

EEL4742C: Embedded Systems

Experiment Eight:

Simple UART Implementation

Ryan Koons 10/22/2020

**Introduction:** This lab is entirely focused on becoming familiar with the UART method of communication. This is primarily done through the implementation (based on defined standard) of initialization, read, and write functions. UART has many different options, and therefore the programmer may be very versatile with how they choose to implement UART into their design. The initialization involves different GPIO pins being diverted to UART (with backchannel USB connection to the PC). This backchannel UART is absolutely crucial for being able to communicate with the PC. Other GPIO pins also have UART and can be used to make connections elsewhere on the msp430 launchpad. UART also has a heavy focus on clock config.—which may involve setting the baud rate (or rate of tx/rx) this rate must match in order to have proper communication. The UART write and read functions are heavily built upon the polling of TX and RX flags to wait until the line is clear and then read or write data (from master to slave). Also, later on in the lab writing strings, and integers will also be explored. These matters can be more complicated since the PC handles ASCII values; so some data manipulation must be done in order to display the result correctly on the PC. The serial window (Tera Term) is also used throughout the lab; this is a serial window that is capable of ‘watching’ a COM port (at a BAUD rate entered by the user) and printing out the information that it sees (ASCII); this is essentially the output of the PC.

**8.1 Transmitting Bytes with UART:** The first part of the lab involves creating simple UART read, write, and initialize functions. This program will be capable of writing 0-9 in an infinite loop to the PC, as well as reading 1/2 to potentially toggle the green LED on the board; additionally, the red LED toggles during each Tx (transmission). Prior to functions though, “#defines” were used to simplify some of the nomenclature for the MSP430; i.e. instead of typing UCRXIFG, just type RXFLAG. This makes the code more discernible and takes a workload off of the programmer. The main function begins with typical initializations as well as, defining a char type for a new line, carriage return, as well as the character that will increment through the infinite loop;

Next, init\_UART is called. First, this function diverts the GPIO pins for UART purposes. These pins have lots of possible applications; however, the default is GPIO. Therefore, the pins need to be diverted using the P3SEL1 and P3SEL0 registers (see Family users guide for bit assignment). It’s important to note that other GPIO pins have UART but they don’t have backchannel UART to USB, so they couldn’t be used for printing to the PC. The next step is to select SMCLK @ 1.0 Mhz (even). The clock dividers and modulators were also found in the family users guide. For the 1Mhz SMCLK, with the desired 960 BAUD (and oversampling) the following configuration was set: `UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;` Most masking here was done with the typical 8 bit (bit 0 to bit 7), where certain hex values or binary values are selected to match the decimal/hex from the guide. Last step is to exit reset state, since this will allow Tx/Rx to begin.

The next function is used to write a char via UART. This function is primarily based upon polling the TXFLAG (#define used for simplicity). The TXFLAG is 1 when ready to transmit a byte and 0 during Tx. Also, this flag will automatically go from 1 to 0 and back to 1. So the function

essentially does nothing while transmission is taking place. And once the flag is 1 again, the char is written to the buffer—"TXBUFFER."

Another function necessary is capable of reading a char via UART. This function polls the RXFLAG (also done automatically). The RXFLAG is 0 when there is no new data, and 1 when the byte is being received. This function defines a temp char, and returns NULL if no byte is received; otherwise, it returns RXBUFFER in the temp variable.

In the main function a infinite for loop is created. Inside, there is another char loop that increments from 0-9 and back to 0 again. Inside that, there is a delay loop; since the delay loop takes up the majority of cycle time this is where the RXBUFFER is checked with the read\_uart function. Two if statements are used here—if the function returns '1', turn the green LED on, if '2' turn the green LED off. This delay loop was implemented to create a slight delay between transmitted characters. Lastly, the code toggles the Red LED, writes a character, and also writes the necessary formatting characters—newline and carriage return.

