CPE 323
MODULE 10
UART Serial Communication

Aleksandar Milenković

Email: milenka@uah.edu

Web: http://www.ece.uah.edu/~milenka

Overview

This module introduces various aspects of communication in embedded systems. You will learn about types of communication (parallel vs. serial, asynchronous vs. synchronous, unidirectional vs. bidirectional) and communication interfaces used in the MSP430 family of microcontrollers. A special emphasis is on serial communication protocols: UART, SPI, and I2C.

Objectives

- *Learners will understand hardware and software aspects of serial communication*
- *Learners will be able to configure and interact with serial communication interfaces*
- *Learners will be able to evaluate pros and cons of each serial communication protocol (speed complexity)*

Contents

1 Introduction

Ability to communicate data is one of the core functionalities of all embedded 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 computer systems (e.g., controller units in your car are all connected through a Controller Area Network 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 because fewer traces need to be routed on the printed circuit board which reduces the size and the manufacturing cost 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 (A to B and B to A). 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 module 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^{2}C). The MSP430 family of devices provide several communication peripherals 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).

`Things to remember 1-1. Data communication in embedded systems.`

Data communication is one of the key functionalities in embedded systems. It can be classified using different criteria: wired vs. wireless, serial vs. parallel, asynchronous vs. synchronous, simplex vs. half-duplex vs. full-duplex. In embedded systems a number of protocols are used to connect different components on a board (intra-board) as well as different systems.

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.

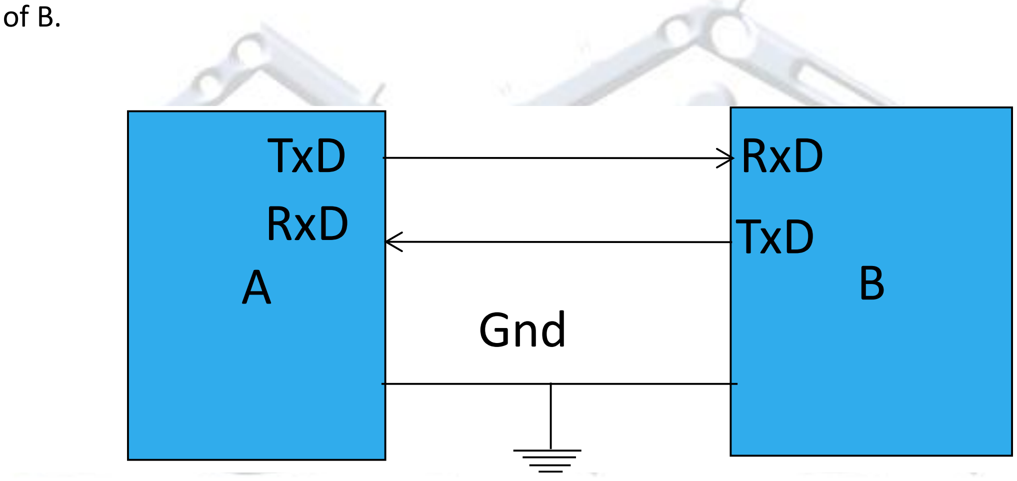


Figure 1. UART communication: a system view.

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 up to 8-bit characters are divided into individual bits that are sent one by one from the transmitter. 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 define 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 shift register and the 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 address bit (AD) and a parity bit (PA). The character is terminated by 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. The transmission takes $12 \cdot T_{BITCLK}$. The serial communication interface is responsible for inserting the start, stop, and parity bits, but both transmitter and receiver should use the same character format.

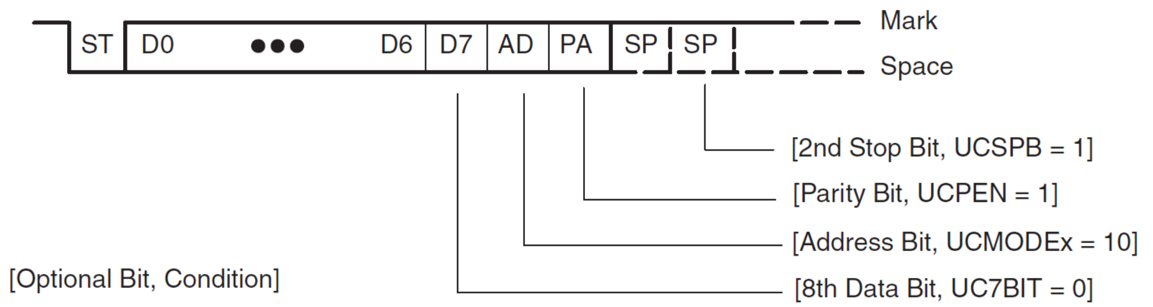

ST
D0
●●●
D6
D7
AD
PA
SP
SP
Mark
Space
[2nd Stop Bit, UCSPB = 1]
[Parity Bit, UCPEN = 1]
[Address Bit, UCMODEx = 10]
[8th Data Bit, UC7BIT = 0]
[Optional Bit, Condition]

Figure 2. Format of a character in UART mode of communication. The transmission starts with a start bit (ST - logic 0) and ends with one or two stop bits (SP – logic 1). Data can be sent LSB (D0) first or MSB first (D7). The parity bit PA can be optionally included as well as address bit (AD) that supports multiprocessing.

```
Things to remember 2-1. UART communication and character format.
```

UART stands for Universal Asynchronous Receiver/Transmitter. It is a widely used serial asynchronous communication protocol for transmitting 8-bit data (character oriented protocol). The TxD pin of the transmitter is connected to the RxD pin of the receiver. The TxD line is held at a logic '1' when there is no data to be sent. The start of a character is marked by the START bit (a logic '0' for 1-bit period) and the character is terminated by one or two STOP bits (logic '1'). The data character (7-bit or 8-bit) can be sent LSB first or MSB first. An optional parity bit can also be sent before the STOP bits.

The receiver uses the start bit to detect that a new character is being received. Its control logic typically works in such a way that RxD input is sampled multiple times during one bit period (e.g., 16 times). If a majority of consecutive samples is at logic '0', the receiver assumes that a new character is being received. The every incoming bit is sampled ideally in the middle of bit period T_{BITCLK}. The challenge we face with serial communication is that though both receiver and transmitter use the same T_{BITCLK}, there are always discrepancies between the bit period on the transmitter side and the bit period on the receiver side. These discrepancies accumulate with every new incoming bit, and they may lead to erroneous interpretation on the receiver side in presence of electrical noise and signal distortion on long wires. That is the main reason why we limit the length of data in UART mode to a single character (up to 8 data bits).

To help detect when an error occurs in transmission a parity bit is often used. The parity bit is computed on the transmitter side using one of the following policies: (a) EVEN parity – parity bit is determined in such a way that the total number of transferred bits with logic '1' is EVEN; or (b) ODD parity – the parity bit is determined so that the total number of transferred bits with logic '1' is ODD. The receiver computes its own parity bit from the incoming data and compares it with the parity bit that is received. If the two match, the likelihood of errorless transmission is very high. If they do not match, we know that we had error in one bit (very likely), or in 3 bits (less likely), or 5 bits, and so on. What happens if we have error on two bits – the parity bit will not help us detect this type of error, but the communication theory teaches us that this event is less likely than having no errors. The parity bit thus can give us reasonable assurances that communication was without errors but cannot detect all errors and can correct them. For that we can use error correction codes that is studies in communication.

```
Example 2-2. Determine parity bit when transferring an ASCII character '0'
if we use ODD parity.
   •  ASCII code for '0': 0x30 or 0011_0000b => even number of bits with
      '1'
   •  The parity bit should be set to P=1 so that the total number of bits
      transferred is ODD.
```

3 Serial Interfaces in MSP430F5529

Figure 3 shows a functional diagram of the MSP430F5529. It includes two Universal Serial Communication Interfaces (USCI), USCI0 and USCI1. Each USCI module supports two channels USCI_Ax and USCI_Bx. USCI_Ax supports UART, SPI, and IrDA (infrared) communication and USCI_Bx supports SPI and I^{2}C communication protocols. Here, we will discuss USCI_Ax operating in UART mode of serial communication.

