# CPE 323: UART Serial Communication

**Aleksandar Milenković**
Email: milenka@uah.edu
Web: http://www.ece.uah.edu/~milenka

## Objective:

*Introduce serial communication, specifically UART mode of serial communication and Universal Serial Communication Interface (USCI) peripheral.*

## Contents

Objective: ..... 1
Contents ..... 1
1 Introduction ..... 2
2 Universal Asynchronous Receiver/Transmitter (UART) ..... 2
3 USCI: UART Mode ..... 4
3.1 USCI Initialization: UART Mode ..... 6
3.2 USCI UART Error Conditions ..... 6
3.3 UART Baud Rate Generation ..... 7
4 Examples ..... 9
5 References ..... 13

# 1 Introduction

Ability to communicate data is one of the core functionalities of any computer systems (the others are sensing the environment, processing data, and storing data). When building embedded systems we often need to provide means to exchange data between different components on a single board (e.g., between a microcontroller and a sensor), between different embedded systems (e.g., controller units in your car are all connected through a bus) or between an embedded computer system and a workstation. To meet a diverse set of requirements and design constraints, a multitude of communication protocols have been developed and used over time.

We can classify communication techniques in embedded systems using different criteria. Depending on the medium used to transfer data, the communication can be wired when data is communicated by sending logic signals through wires, or wireless when data is turned into radio waves through antennas and transferred wirelessly. Here we will focus on wired communication. Based on the number of bits sent or received at a time, we can distinguish between serial communication, where one bit is sent/received at a time, and parallel communication, where multiple bits (>1) are sent/received at a time. Serial communication limits the number of bits that can be communicated in unit of time (typically 1 bit of data is sent/received each clock cycle), but it is less expensive (e.g., fewer traces are needed to be routed on the printed circuit board which reduces the size and the cost of manufacturing or fewer wires are needed to connect to external system). With parallel communication we can transfer more data bits at a time, but it will cost us more. Next, based on the flow of data, communication can be unidirectional, a.k.a. simplex, where data always flow in one direction, e.g., from device A to device B, or bidirectional, a.k.a. duplex, where data can flow in both directions. Further, duplex communication can be half-duplex – data can flow in both directions but only in one direction at a time because the same set of wires is shared to carry information from device A to B and from device B to A, or full-duplex – data can flow in both directions at the same time because separate sets of wires are provided for data flow in each direction. Finally, depending on whether communicating parties share a common clock, communication can be asynchronous when there is no common clock or synchronous where the communicating parties share a common clock.

In this section we exclusively focus on wired serial communication protocols routinely used in embedded systems, such as Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), and Inter-Integrated Circuit Bus ($I^2C$). MSP430 family of devices provide several communication peripheral devices that include hardware support for serial communication. They are Universal Serial Communication Interface (USCI), Universal Serial Interface (USI), and Universal Synchronous/Asynchronous Receiver/Transmitter (USART).

# 2 Universal Asynchronous Receiver/Transmitter (UART)

Asynchronous serial communication is very popular type of communication in embedded systems. It can be used to exchange data between components on the same board or between different systems.

Figure 1 illustrates a system view of UART style of communication between two devices, called A and B. The devices are physically connected using two wires that carry information from A to B (top wire) and from B to A (middle wire). The communicating parties need to share a common ground (Gnd). In this configuration we have a full-duplex asynchronous communication. Each device requires two ports: TxD (Transmit Data) for data transmission and RxD (Receive Data) for receiving data. The TxD port of A is connected to the RxD of B and RxD of A is connected to the TxD of B.

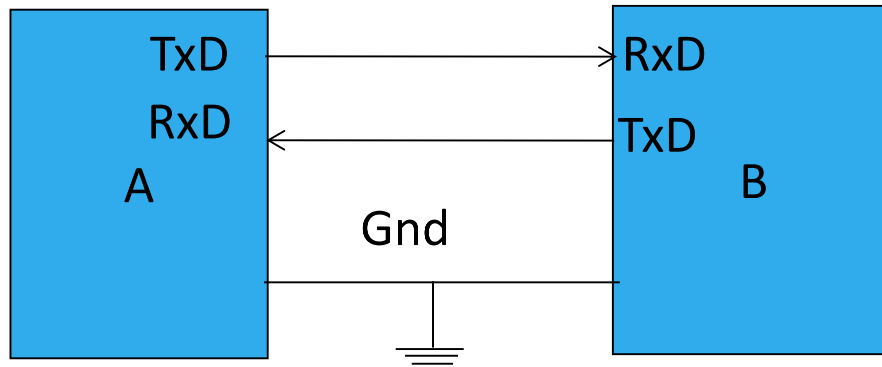


**Figure 1. UART communication: a system view**

UART communication is asynchronous because devices A and B do not have a common clock. In addition, they can be completely different types of devices, each with their own clock subsystem. UART communication is typically character-oriented, where 8-bit characters are divided into individual bits that are sent one by one from the transmitter time. The individual bits are grouped into characters at the receiving side.