### 8.1 CCS Code: Print 0-9 to PC, use keyboard 1/2 to toggle green LED

```
//Lab 8.1: Tx 0-9 to computer in infinite loop, and use keyboard 1/2 for greenLED ON/OFF
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer
#define redLED BIT0 //red LED at P1.0
#define greenLED BIT7 //green LED at P9.7
#define BUT1 BIT1 //Button 1 at P1.1
#define BUT2 BIT2 //Button 2 at P1.2

//function prototypes
void init_UART(void);
void uart_write_char(unsigned char ch);
unsigned char uart_read_char(void);

int main(void)
{
    unsigned int i; //integer for delay loop
    unsigned char ch; //char for producing 0-9 infinite loop
    //formatting parameters
    unsigned char nl='\n';
    unsigned char cr='\r';
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5; //enable GPIO Pins

    //LED init.
    P1DIR |= redLED; //set as output
```

```

P9DIR |= greenLED;
P10OUT &= ~redLED;    //red OFF
P9OUT &= ~greenLED;   //green OFF

init_UART(); //Initialize

for(;;) //infinite loop
{
    for(ch='0'; ch<='9'; ch++)//infinite transmission loop
    {
        //delay loop
        for(i=0; i<50000; i++)//check for keyboard response during delay loop
        {
            unsigned char result=uart_read_char(); //read from keyboard
            //alternatively use if( uart_read_char() == 1)
            if(result == '1')//If user types 1
                P9OUT |= greenLED; //Turn on green LED
            if(result == '2') //If user types 2
                P9OUT &= ~greenLED; //Turn off
        }

        uart_write_char(ch); //Tx current ch to PC
        uart_write_char(nl); //Tx newline (down a line)
        uart_write_char(cr); //Tx carriage return (leftmost column)
        P10OUT ^= redLED; //toggle LED
    }
}

return 0;
}

//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
    P3SEL1 &= ~(BIT4|BIT5); //Using 3.4--UCA1TXD, 3.5--UCA1RXD
    P3SEL0 |= (BIT4|BIT5);

    //Use SMCLK, other settings default
    UCA1CTLW0 |= UCSSEL_2;

    //Configure clock dividers and modulators
    //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
    UCA1BRW =6;
    //bit masking for UCBRS and UCBRF done using 0-7 bit number
    UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;

    //exit reset state, to begin Tx/Rx
    UCA1CTLW0 &= ~UCSWRST;
}

```

```

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission to complete
    while((FLAGS & TXFLAG) == 0){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}

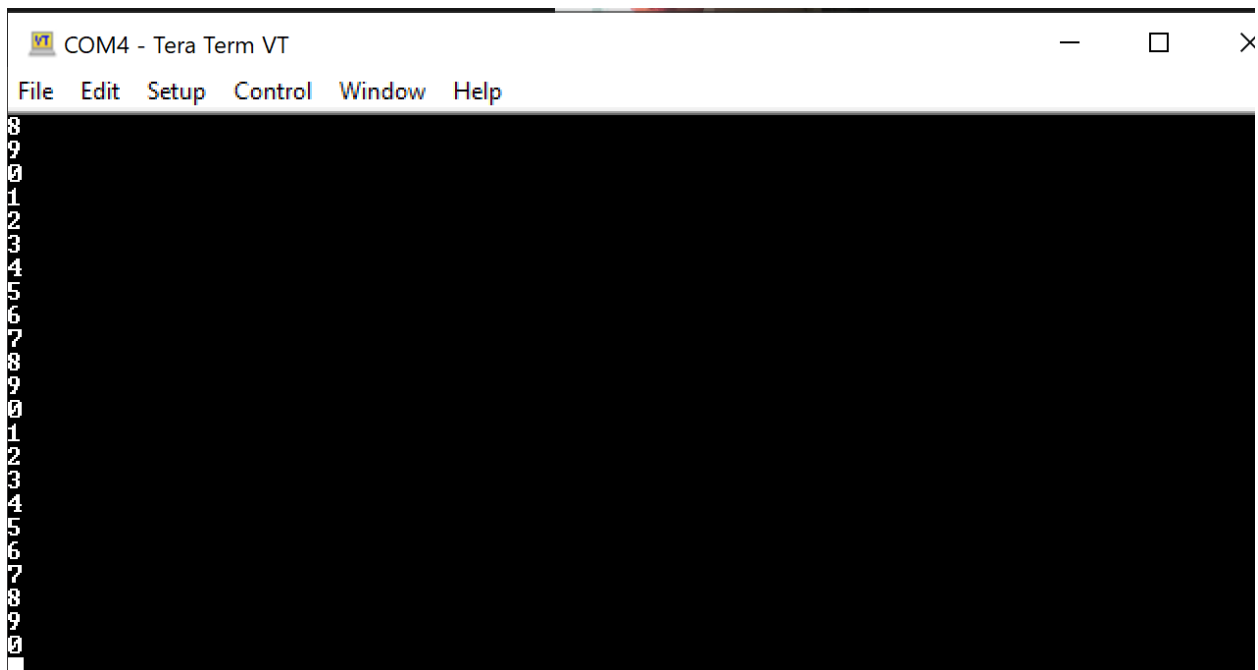
//RXFLAG--0--no new data, 1--byte is being received.
//Copy byte from receive buffer and then flag goes to zero.
unsigned char uart_read_char(void)
{
    unsigned char temp;

    //Return NULL if no byte received
    if((FLAGS & RXFLAG) == 0 )
        return '\0';

    temp= RXBUFFER; //else copy received byte (auto clears flag)
    return temp;
}

