OBJECTIVES

- Learn how to apply asynchronous communication and the **U**niversal **S**ynchronous/**A**synchronous **R**eceiver **T**ransmitter (**USART**) system of the *ATxmega128A1U*.

INTRODUCTION

Beyond **parallel communication**, where multiple bits of data are transferred together via a bus of pins[1], **serial communication** is another method in which a processor can communicate with external devices. Overall, serial communication involves the process of sending one bit of data at a time, while following a specific protocol.

Serial communication is generally classified as either being **asynchronous** or **synchronous**. Unlike synchronous serial communication, where a common clock signal is used to determine when to send, receive, or sample data between multiple devices, there exists no synchronization signal for asynchronous serial communication. Instead, asynchronous communication protocols rely on the hope that a common transfer rate will be upheld by any systems using a chosen protocol; if this is not achieved, data transferred or received could be wrongly interpreted, or even entirely missed.

To facilitate asynchronous serial communication, a device known as a **U**niversal **A**synchronous **R**eceiver/**T**ransmitter (**UART**) is generally used in conjunction with both a transmitting device and a receiving device. A UART utilizes a clock signal to generate a transfer rate, denoted as the **baud rate**[2], and also uses two physical connections to communicate data: one pin to **r**eceive data (**R**x), and one pin to **t**ransmit data (**T**x). Within the *ATxmega128A1U*, several UART modules are available by way of the **U**niversal **S**ynchronous/**A**synchronous **R**eceiver **T**ransmitter (**USART**) system.[3]

LAB STRUCTURE

Within this lab, you will begin to explore the asynchronous capabilities of the USART system within the *ATxmega128A1U*. In § 1, you will research relevant information regarding the USART system. In § 2, you will learn to transmit a text character from your microcontroller to a connected computer, via the USART system. In § 3, you will use an oscilloscope to measure the transmission signal generated from a relevant USART module. In § 4, you will learn to transmit a character string of arbitrary length[4]. In § 5, you will determine how to receive a character from a connected computer via the USART system. In § 6, you will learn to receive a character string of arbitrary length[4]. In § 7, you will configure the reception of data via the USART system to be interrupt-based.

REQUIRED MATERIALS

- Atmel *ATxmega128A1U* AU Manual (doc8331)
- Atmel *ATxmega128A1U* Manual (doc8385)
- *OOTB µPAD v2.0* with USB A/B cable
- **D**igilent **A**nalog **D**iscovery (**DAD**) kit with *WaveForms* software

SUPPLEMENTAL MATERIALS

[1] You have previously utilized parallel communication whenever using an I/O port or the external bus interface (EBI) system.

[2] To learn more about baud rates in the context of this course, see Appendix B of this document.

[3] In addition to providing the functionality of a UART, the USART system also provides a form of synchronous serial communication to the *ATxmega128A1U*.

[4] By "arbitrary length", we mean any length of allottable, contiguous data memory locations.

PRE-LAB PROCEDURE

REMINDER OF LAB POLICY

You must re-read the *Lab Rules and Policies* before submitting any pre-lab assignment and before attending any lab.

1. INTRODUCTION TO USART

In this section, you will explore documentation for the **U**niversal **S**ynchronous/**A**synchronous **R**eceiver **T**ransmitter (**USART**) system within the *ATxmega128A1U*, which provides the microcontroller with forms for both synchronous and asynchronous serial communication. However, in this lab, we will only explore the asynchronous form provided by the system.

1.1. Read any relevant parts of § 23 (*USART*) within the 8331 manual to learn about the asynchronous abilities of the USART system available within the *ATxmega128A1U*.

On the *µPAD*, the *ATxmega128A1U* uses a specific USART module to transfer data, via an EDBG chip, to and from the USB type B port also located on the *µPAD*; this allows communication between the USB type B port and a USB type A port on a computer, whenever the appropriate USB A/B cable is connected. (Note that the EDBG chip is not shown on the *µPAD* schematic due to the request of the manufacturer.)

NOTE: It is said that the *microcontroller receives data transmitted from the EDBG chip*, and that the *EDBG receives data transmitted from the microcontroller*. Following this nomenclature, recognize that the signal labeled *EDBG_USART_CDC_TX* in the *µPAD* schematic is meant to correspond to the Rx signal within the relevant USART module, and the signal labeled *EDBG_USART_CDC_RX* is meant to correspond to the Tx signal within the same USART module.

1.2. Review the relevant *µPAD* schematic to identify which USART module is used to communicate with a connected computer.