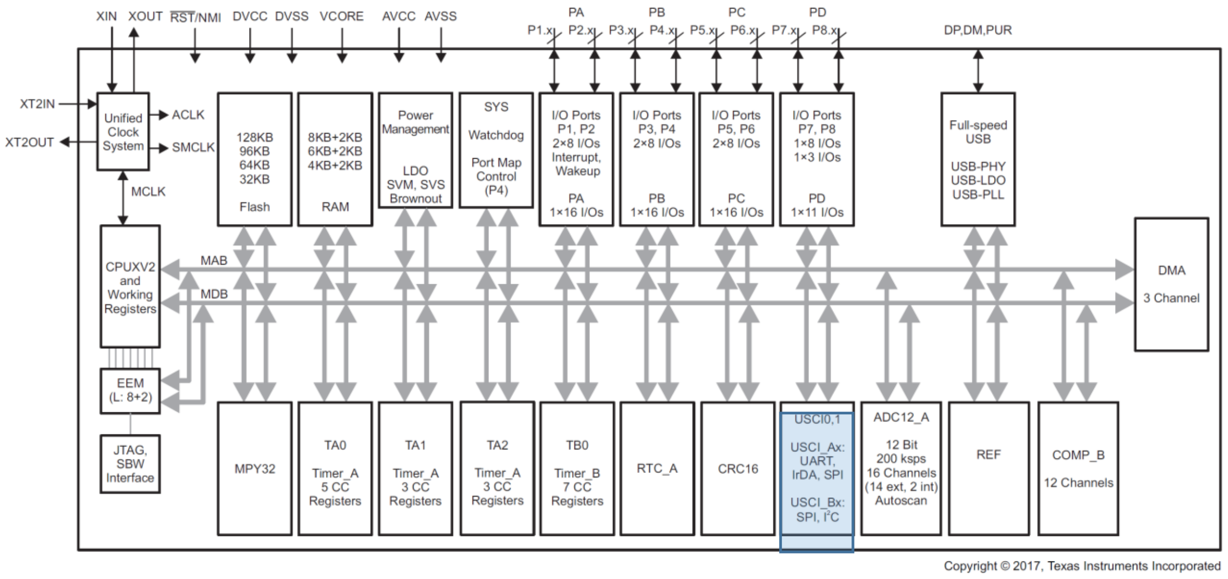

XIN
XOUT
RST/NMI
DVCC
DVSS
VCORE
AVCC
AVSS
PA
PB
PC
PD
P1.x
P2.x
P3.x
P4.x
P5.x
P6.x
P7.x
P8.x
DP,DM,PUR
XT2IN
XT2OUT
Unified Clock System
ACLK
SMCLK
MCLK
128KB 96KB 64KB 32KB Flash
8KB+2KB 6KB+2KB 4KB+2KB RAM
Power Management LDO SVM, SVS Brownout
SYS Watchdog Port Map Control (P4)
I/O Ports P1, P2 2×8 I/Os Interrupt, Wakeup PA 1×16 I/Os
I/O Ports P3, P4 2×8 I/Os PB 1×16 I/Os
I/O Ports P5, P6 2×8 I/Os PC 1×16 I/Os
I/O Ports P7, P8 1×8 I/Os 1×3 I/Os PD 1×11 I/Os
Full-speed USB USB-PHY USB-LDO USB-PLL
CPUXV2 and Working Registers
MAB
MDB
DMA 3 Channel
EEM (L: 8+2)
JTAG, SBW Interface
MPY32
TA0 Timer_A 5 CC Registers
TA1 Timer_A 3 CC Registers
TA2 Timer_A 3 CC Registers
TB0 Timer_B 7 CC Registers
RTC_A
CRC16
USCI0,1 USCI_Ax: UART, IrDA, SPI USCI_Bx: SPI, I²C
ADC12_A 12 Bit 200 ksps 16 Channels (14 ext, 2 int) Autoscan
REF
COMP_B 12 Channels
Copyright © 2017, Texas Instruments Incorporated

Figure 3. Functional Block Diagram of MSP430F5529.

4 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 4 shows a block diagram of the USCI in the UART mode. We can identify the receiver block on the top with the receiver data buffer (UCAxRXBUF), the receive shift register (not visible to programmers), and a connection to the UCAxRXD port pin. The transmitter block at the bottom includes the transmit data buffer (UCAxTXBUF), the transmit shift register (not visible to programmers), and a connection to the UCAxTXD port pin. The middle block is baud rate generator that takes one of the input clocks (UCAxCLK, ACLK, SMCLK) and generates the communication bit clocks (Transmit clock and Receive clock) for both the transmit and receive blocks.

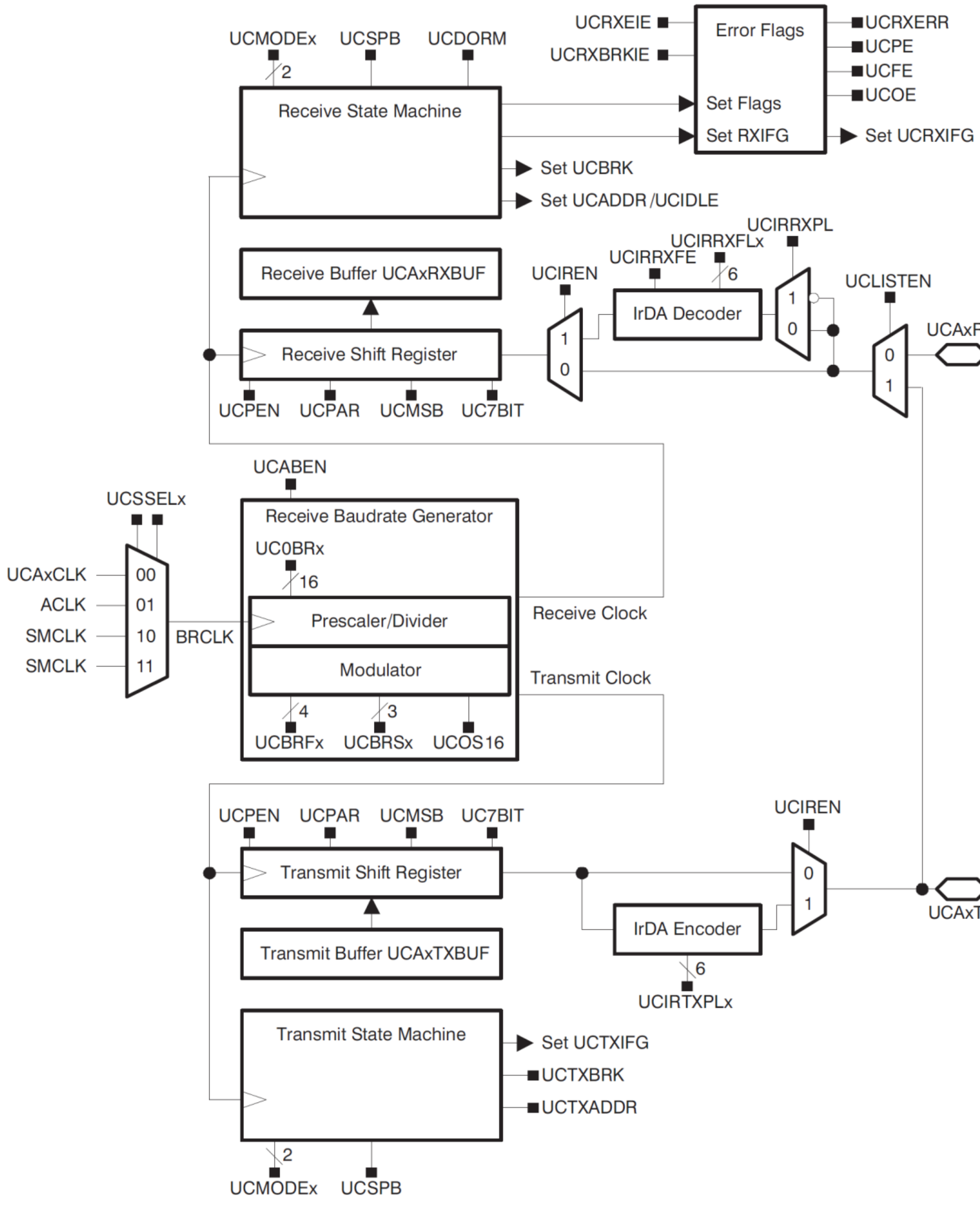
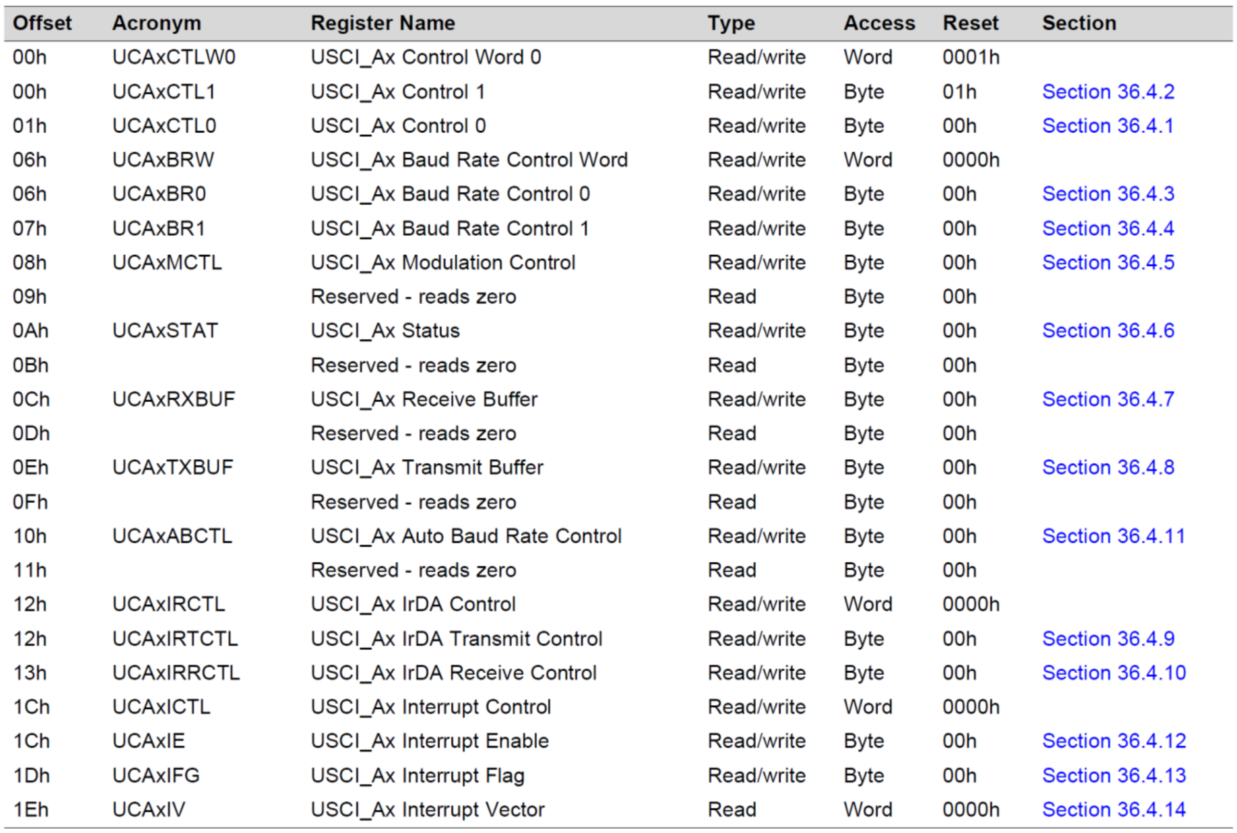