How does UART communication work? Both the transmitter and receiver should properly initialize their respective communication interfaces for UART type of communication. The initialization involves steps to set up the baud rates (or bit rates) that defines at what speed the communication interfaces transmit/receive data (they should be the same for the transmitter and receiver), format of characters, and how to handle errors in communication. Upon initialization, the transmitter device (e.g., A) writes a byte of data into a TxBUF (transmit data buffer) register of its serial communication interface. This character is then typically moved into a serial shift register and control logic of the communication interface takes care of transmitting data, one bit at a time. The communication interface of the receiver shifts in one bit of data at a time in its shift register. When all bits in a character are received, the character is moved into a RxBUF (receive data buffer) register and a flag is set to indicate that a new character has been received.

How does a receiver know that new transmission is underway? To answer this question, let us take a look how the signal look like during transmission of one character as observed on the TxD port pin. Figure 2 shows a format of a character. When there is no transmission the TxD port is

held a logic ‘1’. When a new character transmission starts, a START (ST) bit is transmitted – one bit period at a logic ‘0’. Then the character bits are sent (D0-D7), followed by optional the address bit (AD), parity bit (PA), and one or two stop bits. Thus, to transmit one 8-bit character with a parity bit and 2 stop bits, we need in total 1 (start) + 8 (data) + 1 (parity) + 2 (stop) = 12 bits or $12 \cdot T_{BITCLK}$ in time. The serial communication interface is responsible for inserting start, stop, and parity bits, but both transmitter and receiver should use the same character format.

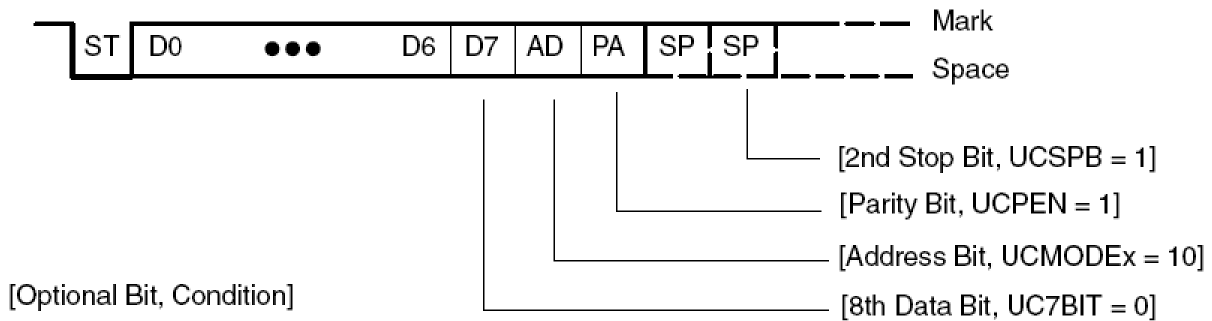


**Figure 2. Format of a character**

# 3 USCI: UART Mode

The Universal Serial Communication Interface or USCI for short is a TI peripheral that supports several serial synchronous and asynchronous communication protocols including UART mode. The UART mode supports several configurable parameters as follows:

- 7 or 8-bit data
- odd, even parity or no parity at all
- MSB or LSB bit is sent first
- programmable baud rate
- status flags for error detection and suppression
- receiver start-edge detection for auto-wake up from LPMx modes
- support for multiprocessing modes.

Figure 3 shows a block diagram of the USCI. We can identify the receiver block on the top with the receiver data buffer (UC0RXBUF), the receive shift register (not visible to programmers), and connection to the UC0RX port pin. The transmitter block at the bottom includes the transmit data buffer (UCA0TXBUF), the transmit shift register (not visible to programmers), and a connection to the UC0TX port pin. The middle block is baud rate generator that takes one of the input clocks (UC0CLK, ACLK, SMCLK) and generates the communication bit clocks for both the transmit and receive blocks.

