**EE 354** **November 13, 2017**
**ARM UART Notes**

For serial communications you should be familiar with the following terms:

UART/USART
Baud rate
Synchronous/Asynchronous communication
Half-Duplex/Full-Duplex

The term UART stands for Universal Asynchronous Receiver/Transmitter and it does the logic functions to make serial communications easier. Effectively, a UART provides an 8-bit parallel interface suitable for a microcontroller bus, to a serial communications channel. The UART can take 8-bits of data from a microcontroller, convert it to a serial data stream, and send it out over a single line at a preprogrammed bit rate. It can simultaneously, in full duplex mode, receive a serial stream, at the same or different bit rate, and convert it to 8-bit parallel for input to a microcontroller bus. A block diagram for the STM32F446 USART is shown in Figure 1 below.

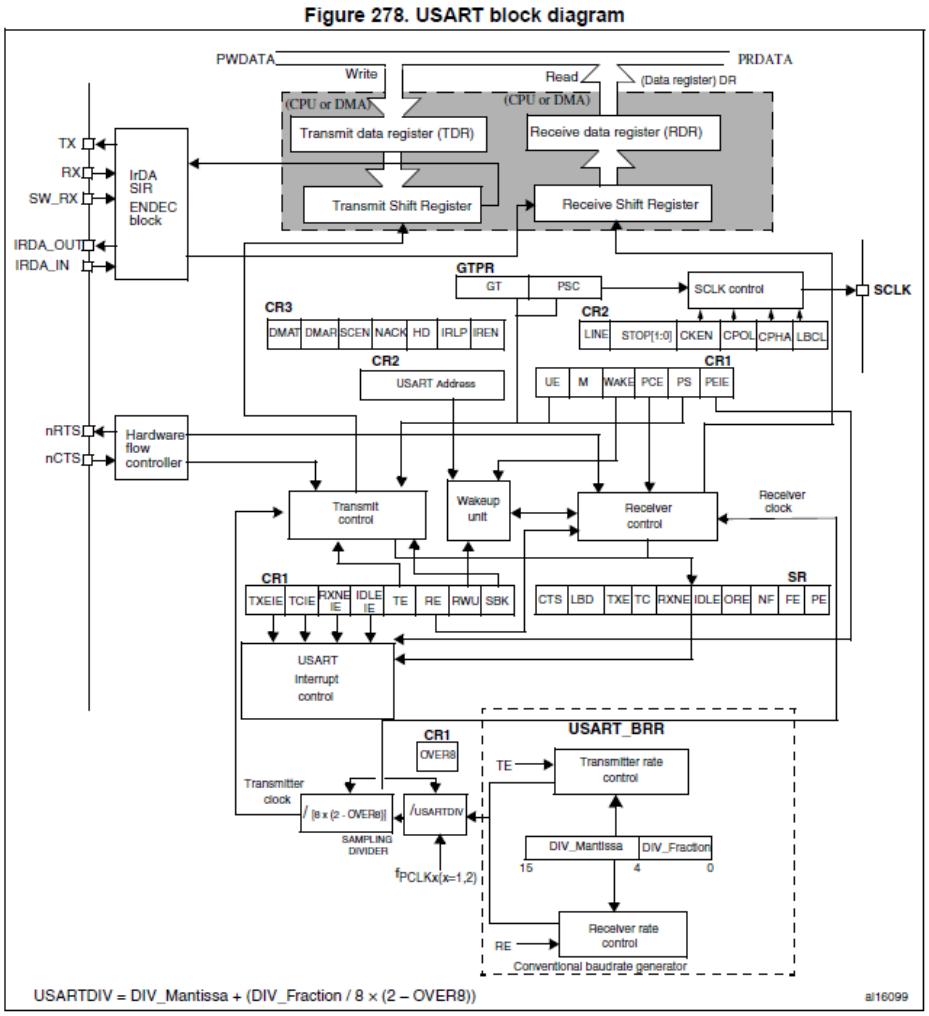


**Figure 1**
The STM32F446 USART block diagram

The STM32F446 has six UARTS – four of them 0, 1, 2, 3, and 6 are USARTS and can operate synchronously or asynchronously. The other two, 4, and 5 can only operate asynchronously.