Figure 4. USCI_Ax block diagram: UART mode.

The USCI registers visible to programmers for USIC_Ax are shown in Figure 5. USCI is an 8-bit peripheral device and all registers are 8-bit long. For USCI_A0 (UCA0), the notable registers are two control registers (UCA0CTL0 and UCA0CTL1), baud rate control registers (UCA0BR0 and

UCA0BR1), modulation control register (UCA0MCTL), status register (UCA0STAT), receive buffer (UCA0RXBUF), and transmit buffer (UCA0TXBUF).



Offset	Acronym	Register Name	Type	Access	Reset	Section
00h	UCAxCTLW0	USCI_Ax Control Word 0	Read/write	Word	0001h	
00h	UCAxCTL1	USCI_Ax Control 1	Read/write	Byte	01h	Section 36.4.2
01h	UCAxCTL0	USCI_Ax Control 0	Read/write	Byte	00h	Section 36.4.1
06h	UCAxBRW	USCI_Ax Baud Rate Control Word	Read/write	Word	0000h	
06h	UCAxBR0	USCI_Ax Baud Rate Control 0	Read/write	Byte	00h	Section 36.4.3
07h	UCAxBR1	USCI_Ax Baud Rate Control 1	Read/write	Byte	00h	Section 36.4.4
08h	UCAxMCTL	USCI_Ax Modulation Control	Read/write	Byte	00h	Section 36.4.5
09h		Reserved - reads zero	Read	Byte	00h	
0Ah	UCAxSTAT	USCI_Ax Status	Read/write	Byte	00h	Section 36.4.6
0Bh		Reserved - reads zero	Read	Byte	00h	
0Ch	UCAxRXBUF	USCI_Ax Receive Buffer	Read/write	Byte	00h	Section 36.4.7
0Dh		Reserved - reads zero	Read	Byte	00h	
0Eh	UCAxTXBUF	USCI_Ax Transmit Buffer	Read/write	Byte	00h	Section 36.4.8
0Fh		Reserved - reads zero	Read	Byte	00h	
10h	UCAxABCTL	USCI_Ax Auto Baud Rate Control	Read/write	Byte	00h	Section 36.4.11
11h		Reserved - reads zero	Read	Byte	00h	
12h	UCAxIRCTL	USCI_Ax IrDA Control	Read/write	Word	0000h	
12h	UCAxIRTCTL	USCI_Ax IrDA Transmit Control	Read/write	Byte	00h	Section 36.4.9
13h	UCAxIRRCTL	USCI_Ax IrDA Receive Control	Read/write	Byte	00h	Section 36.4.10
1Ch	UCAxICTL	USCI_Ax Interrupt Control	Read/write	Word	0000h	
1Ch	UCAxIE	USCI_Ax Interrupt Enable	Read/write	Byte	00h	Section 36.4.12
1Dh	UCAxIFG	USCI_Ax Interrupt Flag	Read/write	Byte	00h	Section 36.4.13
1Eh	UCAxIV	USCI_Ax Interrupt Vector	Read	Word	0000h	Section 36.4.14

Figure 5. USCI_Ax UART Mode Control and Status Registers

`Things to remember 4-1. USCI in UART mode.`

USCI stands for Universal Serial Communication Interface. It supports different types of serial communication protocols: UART, SPI, I^{2}C, IrDA. USCI in UART mode includes 3 distinct submodules: receiver with UCAxRXBUF, transmitter with UCAxTXBUF, and the baud rate generator.

4.1 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);

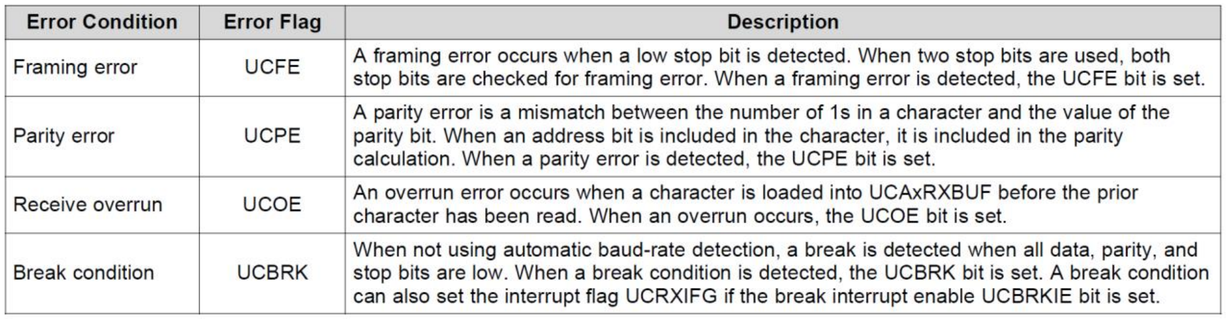
5. Enable interrupts (optional) by setting UCAxRXIE and UCAxTXIE.

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

```
Things to remember 4-2. USCI UART mode initialization.
```
It is recommended to follow guidelines when initializing USCI devices.