*Figure 19–1. USCI_Ax Block Diagram: UART Mode (UCSYNC = 0)*

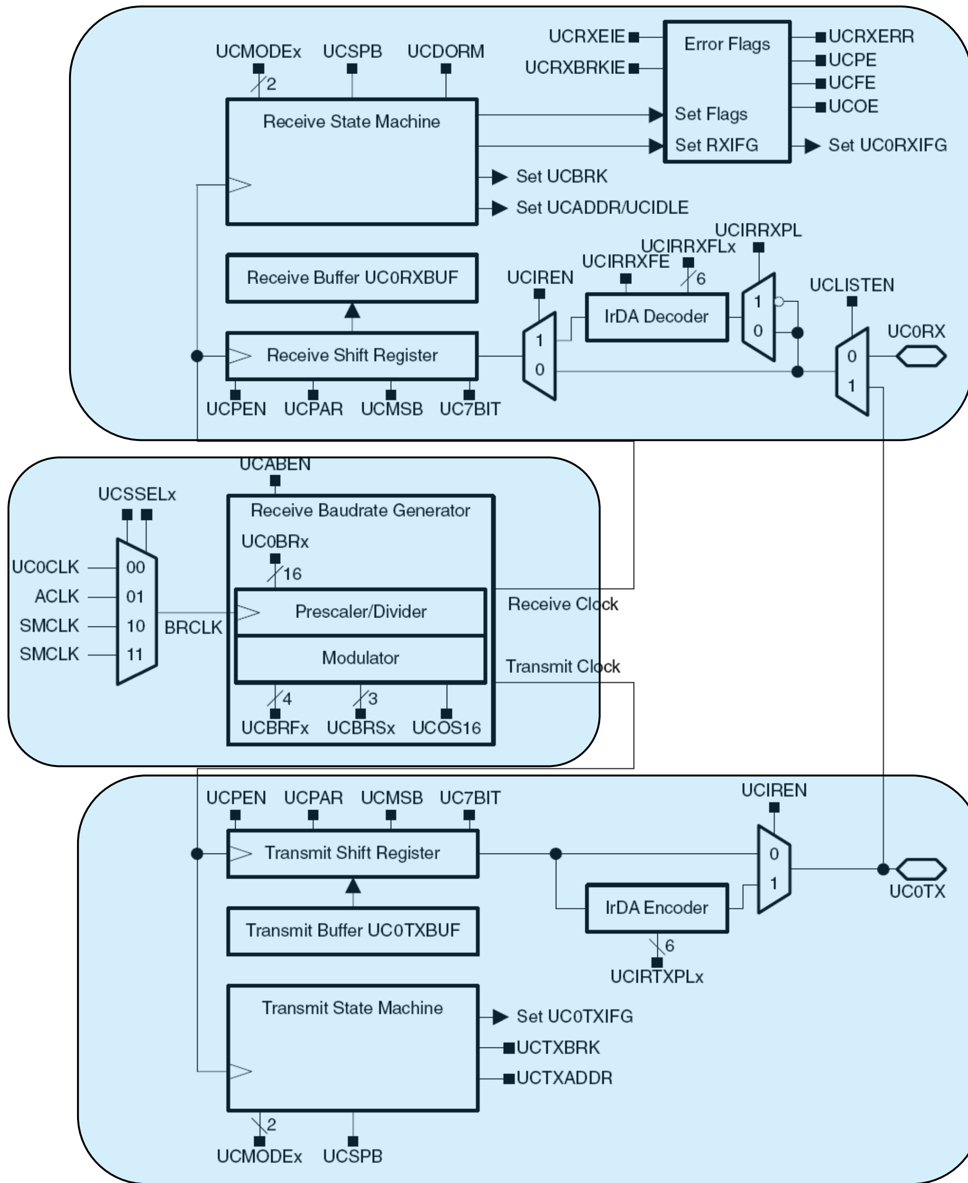


**Figure 3. USCI block diagram: UART mode**

USCI communication interface contains two channels (A0 and A1) and USCI registers visible to programmers are shown in Figure 4. USCI is an 8-bit peripheral devices and all registers and 8-bit long. The notable registers are two control registers (UCA0CTL0 and UCA0CTL1), baud rate control registers (UCA0BR0 and UCA0BR1), a modulation control register (UMA0MCTL), a status register (UCA0STAT), the receive buffer (UC0RXBUF), and the transmit buffer (UC0TXBUF). In addition, the system-wide registers IFG2 and IE2 contain flags of interest for the USCI interface. The channel 1 contains a similar set of registers. To learn more about the format of these registers and their functionality, please consult the user's manual (Chapter 19: Universal Serial Communication Interface: UART Mode).

