Lab 6

Universal Asynchronous Receiver and Transmitter (UART)

In this lab, we will learn using the UART interface and program the backchannel UART link that connects our board to the PC.

6.1 Transmitting Data over UART

UART is a simple interface that allows transmitting bytes between two parties in a point-to-point configuration. UART is usually implemented over two wires that carry data in both directions between the transmitter and the receiver. UART can be implemented over one wire but, in this case, the data always travels in the same direction. The former configuration is known as full-duplex and the latter is known as half-duplex. UART is considered asynchronous since the two parties are tuned to the same frequency but each one generates its own clock signal; they are not synchronized by the edges of the clock.

UART is a very simple transmission scheme and its frame is shown in Figure 6.1. The line is idle at high. The line drops to low for one bit duration to signal the Start Bit. Typically, the receiver is listening for the occurrence of the Start Bit. After that, the data is transmitted bit by bit, usually starting with the

least significant bit (LSB). Finally, the Stop Bit, which has a value of high, is transmitted to signal the end of the transmission.

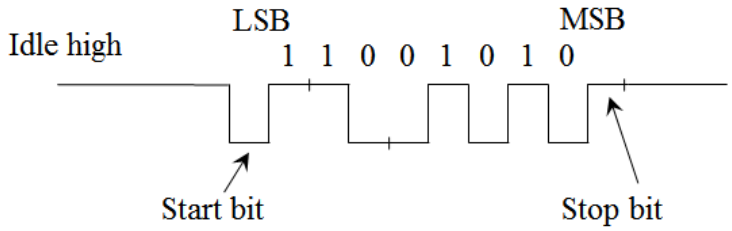

Figure 6.1: UART Transmission. Data byte: 01010011

The bit duration is defined by the baud rate (named after telegraph engineer Emile Baudot) which is simply the transmitter's clock rate. A common rate is 9600 baud which corresponds to a clock frequency of 9600 Hz. This means means that each bit lasts for 1/9600 seconds. Table 6.1 shows the parameters of a UART configuration and highlights the most popular configuration. This is the configuration that we'll use.

Table 6.1: UART Parameters

Parameter	**Meaning**	**Popular Configuration**
Baud rate	Transmission speed	9600
Data size	Number of bits	8-bit
First bit	Either Most- or Least- Significant Bit	LSB
Parity	Bits to detect errors	None
Stop bit	Signals the end of the transmission	1-bit
Flow control	A mechanism to pace the transmission	None

UART reception can be setup in two ways. In the basic approach, the receiver runs a clock at the baud rate (e.g. 9600 Hz) and samples each data bit once. This technique works but doesn't guarantee that the data bit is sampled in the middle of its duration. Therefore, it may not be reliable at high baud rates. A more advance technique is known as oversampling. In this case, the receiver runs a clock at 16x baud rate (e.g. 16x9600 Hz). Now each data bit lasts for 16 cycles at the receiver. The receiver lines up the start of the bit with cycle 1 and uses cycles 7, 8 and 9 to take three samples that are guaranteed to be in the middle of the bit duration. A majority vote is done on the three samples to determine the value of the sampled bit.

The eUSCI Module

Our microcontroller contains a communication module called eUSCI (enhanced Universal Serial Communications Interface) that implements UART and other transmission protocols such as SPI and I2C. The details of UART transmission and reception are handled by the module in the hardware and our

code interfaces with the module using a few registers and bits, which makes the programming simple. We will use the eUSCI module in this lab. The module is described in the Family User's Guide (slau367o) (Chapter 30).

The eUSCI module is organized in two channels. Channel A supports UART and SPI while Channel B supports I2C and SPI, two communication protocols that we'll encounter in future labs. Therefore, we will use Channel A in this lab. An MSP430 chip may have multiple eUSCI modules to enable independent communication channels. These are usually called eUSCI0, eUSCI1, etc.

The LaunchPad Board Setup

On our LaunchPad board, one of the MCU's UART interfaces is routed to the USB connector so that it can reach the PC. This interface is known as the backchannel UART. When this interface is configured, it creates a COM port on Windows or a 'device' on Linux or MacOS. Any application on the PC that reads serial data can interact with the backchannel UART. We can use a serial application (e.g. TeraTerm or PuTTY), the console in CCS or a C or Python application. In this lab, we'll use a terminal application that shows incoming data and transmits data back to the MCU.