```

## 8.1 Output (Tera Term):



**8.2 Sending Unsigned 16-bit Integers over UART:** This program uses the same #defines, and uart\_write\_char function. However, this part of the lab is dedicated to creating a function capable of printing a 16-bit integer (continuously incrementing number) to the display; this is done by using modulo and parsing through the integer. This function will also use the previous urt write function, to print one digit of the integer at a time. The same Uart\_init function is used for this section of the lab (9600 BAUD). Note: Program works for small numbers (i.e. 1,2,3), as well as large numbers (i.e. 65532, 65533, 65534). Note: Program prints several integers at a time on tera term during debug mode; but, just one at a time during regular execution.

The 16-bit uart function declares an int digit that will be the parsed digit that is passed to the uart write char function. Conditional statements were utilized to grab each digit; for the leftmost digits, the condition  $n \geq 10,000$  is checked. This block of code will take the 5-digit number and divide it by 10,000 to get the left-most place. Then modulo operation is used to hack off any numbers to the left of the ten-thousandths place. Lastly, '0' + digit is passed to the UART write char function, since this will pass the proper ASCII character value for the write char function. Similar methodology is used from the one-thousandths place, down to the hundredths place. Lastly, the ones place unconditionally executed, since at the very least ( $n=0$ ), a 0 will need to display on the PC.

Within the main function, the 16 bit integer is initialized to 0, as well as the delay loop counter, and formatting character. After init\_UART is called. Inside of an infinite loop, there is a loop for incrementing the current 16bit int from 0->65534. Then a delay loop ensures time between transmission, and the 16bit write function is called, followed by the write char with the corresponding format parameters.

## 8.2 CCS Code: Printing 16-bit integers in an infinite loop on PC

```
//Lab 8.2: Sending unsigned 16-bit int incrementing loop to PC via UART
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer
#define redLED BIT0 //red LED at P1.0
#define greenLED BIT7 //green LED at P9.7
#define BUT1 BIT1 //Button 1 at P1.1
#define BUT2 BIT2 //Button 2 at P1.2

//function prototypes
void init_UART(void);
void uart_write_char(unsigned char ch);
unsigned char uart_read_char(void);
void uart_write_uint16(unsigned int n);

int main(void)
{
    unsigned int n=0; //16-bit int to pass during function call
    unsigned int i=0; //delay loop between integers
    unsigned char nl='\n'; //formatting parameters
    unsigned char cr='\r';

    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5; //enable GPIO Pins
```

```

init_UART(); //Initialize
while(1)
{
    //increment n loop
    for(n=0; n<=65534; n++){

        uart_write_uint16( n ); //call function
        //delay loops
        for(i=0; i<65000; i++)
        {}
        uart_write_char(nl); //Tx newline (down a line)
        uart_write_char(cr); //Tx carriage return (leftmost column)
    }
}

return 0;
}

//function for properly Tx 16-bit int (0-65535) to PC
void uart_write_uint16(unsigned int n)
{
    int digit; //digit for parsing through

    //check ten=thousandths place
    if(n>=10000)
    {
        //modulo 10 ensures no digits past 16-bit int allowed (out of constraint)
        digit = (n/10000) % 10; //takes leftmost digit first
        uart_write_char('0' + digit); //pass correct ascii value for digit to computer
    }

    //check thousandths place
    if(n>=1000)
    {
        digit = (n/1000) % 10; //takes next leftmost digit
        uart_write_char('0' + digit); //print next digit on screen
    }

    //modulo is also used to hack off other digits of int that is being displayed
    //(i.e. ten thousandths place that already printed to PC)

    //Hundredths place
    if(n>=100)
    {
        digit = (n/100) % 10; //take next digit
        uart_write_char('0' + digit); //print
    }

    //tenths place
    if(n>=10)
    {
        //process repeats similarly
        digit = (n/10) % 10;
        uart_write_char('0' + digit);
    }
}