*Table 19–6. USCI_A0 Control and Status Registers*

| Register | Short Form | Register Type | Address | Initial State |
|---|---|---|---|---|
| USCI_A0 control register 0 | UCA0CTL0 | Read/write | 060h | Reset with PUC |
| USCI_A0 control register 1 | UCA0CTL1 | Read/write | 061h | 001h with PUC |
| USCI_A0 Baud rate control register 0 | UCA0BR0 | Read/write | 062h | Reset with PUC |
| USCI_A0 Baud rate control register 1 | UCA0BR1 | Read/write | 063h | Reset with PUC |
| USCI_A0 modulation control register | UCA0MCTL | Read/write | 064h | Reset with PUC |
| USCI_A0 status register | UCA0STAT | Read/write | 065h | Reset with PUC |
| USCI_A0 Receive buffer register | UCA0RXBUF | Read | 066h | Reset with PUC |
| USCI_A0 Transmit buffer register | UCA0TXBUF | Read/write | 067h | Reset with PUC |
| USCI_A0 Auto Baud control register | UCA0ABCTL | Read/write | 05Dh | Reset with PUC |
| USCI_A0 IrDA Transmit control register | UCA0IRTCTL | Read/write | 05Eh | Reset with PUC |
| USCI_A0 IrDA Receive control register | UCA0IRRCTL | Read/write | 05Fh | Reset with PUC |
| SFR interrupt enable register 2 | IE2 | Read/write | 001h | Reset with PUC |
| SFR interrupt flag register 2 | IFG2 | Read/write | 003h | 00Ah with PUC |

**Figure 4. USCI_A0 Control and Status Registers**

## 3.1 USCI Initialization: UART Mode

To initialize the USCI in UART mode the following sequence of steps is recommended:

1. Set UCSWRST bit (software reset: BIS.B #UCSWRST, &UCAxCTL1) to reset the USCI state machine
2. Initialize all USCI registers with UCSWRST=1 (baud rate control, modulation control, control registers)
3. Configure ports (TxD, RxD special function)
4. Clear UCSWRST (BIS.B #UCSWRST, &UCAxCTL1)
5. Enable interrupts (optional) by setting UCAxRXIE and UCAxTXIE.

The interrupt vector table contains separate entries for interrupts from the receiver and transmitter in the USCI.

## 3.2 USCI UART Error Conditions

The USCI peripheral can detect framing errors, parity errors, overrun errors, and break conditions when receiving characters as shown in Figure 5. The USCI can be configured to generate an interrupt when received erroneous character conditions are detected (UCRXEIE bit in the UCAxCTL1 register). When UCFE, UCPE, UCOE, and UCBRK or UCRXERR is set, the bit remains set until user software resets it or UCAxRXBUF is read. To detect overflows (a new character is received while the previous one has not been read yet) the following flow is recommended. After a character is received and UCAxRXIFG is set, first read UCAxSTAT to check the error flags including OCOE. Read UCAxRXBUF next. This will clear all error flags except UCOE if UCAxRXBUF

was overwritten between the read access to UCAxSTAT and the read access to UCAxRXBUF. To detect this condition (buffer overwrite between these two reads), the OCOE bit should be read after reading UCAxRXBUF.

| Error Condition | Error Flag | Description |
|---|---|---|
| Framing error | UCFE | A framing error occurs when a low stop bit is detected. When two stop bits are used, both stop bits are checked for framing error. When a framing error is detected, the UCFE bit is set. |
| Parity error | UCPE | A parity error is a mismatch between the number of 1s in a character and the value of the parity bit. When an address bit is included in the character, it is included in the parity calculation. When a parity error is detected, the UCPE bit is set. |
| Receive overrun | UCOE | An overrun error occurs when a character is loaded into UCAxRXBUF before the prior character has been read. When an overrun occurs, the UCOE bit is set. |
| Break condition | UCBRK | When not using automatic baud rate detection, a break is detected when all data, parity, and stop bits are low. When a break condition is detected, the UCBRK bit is set. A break condition can also set the interrupt flag UCAxRXIFG if the break interrupt enable UCBRKIE bit is set. |

**Figure 5. Receive Error Conditions**

## 3.3 UART Baud Rate Generation

The USCI baud rate generator can produce standard baud rates from non-standard source frequencies. It provides two modes of operation: low-frequency mode (UCOS16 = 0) and over-sampling mode (UCOS16 = 1).