PRE-LAB EXERCISES

i. The sampling rate of a UART receiver is usually faster than the baud rate of the overall system. Why is this so?

ii. What is the maximum possible baud rate for asynchronous communication within the USART system of the *ATxmega128A1U*, assuming that the microcontroller has a system clock frequency of 2 MHz and that the USART "double-speed mode" is disabled (i.e., the relevant bit CLK2X is set to 0)? In addition to the maximum rate, provide the values of the relevant registers used to configure that rate. Whenever appropriate, support your answer with calculations.

iii. In the context of the USART system within the *ATxmega128A1U*, how many buffers (i.e., memory locations that store temporary data) are used by a transmitter? How many are used by a receiver? Additionally, for both transmitters and receivers, explain how the use of buffers provides greater flexibility to an application involving these components.

iv. If an asynchronous serial communication protocol of 7 data bits, one start bit, one stop bits, even parity, and baud rate of 14.3 kHz was chosen, calculate how many seconds it would take to transmit the ASCII character string **"**`Dr. Schwartz saw seven slick slimy snakes slowly sliding southward.`**"** (This string has 67 characters.) Note that ASCII is a **7-bit** (not an 8-bit) code. Show all work.

2. USART, CHARACTER TRANSMISSION

In this section, you will write an assembly program, **`lab5_2.asm`**, to configure the appropriate USART module within your microcontroller to send data to your computer via the relevant USB ports. For this lab, you must utilize the following asynchronous serial communication protocol: even parity, 8 data bits, 1 start bit, 1 stop bit, and 34,700 bps (bits per second) baud rate, i.e., 34.7 kHz.

2.1. Create an assembly program, **`lab5_2.asm`**. In this program, first create the following two subroutines.

2.1.1. *USART Initialization* (**`USART_INIT`**). This subroutine should initialize the necessary USART module.

2.1.1.1. Set the data direction of the *appropriate* USART transmit pin.

2.1.1.2. Configure the USART module for the appropriate mode (synchronous, asynchronous, etc.), and configure the expected number of data bits, parity type, and number of stop bits.

2.1.1.3. Set the baud rate by storing the appropriate value in the relevant baud rate registers. (See Appendix B for some information regarding baud rates.) You can use the excel workbook given on the course website (Baud Calculator) to verify any baud rate calculations, but **be sure that you know how to calculate the necessary values by hand for quizzes and exams**. (When configuring baud rate registers, it is useful to use assembler directives when programming in assembly, or macros when programming in C, which will be relevant later.)

2.1.2. *Output Character* (`OUT_CHAR`). This subroutine will output a single character to the transmit pin of a chosen USART module. It will be assumed that the relevant character is passed into the subroutine via a general-purpose register (e.g., r16 or r17).

2.1.2.1. At the beginning of this subroutine, check if there is currently an ongoing transmission in the relevant USART module; if there is, wait until it has been

completed. An appropriate interrupt flag should be *polled* to handle this, i.e., do *not* use an interrupt.

2.1.2.2. Transmit the character passed into the subroutine.

2.2. Next, create a main routine within the assembly program to continually transmit the ASCII character **`U`** (i.e., the capital letter U) utilizing the **`OUT_CHAR`** subroutine.

You will need to use a serial terminal program on your computer to view any data transmitted by your microcontroller. Basic information on how to configure/use *PuTTY*, a popular serial terminal program, is given in Appendix A of this document. Some other popular terminal programs are *X-CTU*, *RealTerm*, *Bray Terminal* (also known as *Br@y++ Terminal*), *MobaXterm*, and *HyperTermina*l. There is even a terminal within the *Data Visualizer* extension of *Atmel Studio*. You may use any serial terminal program for this course, as long as it has all features needed. (These features will not be listed here.)

2.3. Use a serial terminal program on your computer, e.g., *PuTTY*, to test your assembly program and verify that the ASCII character **`U`** is continually transmitted.

3. USART, MEASURING BAUD RATE

In this section of the lab, you will use your DAD to measure a baud rate created by a USART module, and then record a transmission frame for the ASCII character **`U`**.

Unfortunately, on the *µPAD*, there is no practical way to measure the physical pins utilized by the USART module connected to your computer. Thus, so that any measurements may be easier, you will utilize some other USART module that has the relevant signals mapped to more accessible pins.

NOTE: When utilizing a separate USART module, you will *not* be able to communicate with a connected computer, since the *µPAD* was not designed to connect any other module to the relevant EDBG chip.

3.1. Create an assembly program, **`lab5_3.asm`**, to configure a USART module that has its Tx signal connected to an I/O pin that can be easily measured via the *µPAD*, and then to continually transmit the ASCII character **`U`** within a main routine. (Other than the different USART module, this program should be unchanged from your previous program.)