```

```

    }

    //ones place always exectures
    digit = n%10; //get rid of all other digits
    uart_write_char('0'+ digit); //print digit on PC

    return;
}

//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
    P3SEL1 &= ~(BIT4|BIT5); //Using 3.4--UCA1TXD, 3.5--UCA1RXD
    P3SEL0 |= (BIT4|BIT5);

    //Use SMCLK, other settings default
    UCA1CTLW0 |= UCSSEL_2;

    //Configure clock dividers and modulators
    //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
    UCA1BRW =6;
    //bit masking for UCBRS aand UCBRF done using 0-7 bit number
    UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;

    //exit reset state, to begin Tx/Rx
    UCA1CTLW0 &= ~UCSWRST;
}

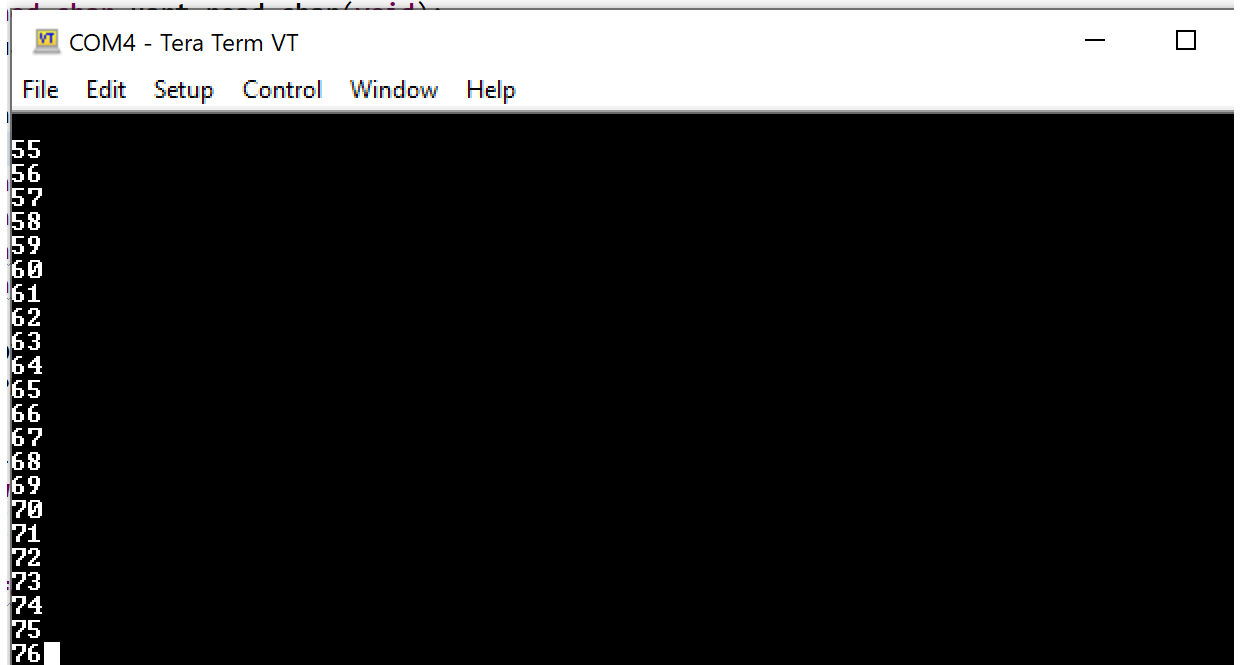
//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission to complete
    while((FLAGS & TXFLAG) ==0 ){} 

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}

