Using the UART and USART to Communicate Using the AVR Microcontroller

Here is the plain english information to understand the USART (Universal Synchronous Asynchronous Transmitter/Receiver). That is to say, the information contained here is an interpretation of the datasheet so it is easily understandable. The USART feature of the AVR microcontroller can communicate with another microcontroller (not necessarily another AVR microcontroller), multiple microcontrollers, or a computer using a voltage level shifter or converter. The USART can transmit data using a buffer and a shift register (more on that later), receive data using a shift register and buffer, create a frame of data that is recognized on both the receiving end and the transmitting end. All of this works according to an agreed upon speed from both sides, or with synchronous mode where the clock line (a wire) is directly connected.

Sending information and receiving information in UART and USART communication is like riding on a train. The train sends you to a destination like it sends data to another computer or microcontroller. The train station is like the internal transmitter and receiver inside the microcontroller. There must be a place that the train riders queue to get on the train. First, a passenger will get the ticket and sit down to wait for the next train to arrive. The passenger then must queue up to get on the train. Then the passenger gets on the train. This is exactly how a USART tranmitter functions. The data is sent to a buffer, which is like the waiting room for the train. The shift registers is like the queue to get on the train. The data moves from the buffer and onto the shif registers. This is after the previous data has left the shift registers. From the shift registers, the data moves along the transmit wire, just like a train moves along a track.

Receiving information is the same but in reverse. In the human analogy, the train arrives to the train station. The person gets off of the train and must wait in a queue so that the people in front of them makes space. In the microcontroller reception of data, the data gets off the wire and goes straight into the shift registers. With the Atmega32, there are two buffers that the data can use to wait to be used by the program in the microcontroller. With the shift register, this gives the receive area three total buffers for the data to sit, waiting to be used. This is established with the chip so Data Over Runs (DOR) are less likely. The DOR is a flag that you can look at.

The train is actually similar to the baud rate. The baud is the clock that pushes the data along the line. There are a couple types of clocks you can apply using UART and USART. The UART only allows you to apply an agreed upon baud rate from both parties (each microcontroller must be set with this specific baud rate). In USART, a clock wire must be connected between each microcontroller. This wire will pulse like a heartbeat. In the case of asynchronous, each microcontroller has its own clock and since the data is being sent with the transmitting microcontroller using this clock (baud rate), the receiving microcontroller must be receiving this data at the same pace, so its clock (baud rate) must be the same.

Clock Modes:

Pick between Synchronous or Asynchronous

```
UCSRC = (1 << UMSEL); //setting the UMSEL bit to 1 for synchronous mode
UCSRC &= ~(1 << UMSEL); //setting the UMSEL bit to 0 for asynchronous
mode
```

$8.50

Asynchronous:

$0.34

Asynchronous is where the microcontroller's clock is not connected together by a wire with the other microcontroller, but they need the same clock beat to process the data on the data line.

This beat, or baud rate is determined by the UBBR formula:

```
UBBR = ( (regular microcontroler clock speed) / 16 * baud ) - 1

Example: UBBR = ( 1,000,000 / 16 * 2400 ) - 1 = ( 1,000,000 / 38,400
) - 1 = 26.0416667 - 1 = 25.0416667 = 25
```

$0.34

It's 25 because the UBBR will not accept a decimal value, but that introduces a 0.2% error, which is acceptable. You can find a table of commonly used UBBR numbers in the "Examples of baud rate settings" under the USART section of the datasheet.

Finding the baud from the UBBR is optional:

```
baud = (regular microcontroler clock speed) / ( 16 * (UBBR + 1) )

Example: baud = 1,000,000 / (16 * (25 + 1)) = 1,000,000 / (16 * 26) =
1,000,000 / 416 = 2403.84615 = 2400
```

$0.34

Pretty close to the standard 2400 baud with 0.2% error which is acceptable. Check your datasheet for acceptable error percentages.

Setting the baud rate:

```
UBBRH &= ~(1 << URSEL);
UBBRH = (unsigned char) (# from table >> 8);
UBBRL = (unsigned char) # from table;
```

The URSEL is set to 0 because UBBRH shares the same I/O as UCSRC, the URSEL must be set to 0 to write to UBBRH. The UBBRH sets the high portion of the baud rate - from bit 8 to bit 11. The UBBRL puts the remaining bits 0 to 7 into the low UBBR register.

Setting the Asynchronous mode:

Normal Asynchronous
Double Speed Asynchronous- U2X bit in UCSRA controls double speed asynchronous

```
UCSRA = (1 << U2X); //setting the U2X bit to 1 for double speed
asynchronous
UCSRA &= ~(1 << U2X); //setting the U2X bit to 0 for normal
asynchronous
```

Where the clock is connected with a wire between the two microcontrollers - DDR_XCK Data Direction Register controls which microcontroller is the master and which is the slave. If the DDR_XCK is set to output, then the XCK pin on that microcontroller is the master since it will be creating the clock output on that XCK pin.

Master Synchronous (This is just the microcontroller that is creating the heartbeat)
Slave Synchronous

Data Frame:

This is the actual data that is transmitted. It's the train with the data riding on-board. This is with 9 bits for the data bits, one start bit, two stop bits, one parity bit which is the maximum frame size.

