1 Introduction

In this document, you will learn about UART, including historical context, background, as well as the specific implementation provided by the ATMega328P.

Contents

1 Introduction **1**

2 Background **1**
2.1 What Is Serial Communication? 1
2.2 Voltage Levels 2
2.3 FTDI, RS232, and Terminology 2
2.4 What is a UART? 3

3 UART Interfacing on the ATMega328P **3**
3.1 Enabling `stdio` 4

4 Exercises **6**

5 Connecting From a Workstation Computer **6**

2 Background

A **UART** (Universal Asynchronous Receiver-Transmitter) is a hardware device which implements asynchronous, full-duplex, serial data transmission between two or more hardware devices. There is a lot to unpack in that sentence, so let's take a step back.

2.1 What Is Serial Communication?

For data to be transmitted **in serial** means for it to be transmitted[1] one bit at a time, generally using a single wire.

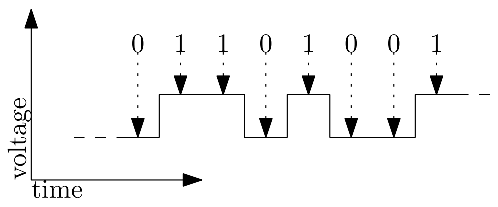

Figure 1: An graph showing the voltage of a wire used for a serial data transmission.

Figure 1 shows an example of an idealized serial connection. We note that the voltage level of the signal is sampled at a constant interval, often referred to as the **sample rate** or **clock rate**. If at the time the n^{th} sample is taken, the voltage level is **logic high**, then we consider the n^{th} bit to be 1, and else 0.

Figure 1 presents an *idealized* view of what a serial connection looks like. In actuality, no electronic device will produce a perfect square wave, where transitions from 0 to 1 bits or vice-versa take place instantly, and the peaks and valleys are perfectly constant voltages. Figure 2 shows a more realistic view - the shaded areas represent time intervals when the value of the signal line is unknown, it may be rising or falling, and sampling during these times would result in an in determinant value.

[1]This is a related but distinct concept from serialization, which usually suggests storing data from memory into some format suitable for archival or storage beyond the lifecycle of of the running program, as used in the phrase "To serialize the data to JSON".

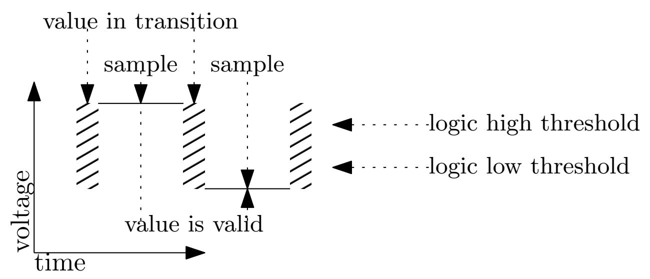

Figure 2: A graph of a serial transmission over time, with transition times shown.

Further, there may be slight fluctuations in the voltage of a signal, even when it is ostensibly held constant. Thus we define any voltage level greater than the logic high threshold to be "logic high", representing a 1, and any voltage level less than the "logic low threshold" to be logic low" representing a 0. For example, for a 5-volt MCU, we might define logic high to be any voltage greater than 4 volts, and logic low to be any voltage less than 1 volt. The vendor-provided data sheet for specific components will indicate what voltage levels are used for logic high and low thresholds.

In short, "serial" is a generic term which refers to the general idea of transmitting data one bit at a time. In the vernacular, it is often used to refer to a collection a collection of communication protocols that may be used in conjunction with a UART (more on this soon).

2.2 Voltage Levels

Often, the voltages that are used to communicate between chips or components are referred to as "TTL voltages" (Transistor-Transistor-Logic voltages). What "TTL voltage" means can vary depending on what specific components are in use, but this is usually meant to imply voltage levels that are used for information signaling purposes internal to an electronic device, as opposed to voltage levels that are used for power supply purposes, or to communicate with the outside world. This distinction is largely historical - at one time it was common to have several different voltage levels supplied to chips (often 12V and 5V), and external connections such as RS232 might run at much higher voltage levels than the internal TTL levels used within devices.

Common TTL voltage levels include 1.8V, 3.3V, and 5V. When connecting devices supporting different voltage levels, a level-shifter must be used. When stepping down voltages, a simple passive resistor-based voltage divider circuit will do, but when stepping voltages up, more complex circuits are required. In the context of this course, we will use pre-manufactured level shifters for these purposes. Keep in mind that connecting a higher voltage level to a device that will not tolerate it, for example directly connecting a 3.3V-tolerant sensor to a 5V MCU may permanently damage the device, and it certainly will not operate correctly.

