

# Experiment 10: Universal Asynchronous Receiver/Transmitter

## 1.0 Introduction

*Asynchronous serial communication using a Universal Asynchronous Receiver/Transmitter (UART) has been a staple of inter-computer communication since its introduction in the 1960s. Although asynchronous serial has been largely supplanted by more specialized communication protocols that are faster or more specially structured, it remains the only interface on your microcontroller that can be readily used for ad hoc interactive bidirectional communication. In this lab, you will use a UART to implement communication with an interactive program. You will examine the asynchronous serial protocol using an oscilloscope to identify the communication payload.*

Step		Points
4.1	Prelab	25
6.1	Wiring	5
6.2	Initializing the USART	20
6.3	Adding support for printf()	10
6.4	Reading from the terminal	15
6.5	Reading and writing with interrupts	30
6.6	Testbench	5
6.7	Post lab submission	*
<b>Total</b>		<b>100</b>

**\*All lab points are contingent on submission of code completed in lab.**

## 2.0 Objectives

1. To understand the configuration and programming of the STM32F0 USART
2. To examine and analyze a serial data stream

## 3.0 Equipment and Software

1. Standard ECE362 Software Toolchain (Installation instructions available in Experiment 0: Getting Started)
2. STM32F0-DISCOVERY development board (For procurement instructions, see Experiment 0: Getting Started)

## 4.0 Prelab

Complete the prelab on the course website

## 5.0 Background

### 5.1 Asynchronous Serial Communication

An asynchronous serial connection allows information to be sent between two devices using three wires: receive (RX), transmit (TX), and ground. Optional "handshake" lines can add flow control. For example a receiver can use a signal to output a "request to send" (RTS) to a transmitter. This tells the transmitter that it is "clear to send" (CTS). Data and handshake lines are connected between devices reciprocally. The RX signal on one device is connected to the TX signal on the other device, and similarly for RTS and CTS.

An asynchronous receiver and transmitter translates between parallel bytes and serial encodings of those bytes framed by start and stop bits. An optional parity bit in each word can be added to detect errors. When such a hardware transceiver can be configured for multiple word sizes, parity types, and number of stop bits, it is called a Universal Asynchronous Receiver/Transmitter (UART). When it can also optionally support Synchronous transmission, it is called a USART.

A USART, at minimum, supports two I/O registers used to put new bytes into the transmitter and read new bytes from the receiver. Other I/O registers are used to determine when the transmitter is ready for another byte and when the receiver can be read. Still other I/O registers can be used to control the generation of interrupts and DMA requests for the receiver and transmitter.

### 5.2 STM32 USART

Your STM32 development system contains two independent USART channels, each of which have two signals that can be routed to the external pins. Each channel uses eight main I/O registers to orchestrate the asynchronous serial communication:

- 1) The USART\_BRR register is used to select a baud rate which is generated by dividing the 48MHz clock by a 16-bit number.
- 2) The USART\_CR1 register is used to set up long-term channel configuration parameters as well as to enable the device (with the UE bit). Most of these parameters may be changed only when the UE bit is clear.
- 3) The USART\_CR2 register is used to configure a few other long-term channel parameters such as the length of the stop bit. The ability to change them also depends on CR1\_UE being clear.
- 4) The USART\_CR3 register is used to configure still more long-term channel parameters. The ability to change many of them also depends on CR1\_UE being clear.
- 5) The USART\_ISR register contains the flags that show the state of the channel as well as interrupt status.
- 6) The USART\_ICR register is used to clear bits in the ISR.
- 7) The USART\_RDR register is used to read a new byte from the receiver.
- 8) The USART\_TDR register is used to write a new byte into the transmitter.

### 5.3 Viewing an asynchronous serial transaction on your lab station oscilloscope

The oscilloscope at your lab station can be used as a high-level monitor for several serial protocols, including asynchronous serial communication. To do so, use the following procedure:

- 1) Connect the channel 1 and 2 probes of the scope to the STM32 RX (PA10) and TX (PA9) signals, respectively. Make sure that the ground clips of both probes are attached to zero volts. (Use a rigid device, such as a LED plugged into the ground strip of a breadboard, to attach the ground clips. Never

attach ground clips to your STM32 development board. You will almost certainly slip, short power and ground pins, and destroy an on-board diode if you do this.)

2) While continuous serial communication is in operation (e.g. press and hold a key in a serial terminal), press the "Auto Scale" button in the upper right of the scope controls. This should allow you to see the square waves. If you press the "Single" button in the upper right corner of the scope controls, you will freeze the display on one snapshot of the communication. You may use the "x pos" dial in the upper middle of the scope controls to slide the captured waveforms to the left or right. Press the "Run/Stop" button to continue scanning.