3.2. To verify that the defined protocol is met (i.e., 34.7 kHz baud rate, even parity, 8 data bits, 1 start bit, and 1 stop bit), use the *Scope* feature within *WaveForms*, along with your DAD, to measure the width of both a single data bit and a single character transmission frame. Take an appropriate screenshot for each measurement. For the character transmission frame screenshot, annotate each element within one frame by type, e.g., *start bit, data bit 0 (D0), data bit 1 (D1)*, etc.

NOTE: The ASCII character **`U`** (equivalent to `0x55 = 0b01010101`) was chosen for the above tasks to allow the elements within a character transmission frame to be easily identified.

4. USART, STRING TRANSMISSION

Now that you have a method to output a single character via the USART system, you should be able to easily create a routine to output a character string of arbitrary length.

4.1. Create an assembly file, **`lab5_4.asm`**. First, copy the subroutines used in § 2. Then, write the following subroutine.

4.1.1. *Output character string* (**`OUT_STRING`**). This subroutine should output a character string stored in program memory, using the appropriate USART module. When this subroutine is called, it will be assumed that the Z register already points to the beginning of a character string within memory, i.e., any main program utilizing this subroutine must properly configure the Z register before calling the subroutine.

4.1.1.1. Read the character pointed to by Z and increment the pointer.

4.1.1.2. For each non-null (i.e., non-zero) character, call the subroutine **`OUT_CHAR`**; when a null character is found, return from the subroutine.

4.2. Create a main routine within the relevant assembly program to output your complete name, using the **`OUT_STRING`** subroutine. Use a terminal program on your computer to test your assembly program. Outputting your name once in the terminal is sufficient.

NOTE: Recall that ASCII characters can be referenced in *Atmel Studio* individually, by using single quotes (e.g., `'A'`), and as a string, by using double quotes (e.g., `"this is a string of ASCII characters"`).

5. USART, CHARACTER INPUT

In this section, you will begin to configure the appropriate USART module to receive serial data from your computer.

5.1. Create an assembly file, **`lab5_5.asm`**. First, copy the subroutines used in § 4, and edit the **`USART_INIT`** subroutine to additionally enable the receiver within the appropriate module and configure the data direction of the pin connected to the Rx signal. Then, write the following subroutine.

5.1.1. *Input character* (**`IN_CHAR`**). This subroutine should receive a single character with the relevant USART module and return the received character to the calling procedure via a specified general-purpose register (e.g., r16 or r17).

5.1.1.1. Check if a character has been received (by polling an appropriate interrupt flag), and if not, keep checking until one has been received.

5.1.1.2. Read the received character from the appropriate buffer and return the character to the calling procedure.

5.2. Design a main routine within the relevant assembly file to continually echo (i.e., transmit back) to your computer any character received. Utilize the **IN_CHAR** and **OUT_CHAR** subroutines whenever appropriate. Use a terminal program on your computer to test your assembly program.

6. USART, STRING INPUT

Now that you have a method to input a single character via the USART system, you should be able to create a routine to input a character string of arbitrary length.

6.1. Create an assembly program (**lab5_6.asm**). First, copy the subroutines used in § 5. Then, write the following subroutine.

6.1.1. *Input character string* (**IN_STRING**). This subroutine should receive a character string of arbitrary length with the relevant USART module and store the received character string to some memory location(s) within data memory via the Y index. Whenever this subroutine is called, it will be assumed that the Y index already points to the beginning of some pre-allocated contiguous memory locations, i.e., any main program utilizing this subroutine must properly configure data memory and the Y index before calling the subroutine.

6.1.1.1. Continually read characters from the appropriate USART module with the **IN_CHAR** subroutine. For each character not equal to the carriage return[5] character (CR, 0x0D) character, nor the backspace character (BS, 0x08), nor the delete character (DEL, 0x7F), store the character in the next appropriate data memory location with the **Y** index; when a backspace character or delete character[6] is found, decrement the Y index to allow for another character to be written, and when a carriage return character is found, store a null character at the end of the input string and return from the subroutine.

6.2. Create a main routine within the relevant assembly program to input your complete name, using the **IN_STRING** subroutine, and then echo the relevant input string. To echo this string stored in data memory, you will not be able to use the **OUT_STRING** designed in § 4 (since this subroutine was designed to read from program memory), however another subroutine with very similar functionality could be created. Make sure to allocate an appropriate amount of data memory as well as configure any necessary indices, e.g., Y. The amount of memory that should be allocated for the input string is, more or less, arbitrary. (In general, this should be dependent on the application.) However, it would be wise to utilize some form of a symbolic constant, e.g., one defined by the '.equ' keyword, so that the amount of data allocated could be readily changed.