2.3 FTDI, RS232, and Terminology

There are many terms used in the vernacular to refer to a family of related devices and protocols. You may have heard some of these terms in your own learnings, or in past classes. This can be confusing, and there is a lot of bad information floating around on the 'net.

- **Serial** - refers to any connection which transfers one bit at a time. Sometimes the term "bit serial" is used to differentiate this idea as opposed to the notion of saving data from memory for later retrieval, as in "to serialize an object".
- **FTDI** - commonly used in the vernacular to mean either RS232 or UART, FTDI in fact refers to Future Technology Devices International, a manufacturer of many commonly used adapter chips for connecting to serial connections via USB or types of connections. Much like "Kleenex" or "Xerox", the name of the company has become largely synonyms with the protocols their chips commonly implement.
- **RS232** - a standard for serial communication of data commonly used in older personal computers, and still commonly used for industrial and special purpose devices. This standard is often associated with the DB-25 and DB-9 connectors that are frequently used to expose it.
- **UART** - a type of hardware used to transmit and receive serial data. Often, many formats and speeds of data transmission are supported. UART may be easily adapted to or from RS232 using inexpensive and ubiquitous chips such as the MAX232 and it's successors.

Notably, UART is not a protocol - it is a piece of hardware which can support data transmission in a variety of speeds and formats that make it suitable for use with various protocols.

2.4 What is a UART?

Because UARTs have been around for a long time, and are very prolific, the term has come to adopt many meanings in the vernacular. UART is often used to refer to the general idea of using serial transmission of 8-bit data frames in a style that is generally similar to RS232. However, though it is a common misconception, UART is not a protocol, it is a type of hardware device.

UART data is usually organized into **frames**, groups of bits (usually 8), sometimes with a parity bit, beginning with a start bit, and ending with a stop bit (sometimes two). Data is transmitted at a particular **baudrate** - the number of bits per second which a device transmits or receives. We referred to this as the **sample rate** earlier in this document.

When a UART receives data, it waits for a "start bit" to be received, which is always a zero bit (logic low). It then samples at the configured baudrate for however many data bits it is configured to read, a parity bit (if it is configured to do so), and either one or two "stop bits". "stop bits" are always one (logic high). If even parity is used, then the parity bit is one if the number of one bits in the data frame is even, else it is zero. If odd parity is used, then the opposite is true.

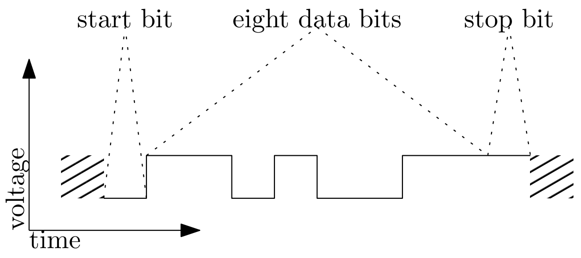

Figure 3: An example of a UART transmission, with 1 stop bit, no parity, and 8 data bits. The value being transmitted is 11010011_2. The hashed areas represent arbitrary, unknown values.

Figure 3 shows an idealized representation of a single data frame of a UART transmission.

The fact that data is divided into distinct frames, which may be separated from one another by any time interval is what makes UART asynchronous. In other words, when no data frame is being transmitted, the line is "idle" in a logic high state until a new frame begins by transitions to logic low, signaling a start bit. This is as opposed to a synchronous serial protocol, where an idle state is signaled by continually repeating a specific bit pattern that the receiver looks for to keep itself synchronized with the transmitter. Synchronous serial will not be used in this course, however the ATMega328P does support it, hence why the datasheet uses the term USART (Universial Synchronous/Asynchronous Receiver Transmitter, rather than UART).

3 UART Interfacing on the ATMega328P

The ATMega328P includes a very featureful USART controller core which supports serial frames of 5 to 9 bits, 1 or 2 stop bits, as well as odd or even parity.