3) Press the "Serial" button (located either between channels 1 & 2, or in the gray box on the middle right of the controls). Choose Serial 1, and set the mode for UART/RS232. Set the signals to tell the scope which channel is connected to RX and TX. For bus config, select 8 bits, no parity, a baud rate of 115.2Kb/s, polarity "idle high", and a bit order of "LSB." Now, when you press "Run/Stop" and "Single" the snapshot will show blue third and fourth rows at the bottom of the screen which interpret the serial information for you. They show a hexadecimal number for each byte read from the RX and TX lines. By using this interpreter, you can avoid having to analyze the waveforms by hand.

#### 5.4 Using a serial terminal program

The computers at your lab station have a program installed called "Tera Term". (If you are using your own computer for this lab experiment, you may skip these instructions.) Invoke Tera Term. On the configuration box that appears, select "Serial" instead of "TCP/IP". Select "COM4" as the serial device. Choose the menu option Setup → Serial Port. Configure the baud rate for 115200, 8 bits, no parity, one stop bit.

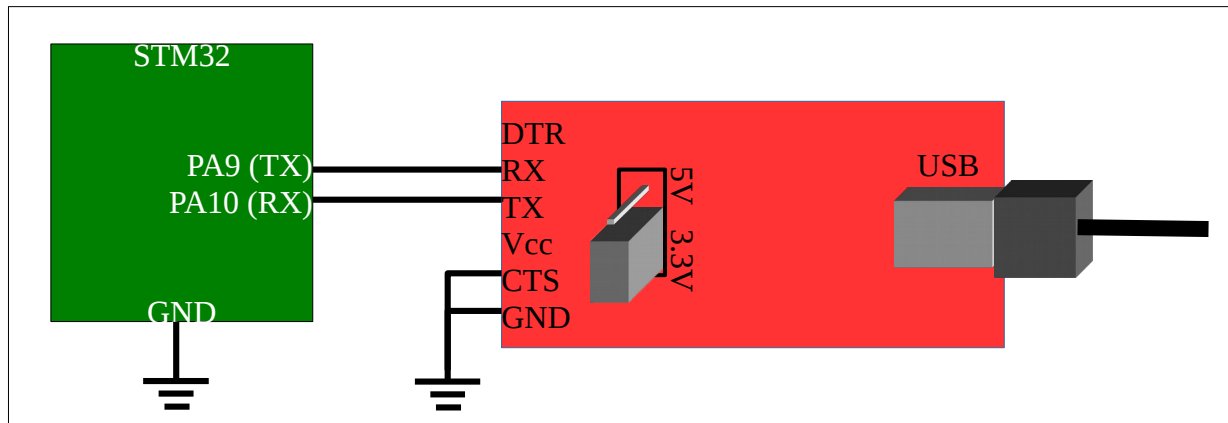


Figure 1: Pinout of the FTDI232 and connections to the STM32.

## 6.0 Experiment

### 6.1 Wiring

Neither the computers in the lab nor your personal computer (likely) have RS-232 serial connectors. Even if they did, RS-232 voltage levels are not compatible with the serial port on the STM32. Instead, you will use a FTDI232 (also known as an FT232RL) to add a UART to the computer that is controlled through a USB connection. The pinout for the FTDI232 and connections to the STM32 are shown in Figure 1. Ensure that the FTDI232 is set for 3.3V operation. To do so, remove the two-pin header from the two pins labeled 5V and replace it on the two pins labeled 3.3V.

Show your completed wiring to your TA. You may continue before demonstrating it.

## 6.2 Initializing the USART

For this section you will need to fill in the **tty\_init()** and **enable\_tty\_irq()** functions. Under **tty\_init()**'s "Student code goes here" section of the subroutine, set up the GPIO registers for pins PA9 and PA10 and then initialize the UART (the corresponding USART for PA9 and PA10) for 115.2Kb/s operation with an 8-bit data word size, and one stop bit. Wait for TEACK and REACK to be set by hardware in the ISR register and finally set the global variable 'interrupt\_mode' to NON\_INTR (see #define).

The **enable\_tty\_irq()** should enable the RXNE interrupt. Remember to enable the right bit in the NVIC registers. It should also set the interrupt\_mode flag to INTR. We're not going to use this function until step 6.5, but you can complete it now just so you don't forget to.

Show this to your TA. You may go on to the next step before demonstrating it.

