UART - Serial communication

Published: 28/09/2016 | Post categories: Communication Interfaces, Learn | Views: 53033

Universal Asynchronous Receiver/Transmitter or **UART** for short represents the hardware - integrated circuit, used for the serial communication through the serial port. UART is a standalone integrated circuit (**IC**) but also as a part of microcontrollers.

It is important to understand the difference between standalone UART modules and the UART modules integrated in MCU modules, and how MCU modules usually control the output of UART, while standalone modules usually do not. Conversion of the UART's output to physical signal on transmitter side and vice versa, is done by a separated circuit depending on the communication standard used, (RS-232, RS-422,RS-485). All of these standards define different methods for physical signal generation which also differs from an MCU's TTL class of digital circuits on the output.

In reality, this means that additional circuit is needed to achieve communication between MCU's UART module and a PC peripheral device or PC serial port - just because there are differences in transmission methods. For example, in the case of TTL, voltages on the UART output can be in range of **0 V** up to **Vcc** where Vcc represents logical 1 and logical 0 is 0 V on output. On the other hand according to the RS-232 logical 0 is positive voltage between 3 and 25 V and logical 1 is negative voltage between -3 and -25 V.

Therefore from the software side MCU's UART module and PC peripheral are the same and most of the embedded devices are easy adaptable to communicate with the PC serial port or even USB which was what replaced the RS-232 when we talk about PC world. This is also the reason why RS-232 is the most important standard from the point of view of an embedded developer. Some rules from RS-232 found it's own usage in the embedded world - Data Flow Control for example, which will be explained with more details later. Another reason is the desire by hardware producers to make their own products also adaptable to the PC peripherals with addition of just one circuit.

Communication

Because the UART is the device used for serial communication, it is good to explain what it stands for. Probably the shortest definition would be that serial communication stands for the process of sending data one bit at a time, sequentially, through the bus or communication channel. Shift registers are a fundamental method of conversion between serial and parallel forms of data, and therefore are an unavoidable part of every UART. Just like serial communication has two primary forms, (synchronous and asynchronous) there are also two forms of UART, known as :

- **UART** - Universal Asynchronous Receiver/Transmitter
- **USART** - Universal Synchronous/Asynchronous Receiver/Transmitter

The asynchronous type of transmitter generates the data clock internally and dependent to the MCU clock cycles. There is no incoming clock signal that is associated with the data, so in order to achieve proper communication between two modules, both of them have to have the same clock cycle length, which means that they must work on the same **baud rate**.

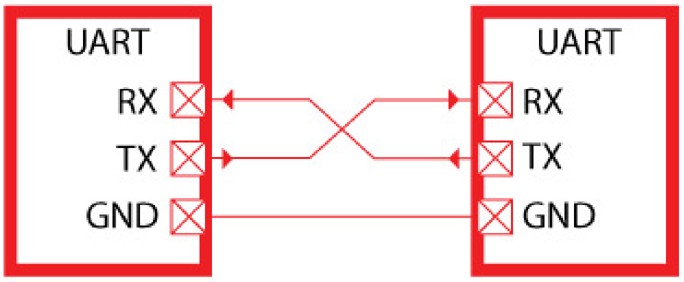
The data is normally transmitted in the form of a byte. On the other hand, a synchronous type of transmitter generates a clock used by the receiver side to recover data from the stream without knowledge of the transmitter's baud rate. In cases where a separated line is used to carry the clock signal, UART can achieve very high transfer rates, up to 4.000.000 bits per second which is high above the UART limits. USART data transfers are usually in the form of a block.

It is important to say that a USART module, (because of higher speed and ability to transfer blocks of data) can generate data in a form corresponding to many different standard protocols such as IrDA, LIN, Smart Card etc. The ability to generate clocked data allows the USART to operate at baud rates well beyond a UART's capabilities. USART does encompass the capabilities of UART. Though, and in many applications, despite having the power of a USART, developers use them as simple UARTs, ignoring or avoiding the synchronous clock generation capability of these powerful peripherals. No wonder so many people use the terms as though they were synonyms.