The sampling rate of the USART controller is defined by a baud rate control register, UBRRn (USART Baud Rate Register). As we intend to operate the USART in asynchronous normal mode, we can compute the value of the UBRRn register using the following equation:

$$\mathrm{UBRR}n = \left\lfloor \frac{f_{\mathrm{OSC}}}{16 \cdot \mathrm{BAUD}} - 1 \right\rfloor \tag{1}$$

In this equation, f_{OSC} refers to the system clock frequency, in our case 16MHz, and BAUD refers to the desired baudrate. Unless otherwise noted, we will use a baudrate of 56.7k (56700) in this course. Therefore, we can compute the value to which we must initialize UBRRn as such:

$$\mathrm{UBRR}n = \left\lfloor \frac{16 \cdot 10^6}{16 \cdot 56700} - 1 \right\rfloor = \lfloor 16.6366 \rfloor = 16 \tag{2}$$

In C, this is accomplished as such:

```c
UBRR0 = (((F_CPU/(UART_BAUDRATE*16UL)))-1); // set baud rate
```

Listing 1: Code for setting UBRRn register for USART0, assuming `F_CPU` is a macro defining the CPU frequency in Hz, and `UART_BAUDRATE` is a macro defining the desired UART baudrate. Notice that `16UL` forces the value 16 to be interpreted as an `unsigned long`, preventing any unwanted type conversion.

If we wish to enable both receiver and transmitter modes, we must write 1 bits into the RXEN (receiver enable) and TXEN (transmitter enable) fields of UCSRnB (USART Control and Status Register n B), as such:

```c
UCSR0B|=(1<<TXEN0); //enable TX
UCSR0B|=(1<<RXEN0); //enable RX
/* alternatively ... */
UCSR0B |= (1<<TXEN0) | (1<<RXEN0); // enable TX and RX in one line
```

Listing 2: Code for setting USART0 into transmit+receive mode.

Finally, we must configure the controller to use 8 data bits, one stop bit, and no parity. This can be done using the UPM (USART Parity mode), USBS (USART Stop Bit Select), and UCSCZ (USART Character SiZe) fields of UCSRnC, as such:

```c
UCSR0C |= (1<<UCSZ00) | (1<<UCSZ01); // no parity, 1 stop bit, 8-bit data
```

Listing 3: Code for setting USART0 into 8 data bits, 1 stop bit, no parity mode.

We note that UPM defaults to no parity mode, and USBS defaults to one stop bit. Thus the following code is equivalent:

```c
UCSR0C |= (1<<UCSZ01) | (1<<UCSZ00); // character size of 8
UCSR0C &= ~(1 << USBS0); // 1 stop bit
UCSR0C &= ~( (1 << UPM01) | (1 << UPM00) ); // disable parity
```

Listing 4: Code for setting USART0 into 8 data bits, 1 stop bit, no parity mode, explicitly setting default values.

We can now transmit data by waiting until the UDREn flag in UCSRnA is 1, indicating that the transmit buffer UDRn is ready to receive data, after which we can transmit data by simply writing the byte we wish to transmit into UDRn. Here is how we might transfer the character 'a':

```c
char data = 'a';
while(!(UCSR0A&(1<<UDRE0))){}; // wait until UDRE0 is high
UDR0 = data; // transmit the byte
```

Listing 5: Code to transmit the character 'a' using USART0.

To make our code a little more readable, we can also use the utility function `loop_until_bit_is_set()`:

```c
char data = 'a';
loop_until_bit_is_set(UCSR0A, UDRE0); // wait until UDRE0 is high
UDR0 = data; // transmit the byte
```

Listing 6: Code to transmit the character 'a' using USART0, in a slightly more readable style.

Data can be received similarly. The RXC (USART Receive Complete) field of UCSRnA is set to 1 when a data frame has been received, but not yet read. For example:

```c
char data;
loop_until_bit_is_set(UCSR0A, RXC0); // wait until RXC0 is high
data = UDR0 // read the data
```

Listing 7: Code to receive a character using USART0.