Let's start by looking at the LaunchPad User's Guide (slau627a) to see which eUSCI module is used by the backchannel UART. The document states (page 10) that the backchannel UART is connected to eUSCI_A1. This refers to eUSCI module #1 Channel A. Next, let's look at the pinout figure to see which I/O pins are shared with the UART signals. Looking in the same document (page 7), we find two cases:

```
Option 1:    P3.4/UCA1TXD    P3.5/UCA1RXD
Option 2:    P5.4/UCA1TXD    P5.5/UCA1RXD
```

The term UCA1 refers to eUSCI module #1 Channel A, the one we're interested in. The terms TXD and RXD refer to Transmit Data and Receive Data. We need to find out whether the backchannel UART is the one on pins P3.4/P3.5 or pins P5.4/P5.5. To find this out, we look in the same document at the schematic (page 33). At the right side of the schematic, the jumpers layout show that the TXD/RXD at pins P3.4/P3.5 are linked to the emulation chip on the board which is connected to the USB port.

Pins on the MSP430 chip are typically shared between multiple functionalities. The chip's data sheet (slas789c) shows how the pins can be configured to the various functionalities. By default, our pins act as P3.4/P3.5. Let's look at the data sheet (page 102) to see how we can configure these pins to UCA1TXD/ UCA1RXD functionalities. We see that P3DIR is X (don't care) while P3SEL1 have 0 for both bits and P3SEL0 have 1 for both bits. The LCDS bits should be 0 (they are by default). Therefore, the piece of code below configures the pins to the backchannel UART.

```
// Configure pins to backchannel UART
// Pins: (UCA1TXD / P3.4) (UCA1RXD / P3.5)
```

```
// (P3SEL1=00, P3SEL0=11) (P2DIR=xx)
P3SEL1 &= ~(BIT4|BIT5);
P3SEL0 |= (BIT4|BIT5);
```

Finally, it would be a good idea to ensure that the jumpers on the LaunchPad board (check the row of jumpers on the board) close the connections between the MSP430 chip and the emulation chip as shown in the figure in (slau627a) page 8. If the jumpers are missing, the backchannel UART won't work.

Configuring the UART Clock Frequencies

There is a set of commonly used baud rates which includes the following: 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400 and others. Such clock frequencies are configured in the eUSCI module by starting with a higher frequency and dividing an modulating to obtain the desired frequencies.

For example, let's use SMCLK at 1 MHz (1,000,000 Hz) to configure a UART clock of 9600 Hz. We need to configure the divider and modulator. The divider is simply (1,000,000/9600=104.16) 104. The modulator compensates for the .16 fraction. Modulation is the idea of mixing cycles from two frequencies F1 and F2 and to accomplish an average frequency that's in between. In this case, the hardware configures a clock frequency of 1MHz/104 = 9,615 Hz and another at 1MHz/105 = 9,523 Hz and mixes cycles of the two frequencies to accomplish an average frequency of 9.600 Hz. All this work is done in the eUSCI hardware. What the programmer needs to do is configuring the divider and modulators values in the eUSCI configuration registers.

The eUSCI module supports UART reception with and without oversampling. Therefore, if we're transmitting and receiving UART at 9600 baud, the transmitter clock is 9,600 Hz and the receiver clock is either 9,600 Hz (without oversampling) or 16x9,600 Hz (with oversampling).

The dividers and modulators are obtained from a table in the Family User's Guide (slau367o) (page 779). Let's look up the parameters for the configuration that we'll use. Starting with the default SMCLK at 1 MHz (1,000,000 Hz) and aiming at a baud rate of 9600 and using oversampling (UCOS16=1), the dividing and modulating parameters are: UCBR=6, UCBRF=8 and UCBRS=0x20. We'll plug these values in the eUSCI configuration registers in the UART initialization function.