## 6.3 Adding support for printf()

Whenever you use **printf()** in your standard C program in a desktop environment, the printf function makes a bunch of system calls into the operating system to put the string on to your screen. Our goal for this section is to port the 'printf' and other stdio output functions so that printf results into an output on the terminal (kermit, Tera-Term, PuTTY, etc) screen.

This may seem like a huge challenge, however it is trivial, all you need to do is implement the **\_\_io\_putchar()** function, so that it puts the character 'ch' (parameter of the function) into the USART transmit register. If you are curious about the details, take look at the syscalls.c file in the 'src' directory. For more information about which C functions need which of these low level functions, consult the Newlib libc-manual.

In this lab experiment, we will gradually build up serial I/O subroutines from direct access to the USART to ones that efficiently buffer input and output characters in FIFO queues. These FIFO queues will be filled and emptied with interrupts. The **\_\_io\_putchar()** and **\_\_io\_putchar()** functions will call the appropriate functions depending on whether your program is using interrupts.

For this section you will need to complete the **putchar\_nonirq()** function. It should perform the following operations:

- If the **ch** variable (function parameter) is a newline,
  - Wait for the USART transmitter to be empty.
  - Transmit a carriage return (a '\r' character).
- Wait for the USART transmitter to be empty.
- Transmit the character in the **ch** variable.
- Return the **ch** variable.

Why must we add a carriage return to a newline? What happens otherwise? Try it.

Uncomment **prob3()**, it should display 'Test!' and 'Successful.' on two separate lines on the serial terminal. Both of these words should be printed at the left edge of the terminal window. Demonstrate this to your TA.

## 6.4 Reading from the terminal

For this section you will complete the `__io_getchar()` functionality by implementing the `getchar_nonirq()` function. The `__io_getchar()` function, similar to `__io_putchar()`, is the lowest level subroutine called by `gets()`, `fgets()` and other stdio input functions.

In this section we will implement `getchar_nonirq()` so that the CPU is actively waiting for a full line of input. In a later section, we will use interrupts to do the same. To do this, it gradually fills a first-in, first-out (FIFO) queue. The FIFO implementation is provided for you in the `fifo.c` and `fifo.h` files. This FIFO is special in that it reports the presence of a newline character. To read a full line of input, simply continue to insert characters into the FIFO until it reports that it contains a newline. Once the FIFO contains a newline, the `getchar_nonirq()` function can remove the characters from the FIFO, one at a time, and return them to the caller. The `getchar_nonirq()` function should perform the following: (We give you some hints for the FIFO functions highlighted in yellow.)

- Check for the overrun flag, to check if we missed some characters. If the overrun flag is set,
  - clear the overrun flag (hint: look at ISR and ICR register of the corresponding USART).
- while the `input_fifo` does not contain a newline, `while(!fifo_newline(&input_fifo))`
  - Wait for the USART's RXNE flag to be set.
  - Read a character from the USART and put it into the `input_fifo` with `insert_echo_char()`.
- Remove a character from the `input_fifo` and return it. `return fifo_remove(&input_fifo);`

Uncomment `prob4()`, when run it should prompt you to type in the terminal. Type something in the serial terminal, once you hit enter (carriage return) it should print whatever it was that you typed once again. The `insert_echo_char()` will echo the characters you type as well as let you hit backspace to correct your mistakes.

Demonstrate this to your TA.

At this point, the code you've written resembles the components of an operating system called a *device driver*. It creates a higher-level abstract interface to a hardware resource without the caller having to know how it works. In this case, the `read()` or `write()`, through `__io_putchar()` or `__io_getchar()`, will not have to deal directly with the USART registers. Furthermore, it intelligently handles newline and carriage returns to interact with the terminal the way the program intended. Nevertheless, there are some deficiencies. When the program outputs a string of characters with `puts()` or `printf()`, it must wait for the (slow) serial line to transmit each character. Similarly, when the program is waiting to read a string it is continually looping waiting on the RXNE bit to show that there is a new character to read from the (slow) serial line from the (even slower) typist. An even greater problem is that if the program is busy with something and the user types some data while the program is not doing a `__io_getchar()`, that data will be lost due to an overrun. To avoid problems like this, we want to use the interrupt mechanism of the serial port to insert newly received characters into the `input_fifo` with an interrupt handler. The same interrupt handler can also automatically send characters from an output queue to the serial output when its transmitter buffer is empty.