4.2 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 6. 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 UCRXIFG if the break interrupt enable UCBRKIE bit is set.

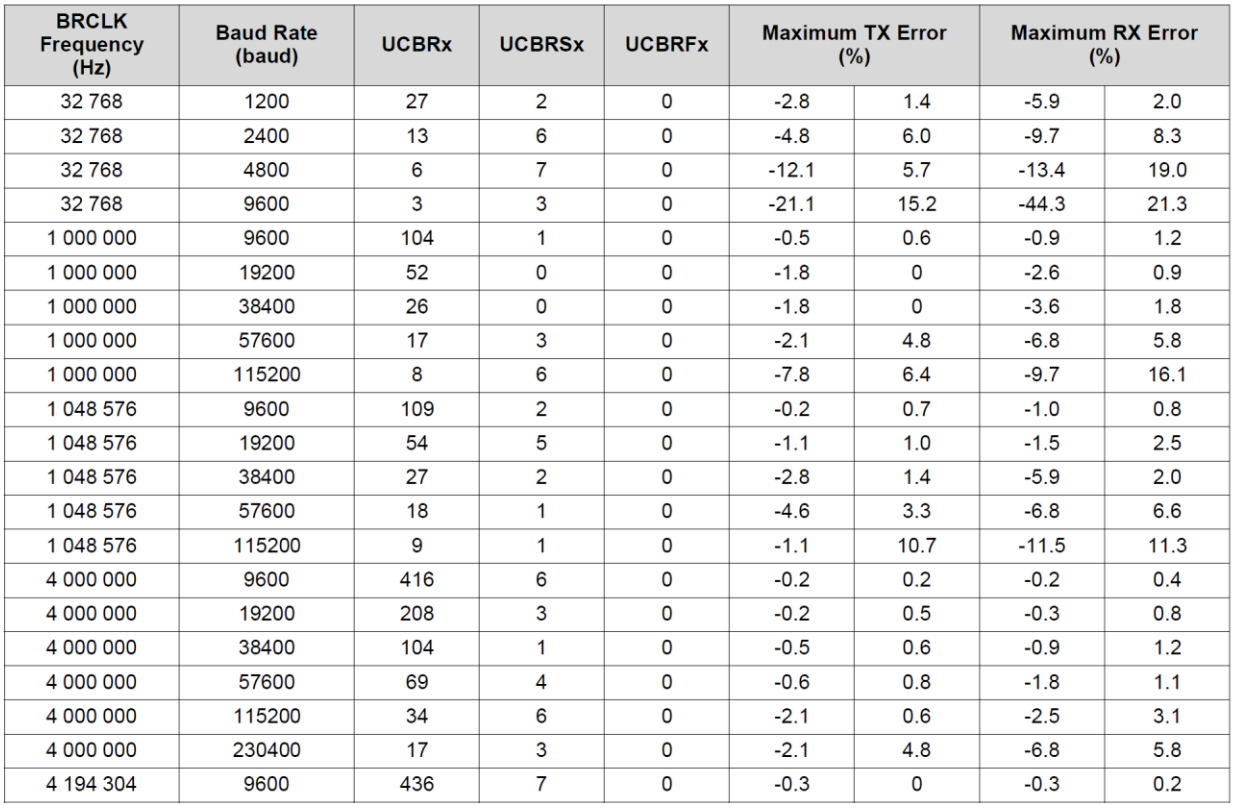
Figure 6. Receive Error Conditions

4.3 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 7 shows common combinations of clock sources and baud rates and how to set the baud rate control registers.



BRCLK Frequency (Hz)	Baud Rate (baud)	UCBRx	UCBRSx	UCBRFx	Maximum TX Error (%)		Maximum 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
4 194 304	9600	436	7	0	-0.3	0	-0.3	0.2

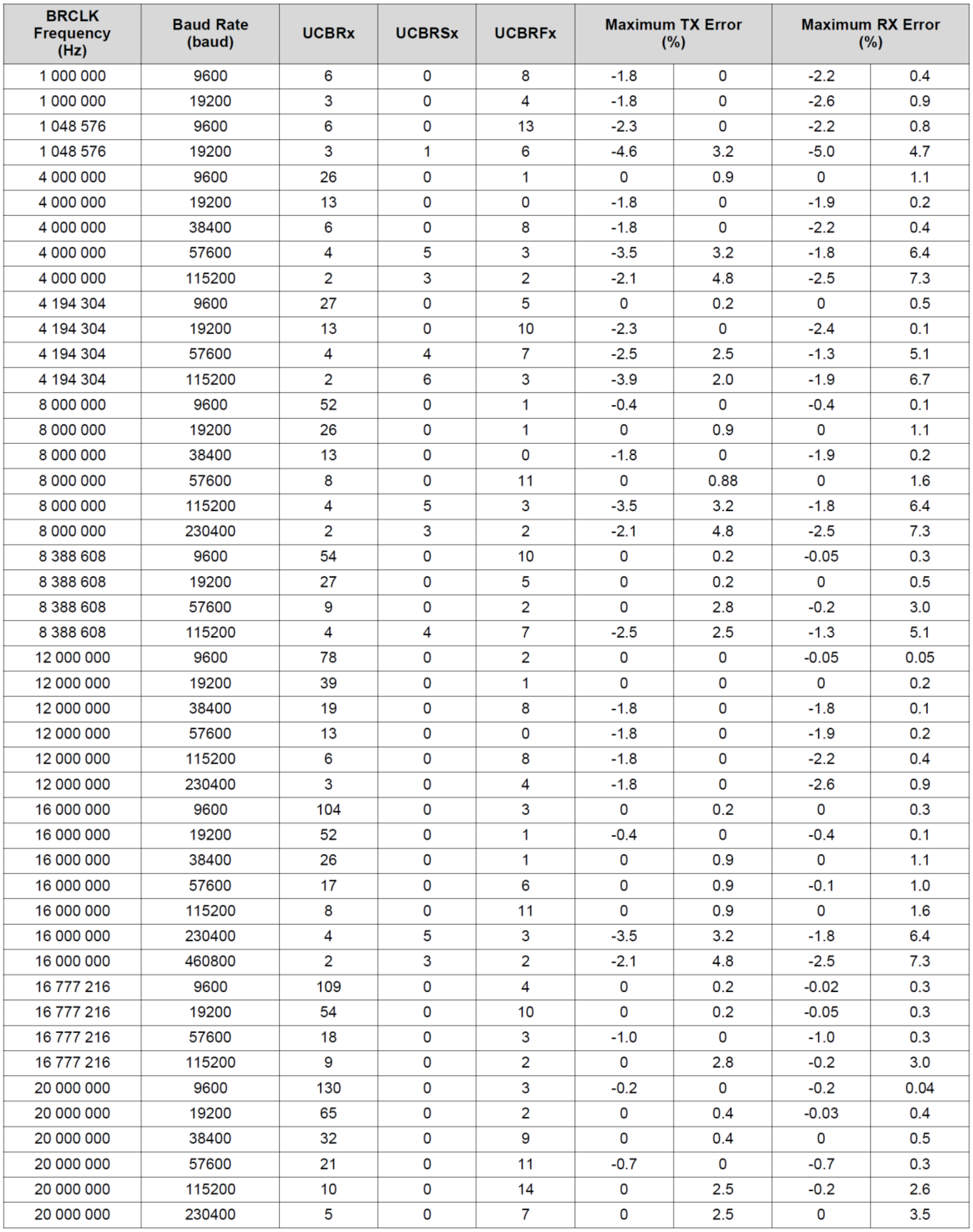
Figure 7. Commonly user baud rates and settings in low-frequency mode (UCOS16=0) – for full table see User's Guide, Table 36-5.

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 8 shows how to setup baud rate control registers in oversampling mode for common combinations of clock sources and baud rates.

`Things to remember 4-3. USCI UART baud rate generator.`

USCI in UART mode can operate in low-frequency (UCOS16=1) or oversampling mode (UCOS16=1). The modes determine how to initialize the prescalar as well as modulation control registers.