3.1 Enabling `stdio`

Now that we can transmit and receive data, we probably want to be able to use functions such as `printf()` and `read()` to print to or receive from the serial console. Fortunately, AVR Libc makes this very easy. All we need to do is provide a function that can read one byte of input, and another function which can write one byte of output.

```c
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/**
 * @brief output one character via UART, on UART0
 *
 * @param c the character
 * @param stream file handle, this is for compatibility
 *
 * @return
 */
int uart_putchar(char c, FILE *stream) {
        /* wait for the transmitter to become ready */
        loop_until_bit_is_set(UCSR0A, UDRE0);

        /* because it is assumed by most serial consoles, we automatically
         * rewrite plain LF endings to CRLF line endings */
        if (c == '\n') {
                uart_putchar('\r', stream);
        }

        /* wait for the transmitter to become ready, since we might have just
         * transmitted a CR */
        loop_until_bit_is_set(UCSR0A, UDRE0);

        /* transmit the character */
        UDR0 = c;

        /* signal success */
        return 0;
}

/**
 * @brief Read one character from UART0, blocking
 *
 * @param stream for compatibility
 *
 * @return the character that was read
 */
int uart_getchar(FILE *stream) {

        /* wait for the data to arrive */
        loop_until_bit_is_set(UCSR0A, RXC0);

        /* return the received data */
        return UDR0;

}

/**
 * @brief Initialize UART for use with stdio
 */
void uart_init(void) {
        /* initialize USART0 */
        UBRR0=(((F_CPU/(UART_BAUDRATE*16UL)))-1); // set baud rate
        UCSR0B|=(1<<TXEN0); //enable TX
        UCSR0B|=(1<<RXEN0); //enable RX

        UCSR0C |= (1 << UCSZ01) | (1 << UCSZ00); // character size of 8
        UCSR0C &= ~(1 << USBS0); // 1 stop bit
        UCSR0C &= ~( (1 << UPM01) | (1 << UPM00) ); // disable parity

        /* initialize file descriptors, notice the functions we wrote earlier
         * are used as callbacks here */
        fdevopen(uart_putchar, NULL);
        fdevopen(NULL, uart_getchar);
        printf("\n\nUART initialized (%i 8N1)\n", UART_BAUDRATE);
}
```

Listing 8: Example code to enable `stdio` via the UART console.

Now, we can simply call `uart_init()` from our `main()` function, after which functions such as `printf()` should begin to work. You should be able to use a command like `picocom -b 57600 /dev/ttyUSB0` to attach to your UART console if everything is working properly. Remember to define values for `UART_BAUDRATE` and `F_CPU`!

HINT: remember that `UART_BAUDRATE` and `F_CPU` are larger than a standard integer on AVR, so you should suffix them with `UL` to signify that they should be interpreted as `unsigned long` values. For example: `#define F_CPU 16000000UL`.

BEST PRACTICE: it is wise to break values such as these out into your `config.mk` file. You can define the value of a macro as a command-line argument to `avr-gcc` using `-D`, for example `-DMACRONAME=123` would define the macro `MACRONAME` to have the value 123.

4 Exercises

You should complete the following **ungraded** exercises to check your understanding. **You do not need to submit solutions to these problems**.

1. Using listing 8 as a starting point, write a program that displays "Hello World!" to the UART console. Make sure it compiles and runs correctly!
2. Using your solution to the previous exercise as a starting point, write a program that asks you for your name, then prints `Hello, <your name here>!`.
3. Using what you have learned from this and the previous handout, write a program which can display the value of one of the unused GPIO pins (such as PD2) every second using UART. Try connecting and disconnecting this pin to VCC while the program is running, and make sure the output displayed changes appropriately.
4. Using your solution to the previous exercise as a starting point, write a program that continually checks the value of PD2, setting the value of PD3 to it's complement, and displaying an informative message using UART any time the value changes. Use an LED, or the Oscilloscope to verify the value is correctly complemented.

5 Connecting From a Workstation Computer

NOTE: this section applies specifically to computers running Linux. You can also use other operating systems, but the specific software programs and steps will be different.

When you connect the USB port on the Arduino board to your computer, a device entry should be created, usually `/dev/ttyACMX` or `/dev/ttyUSBX`, where `X` will be a number (often 0) used to differentiate different serial devices from one another.

One easy way to determine the device entry that a serial device is associated with is to check the output of the `dmesg` command. For example, when connecting an Adafruit Metro 328P to a Linux computer, you might use a command such as `dmesg | grep tty` and get an output that looks like this:

```
[940038.067977] usb 2-1.8.5.3: cp210x converter now attached to ttyUSB0
```

This would suggest that the device entry is `/dev/ttyUSB0`.

You will also need a program to interface with the device entry. There are many available, in this course, we will be using `picocom`.

If you wish to install `picocom` on your personal computer, it can be retrieved from this URL: `https://github.com/npat-efault/picocom`. It will already be installed in the university computer lab for you to use.

To attach `picocom` to a serial device, you should use a command like `picocom -b 57600 /dev/ttyUSB0`, replacing `/dev/ttyUSB0` with the correct device entry if it isn't already. You can later exit `picocom` by pressing Ctrl + a followed by Ctrl + x.

References

- An Introduction to Microcomputers, Volume 1, 1977, by Adam Osborne: `https://archive.org/details/AnIntroductionToMicroprocessorsVolume1`
- ATMega328P data sheet`https://www.microchip.com/wwwproducts/en/ATmega328p`