6.3. Use a serial terminal on your computer to test your program. Verify that backspace functionality is correct for at least one of [1] the backspace character (BS, 0x08) or [2] the delete character (DEL, 0x7F); it is not expected that both of these specified characters actually be utilized on some connected keyboard (e.g., DEL may not function as expected).[6]

7. USART, INTERRUPT-BASED RECEIVING

In this section of the lab, you will learn how to configure interrupt-based receiving within the USART system by creating an interrupt-driven echo program for the appropriate USART module.

7.1. Create an assembly program, **lab5_7.asm**, that utilizes the receive complete (RxC) interrupt within the appropriate USART module to echo (i.e., transmit back) to your computer any character received. Additionally, to demonstrate that your serial interrupt is independent from the rest of your program, continually toggle the *BLUE_PWM* LED within the main routine of your program. (See the relevant *µPAD* schematic, if necessary.) Use a serial terminal program to test your assembly program. Within the serial terminal, make sure that local echoing, or anything similar, is disabled. (See § A.3 within Appendix A of this document to learn how to disable local echoing within *PuTTY*.)

[5] To denote when text should start to appear at the beginning of a following line, a special sequence of values, comprising what is known as a *newline*, is utilized. Usually, a newline is generated by an Enter key or Return key on a computer keyboard, and, with most operating systems, a newline consists of at least a *line feed* (LF, 0x0A, \n) character. However, with *PuTTY* and some other serial terminal programs, only a *carriage return* (CR, 0x0D, \r) character is used to represent a newline, at least by default.

In the *Windows*® operating system, a newline is a carriage return character, followed by a line feed character, to designate that a relevant cursor should both return to the beginning of a line of text and advance to the next line of text, respectively. This sequence of characters originates from typewriter times, when starting a new line of text involved the two-step process of turning a carriage to the position denoting the beginning of a line and then turning the platen (wheel) to the following line.

In systems based on *Unix*® or *Linux*®, a newline character consists of only a line feed character.

[6] The delete character code is included here since serial terminal programs (e.g., *PuTTY*) utilize, by default, the delete character code to represent a "backspace".

NOTE: Recall that interrupt service routines should generally be as short as possible; thus, it would generally be *unwise* to call a subroutine within an ISR.

PRE-LAB PROCEDURE SUMMARY

1) Answer pre-lab exercises when applicable.
2) Become introduced to the USART system within the *ATxmega128A1U* in § 1.
3) Learn how to transmit a character via the USART system in § 2.
4) Measure USART character transmissions with an oscilloscope in § 3. Take an appropriate screenshot for each relevant measurement and annotate when appropriate.
5) Learn to transmit a character string of arbitrary length via the USART system in § 4.
6) Create a subroutine to receive a character via the USART system in § 5.
7) Implement a subroutine to receive a character string of arbitrary length via the USART system in § 6.
8) Create an interrupt-driven echo program in § 7.

APPENDIXES

A. PUTTY

PuTTY is a lightweight terminal program with many features and settings, but for the purposes of this course, we will only need to use its serial operating mode. To start using *PuTTY*, you will first need to download the program to your PC.

A.1. To download *PuTTY* for 64-bit operating systems, click here; for 32-bit operating systems, click here. Once downloaded, run the executable.

Upon the program opening, the *PuTTY* configuration menu should be displayed, as shown in Figure 2. This configuration menu is used to select the operation(s) of the terminal application. There are a few things that must be changed before we can start communication between our computer and the microcontroller.

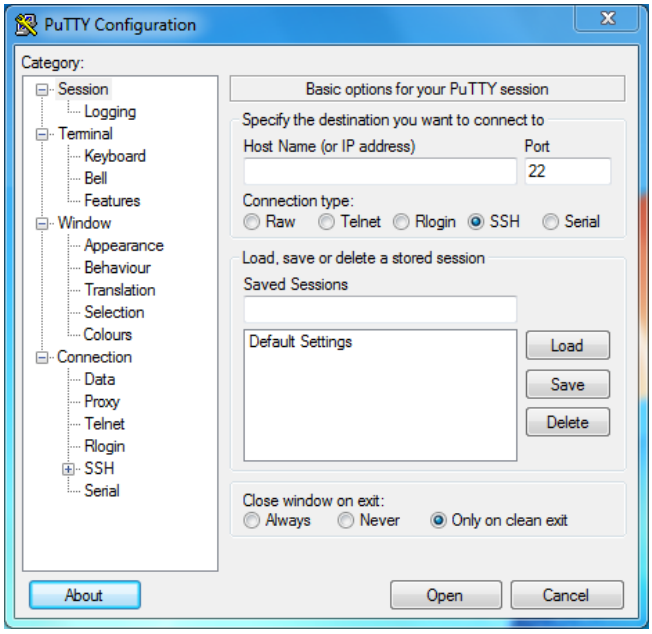

Figure 2: *PuTTY* Configuration Menu

A.2. In the configuration menu of *PuTTY*, do the following:

A.2.1. Select the *Session* tab at the top left. Choose *Serial* as the connection type (on the right, next to *SSH*).

A.2.2. Select the *Serial* sub-tab on the left of the *PuTTY* configuration menu (located at the bottom of the *Connection* tab list).

A.2.3. Choose and enter the correct COM (communication) port that corresponds to the *µPAD*. To determine which COM port on your computer represents the *µPAD*,

A.2.3.1. Open the preinstalled *Device Manager* application in *Windows*, expanding the *Ports* section.

A.2.3.2. Disconnect your *µPAD* USB cable and make a note of the COM ports that are available. (It is possible that no ports may be shown.)

A.2.3.3. Re-connect your *µPAD* USB cable and notice the COM port was added to the list. This is the COM port that you should use in *PuTTY*, e.g., **`COM1`**, **`COM2`**, etc.

A.2.4. Enter the correct baud rate in the *Speed (baud)* textbox, select the correct number of data and stop bits, and also select the correct type of parity. Additionally, set *Flow Control* to ***None***. (Make sure that the data bits, stop bits, and type of parity are all configured as they are in the USART system within the *ATxmega128A1U*.)

A.2.5. Once everything is configured, you can save your configuration settings so that you do not have to change them every time. To do this, do the following:

A.2.5.1. Navigate back to the *Session* menu, and in the textbox located under *Saved Sessions*, type something such as ***4744 UART Config***. This will be the name used for your configuration.

A.2.5.2. Next, click the button to the right labeled ***Save***. This will save your current configuration, so that you can access it for the next time you use *PuTTY*. (To load a saved configuration, you will need to first click on the appropriate configuration listed within *Saved Sessions* and then click the button to the right labeled ***Load***.)

Now that everything is configured, you can open the terminal window by clicking the ***Open*** button located at the bottom right of the window.

NOTES:

- Configure *PuTTY* and open the terminal window **BEFORE** you debug/run your program in *Atmel Studio*.
- It is possible that the COM channel will change if you have different USB devices connected to your PC, or if you connect any USB devices in a different order. If this occurs, just repeat items 2 and 3 above to determine the proper COM port for your microcontroller.

Additionally, there is a setting in *PuTTY* that causes characters typed to the terminal to be echoed, i.e., displayed to the terminal automatically. This can be mistaken as a properly-working echo program, when in fact *PuTTY* might be the only source of echoing.

A.3. When applicable, to turn off the automatic echo setting, do the following:

A.3.1. Open the configuration menu of *PuTTY*.

A.3.2. Select the *Terminal* tab.

A.3.3. Under *Local echo*, select ***Force off***.

You can find more detailed information on *PuTTY's* website, or by clicking on the following link: *PuTTY* User Manual.

B. BAUD RATE VS. HZ VS. BITS/SECOND

In general, a **baud rate** represents the rate of **symbols per second**, where a **symbol** represents a relevant unit. In this course, the relevant symbol for a baud rate is a *bit*. Therefore, the overall unit of any baud rate in this course is **bits/second** (**bps**).

Additionally, since one bit of data is transmitted per one cycle of a respective clock signal, and since the unit of hertz (Hz) represents *cycles per second*, we can state that a baud rate of 1 bps implies that the respective clock signal has a frequency of 1 Hz.

To learn more about communication theory, take *EEL4514: Communication Systems and Component*

C. DEBUGGING WITH UART

The *ATxmega128A1U* UART module **does not** continue to run while the processor is halted at a breakpoint. Consequently, if you are trying to debug your code by setting breakpoints anywhere near your UART functions, or code that calls your UART functions, your character transmissions are very likely to become **corrupted**. If you use breakpoints, you should be aware of this fact.

Fortunately, you have many tools at your disposal. Halting the program to view IO memory and registers can still be useful, as long as you are aware that stepping through the code line by line will interfere with UART communication. Additional debugging techniques include outputting useful information (such as the USART STATUS register) via LEDs at runtime. Another common debugging technique is printing information to the terminal.