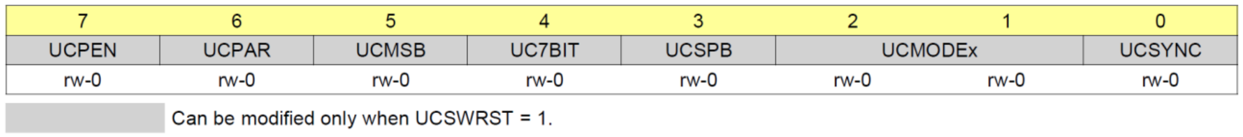
BRCLK Frequency (Hz)	Baud Rate (baud)	UCBRx	UCBRSx	UCBRFx	Maximum TX Error (%)		Maximum RX Error (%)	
1 000 000	9600	6	0	8	-1.8	0	-2.2	0.4
1 000 000	19200	3	0	4	-1.8	0	-2.6	0.9
1 048 576	9600	6	0	13	-2.3	0	-2.2	0.8
1 048 576	19200	3	1	6	-4.6	3.2	-5.0	4.7
4 000 000	9600	26	0	1	0	0.9	0	1.1
4 000 000	19200	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
4 194 304	9600	27	0	5	0	0.2	0	0.5
4 194 304	19200	13	0	10	-2.3	0	-2.4	0.1
4 194 304	57600	4	4	7	-2.5	2.5	-1.3	5.1
4 194 304	115200	2	6	3	-3.9	2.0	-1.9	6.7
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
8 388 608	9600	54	0	10	0	0.2	-0.05	0.3
8 388 608	19200	27	0	5	0	0.2	0	0.5
8 388 608	57600	9	0	2	0	2.8	-0.2	3.0
8 388 608	115200	4	4	7	-2.5	2.5	-1.3	5.1
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
16 000 000	9600	104	0	3	0	0.2	0	0.3
16 000 000	19200	52	0	1	-0.4	0	-0.4	0.1
16 000 000	38400	26	0	1	0	0.9	0	1.1
16 000 000	57600	17	0	6	0	0.9	-0.1	1.0
16 000 000	115200	8	0	11	0	0.9	0	1.6
16 000 000	230400	4	5	3	-3.5	3.2	-1.8	6.4
16 000 000	460800	2	3	2	-2.1	4.8	-2.5	7.3
16 777 216	9600	109	0	4	0	0.2	-0.02	0.3
16 777 216	19200	54	0	10	0	0.2	-0.05	0.3
16 777 216	57600	18	0	3	-1.0	0	-1.0	0.3
16 777 216	115200	9	0	2	0	2.8	-0.2	3.0
20 000 000	9600	130	0	3	-0.2	0	-0.2	0.04
20 000 000	19200	65	0	2	0	0.4	-0.03	0.4
20 000 000	38400	32	0	9	0	0.4	0	0.5
20 000 000	57600	21	0	11	-0.7	0	-0.7	0.3
20 000 000	115200	10	0	14	0	2.5	-0.2	2.6
20 000 000	230400	5	0	7	0	2.5	0	3.5

Figure 8. Commonly user baud rates and settings in oversampling mode (UCOS16=1).

4.4 USCI Control Registers

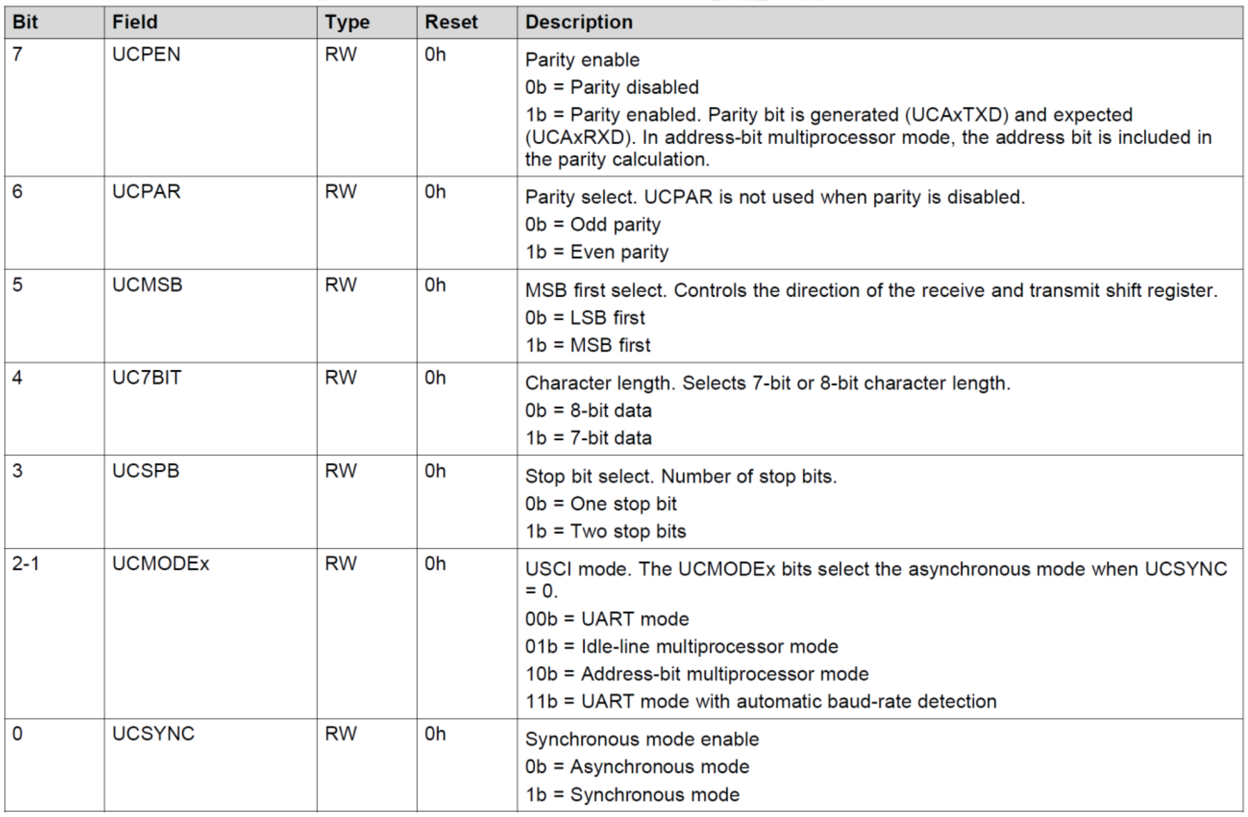
Figure 9 and Figure 10 show the format and description of relevant bits for UCAxCTL0 and UCA0xCTL1, respectively. Figure 11 shows the format of modulation register. Figure 12 shows the format of UCAxSTAT register. The UCAxBR0 and UCAxBR1 registers contain lower and upper byte, respectively, of the prescalar setting for the baud rate generator.

Figure 13 shows the format of registers related to the UART interrupts. The UCTXIFG interrupt flag is set by the transmitter to indicate that the UCAxTXBUF is ready to accept another character. The Interrupt request is generated if UCTXIE and GIE are also set. UCTXIFG is automatically reset if a character is written to UCAxTXBUF. The UCRXIFG flag is set when a new character is received and loaded into UCAxRXBUF. An interrupt request is generated if UCRXIE and GIE are also set. UCRXIFG is automatically reset when UCAxRXBUF is read.



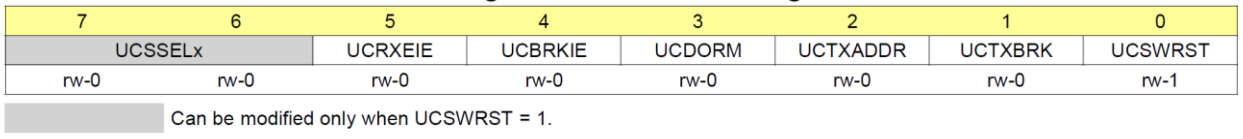
7	6	5	4	3	2	1	0
UCPEN	UCPAR	UCMSB	UC7BIT	UCSPB	UCMODEx		UCSYNC
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

▨ Can be modified only when UCSWRST = 1.