A UART transmits and receives asynchronous data serially in the format shown in Figure 2.

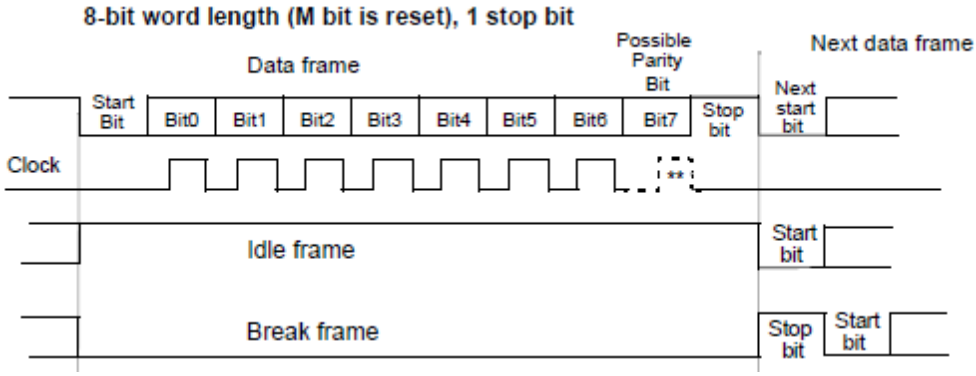


**Figure 2**
Format of serial data received and transmitted by a UART.

Note that:

1. The data is transmitted with the LSB first in time.
2. The serial line is normally high and a low going signal indicates a start bit for transmission.
3. A high going signal after 8 data bits indicates a stop bit.
4. Typical serial data consists of the 7-bit ASCII code plus a parity bit so that characters are sent in 10-bit packets consisting of a start bit, 7-bits of data, a parity bit, and a stop bit.
5. Most UARTs also support an 11-bit format which has a start bit, 8-bits of data, a parity bit, and a stop bit. This is seldom used.
6. The term UART implies *asynchronous* data transmission in which the clock is derived from the data. Many serial systems also use a USART which is capable of *synchronous* data transmission in which the clock is transmitted separately on another line.

For serial transmission the baud rate is the same as the serial bit rate. The lowest standard baud rate is 110 baud which typically has two stop bits instead of one making an information packet have 11-bits instead of 10. All other standard baud rates have only 10-bits/packet. These are: 600, 1200, 2400, 4800, 7200, 9600, 14400, 19200, 28800, 38400, 57600, 115200, and 230400. Divide these numbers by 10 to get the number of characters per second transmitted or received.

To transmit data we need an oscillator which will run at the bit rate we want to send and receive at. This is typically done by way of one of the on-board timers for a microcontroller. For example, at 1200 baud we would need a timer that runs at 1/1200 = 0.83333 milliseconds.

Since the clock is part of the signal for asynchronous transmission, it must be retrieved on the receiving end. So, for data sent at 1200 baud, the receiver waits for the start bit and half way through the start bit it starts its own 1200 bit/second oscillator. This oscillator clocks in the incoming bits until a stop bit is received. If the stop bit is not high, a "framing" error occurs.

The transmitter's clock and the receiver's clock are independent of one another but must be at approximately the same frequency. From the beginning of the start bit to the end of the stop bit there are 10 bit times. The two oscillators must be within half of a bit time of each other over this period or a framing error will occur. One-half of a bit in 10-bits is about 5%. In general, the clock rates of the transmitter and receiver are set to within ±2.5% of the agreed upon baud rate so that in the worst case the two clocks will be with 5% of one another.

There are four USARTs and two UARTS on the STM32F446 processor. The USARTs can be set up to be asynchronous but the UARTS cannot be set up to be synchronous. For these notes we are going to look only at the asynchronous mode of operation.

