PHYS 402 – Introduction to Embedded Microsystems

Laboratory #3 – Serial Basics: the USART and RS-232

Total Points: 0x46

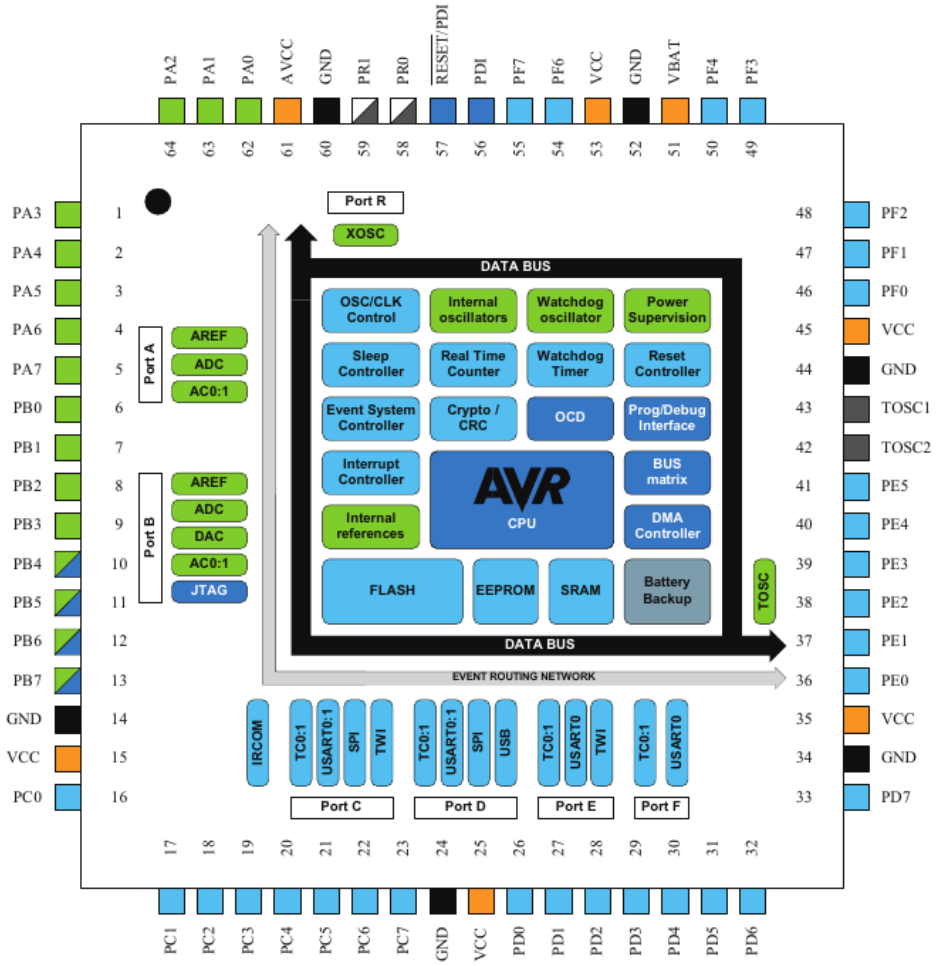
Finish this lab EOC: Sept. 19, 2019
Code for credit by: Sept. 24, 2019

Objective

This lab will introduce the USART and RS-232 serial communications, as well as the XMega MCU and the XPlain board. We will use our knowledge of interrupts to allow the serial port to run bidirectionally with minimal overhead. DO NOT spend time optimizing USART ISRs in this lab; I will give you an optimized USART driver module to use in your projects.

Part I – The XMega128A USARTS

The ATXMegaA3BU has 6 USARTs (way more than you'll ever need!), as well as dedicated ports for SPI (2), TWI (2), and USB. Tables 33.1 and 33.2 in the datasheet show the alternate functions for each of the ports. As you can see, ports A and B are



intended primarily for use as ADC/DAC combinations. Ports C, D, E, F are the timer/serial ports. We'll focus on those for the next two labs.