```

## 8.2 Output (Tera Term):



A screenshot of a Tera Term VT window. The window title is "COM4 - Tera Term VT". The menu bar includes "File", "Edit", "Setup", "Control", "Window", and "Help". The main area is a black terminal window with white text. On the left side, a list of line numbers is displayed, starting from 55 and ending at 76. The numbers are 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, and 76. A white cursor is visible at the end of line 76.

**8.3 Sending an ASCII string over UART:** This part of the lab focuses on designing a function that can iterate through the characters of a string, and pass one at a time to the `uart_write_char` function. Note: Strings with various sizes, special characters, and spaces are all compatible with the program.

The main function is very simple for this section. It simply defines the string to be transmitted to the PC. And passes it to the `uart_write_string` function which is expecting a data type of character pointer. The function that parses the string is also very simple. It defines a parsing integer. And, it uses a while loop to increment through the string. That is, while the current character is not NULL, take it and pass it to the `uart_write_char` function from the 8.1 code. It should also be clarified that this code also uses the same `init_uart` function as the two previous sections.

### 8.3 CCS Code: Sending simple string to PC

```
//Lab 8.3: Sending an ASCII string over UART
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer
#define redLED BIT0 //red LED at P1.0
#define greenLED BIT7 //green LED at P9.7
#define BUT1 BIT1 //Button 1 at P1.1
#define BUT2 BIT2 //Button 2 at P1.2
```

```

void init_UART(void);
void uart_write_char(unsigned char ch);
void uart_write_string(char * str);

int main(void)
{
    //string to Tx to PC
    char mystring[] = "Hello World!";

    WDTCTL = WDTPW | WDTHOLD;    // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5;        //enable GPIO Pins

    init_UART(); //Initialize
    uart_write_string( &mystring[0] ); //call function

    return 0;
}

void uart_write_string(char * str)
{
    int i=0; //parsing integer
    while( str[i] != '\0') //while not at end of string
    {
        uart_write_char(str[i]); //pass current char
        i++;
    }
    return;
}

//UART config
//9600 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: SMCLK @ 1.000 MHz with Oversampling (16x)
void init_UART(void)
{
    //Divert GPIO pins for UART
    P3SEL1 &= ~(BIT4|BIT5); //Using 3.4--UCA1TXD, 3.5--UCA1RXD
    P3SEL0 |= (BIT4|BIT5);

    //Use SMCLK, other settings default
    UCA1CTLW0 |= UCSSEL_2;

    //Configure clock dividers and modulators
    //UCBR=6, UCBRF=8, UCBRS=0x20, UCOS16=1 (over-sampling)
    UCA1BRW =6;
    //bit masking for UCBRS and UCBRF done using 0-7 bit number
    UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;

    //exit reset state, to begin Tx/Rx
    UCA1CTLW0 &= ~UCSWRST;
}