The communication goes through the two independent lines : **TX** (transmission) and **RX**(reception). It can be :

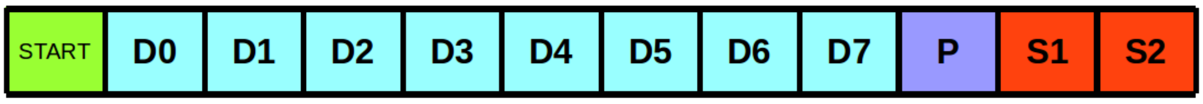
- **Simplex** - One direction only, transmitter to receiver
- **Half Duplex** - Devices take turns transmitting and receiving
- **Full Duplex** - Devices can send and receive at the same time

From the point of the transmitter the TX line is the same as the RX line from the point of receiver, so the data stream always has the same direction through every line. In the idle states lines are pulled high. This allows recognition by the receiver side that there is a transmitter on the other side which is connected to the bus. Data frames can have different lengths depending on configuration.



Frame starts with "**Start Bit**" which is logic low and is used to signal the receiver that a new frame is coming. Next **5-9** bits carry the data and parity bit. The part of the frame is configurable depending on the code set employed. After the data, is the **parity bit,** if the data length is not 9 bits. The parity bit is not used very often, but in the case of noisy buses, parity bits can be a good method to avoid reception of the wrong data packets. It represents the sum of the data bits. Even parity means when the sum is even this bit will be 1, and in the case of odd this bit will be 0. If odd parity is used, the bit values will be reversed, (even sum is 0, and odd is 1).

The end of the frame can have one or two **stop bits**. Stop bits are always logical high. The difference in the logic between start and stop bits allows break detection on the bus. These bits are also called synchronization bits because they mark the beginning and the end of the packet.



The baud rate specifies how fast data is sent over the bus and it is specified in bits per second. It is allowed to choose any speed for the baud rate but actually there are values that are used as a standard. The most common and standardized value is **9600.** Other standard baud rates are : 1200, 2400, 4800, 19200, 38400, 57600 and 115200. Baud rates higher then 115200 can be used but usually that causes a lot of errors in transmission.

The shortcuts used for describing the UART bus configuration are usually in form :

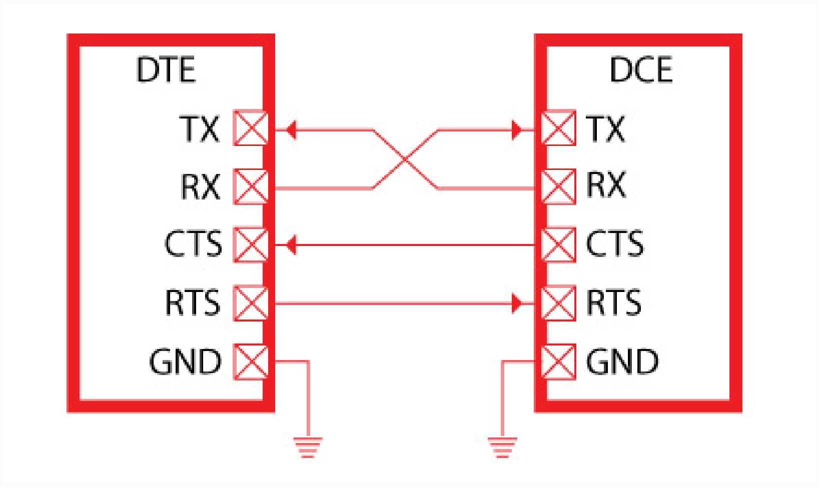
<DATA_BITS><STOP_BITS>

Examples :

• **9600 8N1**	9600 baud rate, 8 bits of data, No parity, 1 stop bit
• **115200 8E2**	115200 baud rate, 8 bits of data, Even parity, 2 stop bits

An important aspect of the communication might be the negotiation between modules. RS-232 standard defines the usage of data flow control. Today there is hardware and software flow control. Both of them are doing the same thing but let's say, for now, that hardware is much more effective and safer.

Data flow control in RS-232 introduced the usage of two additional lines known as **RTS** and **CTS.** These two lines from the view of an embedded developer are nothing more than simple GPIO pins, but usage of them from the view of a software developer can bring a lot of benefits - we will discuss this later. To properly understand what data flow control is we have to know what **DTE** and **DCE** are**.**



RS-232 defines the serial communication between DTE (data terminal equipment) and DCE (data communication equipment). DTE can be PC or MCU and DCE can be some PC modem or some GPS module for example. Now let's go back to the data flow control. The very first defined flow control rules were that DTE has to assert RTS line to indicate the desire for the transfer, and after that the DCE responds by asserting the CTS line if it is ready to receive more data. What is visible on the first look is that this method is only one directional. There is no way for DTE to stop data from the DCE and that is the flaw of this rule.

