a bit? Do some research on the Internet.

c) Start the *HyperTerminal* application (*Accessories* → *Communications* → *HyperTerminal* in the start menu). Enter an arbitrary profile name and select a valid COM port. Have a look at the available settings. List the standard baud rates that are available.

d) Read the ATmega168 manual to find out which register is used to configure the baud rate of the target. Which of the baud rates from exercise **2.1 c)** are theoretically supported by ATmega168 at a clock speed of 16 MHz? Read the ATmega168 manual to find the respective formula and *calculate* the possible baud rates.

e) Which value should we set UBRR to for a given baud rate? Write down the formula for asynchronous normal mode. Which bits of the UBRR value have to be put into which hardware register?

f) List the registers and the corresponding bit fields that are relevant for storing other UART related settings and data in ATmega168.

2 Polling-based UART communication

Programming a microcontroller without a debugger is a painful task. In this section, we will try to build a UART based debugger. The idea is to forward the standard output channel (written to by C functions like `putchar()`, `puts()`, `printf()`, etc.) via a UART connection to the PC and use a terminal application to visualize the output.

Exercise 2.2

a) Write a function `uart_init()` that initializes the UART of ATmega168 according to the following parameters: Asynchronous normal mode, 9,600 baud, 8 data bits, no parity bit, 1 stop bit. Make sure your code is readable (i.e., use bit manipulation to calculate the register values where appropriate).

b) Develop a function named `uart_putchar()` for sending a single character `c` to the terminal. The signature of the function should be (we will later see why; also include `<stdio.h>`):

```
int uart_putchar(char c, FILE* stream);
```

Let your function always return `0` ("no error") and ignore the `stream` argument. Establish a *HyperTerminal* connection with the same baud rate, data bits, stop bits and parity mode as on the target and disable flow control. Test your `uart_putchar()` function.

c) Based on your `uart_putchar()` implementation, implement and test a `uart_puts()` function with the following signature that can print a whole character string to the console:

```
int uart_puts(const char* s);
```

For reasons of clean coding, the function should check the return code from `uart_putc()` for errors (values unequal to 0) and stop processing and return the error code when it sees one. Always append a Windows line break (`"\r\n"`) at the end of a string transmission. Test your implementation by sending `"Hello world!"`.

Hints

- There are two serial connectors on the development board. The one named *RS232 CTRL* is used to program the device and *RS232 SPARE* is used for normal communication (e.g., our debug console). We recommend to attach both serial cables to the host PC so that you do

not need to reconnect the cable between programming and communicating. Each host PC should have an additional serial port connector below the VGA slot.

- In the C programming language, strings are terminated with a special character '\0', which is automatically inserted by the compiler. For example, assume we define a string using the following statement:

```
char str[13] = "Hello world!";
```

Although the string only has a length of 12 characters, we need to request 13 bytes of memory to store `str`, because the last byte is used to store the terminating *null character* '\0'. The standard C `puts()` function continues to output a string until a null character terminator is detected. Implement your own `uart_puts()` function in the same way.

- The `uart_putchar()` and `uart_puts()` functions developed in this exercise will serve you well during the rest of the lab course as they facilitate debugging of the target device. Hence, you should make sure that they work properly before continuing.

Exercise 2.3

Up to now we can only print a constant string to the terminal. To build a debug console, it is important that we can print the current value of a certain variable. The C standard function `printf()` can be used to do this.

a) Familiarize yourself with the `printf()` function (e.g., by "Googling"). Write down some example format strings for formatting character strings, decimal integers and hexadecimal integers.

b) The standard C I/O functions (e.g., `puts()` and `printf()`) forward the output string to a predefined channel called `stdout`. On a PC, this channel is normally bound to a console window, a file or a different process. We will now replace `stdout` with the serial port, so that the output can be visualized at the host side. Add the following statements to your `main()` function (and include `<stdio.h>`):

```
/* File handle for standard output via UART */
static FILE uart_stdout =
    FDEV_SETUP_STREAM(uart_putchar, 0, _FDEV_SETUP_WRITE);
stdout = &uart_stdout;
```

In these statements, we created a "fake" file called `uart_stdout` that redirects to our UART character printing function and replace the default `stdout` object with our own file handle. Now if you use `printf()` function, the characters are redirected to the serial port by calling the `uart_putchar()` function, which you have developed in the previous steps. The reason why we chose the specific signature in exercise **2.2 b)** is that this function expects a pointer to a function with exactly that signature. Test whether the `printf()` function works as you expected using different format specifiers.

Hints

- To make formatting floating-point numbers using `printf()` work, you will need to tweak the project's *Configuration Options* a little bit: In the *Libraries* tab, add the objects `libm.a` and `libprintf_flt.a`. In the *Custom Options* tab, select the `[Linker Options]` item on the left and add the value `-Wl,-u,vfprintf` to the parameter list.

Exercise 2.4

a) Write a function `uart_getchar()` that waits until a single character is received on the serial port and returns that character. To prove that the function works, let the LEDs attached to `PORTB` display the less significant 6 bits of the (ASCII) character code of the received character. Find out the character code of the digit '2' by sending it to the target and interpreting the status of the LEDs.

b) Write an *echo* application. Upon receiving any data from the host side, send back exactly the same data.

Hints

- Displaying the (ASCII) character code of the received character using the LEDs is actually very simple. Characters are already treated as numbers internally. Do not forget to invert the bits because of the inverse logic of the LEDs.

Notes

- It is not a problem if you do not finish all exercises within one session. For reasons of structure, some exercises will be longer, others shorter. You can also work on the solutions in the next session. The following exercise is for those among you who are faster and are not required to be solved.

Exercise 2.5 (optional)

a) Optional: Create a generalized implementation of `uart_init()` named `uart_init2()` that takes the following parameters:

- Baud rate (as a numeric value)
- Mode (asynchronous normal, asynchronous double speed, synchronous master)
- Parity mode (none, even, odd)
- Number of stop bits (1, 2)
- Number of data bits (5, 6, 7, 8, 9)

Use the enumerations from the `uart_params.h` file. Test your implementation with various settings on the target and the host side.

b) Optional: Generalize your set of UART functions so that they can handle 9 data bits. Name them `uart_getchar2()` and `uart_putchar2()` so that you can still use the "normal" ones. For specifying the 9^{th} data bit, either use an additional parameter of type `bool` or change the data type to `short` or `uint16_t`.

c) Optional: `printf()` is a *variable-argument function*, which means that it accepts a variable number of input parameters. If you are interested (or have some spare time), ask the tutor to provide you a `printf()` implementation. This implementation works exactly the same as standard C. Have a look at the `printf()` implementation to find out how to declare and implement a variable-argument function.