```

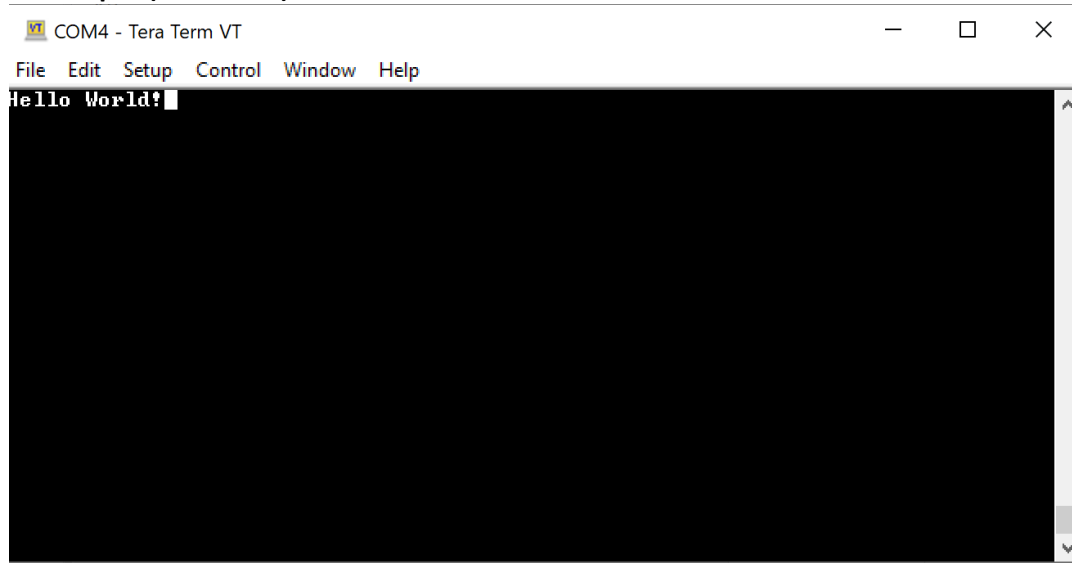
```

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission to complete
    while((FLAGS & TXFLAG) ==0 ){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}

```

### 8.3 Output (Tera Term):



**8.4 Alternate UART Configuration:** This program is very similar to the program from section 8.2; however, it operates with a different UART config. That is the primary focus in this section. Init\_UART2 seeks to utilize the ACLK 32768 crystal soldered to the board (that can be configured via GPIO). Default UART config is mainly used except for a 4800 BAUD rate. The same GPIO diversion from previous sections is also used here. Then UCSSEL\_1 is selected to choose the ACLK. It is important to note that the convert to 32kHz function should already be called prior to this event, so that the 32768 Hz is properly passed to the init UART2 function. Then different modulators and clock dividers are used; this information is found from the chip's family user's guide. For 32768 Hz (ACLK) with 4800 BAUD—UCBR is 6 and UCBRS (set bit-by-bit) is 0xEE. Last but not least, exit reset state to begin Tx.

Within the main function, the incrementing 16-bit integer, delay loop integer, and formatting characters are initialized. Then, the 32kHz function is called, followed by the init UART2 function. Within the infinite loop, there is once more a delay loop between numbers, and then each 16-bit int is passed to the write 16-bit function from 8.2 (as well as formatting characters are printed to the PC).