Idle state: The train track - idle state of the signal - always high (5v)
Bit 01: The train engine - Bit 1: Start bit - always low (0)
Bit 02: Person #1 on the train - Data bit #0 - high or low depending on the data
Bit 03: Person #2 on the train - Data bit #1 - high or low depending on the data
Bit 04: Person #3 on the train - Data bit #2 - high or low depending on the data
Bit 05: Person #4 on the train - Data bit #3 - high or low depending on the data
Bit 06: Person #5 on the train - Data bit #4 - high or low depending on the data
Bit 07: Person #6 on the train - Data bit #5 - high or low depending on the data
Bit 08: Person #7 on the train - Data bit #6 - high or low depending on the data
Bit 09: Person #8 on the train - Data bit #7 - high or low depending on the data
Bit 10: Person #9 on the train - Data bit #8 - high or low depending on the data
Bit 11: Train car just before the caboose watching over everything - Parity bit - high or low depending on the 1's of the data
Bit 12: The caboose - Stop Bit - always high
Bit 13: The extra caboose - stop bit - always high and always ignored by the receiver back to high idle state - or to a new start bit.

There is a maximum of 13 bits in the largest data frame.

Setting the Data Bit Size - Number of data bits you want in the frame. Use the UCSZ2:0 (UCSZ0, UCSZ1, UCSZ2) bits in the UCSRC register. Note, keeping all the bits in UCSZ2:0 not set establishes a 5-bit data bit length.

UCSZ - Character Size

```
UCSRC |= (1 << UCSZ0); //6-bit data length

UCSRC |= (1 << UCSZ1); //7-bit data length, or
UCSRC |= (2 << UCSZ0); //Alternative code for 7-bit data length

UCSRC |= (1 << UCSZ1) | (1 << UCSZ0); //8-bit data length, or
UCSRC |= (3 << UCSZ0); //Alternative code for 8-bit data length

UCSRC |= (1 << UCSZ2) | (1 << UCSZ1) | (1 << UCSZ0); //8-bit data
length, or
UCSRC |= (7 << UCSZ0); //Alternative code for 8-bit data length
```

Huh, what's that "2, 3 and 7"?!? Well, it's about time you learned a shorthand way of setting bits. The UCSZ0 is the equavilant to the one's place in binary. If you put another number, like

7, you are actually putting the binary equivalent of the number 7 in that location which fills up the next binary digits, because the number 7 is 111 in binary.

Note: If you want to change any of this frame stuff shown below, take a look at the TXC and RXC flags and make sure they are not set. If they are set, then there is still transmissions going on. For most microcontroller projects, this may never be needed, unless you want control of the port I/O pins onthe chip where TX and RX is.

Another note: For some strange reason, UBBRH and UCSRC share the same I/O location, so you need to make sure:

> URSEL bit is set in the UBBRH if you want to put stuff in that register, or
> URSEL bit is set in the UCSRC if you want to put stuff in that register.

Setting the Parity Mode - This is where you set the error checking feature of UART and USART communication. The parity is set only by the transmitter.

> If the parity is set for "**even**" then there are an odd number of ones in the data bits, the transmitter will set the parity bit to a "1" so there will be an even number of ones including the parity bit.
>
> If the parity is set for "**odd**" then there are an even number of ones in the data bits, the transmitter will set the parity bit to a "1" so there will be an odd number of ones including the parity bit.

UPM - Parity Mode

```
UCSRC |= (1 << UPM1); //Sets parity to EVEN

UCSRC |= (1 << UPM1) | (1 << UPM0); //Sets parity to ODD, or
UCSRC |= (3 << UPM0); //Alternative way to set parity to ODD
```

Setting the number of Stop Bits to use - Do you want one or two cabooses on the end of your train? Remember that the receiver will ignore the second stop bit, so why did they put it in there in the first place? Who knows, but I think that it adds a "bit" of breathing room for the next data frame. How do you like that pun? Huh!

USBS - Stop Bit Select

```
UCSRC |= (1 << USBS); //Sets 2 stop bits

UCSRC &= ~(1 << USBS); //clears the USBS for 1 stop bit, only needed
if the bit was already set
```

Note: If the microcontroller receiving the data frame (the choo choo train) sees a stop bit that is low (it is supposed to be high), then the Frame Error Flag Bit (FE) will be set.

Example Initialization for the USART or UART:

In this example, the baud is being set, the transmitter (TXEN - Transmitter Enable) and receiver (RXEN - Receiver Enable) is being enabled, and in the UCSRC, the URSEL is set so we can modify UCSRC, 2 stop bits used and the length of the data bits is 8-bit.