The A3BU Xplained is much more economical in its use of ports. Header J1 breaks out the TWI (I2C), SPI, and USART functions of port C. J4 breaks out the TWI and USART functions of port E. So we have some resources to work withTake a few moments to familiarize yourself with the chip layout and where things are on the board.

Part II (0x14 points) – Basic Output

Remember from lecture there are two basic types of serial (or parallel) communications: synchronous and asynchronous. In the former case, a clock signal is sent with the data to provide the receiving end with a means for recovering the data. In the latter case, the two ends "agree" beforehand on the data rate, and the receiver then must have circuitry to recover the data based on a clock of that frequency (essentially, it must determine the phase of the incoming data).

Synchronous interfaces are most often used for communication between devices on a board or between boards in a system. The Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C, disingenuously called Two-Wire Interface, or TWI, by Atmel) standards are commonly used this way. Asynchronous interfaces are used for communication between systems, over longer distances; for communicating between computers and devices, you're familiar with the Universal Serial Bus (USB) and maybe RS-232 serial, but many of the long haul standards like ATM and Ethernet are also asynchronous.

Let's look at the synchronous ones first. The upper part of port C (pins 5-9 on J1) can be used as an SPI port. We're going to build a simple program to just repeatedly transfer a byte out and look at what we get on the oscilloscope.

Use the timer and its ISR to output 1 byte of serial data via the SPI on a regular basis. You'll need to read about the CTRL register and the other SPI setup. Use only the timer ISR to output the same byte over and over again, at regular intervals. You'll want to make sure the intervals are long enough for the transmission to complete, but short enough that you don't have to hunt for the signal on the scope (use the trigger function!).

Once you capture it, look carefully at the scope trace. Can you see where the data is embedded in the signal?

Now let's use port C for serial output. This is a little trickier but will let us talk to a laptop! Remember that RS-232 has a different physical specification, which single-supply processors can't do, so we need a transceiver to produce the right levels. Then we'll use the RS-232/USB converter to let you have a serial port on your laptop.

Use the timer code to output 1 byte via serial USART (port C) on the board. Look at it on the trace and capture the 1 byte transmission to save for comparison.

Now obtain a serial transceiver from me and wire it up using the J1 header on the board. You'll need to supply V_{CC} and GND to the board as well as the RS-232 Rx and Tx signals. You'll also need a DB9 cable and a USB/serial adapter for your computer. You'll want a terminal emulator program like PuTTY or RealTerm so you can see the serial data.

Re-run the code and see what the serial data looks like on the downstream end of the transceiver. See if you can identify the bits. Change the byte you send and see if the bit pattern changes the way you think it should.

Part II (0x14 points) – The Real Thing

Now let's get the USART working. First, we need to set up the USART to time and frame the data correctly, as we discussed in class. We'll use USARTC0 to get started. We want to set the control registers up to enable interrupts on transmit, 57600 baud, 8 data bits, 1 stop bit, and no parity. The framing part is easy; we set the CTRLC register (`USARTC0_CTRLC`) for 8 data bits, no stop bit, and no parity (8N1 format).

To set the baud rate on the XMega, we must do a bit of calculation. We'll tabulate more of these values later on, but for now, trust me that at 57.6 kBaud you need a BScale factor of -4. We'll calculate another variable based on this, so you'll want to declare two of them:

```
int nBScale;
unsigned long nBSel;
```

The formula you need is

$$nBSel = \frac{f_{PER}}{2^{nBScale} * 16 * f_{BAUD}} - 1$$

The frequencies are for the peripheral clock, f_{PER} (which is `pClk` in the setup of part I), and the desired baud rate, f_{BAUD} (57600 in this case). You can use the `pow()` function in math.h to compute the power of 2 in the equation. Then you can set the baud rate control registers for USARTC0:

```
USARTC0_BAUDCTRLA = (unsigned char)(nBSel & 0x00FF);
USARTC0_BAUDCTRLB = (char)( ((nBScale & 0x000F) << 4) |
      ((nBSel & 0x0F00) >> 8) );
```

Finally, we set the interrupt level for the serial transceivers via `USARTC0_CTRLA` to low priority, which is fine for serial communications.