## 8.4 CCS Code: Alternate UART Config.

//Lab 8.4: Alternative UART Config, counting up infintely

```
#include <msp430.h>
#define FLAGS UCA1IFG //Contains the Tx and Rx flags
#define RXFLAG UCRXIFG //Rx Flag
#define TXFLAG UCTXIFG //Tx Flag
#define TXBUFFER UCA1TXBUF// Tx Buffer
#define RXBUFFER UCA1RXBUF// Rx Buffer
#define redLED BIT0 //red LED at P1.0
#define greenLED BIT7 //green LED at P9.7
#define BUT1 BIT1 //Button 1 at P1.1
#define BUT2 BIT2 //Button 2 at P1.2

void init_UART2(void); //alternate UART Init. Function
void uart_write_char(unsigned char ch);
void uart_write_uint16(unsigned int n);
void convert_32khz(void); //use 32kHz crystal soldered to the board
```

```
void main(void)
{
    unsigned int n=0; //incrementing 16-bit integer
    unsigned int i; //delay loop
    unsigned char nl='\n';//formatting parameters
    unsigned char cr='\r';
    WDTCTL = WDTPW | WDTHOLD; // stop watchdog timer
    PM5CTL0 &= ~LOCKLPM5; //enable GPIO Pins

    //LED init.
    P1DIR |= redLED; //set as output
    P9DIR |= greenLED;
    P1OUT &= ~redLED; //red OFF
    P9OUT &= ~greenLED; //green OFF
    convert_32khz();
    init_UART2(); //Initialize

    while(1)
    {
        //delay between numbers
        for(i=0; i<65000; i++){
            P1OUT ^= redLED; //toggle LEDS
            uart_write_uint16(n); //print number to screen
            uart_write_char(nl); //Tx newline (down a line)
            uart_write_char(cr); //Tx carriage return (leftmost column)

            n++;
        }
    }
}
```

```

//Function for Converting ACLK to 32768 Hz
void convert_32khz()
{
    //By default, ACLK runs on LFMODCLK at 5Mhz/128= 39 Khz

    //Reroute pins to LFXIN/LFXOUT functionality
    PJSEL1 &= ~BIT4;
    PJSEL0 |= BIT4;

    //wait until oscillator fault flags are cleared
    CSCTL0 = CSKEY; //unlock CS registers
    do {
        CSCTL5 &= ~LFXTOFFG; //local fault flag
        SFRIFG1 &= ~OIFG; //global fault flag
    } while((CSCTL5 & LFXTOFFG) != 0 );

    CSCTL0_H = 0; //Lock CS registers
    return;
}

//UART config
//4800 Baud, 8-bit data, LSB first, no parity, 1 stop bit, no flow CTL
//Initial clock: ACLK @ 32768 Hz, no oversampling
void init_UART2(void)
{
    //Divert GPIO pins for UART
    P3SEL1 &= ~(BIT4|BIT5); //Using 3.4--UCA1TXD, 3.5--UCA1RXD
    P3SEL0 |= (BIT4|BIT5);

    //Use ACLK, other settings default
    UCA1CTLW0 |= UCSSEL_1;

    //Configure clock dividers and modulators
    //UCBR=6, UCBRS=0xEE
    UCA1BRW =6;
    UCA1MCTLW = UCBRS7|UCBRS6|UCBRS5|UCBRS3|UCBRS2|UCBRS1;

    //exit reset state, to begin Tx/Rx
    UCA1CTLW0 &= ~UCSWRST;
}

void uart_write_uint16(unsigned int n)
{
    int digit; //digit for parsing through

    if(n>=10000)
    {
        //modulo 10 ensures no digits past 16-bit int allowed (out of constraint)
        digit = (n/10000) % 10; //takes leftmost digit first
        uart_write_char('0' + digit); //pass correct ascii value for digit to computer
    } //1000

    if(n>=1000)
    {

```

```

    digit = (n/1000) % 10; //takes next leftmost digit
    uart_write_char('0' + digit);
}
//modulo is also used to hack off other digits of int that is being displayed
if(n>=100)
{
    digit = (n/100) % 10; //take next digit
    uart_write_char('0' + digit); //print
}

if(n>=10)
{ //process repeats similarly
    digit = (n/10) % 10;
    uart_write_char('0' + digit);
}
//ones place always executes
digit = n%10;
uart_write_char('0'+digit);

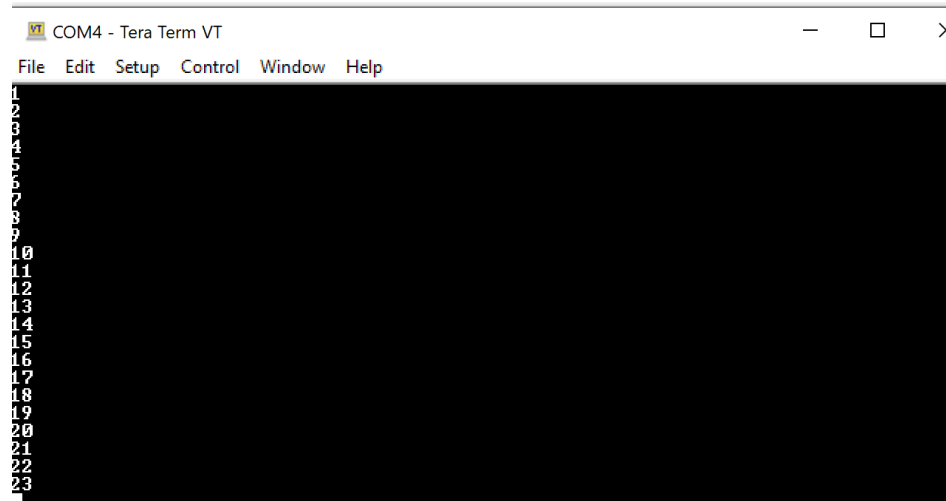
return;
}

//TXFLAG: 1 When ready to transmit, 0 during transmission
void uart_write_char(unsigned char ch)
{
    //wait for any ongoing transmission to complete
    while((FLAGS & TXFLAG) ==0){}

    //Write byte to the transmit buffer
    TXBUFFER= ch;
}