## 6.5 Reading and writing with interrupts

For this section you will complete **USART1\_IRQHandler()**, **getchar\_irq()**, and **putchar\_irq()** functions. The structure for each functions is described below. In the margin, highlighted in yellow, we show you how to write some of the FIFO functions.

The ISR should do the following:

- If the RXNE flag is set,
  - Read a character from the USART and pass it to **insert\_echo\_char()**.
- If the TXE flat is set,
  - If the FIFO is empty, `if (fifo_empty(&output_fifo))`
    - Turn off the TXEIE interrupt enable.
  - else,
    - Transmit a char from the output\_fifo. `fifo_remove(&output_fifo)`

The use of interrupts allows us to simplify **getchar\_isr()** compared to its nonirq variant. It should do the following:

- While the input\_fifo does not contain a newline, `while(!fifo_newline(&input_fifo))`
  - Wait for an interrupt. `asm("wfi");`
- Return a character removed from input\_fifo. `return fifo_remove(&input_fifo);`

The use of interrupts does not allow us to simplify **putchar\_isr()** compared to its nonirq variant. It looks more complicated, but it's mostly for the sake of avoiding the problems we saw when not using interrupts. The **putchar\_isr()** function should do the following:

- While the output\_fifo is full, `while (fifo_full(&output_fifo))`
  - Wait for an interrupt. `asm("wfi");`
- If **ch** is a newline ('\n'), `fifo_insert(&output_fifo, '\r');`
  - Insert a 'r' into the output\_fifo.
- else, `fifo_insert(&output_fifo, ch);`
  - Insert the **ch** variable into the output\_fifo.
- If the TXE interrupt enable bit is not set,
  - Set the TXEIE bit.
  - Call **USART1\_IRQHandler()** to start sending.
- If the ch variable is newline,
  - While the output\_fifo is full, `while (fifo_full(&output_fifo))`
    - Wait for an interrupt. `asm("wfi");`
  - Insert '\n' in the output\_fifo.

Uncomment **prob5()**, when run it should prompt you to type in the terminal. Type something in the serial terminal, once you hit enter (carriage return) it should print whatever it was that you typed once again.

Demonstrate this to your TA.

## 6.6 Testbench

Developing a project that contains a microprocessor can be frustrating if debugging it means changing code, recompiling, and restarting a program for... Every. Single. Change. One of the most useful ways of interacting with a microprocessor system in development is to use a serial terminal to interact with a program running on the microprocessor. This program can be customized to perform various utility functions directly on the hardware rather than having to rewrite software on the development host. When you examine the internals of almost any microprocessor-based product, you will find a serial test interface. More likely than not, this interface is still functional in the completed product!

The **testbench()** subroutine waits for a line of input and splits it apart into words. It passes the array of words to the **action()** subroutine. To complete this section modify the **action()** subroutine to do something with hardware. You should do something that shows you know how to use your microcontroller. A few examples are provided for you. The **action()** subroutine has code to recognize the following commands:

```
init lcd
display1
display2
```

The **testbench()** also recognizes the **display1** and **display2** commands and does not split apart the words following the commands so that they are passed in their entirety to the **action()** subroutine in the **words[1]** array entry. That allows you to type something like:

```
display1 this is a test
```

and the **action()** subroutine would see entries in the **words** array like this:

```
words[0] = "display1"
words[1] = "this is a test"
```

Another example supported by the **action()** subroutine is the **alpha** command. For this, character echo is turned off so that any character input does not add to the (continually full) output buffer.

You must add something to **action()** subroutine to do something with hardware. You might choose to add your OLED LCD display and the driver software you wrote in lab 8, and let the initialization and display commands initialize and display strings on the two lines. You might choose something to do with your mini-project. You might add commands to turn the green LED on or off like this:

```
green on
green off
```

Use the existing commands as a template. You might create commands to set and clear bits in the PORTC ODR like this:

```
set portc 100000000
clear portc 100000000
```

You could use the **strtoul()** function to convert the binary string in the **words[2]** array entry into an integer value like this:

```
int value = strtoul(words[2], 0, 2);
```

*You get to decide. Use your imagination. Do something interesting.*

Uncomment **testbench()** and demonstrate it. Your TA will decide if it is good enough.

## 7.0 Complete post lab and submit code

Submit your code and complete the post lab on the website. The portal for post-lab will close 10 minutes after your scheduled lab.

You need only upload the code you wrote in main.c. If you add code to the **action()** function to recognize the command

**engage warpdive**

and invoke an equally provocative subroutine, we will, naturally, be curious, but you get to keep that code to yourself. The course staff will look forward to reviewing your mini-project.