Bit	Field	Type	Reset	Description
7	UCPEN	RW	0h	Parity enable 0b = Parity disabled 1b = Parity enabled. Parity bit is generated (UCAxTXD) and expected (UCAxRXD). In address-bit multiprocessor mode, the address bit is included in the parity calculation.
6	UCPAR	RW	0h	Parity select. UCPAR is not used when parity is disabled. 0b = Odd parity 1b = Even parity
5	UCMSB	RW	0h	MSB first select. Controls the direction of the receive and transmit shift register. 0b = LSB first 1b = MSB first
4	UC7BIT	RW	0h	Character length. Selects 7-bit or 8-bit character length. 0b = 8-bit data 1b = 7-bit data
3	UCSPB	RW	0h	Stop bit select. Number of stop bits. 0b = One stop bit 1b = Two stop bits
2-1	UCMODEx	RW	0h	USCI mode. The UCMODEx bits select the asynchronous mode when UCSYNC = 0. 00b = UART mode 01b = Idle-line multiprocessor mode 10b = Address-bit multiprocessor mode 11b = UART mode with automatic baud-rate detection
0	UCSYNC	RW	0h	Synchronous mode enable 0b = Asynchronous mode 1b = Synchronous mode

Figure 9. UCAxCTL0.



7	6	5	4	3	2	1	0
UCSSELx		UCRXEIE	UCBRKIE	UCDORM	UCTXADDR	UCTXBRK	UCSWRST
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-1

▨ Can be modified only when UCSWRST = 1.

Bit	Field	Type	Reset	Description
7-6	UCSSELx	RW	0h	USCI clock source select. These bits select the BRCLK source clock. 00b = UCAxCLK (external USCI clock) 01b = ACLK 10b = SMCLK 11b = SMCLK
5	UCRXEIE	RW	0h	Receive erroneous-character interrupt enable 0b = Erroneous characters rejected and UCRXIFG is not set. 1b = Erroneous characters received set UCRXIFG.
4	UCBRKIE	RW	0h	Receive break character interrupt enable 0b = Received break characters do not set UCRXIFG. 1b = Received break characters set UCRXIFG.
3	UCDORM	RW	0h	Dormant. Puts USCI into sleep mode. 0b = Not dormant. All received characters set UCRXIFG. 1b = Dormant. Only characters that are preceded by an idle-line or with address bit set UCRXIFG. In UART mode with automatic baud-rate detection, only the combination of a break and synch field sets UCRXIFG.
2	UCTXADDR	RW	0h	Transmit address. Next frame to be transmitted is marked as address, depending on the selected multiprocessor mode. 0b = Next frame transmitted is data. 1b = Next frame transmitted is an address.
1	UCTXBRK	RW	0h	Transmit break. Transmits a break with the next write to the transmit buffer. In UART mode with automatic baud-rate detection, 055h must be written into UCAxTXBUF to generate the required break/synch fields. Otherwise, 0h must be written into the transmit buffer. 0b = Next frame transmitted is not a break. 1b = Next frame transmitted is a break or a break/synch.
0	UCSWRST	RW	1h	Software reset enable 0b = Disabled. USCI reset released for operation. 1b = Enabled. USCI logic held in reset state.

Figure 10. UCAxCTL1.

7	6	5	4	3	2	1	0
UCBRFx				UCBRSx			UCOS16
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

▨ Can be modified only when UCSWRST = 1.

Bit	Field	Type	Reset	Description
7-4	UCBRFx	RW	0h	First modulation stage select. These bits determine the modulation pattern for BITCLK16 when UCOS16 = 1. Ignored with UCOS16 = 0. Table 36-2 shows the modulation pattern.
3-1	UCBRSx	RW	0h	Second modulation stage select. These bits determine the modulation pattern for BITCLK. Table 36-2 shows the modulation pattern.
0	UCOS16	RW	0h	Oversampling mode enabled 0b = Disabled 1b = Enabled

Figure 11. UCAxMCTL. UCBRFx field defines modulation in oversampling mode (UCOS16=1). UCBRSx field defines modulation for low-frequency mode (UCOS16=0).

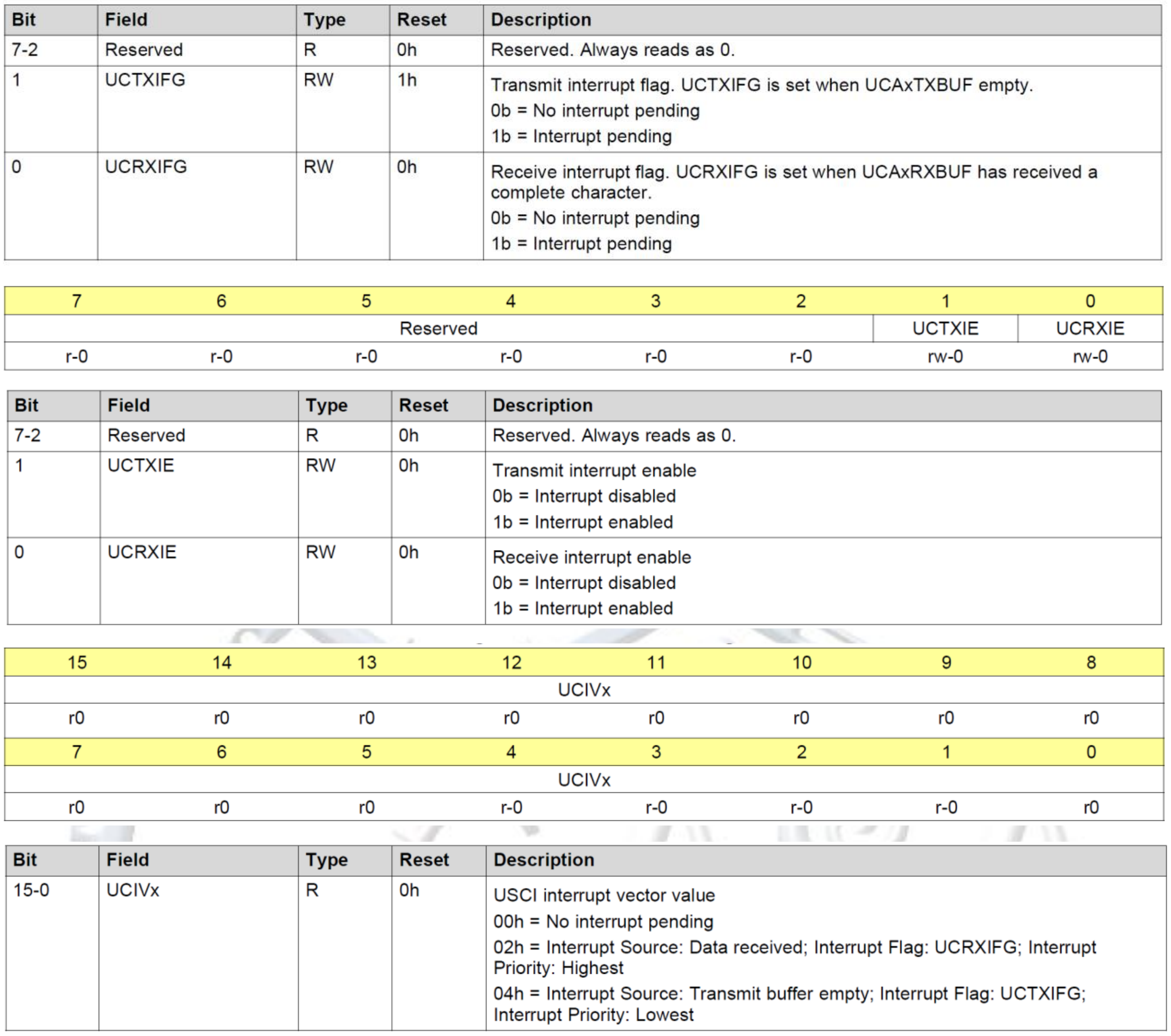
7	6	5	4	3	2	1	0
UCLISTEN	UCFE	UCOE	UCPE	UCBRK	UCRXERR	UCADDR/ UCIDLE	UCBUSY
rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	r-0

▨	Can be modified only when UCSWRST = 1.