```

## 8.4 Output (Tera Term):



### Student Q&A:

- 1) UART is the communication protocol that is specifically designated for communication between multiple devices. This protocol has lots of different configurations, baud rates, and various options to ensure the signal is correctly passed between devices. The eUSCI is a module inside of the microprocessor. This module is responsible for the various channels and types of communication compatible with the MSP430. The eUSCI handles the particular details/characteristics (in this case) for UART; it allows for easy programming and interaction via configuration registers and flags.
- 2) The backchannel UART is what allows the board to communicate with the PC. There is more UART capable of GPIO pins; however, they do not connect to the backchannel UART. This connection is crucial, although it isn't its primary target, the backchannel UART has USB compatibility that allows UART connection from microcontroller to PC, and vice versa.
- 3) These lines of code divert the GPIO pins functionality. The P3SEL0 and P3SEL1 are designated in the family users guide. The MSP430 defaults to the GPIO functionality; however, the pins necessary for backchannel UART can be configured based on setting configuration registers. One bit is required to be set and the other needs to be cleared in order to divert the pins functionality for the backchannel UART.
- 4) First table 30-5 of the chip's user guide should be consulted. Assuming everything else is taken care of during UART initiation (GPIO diversion, exit reset state etc.). Then all that needs to be configured is the clock select for UART and the configuration of clock dividers and modulators; the chip user guide can be consulted to find the corrected configuration for both of these requirements. The guide references that in order to utilize SMCLK use the following code: `UCA1CTLW0 |= UCSSEL_2;` . After the clock is configured, all that remains is the clock dividers and modulators. Page 779 of the user guide states that for 960 baud with a 1Mhz clock (assume no oversampling) `UCBRx=6`, `UCBRFx=9`, and `UCBRSx=0x20`. A value of 6 for `UCBRx` can be obtained with `UCA1BRW =6;` The other two configurations require more effort. Bitmasking based on the binary value (converted from hex or decimal) must be used. So, for `UCBRFx=8` (with bits numbered 0-7), bit #3 should be set: `UCA1MCTLW=UCBRF3`. Likewise `0x20` becomes (`0x2`, `0x0`) just bit 5 being set: `UCA1MCTLW=UCBRS5`. The full bit-masking result is then: `UCA1MCTLW = UCBRS5|UCBRF3|UCOS16;` .
- 5) If no oversampling is used, then the TX rate is the same as the RX rate. In other words, the Rx clock frequency is also 1200 Hz. Although, this does not mean that the clocks are exactly synced up.
- 6) Assuming 16x oversampling is used, then the Rx is at  $16 \times 1200 = 19,200$  Baud. Oversampling is beneficial because you can take the averaging of a bit, or try to take the center of the clock samples to ensure validity of the bit being measured.

**Conclusion:** This lab was very helpful at becoming more familiar with UART and how to take advantage of it in embedded systems programming. The functions that were created today certainly will prove to be useful in future labs. However, the read and write functions are only in specific data types; this is certainly a limitation since other data types (such as hexadecimal) will require the creation of new functions in order to properly communicate between the board and PC. Two Configurations were used today, but UART has many different options and baud rates; these can be configured to conserve on power or to increase performance. As stated earlier, the BAUD rates must match on the Tera Term serial window (and UART initialization function) in order to see the data come through correctly. Oversampling may be used (in which the Rx is 16x as fast as the transmitter). This technique not only ensures validity of the signal, but it also is taken care of in a configuration register, yielding a very simple programming implementation. Moving into future labs, it will certainly be interesting how much information from the UART program will be relevant and helpful when designing other communication methods such as I2C and SPI.