The low-frequency mode allows generation of baud rates from low frequency clock sources that reduce energy consumed by the communication interface. For example, we may have $F_{BAUD}$=9600 bps, and the source clock is BRCLK=ACLK= 32,768 Hz. By dividing 32,768 with 9,600 we get N=3.41. The challenge is that the baud rate divider cannot use fractions. Instead we initialize the baud rate registers UCBRx = INT(N) = 3, and the UCBRSx field UCBRSx = round((N – INT(N))*8)=3. THe UCBRSx 3-bit field controls the second modulation stage. The way this works is as follows: 5 bits (or 8 – UCBRSx bits) will have duration of 3 source clock periods (BRCLK) and 3 bits (UCBRSx bits) will have duration of 4 (N+1 in general) source clock periods, BRCLK, providing an average to be close to 3.41. Thus, some bits during transmission take 3 BRCLK periods and some take 4 BRCLK periods. The duration is modulated in such a way to minimize the error in communication

from the targeted bit rate for each bit period. Figure 6 shows common combinations of clock sources and baud rates and how to set the baud rate control registers.

*Table 19–4.Commonly Used Baud Rates, Settings, and Errors, UCOS16 = 0*

| BRCLK Frequency [Hz] | Baud Rate [Baud] | UCBRx | UCBRSx | UCBRFx | Max TX Error [%] | | Max RX Error [%] | |
|---|---|---|---|---|---|---|---|---|
| **32,768** | **1200** | 27 | 2 | 0 | −2.8 | 1.4 | −5.9 | 2.0 |
| **32,768** | **2400** | 13 | 6 | 0 | −4.8 | 6.0 | −9.7 | 8.3 |
| **32,768** | **4800** | 6 | 7 | 0 | −12.1 | 5.7 | −13.4 | 19.0 |
| **32,768** | **9600** | 3 | 3 | 0 | −21.1 | 15.2 | −44.3 | 21.3 |
| **1,000,000** | **9600** | 104 | 1 | 0 | −0.5 | 0.6 | −0.9 | 1.2 |
| **1,000,000** | **19200** | 52 | 0 | 0 | −1.8 | 0 | −2.6 | 0.9 |
| **1,000,000** | **38400** | 26 | 0 | 0 | −1.8 | 0 | −3.6 | 1.8 |
| **1,000,000** | **57600** | 17 | 3 | 0 | −2.1 | 4.8 | −6.8 | 5.8 |
| **1,000,000** | **115200** | 8 | 6 | 0 | −7.8 | 6.4 | −9.7 | 16.1 |
| **1,048,576** | **9600** | 109 | 2 | 0 | −0.2 | 0.7 | −1.0 | 0.8 |
| **1,048,576** | **19200** | 54 | 5 | 0 | −1.1 | 1.0 | −1.5 | 2.5 |
| **1,048,576** | **38400** | 27 | 2 | 0 | −2.8 | 1.4 | −5.9 | 2.0 |
| **1,048,576** | **57600** | 18 | 1 | 0 | −4.6 | 3.3 | −6.8 | 6.6 |
| **1,048,576** | **115200** | 9 | 1 | 0 | −1.1 | 10.7 | −11.5 | 11.3 |
| **4,000,000** | **9600** | 416 | 6 | 0 | −0.2 | 0.2 | −0.2 | 0.4 |
| **4,000,000** | **19200** | 208 | 3 | 0 | −0.2 | 0.5 | −0.3 | 0.8 |
| **4,000,000** | **38400** | 104 | 1 | 0 | −0.5 | 0.6 | −0.9 | 1.2 |
| **4,000,000** | **57600** | 69 | 4 | 0 | −0.6 | 0.8 | −1.8 | 1.1 |
| **4,000,000** | **115200** | 34 | 6 | 0 | −2.1 | 0.6 | −2.5 | 3.1 |
| **4,000,000** | **230400** | 17 | 3 | 0 | −2.1 | 4.8 | −6.8 | 5.8 |
| **8,000,000** | **9600** | 833 | 2 | 0 | −0.1 | 0 | −0.2 | 0.1 |
| **8,000,000** | **19200** | 416 | 6 | 0 | −0.2 | 0.2 | −0.2 | 0.4 |
| **8,000,000** | **38400** | 208 | 3 | 0 | −0.2 | 0.5 | −0.3 | 0.8 |
| **8,000,000** | **57600** | 138 | 7 | 0 | −0.7 | 0 | −0.8 | 0.6 |
| **8,000,000** | **115200** | 69 | 4 | 0 | −0.6 | 0.8 | −1.8 | 1.1 |