In the late eighties this rule was replaced by the bidirectional data flow control. The basic meaning of the RTS line was replaced by the **RTR** (ready to receive). RTR means that DTE is ready to receive data and compared to the previous method used for data flow control, these two lines are totally independent. The summary is that DTE asserts the RTS (RTR) line whenever it is ready to receive data and DCE does the same with the CTS line.

UART in MikroC

All of our MikroC compilers have UART library for easier access to the UART module built into the MCU. As we said earlier not all MCUs have a UART module but when you have that kind of MCU there are software UART libraries which use the so-called "bit-banging" technique to emulate a real UART module with the usage of the simple GPIO pins. As you probably conclude the real hardware UART module and software emulation can't be compared if we talk about performance, so the software one should be used only in cases when the MCU has no available UART module.

If we talk about an MCU's UART module, which is the most common case for embedded developers, every module has it's own set of registers responsible for the different aspects of the communication, configuration and state of the device. The organization of these registers inside the MCU vary from the platform to platform. Sometimes an MCU on the same platform doesn't have the same registers names for UART modules but MikroC makes all platforms look the same for the developer. The function set for UART control is pretty same on all our compilers and there are just few differences in the initialization routines.

The very first thing needed to do is initialization of the UART module. All compilers have two possible ways to do this: UARTx_Init and UARTx_Init_Advanced. **UARTx_Init** is common for all compilers and requires the one parameter which represents the baud rate of the UART module. The rest of the configuration parameters are the most common UART configuration - **8N1** - 8 data bits, no parity and one stop bit. **UART_Init_Advanced** allows us to configure every aspect of the communication when it is needed.

```
UART1_Init_Advanced( 115200, _UART_8_BIT_DATA,
                             _UART_NOPARITY,
                             _UART_ONE_STOPBIT,
                             &_GPIO_MODULE_USART1_PA9_10 );
```

This code shows an example of the initialization routine on the STM32 microcontroller. Advanced initialization routine arguments vary from platform to platform but it is pretty understandable what every argument means, and compared to the raw registers, editing it is less painful especially if you are new inside this world or you just have switched to your favorite platform.

To continue our journey let's go to the transmission process and functions UARTx_Write and UARTx_Write_Text. It is good to know what actually goes on when we call some of these functions. Both of the functions are doing the same thing except UARTx_Write_Text is sequentially writing until it arrives to the end of the string (null terminator). When the write function is called the argument is copied to the UART's transmission (**TX**) register. After that, it depends on the type of MCU, usually a few additional actions, like asserting transmission bits inside the control register, are needed to start the transmission process. All of this is handled by the UARTx_Write function.

Sometimes this function can rewrite the content of the TX register before the data from register is shifted out to the bus. If we want to be sure that we have avoided that occurrence function UARTx_Tx_Idle can be used to check if the UART module has finished the transmission of the TX register. You might be wondering why this check is not included to the existing write function - the answer is, sometimes we want to rewrite existing content of the TX register. Also in that case UART_Write becoming a blocking function which is not so good when we have no multiple threads.

```
if( UART1_Tx_Idle() ) {

    UART1_Write( 55 );
}
```

In the case of UART, it is much harder to handle its' reception process. Compared to the previous explained serial communication protocols like I2C or SPI where we have a master on the bus which initiates the transfer, UART is totally different. Reception of the data by UART module starts with reception of the signal by using the shift register process for populating the UART receive register (**RX**). After that, a few additional occurrences happen, for example UART asserts a bit in the UART control register to inform the user that new data has arrived. It is important to say that every new data received will overwrite the existing content of the RX registers. This means that if we haven't already read the RX register the data is lost.

This problem can be solved in two ways :

- Polling
- Interrupt

Polling is much more understandable. We are constantly querying the UART control register for the information "has new data arrived or not?". That is actually what the UARTx_Data_Ready function is for. When the query result says that there is new content inside the RX register, then data can be obtained with the UARTx_Read function.

```
while( 1 )
{
    if( UART1_Data_Ready() ){
        receive = UART1_Read();
        break;
    }
}
```

This actually blocks the execution of the whole program until data becomes available, which is a very bad, dangerous and inefficient solution. This can only be acceptable in the case of multi-threading embedded where we can have threads reserved only for polling.

A much more efficient but at the same time more complex way is the usage of interrupts. The UART module inside the MCU can be set up to execute a special kind of function called an Interrupt Service Routine (**ISR**) in the case of some event related to the UART module itself occurring. The interrupts are a special kind of MCU routine which will be processed in one of the next articles, but for now, we are going to just explain how interrupts can be used to avoid polling.