eUSCI Module Configuration

Let's configure the eUSCI module for a baud rate of 9600 based on the popular configuration (shown in Table 6.1) with oversampling reception enabled and using SMCLK=1MHz (1,000,000 Hz). The configuration registers of eUSCI in UART mode are found in the Family User's Guide (slau367o) at the end of Chapter 30 (starting on page 783). The eUSCI module is in reset state by default, which means that it's not operational and is not drawing current. The configuration should be modified only when the module is in reset state. Therefore, the initialization function below configures the registers and, finally, exists the reset state.

```
// Configure UART to the popular configuration
// 9600 baud, 8-bit data, LSB first, no parity bits, 1 stop bit
// no flow control, oversampling reception
// Clock: SMCLK @ 1 MHz (1,000,000 Hz)
void Initialize_UART(void){
    // Configure pins to UART functionality
    P3SEL1 &= ~(BIT4|BIT5);
    P3SEL0 |= (BIT4|BIT5);

    // Main configuration register
    UCA1CTLW0 = UCSWRST;    // Engage reset; change all the fields to zero
    // Most fields in this register, when set to zero, correspond to the
    // popular configuration
    UCA1CTLW0 |= UCSSEL_2;  // Set clock to SMCLK

   // Configure the clock dividers and modulators (and enable oversampling)
    UCA1BRW = 6;                // divider
    // Modulators: UCBRF = 8 = 1000 --> UCBRF3 (bit #3)
    // UCBRS = 0x20 = 0010 0000 = UCBRS5 (bit #5)
    UCA1MCTLW = UCBRF3 | UCBRS5 | UCOS16;

    // Exit the reset state
    UCA1CTLW0 &= ~UCSWRST;
}
```

Programming Model

The eUSCI hardware handles UART transmission and reception and interfaces with the code using a few registers and flags. These are listed at the end of the UART chapter in the family user's guide and summarized in Table 6.2.

Table 6.2: UART Registers and Flags

Flag (1-bit)	Register	Use
-	UCA1TXBUF	Transmit buffer contains the transmit byte
-	UCA1RXBUF	Receive buffer contains the received byte
UCTXIFG	UCA1IFG	0: transmission in progress; 1: ready to transmit
UCRXIFG	UCA1IFG	0: no new data; 1: new byte received

To make our code more readable, we'll rename the flags and registers with user-friendly names by defining these symbolic constants at the top of the code.

```
#define FLAGS      UCA1IFG  // Contains the transmit & receive flags
#define RXFLAG     UCRXIFG         // Receive flag
#define TXFLAG     UCTXIFG         // Transmit flag
#define TXBUFFER   UCA1TXBUF       // Transmit buffer
#define RXBUFFER   UCA1RXBUF       // Receive buffer
```

The transmit flag is 1 when the module is ready to transmit. We should not write to the transmit buffer if the transmit flag is zero as this would disrupt the ongoing transmission. When a byte is copied to the transmit buffer, the transmit flag goes to zero and the transmission starts automatically. When the transmission finishes, the transmit flag goes back to one on its own. Accordingly, this is the function that transmits a byte over UART.

```
void uart_write_char(unsigned char ch){
  // Wait for any ongoing transmission to complete
  while ( (FLAGS & TXFLAG)==0 ) {}

  // Copy the byte to the transmit buffer
  TXBUFFER = ch; // Tx flag goes to 0 and Tx begins!
  return;
}
```

The receive flag becomes 1 when a new data arrives. When the receive buffer is read, the receive flag is cleared automatically. The functions below checks if a new byte was received and returns it. If no byte was received, it returns the null character (ASCII=0).

```
// The function returns the byte; if none received, returns null character
unsigned char uart_read_char(void){
  unsigned char temp;

  // Return null character (ASCII=0) if no byte was received
  if( (FLAGS & RXFLAG) == 0)
    return 0;

  // Otherwise, copy the received byte (this clears the flag) and return it
  temp = RXBUFFER;
  return temp;
}
```

When we transmit bytes over UART to the terminal application on the PC, the latter interprets the bytes as ASCII characters. For example, if the MCU transmits a byte of value 65, the character ‘A’ appears on the terminal. And, vice versa, if we type ‘B’ on the terminal, a byte of value 66 is received by

the MCU. Hence, we called our functions, `uart_write_char()` and `uart_read_char()`.

The two functions above are based on manually checking the flags. Alternatively, the UART transmit and receive flags can be setup to raise interrupts. The interrupt enable flags can be found in the configuration registers at the end of Chapter 30 in the Family User's Guide.

Testing the UART Transmission

Test the UART transmission by writing a code that transmits the numbers 0 to 9 repetitively over UART. Each number should appear on a new line. Introduce a small delay so the numbers don't go too fast and flash the red LED to indicate the ongoing activity. At the same time, read any incoming data over UART. If the user types 1 on the keyboard, turn the green LED on and if the user types 2, turn the green LED off. Incorporate the three functions above in your code along with the symbolic constants definitions.

A new line can be transmitted by transmitting the line feed (LF) character '\n' followed by the carriage return (CR) character '\r'. The first character brings the cursor one line down and the second character rewinds the cursor to the leftmost colon on the screen.