**Figure 6. Commonly user baud rates and settings in low-frequency mode**

For oversampling mode, the baud rate generator first generates a clock $f_{BIT16CLK}$ that is 16 times $f_{baud}$. To illustrate settings for the baud rate generator, let us assume that our target baud rate is $f_{baud}$ = 9600 Hz and the source clock is $f_{BRCLK} = 2^{20}$ Hz. One bit period, $T_{baud}$, thus contain N = $f_{BRCLK}/f_{baud}$ = 109.22 > 16 source clock periods. Dividing N with 16 we get 6.83, i.e., one period of $T_{BIT16CLK}$ contain 6.83 source clock periods. In this case the baud rate register UCABRx is set to INT(N/16) = 6, and the first stage modulator to UCBRFx= round ( (N/16 – INT(N/16))*16) = 13. The meaning of this is as follows: out of 16 bit periods $T_{BIT16CLK}$ in one $T_{BAUD}$, 13 BIT16CLK cycles will have 7 (or N+1 in general) BRCLK clocks and 3 BIT16CLK cycles will have 6 (or N in general) BRCLK clocks, giving on average 6.83 BRCLK clock cycles. The modulator ensures that these different BIT16CLK clocks are spread in such a way to minimize error in communication. Figure 7 shows how to setup baud rate control registers in oversampling mode for common combinations of clock sources and baud rates.

*Table 19–5.Commonly Used Baud Rates, Settings, and Errors, UCOS16 = 1*

| BRCLK frequency [Hz] | Baud Rate [Baud] | UCBRx | UCBRSx | UCBRFx | Max. TX Error [%] | | Max. RX Error [%] | |
|---|---|---|---|---|---|---|---|---|
| 1,000,000 | 9600 | 6 | 0 | 8 | −1.8 | 0 | −2.2 | 0.4 |
| 1,000,000 | 19200 | 3 | 0 | 4 | −1.8 | 0 | −2.6 | 0.9 |
| 1,048,576 | 9600 | 6 | 0 | 13 | −2.3 | 0 | −2.2 | 0.8 |
| 1,048,576 | 19200 | 3 | 1 | 6 | −4.6 | 3.2 | −5.0 | 4.7 |
| 4,000,000 | 9600 | 26 | 0 | 1 | 0 | 0.9 | 0 | 1.1 |
| 4,000,000 | 19200 | 13 | 0 | 0 | −1.8 | 0 | −1.9 | 0.2 |
| 4,000,000 | 38400 | 6 | 0 | 8 | −1.8 | 0 | −2.2 | 0.4 |
| 4,000,000 | 57600 | 4 | 5 | 3 | −3.5 | 3.2 | −1.8 | 6.4 |
| 4,000,000 | 115200 | 2 | 3 | 2 | −2.1 | 4.8 | −2.5 | 7.3 |
| 8,000,000 | 9600 | 52 | 0 | 1 | −0.4 | 0 | −0.4 | 0.1 |
| 8,000,000 | 19200 | 26 | 0 | 1 | 0 | 0.9 | 0 | 1.1 |
| 8,000,000 | 38400 | 13 | 0 | 0 | −1.8 | 0 | −1.9 | 0.2 |
| 8,000,000 | 57600 | 8 | 0 | 11 | 0 | 0.88 | 0 | 1.6 |
| 8,000,000 | 115200 | 4 | 5 | 3 | −3.5 | 3.2 | −1.8 | 6.4 |
| 8,000,000 | 230400 | 2 | 3 | 2 | −2.1 | 4.8 | −2.5 | 7.3 |
| 12,000,000 | 9600 | 78 | 0 | 2 | 0 | 0 | −0.05 | 0.05 |
| 12,000,000 | 19200 | 39 | 0 | 1 | 0 | 0 | 0 | 0.2 |
| 12,000,000 | 38400 | 19 | 0 | 8 | −1.8 | 0 | −1.8 | 0.1 |
| 12,000,000 | 57600 | 13 | 0 | 0 | −1.8 | 0 | −1.9 | 0.2 |
| 12,000,000 | 115200 | 6 | 0 | 8 | −1.8 | 0 | −2.2 | 0.4 |
| 12,000,000 | 230400 | 3 | 0 | 4 | −1.8 | 0 | −2.6 | 0.9 |