Do all this before turning on the interrupt system with `sei()`. After that, you can turn on the transmitter via the last control register:

```
USARTC0_CTRLB = 0x08;
```

(and 0x10 in the bit mask would turn on the Rx as well).

Now set your terminal emulator program (e.g. PuTTY) for the same baud rate and framing and hook up the board. Get to know PuTTY or RealTerm (or Minicom if you're masochistic); it will be a close friend in this course.

Use your timer interrupt code to set up a program where a new string is output every second, with at least 5 different permutations, and no trivial solution. An example might be {"one", "two", "three", "four", "five"}, but not {"1", "2", "3", "4", "5"}. You can start by doing this with a semaphore-queued blocking function in your main while loop (e.g. output a character, wait for it to finish, output the next, etc.).

Once you have gotten this to work and experienced the pain firsthand, you can take a shot at setting up the serial Tx ISR. Buffer a string, output the first character, and code the ISR so that it checks for the end of string after each transmit complete interrupt fires, then only outputs the next character if there is more to go. This way, when the string has finished sending, you will not get any more Tx interrupts.

SHOW ME: your serial output, when it works.

Part III (0x0F points) – It Goes Both Ways

Now let's have a two-way conversation. A bonafide serial port driver is a complex thing; coding a good reliable one with all the functionality one might want is beyond the scope of this class. Fortunately, some nice folks have done that for you.

In the AVRXlib files is a library and header for `AVRX_Serial`. There might even be a Doxygen file for it, but there's also embedded documentation. Read through this and familiarize yourself with the library and its use.

Change your code so that when you type in the numbers 1-5 on the PC keyboard, the MCU responds with the appropriate string (e.g. "one" through "five", or "uno" a "cinco", etc.). The MCU should not output any data until the PC user sends a request string (i.e. one of the numbers) and should stop output after the response is complete.

Make sure you set up your program so that both the receiver and transmitter are enabled and their interrupts are on. You need the instantiating ISR calls in your main file, as documented in the AVRX_Serial.h header and (perhaps) the Doxygen.

SHOW ME: your bi-directional transfer, when it works.

Part IV (0x14 points) – The Real Thing

One thing to remember about computers is that they can react to inputs and change outputs much faster than humans. In some ways, your interaction through Putty may have "given your code a break" because humans can't type very fast compared with how fast computers can output character data.

Change your program so that it does the following:

1. Wait for an input on the serial port. Only accept the ASCII characters corresponding to numbers 1-16 as input.
2. When a valid input is received, set the LED output counter to that value. Thus, if a string ("6") that decodes as the value 6 is received, the LEDs corresponding to bits 3 and 1 should illuminate.
3. After displaying the valid input on the LED counter, the program should output the received value + 1 (e.g. "7" in the example above) on the serial port and continue to wait.
4. Use an external interrupt and a switch to tell the MCU to initiate the count sequence by outputting the string "1" on the serial port.

Work with one of the other teams on the rest of this experiment. Hook your two MCUs together (being careful to cross Rx and Tx between ports). If no one else is ready when you are, I'll bring you a board that will "cooperate" with yours.

SHOW ME: the "cooperative count" algorithm in action.

Part V (0x0F points) – Taking the Show on the Road

Since we don't have servo motors large enough to drag a PC monitor or laptop around, we'll have to find another way to display information from our robot board. Fortunately the Xplained boards have some small LCD displays. The display module uses an ST microcontroller (ST7565R) to drive the display and it can be accessed via SPI with the XMega as the master.

Atmel provides a low level driver for the ST MCU called `st7565r.c` (and its header `st7565r.h`) in the board support package for the XPlained board. It's part of the ASF, but don't use the whole thing. Read through the driver and figure out how to incorporate it into your demo program.

Use the same algorithm you used in Part I to output a given string every second. Pay careful attention to how the LCD has to be set up using control codes. Can you make the strings scroll off the display?

If you're feeling adventurous, you might try some simple graphics operations using the `gfx_mono.c` set of drivers.