On the PC, open the terminal application and setup the same configuration we did at the MCU as summarized in Table 6.1. On Windows, you can use Device Manager to find the COM port used by the backchannel UART. Some terminal applications, like TeraTerm, show the active COM ports within the application.

Perform the following:

- Complete the code and demo it to the TA.
- Submit the code in your report.

6.2 Transmitting Integers & Strings over UART

Write two functions that transmit integers and strings over UART. The function headers are shown below. The first function transmits 16-bit unsigned integers. Since bytes transmitted over UART appear as ASCII characters on the PC terminal (e.g. 65 appears as A), the integer on the MCU needs to be parsed into digits. For example, to transmit the integer n=4567, it is parsed into the digits (4,5,6,7) and the ASCII of these digits are transmitted. Test this function for small values and for large values.

```
void uart_write_uint16 (unsigned int n);
void uart_write_string (char * str);
```

The second function is very simple as it transmits each character of the string over UART. The string is null-terminated. These two functions should use the functions that we implemented earlier.

Perform the following:

- Complete the code and demo it to the TA.
- Test the function with small values and large values.
- Submit the code in your report.

6.3 Modifying the UART Configuration

Modify the UART configuration by making the following changes. Use ACLK = 32 KHz instead of SMCLK and setup a baud rate of 4800 baud. Note that in this case we can't do oversampling reception since the 32 KHz frequency is not higher than 16x4800. Therefore, don't check the oversampling bit in the configuration.

This is how we should approach this modification. First, lookup the dividers and modulators from the Family User's Guide (Chapter 30) for the case of ACLK=32,768 Hz and 4800 baud. These are found in the large table and are called: UCBR, UCBRF, UCBRS, UCOS16. Note that UCBRF is not used since we're not doing oversampling reception. Secondly, look at the registers at the end of Chapter 30 and modify our UART initialization function and call it `void Initialize_UART_2()`. Finally, make sure that the settings in the terminal application are modified to match the settings at the MCU.

Perform the following:

- Complete the code and demo it to the TA.
- Test the new configuration by transmitting data to/from the PC (test in both directions).
- Submit the code in your report.

6.4 Application: Airport Runway Control

Many airports have intersecting runways which require careful coordination of takeoffs and landings to avoid simultaneous clearances. As an embedded engineer, you are in charge of designing a system that prevents simultaneous clearances to be granted on intersecting runways.

This is the flow of requests for an airplane to depart on runway 1 that intersects with runway 2. The airplane pilot contacts the tower operator and requests takeoff clearance for runway 1. Since this runway intersects other runways, the tower operator must get a "release" for that runway from the central operator before giving the airplane a clearance. Once the central operator grants the release, the tower operator clears the airplane to takeoff. Once the airplane has departed, the tower operator must contact the central operator and forfeit the release. This enables the central operator to grant another release on the same runway or on an intersecting runway. The main duty of safety falls with the central operator who never grants releases simultaneously on two intersecting runways.

In this application, the tower operators use the PC (keyboard and monitor) to make requests and monitor the current status. Such requests are transmitted over UART to the MCU board. The central

operator uses the MCU board LEDs and buttons to grant releases and monitor the current status. Finally, the central operator should have a way of sending an inquiry message to the tower operator if they held to a release for a long time.

A demo of the system can be found at the link below:

https://youtu.be/tdRD_unS3xk

The following commands are helpful in implementing a dashboard on the terminal application. They control to cursor and allow printing on a specific screen location. These commands are transmitted as strings over UART.

```
COMMANDS TO MOVE THE CURSOR
\033[2J    // Clear the whole screen
\033[1;1H  // Move cursor to 1,1 position (top left) format is: line/column
\033[10B   // Move cursor down 10 lines
\033[5A    // Move cursor up 5 lines
```

Perform the following:

- Complete the code and demo it to the TA.
- Submit the code in your report.

Student Q&A

1. What's the difference between UART and eUSCI?
2. What is the backchannel UART?
3. What's the function of the two lines of code that have P3SEL1 and P3SEL0?
4. The microcontroller has a clock of 1,000,000 Hz and we want to setup a UART connection at 9600 baud. How do we obtain a clock rate of 9600 Hz? Explain the approach at a high level.
5. A UART transmitter is transmitting data at at 1200 baud. What is receiver's clock frequency if oversampling is not used?
6. A UART transmitter is transmitting data at at 1200 baud. What is receiver's clock frequency if oversampling is used? What's the benefit of oversampling?