**Figure 7. Commonly user baud rates and settings in oversampling mode**

# 4 Examples

```
/*------------------------------------------------------------------------------
 * File:          Lab8_D1.c
 * Function:      Echo a received character, using polling.
 * Description:   This program echos the character received from UART back to UART.
 *                Toggle LED4 with every received character.
 *                Baud rate: low-frequency (UCOS16=0);
 *                1048576/115200 = ~9.1 (0x0009|0x01)
 * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
 *
 * Instructions: Set the following parameters in putty
 * Port : COM1
 * Baud rate : 115200
```

```
13   * Data bits: 8
14   * Parity: None
15   * Stop bits: 1
16   * Flow Control: None
17   *
18   *    MSP430xG461x
19   *   -----------------
20   * /|\ |            XIN|-
21   *  |  |               | 32kHz
22   *  |--|RST        XOUT|-
23   *     |               |
24   *     |   P2.4/UCA0TXD|------------>
25   *     |               | 115200 - 8N1
26   *     |   P2.5/UCA0RXD|<------------
27   *     |           P5.1|----> LED4
28   *
29   * Input:     None (Type characters in putty/MobaXterm/hyperterminal)
30   * Output:    Character echoed at UART
31   * Author:    A. Milenkovic, milenkovic@computer.org
32   * Date:      October 2018
33   *--------------------------------------------------------------------------------*/
34  #include <msp430xG46x.h>
35  void main(void) {
36    WDTCTL = WDTPW+WDTHOLD;         // Stop WDT
37    P5DIR |= BIT1;                  // Set P5.1 to be output
38    UCA0CTL1 |= UCSWRST;            // Set software reset during initialization
39    P2SEL |= BIT4 + BIT5;           // P2.4,5 = USCI_A0 RXD/TXD
40    UCA0CTL1 |= UCSSEL_2;           // BRCLK=SMCLK
41    UCA0BR0 = 0x09;                 // 1MHz/115200 (lower byte)
42    UCA0BR1 = 0x00;                 // 1MHz/115200 (upper byte)
43    UCA0MCTL |= BIT2;               // Modulation (UCBRS0=0x01)(UCOS16=0)
44    UCA0CTL1 &= ~UCSWRST;           // **Initialize USCI state machine**
45    while (1) {
46         while(!(IFG2&UCA0RXIFG));  // Wait for a new character
47         // new character is here in UCA0RXBUF
48         while(!(IFG2&UCA0TXIFG));  // Wait until TXBUF is free
49         UCA0TXBUF = UCA0RXBUF;     // TXBUF <= RXBUF (echo)
50         P5OUT ^= BIT1;             // Toggle LED4
51    }
52  }
```

**Figure 8. Echo a character using polling**

```
1   /*--------------------------------------------------------------------------------
2    * File:         Lab8_D2.c
3    * Function:     Echo a received character, using receiver ISR.
4    * Description:  This program echos the character received from UART back to UART.
5    *               Toggle LED4 with every received character.
6    *               Baud rate: low-frequency (UCOS16=0);
7    *               1048576/115200 = ~9.1 (0x0009|0x01)
8    * Clocks:       ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
9    *
10   * Instructions: Set the following parameters in putty
```

```
11   * Port : COM1
12   * Baud rate : 115200
13   * Data bits: 8
14   * Parity: None
15   * Stop bits: 1
16   * Flow Control: None
17   *
18   *    MSP430xG461x
19   *   -----------------
20   * /|\ |            XIN|-
21   *  |  |               | 32kHz
22   *  |--|RST        XOUT|-
23   *     |               |
24   *     |   P2.4/UCA0TXD|------------>
25   *     |               | 115200 - 8N1
26   *     |   P2.5/UCA0RXD|<------------
27   *     |           P5.1|----> LED4
28   *
29   * Input:     None (Type characters in putty/MobaXterm/hyperterminal)
30   * Output:    Character echoed at UART
31   * Author:    A. Milenkovic, milenkovic@computer.org
32   * Date:      October 2018
33   *-------------------------------------------------------------------------------*/
34  #include <msp430xG46x.h>
35  void main(void) {
36    WDTCTL = WDTPW+WDTHOLD;           // Stop WDT
37    P5DIR |= BIT1;                    // Set P5.1 to be output
38    UCA0CTL1 |= UCSWRST;              // Set software reset during initialization
39    P2SEL |= BIT4 + BIT5;             // P2.4,5 = USCI_A0 RXD/TXD
40    UCA0CTL1 |= UCSSEL_2;             // BRCLK=SMCLK
41    UCA0BR0 = 0x09;                   // 1MHz/115200 (lower byte)
42    UCA0BR1 = 0x00;                   // 1MHz/115200 (upper byte)
43    UCA0MCTL |= BIT2;                 // Modulation (UCBRS0=0x01)(UCOS16=0)
44    UCA0CTL1 &= ~UCSWRST;             // **Initialize USCI state machine**
45    IE2 |= UCA0RXIE;                  // Enable USCI_A0 RX interrupt
46    _BIS_SR(LPM0_bits + GIE);         // Enter LPM0, interrupts enabled
47  }
48
49  // Echo back RXed character, confirm TX buffer is ready first
50  #pragma vector=USCIAB0RX_VECTOR
51  __interrupt void USCIA0RX_ISR (void) {
52    while(!(IFG2&UCA0TXIFG));         // Wait until can transmit
53    UCA0TXBUF = UCA0RXBUF;            // TXBUF <= RXBUF
54    P5OUT^=BIT1;                      // Toggle LED4
55  }
```