Bit	Field	Type	Reset	Description
7	UCLISTEN	RW	0h	Listen enable. The UCLISTEN bit selects loopback mode. 0b = Disabled 1b = Enabled. UCAxTXD is internally fed back to the receiver.
6	UCFE	RW	0h	Framing error flag. UCFE is cleared when UCAxRXBUF is read. 0b = No error 1b = Character received with low stop bit
5	UCOE	RW	0h	Overrun error flag. This bit is set when a character is transferred into UCAxRXBUF before the previous character was read. UCOE is cleared automatically when UCxRXBUF is read, and must not be cleared by software. Otherwise, it does not function correctly. 0b = No error 1b = Overrun error occurred
4	UCPE	RW	0h	Parity error flag. When UCPEN = 0, UCPE is read as 0. UCPE is cleared when UCAxRXBUF is read. 0b = No error 1b = Character received with parity error
3	UCBRK	RW	0h	Break detect flag. UCBRK is cleared when UCAxRXBUF is read. 0b = No break condition 1b = Break condition occurred
2	UCRXERR	RW	0h	Receive error flag. This bit indicates a character was received with error(s). When UCRXERR = 1, on or more error flags, UCFE, UCPE, or UCOE is also set. UCRXERR is cleared when UCAxRXBUF is read. 0b = No receive errors detected 1b = Receive error detected
1	UCADDR/UCIDLE	RW	0h	UCADDR: Address received in address-bit multiprocessor mode. UCADDR is cleared when UCAxRXBUF is read. 0b = Received character is data. 1b = Received character is an address. UCIDLE: Idle line detected in idle-line multiprocessor mode. UCIDLE is cleared when UCAxRXBUF is read. 0b = No idle line detected 1b = Idle line detected
0	UCBUSY	R	0h	USCI busy. This bit indicates if a transmit or receive operation is in progress. 0b = USCI inactive 1b = USCI transmitting or receiving

Figure 12. UCAxSTAT.

7	6	5	4	3	2	1	0
Reserved						UCTXIFG	UCRXIFG
r-0	r-0	r-0	r-0	r-0	r-0	rw-1	rw-0



Bit	Field	Type	Reset	Description
7-2	Reserved	R	0h	Reserved. Always reads as 0.
1	UCTXIFG	RW	1h	Transmit interrupt flag. UCTXIFG is set when UCAxTXBUF empty. 0b = No interrupt pending 1b = Interrupt pending
0	UCRXIFG	RW	0h	Receive interrupt flag. UCRXIFG is set when UCAxRXBUF has received a complete character. 0b = No interrupt pending 1b = Interrupt pending

7	6	5	4	3	2	1	0
Reserved						UCTXIE	UCRXIE
r-0	r-0	r-0	r-0	r-0	r-0	rw-0	rw-0

Bit	Field	Type	Reset	Description
7-2	Reserved	R	0h	Reserved. Always reads as 0.
1	UCTXIE	RW	0h	Transmit interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled
0	UCRXIE	RW	0h	Receive interrupt enable 0b = Interrupt disabled 1b = Interrupt enabled

15	14	13	12	11	10	9	8
UCIVx							
r0	r0	r0	r0	r0	r0	r0	r0
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
UCIVx							
r0	r0	r0	r-0	r-0	r-0	r-0	r0

Bit	Field	Type	Reset	Description
15-0	UCIVx	R	0h	USCI interrupt vector value 00h = No interrupt pending 02h = Interrupt Source: Data received; Interrupt Flag: UCRXIFG; Interrupt Priority: Highest 04h = Interrupt Source: Transmit buffer empty; Interrupt Flag: UCTXIFG; Interrupt Priority: Lowest

Figure 13. UCAxIFG, UCAxIE, and UCAxIV Registers.

The following examples illustrate how to determine initial values for the prescalar and modulation registers.

```
Example 4-1. Determine UCAxBR0, UCAxBR1, UCAxMCTL registers under the
following conditions. Source clock is SMCLK=2^20 Hz, Baud rate is 38,400.
```

- `Step 1: N = 2^20/38400 = 27.3066`
- `Step 2: N = 27.3066; this is >16, but only slightly`
  `=> use the low-frequency mode, UCOS16=0`
- `Step 2: UCAxBRx=INT(N) = 27 (UCAxBR0=27, UCAxBR1=0)`

- Step 3: UCBRS = round[(N – INT(N))*8]=round(0.3066*8)=round(2.453)=2
  UCAxMCTL |= UCBRS_2

Example 4-2. Determine UCAxBR0, UCAxBR1, UCAxMCTL registers under the following conditions. Source clock is SMCLK=2^{22} Hz, Baud rate is 38,400.

- Step 1: N = 2^{22}/38400 = 109.2266
- Step 2: N = 109.2266; this is >16 => use the oversampling mode, UCOS16=1
- Step 2: N/16 = 6.82 => UCAxBRx=INT(N/16) = 6 (UCAxBR0=6, UCAxBR1=0)
- Step 3: UCBRF = round[(N/16 – INT(N/16))*16] = round(0.82*16) = round(13.2266) = 13
  UCAxMCTL |= UCBRF_13 | UCOS16

5 Code Examples

Code 1 shows a program that echoes a character received from the developer workstation using the MSP-EX430F529 launchpad. Connect the board as described in the program header below. We initialize the UCA0 for UART communication with 115,200 baud rate. As we use the default SMCLK of 2^{20} Hz, we will use the low-frequency mode. Thus, the prescalar value is 0x0009 (BR0=0x09, BR1=0x00). The modulation filed is 0x01 (or bit 1 of the UCA0MCTL register should be set). It helps to write a subroutine that includes all steps necessary to initialize of UCA0 (UART_Setup).

The main program calls the UART_Setup() and enters an infinite loop. In the loop, we use polling to detect when a new character is received. The while statement in line 69 checks the register UCA0IFG, bit UCRXIFG. If no character is received we check again. Once the character is received we exit the while loop. The next step is to check whether the transmit buffer is ready as we want to echo the character (send it back to the workstation). If it is ready, we read the character from the UCA0RXBUF and write it into UCA0TXBUF. Please note that we do not need to explicitly clear the flag for UCRXIFG even though we are using polling. The flag is automatically cleared once we read from the UCA0RXBUF. The LED1 is toggled.

```
1   /*------------------------------------------------------------------------------
2    * File:          Lab8_D1.c
3    *
4    * Function:      Echo a received character, using polling.
5    *
6    * Description:   This program echos the character received from UART back to UART.
7    *                Toggle LED1 with every received character.
8    *                Baud rate: low-frequency (UCOS16=0);
9    *                1048576/115200 = ~9.1 (0x0009|0x01)
10   *
11   * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
12   *
```

```
 * Board:         MSP-EXP430F5529
 *
 * Instructions: Set the following parameters in putty
 * Port: COMx
 * Baud rate: 115200
 * Data bits: 8
 * Parity: None
 * Stop bits: 1
 * Flow Control: None
 *
 * Note:       If you are using Adafruit USBtoTTL cable, look for COM port
 *             in the Windows Device Manager with the following text:
 *             Silicon Labs CP210x USB to UART Bridge (COM<x>).
 *             Connecting Adafruit USB to TTL:
 *              GND - black wire - connect to the GND pin (on the board or BoosterPack)
 *              Vcc - red wire - leave disconnected
 *              Rx    white wire (receive into USB, connect on TxD of the board P3.3)
 *              Tx -  green wire (transmit from USB, connect to RxD of the board P3.4)
 *        MSP430F5529
 *     -----------------
 * /|\ |             XIN|-
 *  |  |                | 32kHz
 *  |--|RST         XOUT|-
 *     |                |
 *     |   P3.3/UCA0TXD|------------>
 *     |                | 115200 - 8N1
 *     |   P3.4/UCA0RXD|<------------
 *     |          P1.0|----> LED1
 *
 * Input:     None (Type characters in putty/MobaXterm/hyperterminal)
 * Output:    Character echoed at UART
 * Author:    A. Milenkovic, milenkovic@computer.org
 * Date:      October 2018, modified August 2020
 *------------------------------------------------------------------------------*/
#include <msp430.h>

void UART_setup(void) {

    P3SEL |= BIT3 + BIT4;   // Set USCI_A0 RXD/TXD to receive/transmit data
    UCA0CTL1 |= UCSWRST;    // Set software reset during initialization
    UCA0CTL0 = 0;           // USCI_A0 control register
    UCA0CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA0BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;         // upper byte
    UCA0MCTL |= UCBRS0;     // Modulation (UCBRS0=0x02, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
}

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;        // Stop WDT
    P1DIR |= BIT0;                   // Set P1.0 to be output
    UART_setup();                    // Initialize UART

    while (1) {
       while(!(UCA0IFG&UCRXIFG));    // Wait for a new character
       // New character is here in UCA0RXBUF
       while(!(UCA0IFG&UCTXIFG));    // Wait until TXBUF is free
       UCA0TXBUF = UCA0RXBUF;        // TXBUF <= RXBUF (echo)
73          P1OUT ^= BIT0;              // Toggle LED1
74      }
75  }
```

Code 1. Echo a character using polling.