When we define the ISR and enable the interrupts, inside the ISR we can process the received data and achieve the same thing we did with polling without any blocking process. The ISR will be called any time the UART module receives some data so it is important to say that it is not acceptable for a lot of time to be wasted inside of the ISR. The best solution is that the ISR routine is used to just store the received data to the buffer which will be later processed outside of the ISR. Let's see that in the example on the **ATMEGA32**.

```
sbit RTR at PORTA5_bit;                                                 // RTR pin
sbit CTS at PORTD2_bit;                                                 // CTS pin

volatile bool flag;                                                     // Process flag
volatile char received[8];                                              // Buffer
volatile unsigned int count;                                            // Buffer counter

extern void process_data( char data );

main()
{
    DDA5_bit = 1;                                                       // Set RTR as output
    DDD2_bit = 0;                                                       // Set CTS as input

    UART1_Init( 9600 );                                                 // Peripheral serial

    UCSRB |= ( 1 << RXCIE );                                            // Enable UART RX interru
pt
    SREG  |= ( 1 << SREG_I );                                           // Enable Global interrup
ts

    while( 1 )
    {
        if( flag ) {

            while( count-- )                                            // Process the data
                process_data( received[count] );

            flag = false;
        }
    }
}

void UART_RX_ISR() iv IVT_ADDR_USART__RXC ics ICS_AUTO                   // ISR function
```

```
{
    char tmp = UART1_Read();

    if( ( tmp == 13 ) || ( count == 8 ) )
        flag = true;
    else
        received[count++] = tmp;
}
```

Well this looks good but if we look closer - it is not so safe. ISR is accessing the global variables which are also used by process_data so the program's behavior is not predictable at marked lines. Let's imagine the process_data function refreshing the TFT screen and writing the characters received. That process takes much more time so the chance that something went wrong are increased so far. The simple solution might be copying the content of the buffer to another buffer which is not used in the ISR but even then the chance still exists and alongside with that we are capturing one more part of the RAM for it. Now, remember the part from the beginning with RS-232 rules for data flow controls?

When processing of received data takes a long time, data flow control should take over, stopping the data flow from DCE until the DTE processes the collected data. Using data flow control will avoid losing bytes received, while the DTE is still processing previously received bytes. Of course this is the case when DCE supports it. From the DTE side, where the MCU usually is in our case, RTS (RTR) and CTS lines are, as we said, nothing more than a GPIO. This also means they are easy for handling, especially in cases when DCE firmware supports the handshake method for the "negotiation".

```
...

void send_command( char *buffer, unsinged int count )
{
    while( count ) {
        if( !CTS ) {                                                     // If DCE is ready send data
            UART11_Write( *buffer++ );
            count--;
        }
    }
}

main()
{
    ...

    while( 1 )
    {
        if( flag ) {
            RTS = 0;                                                    // DTE is not ready because of d
ata processing
            while( count-- )
                process_data( received[count] );
            RTS = 1;                                                    // Data processing finished DTE
is ready again
            flag = false;
        }
    }
}

...
```

Careful reading of the datasheet about hardware flow control is needed because, as we said earlier, there are two different methods for this. There also might be some timing requirement by the device we are using. For example, sometimes with assertion of the RTR the data flow doesn't stop immediately. DCE needs time to detect the signal and stop the stream and therefore something must be implemented to handle this problem.

UART is also very useful for debugging. It is usually used to log the data on the reception. So if we imagine that our ATMEGA32 has two UARTs we could do something like this.

```
void UART_RX_ISR() iv IVT_ADDR_USART__RXC ics ICS_AUTO                          // ISR function
{
    char tmp = UART1_Read();
    ...

#ifdef _DEBUG_
    UART2_Write( tmp );
#endif
}
```

This last one was just an example because ATMEGA32 has only one UART module but usually MCUs have more than one. In addition, it is good to say that, we should initialize "debug" UART with a higher baudrate than UART used for the communication with the peripheral. This method is extremely usable in cases like the previous one where we are retrieving data through the interrupt because the debug breakpoints can't stop the data flow from the DCE.

Summary

Serial communication is a fundamental type of communication between two devices and is used everywhere around us. I2C, SPI, CAN, LIN and so on are all some kind of serial communication. Serial communication is simply the connection between PC and the embedded world allowing PC software developers to handle embedded devices and vice versa for embedded developers. MikroElektorinka offers a wide spread of devices that are excellent for learning and exploring the world of UARTs.