**Figure 9. Echo a character using receiver ISR**

```
1   /*------------------------------------------------------------------------------
2    * File:          Lab8_D3.c
3    * Function:      Displays real-time clock in serial communication client.
4    * Description:   This program maintains real-time clock and sends time
5    *                (10 times a second) to the workstation through
```

```
6    *                  a serial asynchronous link (UART).
7    *                  The time is displayed as follows: "sssss:tsec".
8    *
9    *                  Baud rate divider with 1048576hz = 1048576/19200 = ~54
10   * Clocks:          ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO = 1048576Hz
11   * Instructions:    Set the following parameters in putty/hyperterminal
12   * Port : COM1
13   * Baud rate : 19200
14   * Data bits: 8
15   * Parity: None
16   * Stop bits: 1
17   * Flow Control: None
18   *
19   *        MSP430xG461x
20   *     -----------------
21   * /|\ |             XIN|-
22   *  |  |                | 32kHz
23   *  |--|RST         XOUT|-
24   *     |                |
25   *     |    P2.4/UCA0TXD|------------>
26   *     |                | 19200 - 8N1
27   *     |    P2.5/UCA0RXD|<------------
28   *     |            P5.1|----> LED4
29   *
30   * Author:        A. Milenkovic, milenkovic@computer.org
31   * Date:          October 2018
32  --------------------------------------------------------------------------------*/
33  #include <msp430xG46x.h>
34  #include <stdio.h>
35
36  // Current time variables
37  unsigned int sec = 0;           // Seconds
38  unsigned int tsec = 0;          // 1/10 second
39  char Time[8];                   // String to keep current time
40
41  //Function Declarations
42  void SetTime(void);
43  void SendTime(void);
44
45  void UART_Initialize(void) {
46      UCA0CTL1 |= UCSWRST;            // Set software reset during initialization
47      P2SEL |= BIT4 + BIT5;           // Set UC0TXD and UC0RXD to transmit and receive
48      UCA0CTL0 = 0;                   // USCI_A0 control register
49      UCA0CTL1 |= UCSSEL_2;           // Clock source SMCLK
50      UCA0BR0 = 54;                   // 1048576 Hz / 19200 = 54 | 5
51      UCA0BR1 = 0;
52      UCA0MCTL = 0x0A;                // Modulation
53      UCA0CTL1 &= ~UCSWRST;           // Clear software reset
54  }
55
56  // Sets the real-time clock variables
57  void SetTime(void) {
58      tsec++;
59      if (tsec == 10){
60          tsec = 0;
```

```
61            sec++;
62            P5OUT ^= BIT1;
63        }
64    }
65
66    // Sends the time through a serial link
67    void SendTime(void) {
68        int i;
69
70        sprintf(Time, "%05d:%01d", sec, tsec);  // Prints time to a string
71        for (i = 0; i < 8; i++) {               // Send character by character
72            while (!(IFG2 & UCA0TXIFG));
73            UCA0TXBUF = Time[i];
74        }
75        while (!(IFG2 & UCA0TXIFG));
76        UCA0TXBUF = 0x0D;                      // Carriage Return
77    }
78
79    void main(void) {
80        WDTCTL = WDTPW + WDTHOLD;           // Stop watchdog timer
81        UART_Initialize();                  // Initialize UART
82        //Initialize Timer A to measure 1/10 sec
83        TACTL = TASSEL_2 + MC_1 + ID_3;     // Select SMCLK/8 and up mode
84        TACCR0 = 13107;                     // 100ms interval
85        TACCTL0 = CCIE;                     // Capture/compare interrupt enable
86        P5DIR |= BIT1;                      // P5.1 is output;
87        while (1) {                         // Main loop
88            _BIS_SR(LPM0_bits + GIE);       // Enter LPM0 w/ interrupts
89            SendTime();                     // Send Time to HyperTerminal
90        }
91    }
92
93    // Interrupt for the timer
94    #pragma vector=TIMERA0_VECTOR
95    __interrupt void TIMERA_ISA(void) {
96        SetTime();                         // Set Clock
97        _BIC_SR_IRQ(LPM0_bits);            // Clear LPM0 bits from 0(SR)
98    }
```

**Figure 10. Display real-time clock**

# 5 References