Code 2 shows a program that echoes the character that uses the interrupt service routine instead of polling. The main program initializes the UCA0 and enters a low-power mode 0 (the cpu is turned off). In the UART_Setup function we enable interrupts when a character is received (line 50). When a character is received, an interrupt request is presented from UCA0 that wakes the processor up and the USCI_A0_VECTOR interrupt service routine is entered. Inside the ISR we check whether the UCA0TXBUF is ready and it it is ready, echo the received character and toggle the LED1. Upon exiting the ISR, the previous conditions are restored and the processor goes back into the low-power mode 0.

```
1   /*------------------------------------------------------------------------------
2    * File:          Lab8_D2.c
3    *
4    * Function:      Echo a received character, using receiver ISR.
5    * Description:   This program echos the character received from UART back to UART.
6    *                Toggle LED1 with every received character.
7    *                Baud rate: low-frequency (UCOS16=0);
8    *                1048576/115200 = ~9.1 (0x0009|0x01)
9    * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO
10   *
11   * Instructions: Set the following parameters in putty
12   * Port: COMx
13   * Baud rate: 115200
14   * Data bits: 8
15   * Parity: None
16   * Stop bits: 1
17   * Flow Control: None
18   *
19   *        MSP430f5529
20   *     -----------------
21   * /|\ |             XIN|-
22   *  |  |                | 32kHz
23   *  |--|RST         XOUT|-
24   *     |                |
25   *     |    P3.3/UCA0TXD|------------>
26   *     |                | 115200 - 8N1
27   *     |    P3.4/UCA0RXD|<------------
28   *     |            P1.0|----> LED1
29   *
30   * Input:     None (Type characters in putty/MobaXterm/hyperterminal)
31   * Output:    Character echoed at UART
32   * Author:    A. Milenkovic, milenkovic@computer.org
33   * Date:      October 2018
34   *----------------------------------------------------------------------------*/
35  #include <msp430.h>
36
37  // Initialize USCI_A0 module to UART mode
38  void UART_setup(void) {
39
40      P3SEL |= BIT3 + BIT4;   // Set USCI_A0 RXD/TXD to receive/transmit data
41      UCA0CTL1 |= UCSWRST;    // Set software reset during initialization
42      UCA0CTL0 = 0;           // USCI_A0 control register
```

```
    UCA0CTL1 |= UCSSEL_2;   // Clock source SMCLK

    UCA0BR0 = 0x09;         // 1048576 Hz  / 115200 lower byte
    UCA0BR1 = 0x00;         // upper byte
    UCA0MCTL |= UCBRS0;     // Modulation (UCBRS0=0x02, UCOS16=0)

    UCA0CTL1 &= ~UCSWRST;   // Clear software reset to initialize USCI state machine
    UCA0IE |= UCRXIE;       // Enable USCI_A0 RX interrupt
}

void main(void) {
    WDTCTL = WDTPW + WDTHOLD; // Stop WDT
    P1DIR |= BIT0;            // Set P1.0 to be output
    UART_setup();             // Initialize USCI_A0 in UART mode

    _BIS_SR(LPM0_bits + GIE); // Enter LPM0, interrupts enabled
}

// Echo back RXed character, confirm TX buffer is ready first
#pragma vector = USCI_A0_VECTOR
__interrupt void USCIA0RX_ISR (void) {
    while(!(UCA0IFG&UCTXIFG));  // Wait until can transmit
    UCA0TXBUF = UCA0RXBUF;      // TXBUF <-- RXBUF
    P1OUT ^= BIT0;              // Toggle LED1
}
```

Code 2. Echo a character using receiver ISR

Code 3 shows a program that utilizes UCA0 and TA0 to implement a real-time clock that sends time via UART to the workstation. The time is measured with resolution of a decisecond (1/10th of a second). The main program is organized as follows. We configure UCA0 for serial communication in UART mode for 9,600 bps. We configure TA0 to generate an interrupt every 100 ms (1/10th of a second).

The main loop of the program starts with an entering a low power mode. The ISR for TA0 is entered once every 100 ms. Inside we update the variables that keep track of current time (tsec and sec) and change the copy of the status register on the program stack to make sure that once we exit the ISR we do not go back to the low-power mode, but rather remain in active mode. Line 97 reaches to the top of the stack and changes the bits in the copy of the status register. Once we exit the ISR, we continue execution in the main loop by invoking SendTime(). Here the current time is printed into a character array using sprintf library function. The time message is then sent over UART character by character.

```
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/(16*9600) = ~6.8 [16 from UCOS16]
10   * Clocks:        ACLK = LFXT1 = 32768Hz, MCLK = SMCLK = default DCO = 1048576Hz
11   * Instructions:  Set the following parameters in putty/hyperterminal
```

```c
 * Port: COMx
 * Baud rate: 19200
 * Data bits: 8
 * Parity: None
 * Stop bits: 1
 * Flow Control: None
 *
 *         MSP430F5529
 *      -----------------
 * /|\ |             XIN|-
 *  |  |                | 32kHz
 *  |--|RST         XOUT|-
 *     |                |
 *     |    P3.3/UCA0TXD|------------>
 *     |                | 9600 - 8N1
 *     |    P3.4/UCA0RXD|<------------
 *     |            P1.0|----> LED1
 *
 * Author:      A. Milenkovic, milenkovic@computer.org
 * Date:        October 2018
--------------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdio.h>

// Current time variables
unsigned int sec = 0;              // Seconds
unsigned int tsec = 0;             // 1/10 second
char Time[8];                      // String to keep current time

void UART_setup(void) {
    P3SEL = BIT3+BIT4;                        // P3.4,5 = USCI_A0 TXD/RXD
    UCA0CTL1 |= UCSWRST;                      // **Put state machine in reset**
    UCA0CTL1 |= UCSSEL_2;                     // SMCLK
    UCA0BR0 = 6;                              // 1MHz 9600 (see User's Guide)
    UCA0BR1 = 0;                              // 1MHz 9600
    UCA0MCTL = UCBRS_0 + UCBRF_13 + UCOS16;   // Mod. UCBRSx=0, UCBRFx=13,
                                              // over sampling
    UCA0CTL1 &= ~UCSWRST;                     // **Initialize USCI state machine**
}

void TimerA_setup(void) {
    TA0CTL = TASSEL_2 + MC_1 + ID_3; // Select SMCLK/8 and up mode
    TA0CCR0 = 13107;                 // 100ms interval
    TA0CCTL0 = CCIE;                 // Capture/compare interrupt enable
}

void UART_putCharacter(char c) {
    while (!(UCA0IFG&UCTXIFG));    // Wait for previous character to transmit
    UCA0TXBUF = c;                  // Put character into tx buffer
}

void SetTime(void) {
    tsec++;
    if (tsec == 10){
        tsec = 0;
        sec++;
        P1OUT ^= BIT0;              // Toggle LED1
    }
}

```

```c
void SendTime(void) {
    int i;
    sprintf(Time, "%05d:%01d", sec, tsec);// Prints time to a string

    for (i = 0; i < sizeof(Time); i++) {  // Send character by character
        UART_putCharacter(Time[i]);
    }
    UART_putCharacter('\r');        // Carriage Return
}

void main(void) {
    WDTCTL = WDTPW + WDTHOLD;       // Stop watchdog timer
    UART_setup();                   // Initialize UART
    TimerA_setup();                 // Initialize Timer_B
    P1DIR |= BIT0;                  // P1.0 is output;

    while (1) {
        _BIS_SR(LPM0_bits + GIE);   // Enter LPM0 w/ interrupts
        SendTime();                 // Send Time to HyperTerminal/putty
    }
}

#pragma vector = TIMER0_A0_VECTOR
__interrupt void TIMERA_ISA(void) {
    SetTime();                       // Update time
    _BIC_SR_IRQ(LPM0_bits);          // Clear LPM0 bits from 0(SR)
}

```

Code 3. Display real-time clock.

6 Exercises