You'll notice that, first, this is a function and, second, an unsigned int is being passed into this function. The void in the beginning means that this function will not return anything when it

completes. This is the same example that the Atmega324 datasheet shows. Notice that the parity is not set in this example.

```
void USART_Init (unsigned int baud)
{
   //Put the upper part of the baud number here (bits 8 to 11)
   UBBRH = (unsigned char) (baud >> 8);

   //Put the remaining part of the baud number here
   UBBRL = (unsigned char) baud;

   //Enable the receiver and transmitter
   UCSRB = (1 << RXEN) | (1 << TXEN);

   //Set 2 stop bits and data bit length is 8-bit
   UCSRC = (1 << URSEL) | (1 << USBS) | (3 << UCSZ0);
}
```

Note: When TXEN is set (Transmitter Enabled), the General Purpose function of that pin is not availble until the TXEN is disabled. In other words, you cannot use TX pin to light up LEDS, or receive button presses and stuff like that. Same idea goes for the RXEN and the RX pin. One other thing, the TXEN cannot be disabled while there is a pending transmission. If you are using the XCK pin for synchronous operation, then the other general purpose function at the XCK pin is disabled.

Transmit Something!!

In this next example, we will actually transmit something. We are gonig to throw our data into the UDR (USART I/O Data Register) train station. I will let you in on a secret here, because you may get confused a bit later: You will put the data you want to transmit in the UDR register, and you will also get received data from the UDR as well. That seems totally off, doesn't it. If that's true then how would we be able to transmit and receive at the same time when we only have one data register to use... I mean, if full duplex is available, why is there only one register to transmit data and receive data. Seems odd.

Well, here is the secret! Shhhhh, don't tell anyone! The UDR actually has two places. The secret locations are called TXB and RXB (these are 8-bit locations). The microcontroller is so smart that when you place information into the UDR register, the stuff you put in is actually going into the TXB location. If you take the data from the UDR register, then you are actually taking the stuff out of the RXB location. You might be asking, what if we decided to set our data-bit lenght to 9 bits, cause TXB and RXB is 8-bits only? Those bits are located in the UCSRB register and their called, you guessed it, TXB8 and RXB8. Oh, one more thing, this is a polling example which means that we are going to wait (tap our toes) for the UDRE (USART Data Register Empty) signal to say its ok to go. It's like a stop light. You can't put anything into the UDR if the system is not ready for you. Ok enough jibber jabber, lets get to the example:

```
void USART_Transmit (unsigned int data)
{
   //Wait until the Transmitter is ready
   while (! (UCSRA & (1 << UDRE)) );

   //Make the 9th bit 0 for the moment
   UCSRB &=~(1 << TXB8);

   //If the 9th bit of the data is a 1
   if (data & 0x0100)

      //Set the TXB8 bit to 1
      UCSRB |= (1 << TXB8);

   //Get that data outa here!
   UDR = data;
}
```

The condition in the while statement may look foriegn to you. It is seemingly complicated, but it is the same type of (and) and (not) bitwise operation that we learned in an earlier video. The ! is a (not) and & is an (and). Remember when we say register &=~(1 << bit)? We are doing an & (and) calculation and reversing the bits with the ! (not). We can also put this in a condition like this: If you want to check if a bit is (not) 0, then we use (! (register & (1 << bit))).

Ok, so here is an explanation of the flags that inform us about the transmit process:

Polling Resources:

UDRE = USART Data Register Empty - This is cleared when the UDR is written to.
TXC = Transmit complete - You can use this resource if your are doing half duplex. This flag happens when the data leaves the shift register. Remember that queue that we talked about earlier.

Interrupt Resources: Remember to set your global interrupt variable before using these cool resources!

UDRIE = Data Register Empty Interrupt Enable if UDRE is set to 1 - This will be handy when using interrupts. The microcontroller will stop what you are doing so you can go and put your data into the UDR train station.
TXCIE = Transmit Complete Interrupt Enable - Use this if you are doing half-duplex and you want the microcontroller to interrupt you so you can go and put your data into the UDR train station.

Receive Something!!

The process for the microcontroller to receive data that exists on a wire from the other microcontroller goes like this:

- The microcontroller detects a start bit
- The data bits come next and there are taken in concert with the baud rate (or XCK for sychronous) heart beat
- The data goes to the shift register (like people getting off the train and waiting in queue to get off of the train platform)
- This happens until the first stop bit is received (remember, the second stop bit is ignored)
- The contents of the data then go directly to the UDR (well, really the RXB, but we still get it from the UDR)

Here is an example how to receive data:


```
unsigned char USART_Receive( void )
{
   while ( !(UCSRA & (1 << RXC)) ); //Wait for the RXC to not have 0
   return UDR; //Get that data outa there and back to the main program!
}
```

Do you notice the unsigned char at the beginning of our function statement? That is the type of return we will provide. The UDR will be an 8-bit unsigned value (char). We use the return statement to release the data from the UDR and pass it back to the main program.

Let me know if you want to see a 9-bit example.

Ok, so here is an explanation of the flags that inform us about the receive process:

Polling Resource:

> **RXC** = Receive complete - Indicates unread data in the buffer if this flag is set (1), or 0 if empty.

Interrupt Resource: Remember to set your global interrupt variable when using this!

> **RXCIE** = Receive Complete Interrupt Enable - Use this so you can go and get your data out of the UDR register. Super important!! Make sure you clear the RXC flag right after you get the data out of the UDR register!