The UART registers are summarized in Table 1. This table is given in more detail in the User's Manual[1] pp. 936-847

**Table 1**
UART Registers

| **Register** | **Mnemonic** | **Function** |
|---|---|---|
| Status register | `USART_SR` | Has bits for transmit complete, transmit empty, data received, and various error bits |
| Data register | `USART_DR` | This is really two registers. If you read you get the received data and if you write the data is transmitted. Data is 8-bits |
| Baud Rate Register | `USART_BRR` | Bits 15 to 4 are the Baud Rate Divider int and bits 3 to 0 are the Baud Rate Divider fraction |
| Control Register 1 | `USART_CR1` | Bits for UART enable, oversample mode, TX and RX interrupt enable, Parity enable |
| Control Register 2 | `USART_CR2` | Has number of stop bits |
| Control Register 3 | `USART_CR3` | CTS & RTS enable, DMA enable, Smartcard enable |
| Guard Time&Prescale | `USART_GTPR` | Prescaler for low power mode |

*Status Register:* The status register has bits to determine when a data byte has been sent. You can use this bit as an interrupt or you can poll the bit to determine when to send the next character. Likewise, there is a bit which tells when a character has been received.

*Data Register:* This is actually two registers depending on whether you are reading or writing. If you are reading you get the register which holds the received data. If you are writing you write into the send data buffer. All of the data is 8-bits wide.

---

1 http://www.st.com/content/ccc/resource/technical/document/reference_manual/4d/ed/bc/89/b5/70/40/dc/DM00135183.pdf/files/DM00135183.pdf/jcr:content/translations/en.DM00135183.pdf

*Baud Rate Register:* This holds the divisor that divides the peripheral clock to get the baud rate. It is broken to a 12-bit integer part and a 4-bit fractional part. The baud rate is calculated from the following: $TxBaud = \frac{f_{pClk}}{8(2-over8)*BaudRateDiv}$

Where $f_{pClk}$ is the peripheral clock (usually 8 MHz) and *over8* is either 1 or 0 depending on whether you choose oversampling by 8 (over8 =1) or 16 (over8 = 0).

Solving this equation for the *BaudRateDiv* gives

$$BaudRateDiv = \frac{f_{pCl,}}{8(2-over8)*TxBaud}$$

For example, if the baud rate we want is 38400 and $f_{pClk}$ = 16 MHz and we do oversampling by 16 we get

$$BaudRateDiv = \frac{16000000}{8(2-0)*38400} = 26.0417$$

The integer part in binary as a 12-bit number: $26_{10}$ = 0000 0001 1010. The fractional part is 0.0417 = 0x0.5 + 0x0.25 + 0x0.125 + 0x 0.0625 + 1x0.03125 … = 0.00001… or, to 4-bits the fractional part is 0000. The 16-bit Baud Rate Divisor is therefore
0000 00010 1010 0000 = 0x1A0. The following MATLAB® code will do the math for you.

```
%BaudRate.m
fclk = 16000000;  %Peripheral clock
over8 = 0;
TxBaud = 38400;
BaudRateDiv = fclk/(TxBaud*(8*(2 - over8)));
disp(['Baud rate divisor in base 10 = ' num2str(BaudRateDiv)]);
fraction = BaudRateDiv - fix(BaudRateDiv);
hexBaudRateDiv = 16*fix(BaudRateDiv) + fix(fraction*16);
disp(['Baud rate divisor in hex is ' dec2hex(hexBaudRateDiv)])
actualBaudRate = fclk/(8*(2 - over8)*hexBaudRateDiv/16);
disp(['Actual Baud rate is ' num2str(actualBaudRate)]);
```

*Control Register 1:* The upper 16-bits of this register are not used. The remaining 16-bits have the following functions:

Bit 15 **OVER8**: Oversampling mode. 0: oversampling by 16, 1: oversampling by 8
Bit 14 Reserved, must be kept at reset value
Bit 13 **UE**: USART enable. 1 = enable
Bit 12 **M**: Word length 0: 1 Start bit, 8 Data bits, n Stop bit, 1: 1 Start bit, 9 Data bits, n Stop bit
Bit 11 **WAKE**: Wakeup method
Bit 10 **PCE**: Parity control enable. 0: Parity control disabled, 1: Parity control enabled
Bit 9 **PS**: Parity selection. 0: Even parity, 1: Odd parity
Bit 8 **PEIE**: PE interrupt enable. Parity error interrupt enable
Bit 7 **TXEIE**: TXE interrupt enable 1: An USART interrupt is generated whenever TXE=1
Bit 6 **TCIE**: Transmission complete interrupt enable 1: An USART interrupt is generated whenever TC=1 in the USART_SR register
Bit 5 **RXNEIE**: RXNE interrupt enable. 1: An USART interrupt is generated whenever ORE=1 or RXNE=1 in the USART_SR register
Bit 4 **IDLEIE**: IDLE interrupt enable

```
Bit 3 TE: Transmitter enable 1: Transmitter is enabled
Bit 2 RE: Receiver enable 1: Receiver is enabled and begins searching for a start
   bit
Bit 1 RWU: Receiver wakeup
Bit 0 SBK: Send break
```

*Control Register 2:* You can ignore this register and use the default values unless you are doing synchronous transmission.

*Control Register 3:* You can ignore this register and use the default values unless you are doing handshaking with the clear to send and ready to send bits.

*Guard Time and Prescale Register:* This register can be ignored unless you are using infrared communication.

*Example 1:*
Use USART 6 to transmit the 'U' character continuously at 38,400 baud. Note that 'U' in ASCII code is 0x55 = 0101 0101 so that the baud rate is easily verified on an oscilloscope.

********** Example 1 **********

```
//UARTEx1.c
/* Transmits the character 'U' at 38400 baud on USART6 PC6 forever.
*/
#include "stm32f446.h"
void ConfigureUART(unsigned int baudDivisor);
void UARTPutChar(char ch);
void SendMsg(const char msg[]);

int main()
  {  //Clock bits
   RCC_AHB1ENR |= 4;          //Bit 3 is GPIOC clock enable bit
   //RCC_APB1ENR |= (1 << 4);  //Enable peripheral timer for timer 6
   RCC_APB2ENR |= (1 << 5);  //Enable USART6 clock
   //UART PIN Bits
   GPIOC_AFRL = 0x88000000;  //Alternate Func PC 6-7 to USART6
   GPIOC_MODER |= 0x0A000;    //Bits 15-12 = 1010 for Alt Func on PC6, PC7
   //OTYPER register resets to 0 so it is push/pull by default
   GPIOC_OSPEEDER |= 0x3000;  //Bits 7-6 = 11 for high speed on PC6
   //PUPDR defaults to no pull up no pull down
   //USART1 bits
   GPIOA_MODER |= 0x40000; //Bits 18-19 are 01 for digital output on PA9
   GPIOA_OSPEEDER |= 0xC0000; //Bits 18-19 are 11 for high speed on PA9
   //PA10 is input by default
   ConfigureUART(0x1A0);
   while(1)
     UARTPutChar('U');
  }
//
void ConfigureUART(unsigned int baudDivisor)
  {USART6_CR1 = 0;        //Disable during set up. Wd len = 8, Parity = off
   USART6_BRR = baudDivisor;  //Set up baud rate
   USART6_CR2 = 0;            //1 stop bit
   USART6_CR1 = 0x200C;
   USART6_CR3 = 0;            //Disable interrupts and DMA
  }
//
void UARTPutChar(char ch)
  {//Wait for empty flag
   while((USART6_SR & 0x80) == 0);
   USART6_DR = ch;
  }
//
void SendMsg(const char msg[])
  {int i = 0;
   while(msg[i] != 0)
      {UARTPutChar(msg[i]);
       i++;
      }
  }
```