Experiment 11

Asynchronous Serial Interfacing (UART)

Objective

The objective of this lab is to utilize the Universal Asynchronous Receiver/Transmitter (UART) to connect the Stellris Launchpad board to the host computer. In the example project, we send characters to the microcontroller unit (MCU) of the board by pressing keys on the keyboard. These characters are sent back (i.e., echoed, looped-back) to the host computer by the MCU and are displayed in a hyperterminal window.

Asynchronous Communication

The most basic method for communication with an embedded processor is asynchronous serial. It is implemented over a symmetric pair of wires connecting two devices (referred as host and target here, though these terms are arbitrary). Whenever the host has data to send to the target, it does so by sending an encoded bit stream over its transmit (TX) wire. This data is received by the target over its receive (RX) wire. The communication is similar in the opposite direction. This simple arrangement is illustrated in Fig. 10.1. This mode of communications is called asynchronous because the host and target share no time reference (no clock signal). Instead, temporal properties are encoded in the bit stream by the transmitter and must be decoded by the receiver.

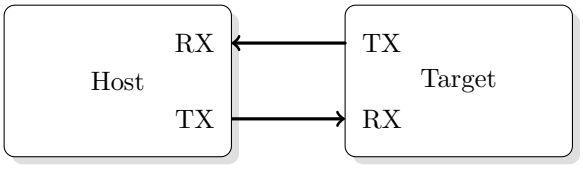

Figure 10.1: Basic Serial Communication

A commonly used device for encoding and decoding such asynchronous bit streams is a Universal Asynchronous Receiver/Transmitter (UART). UART is a circuit that sends parallel data through a serial line. UARTs are frequently used in conjunction with the RS-232 standard (or specification), which specifies the electrical, mechanical, functional, and procedural characteristics of two data communication equipment.

A UART includes a transmitter and a receiver. The transmitter is essentially a special shift

register that loads data in parallel and then shifts it out bit by bit at a specific rate. The receiver, on the other hand, shifts in data bit by bit and reassembles the data. One of the basic encodings used for asynchronous serial communications is illustrated in Fig. 10.2. Every character is transmitted in a frame which begins with a (low) start bit followed by eight data bits and ends with a (high) stop bit. The data bits are encoded as high or low signals for (1) and (0), respectively. Between frames, an idle condition is signaled by transmitting a continuous high signal. Thus, every frame is guaranteed to begin with a high-low transition and to contain at least one low-high transition. Alternatives to this basic frame structure include different numbers of data bits (e.g. 9), a parity bit following the last data bit to enable error detection, and longer stop conditions. Fig. 10.2

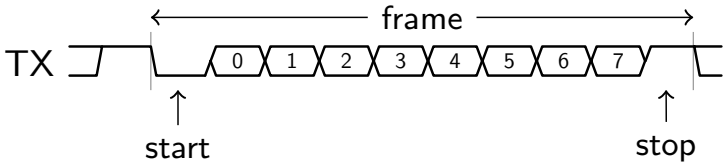

Figure 10.2: Transmission of a byte

There is no clock directly encoded in the signal the start transition provides the only temporal information in the data stream. The transmitter and receiver each independently maintain clocks running at (a multiple of) an agreed frequency commonly called the baud rate. These two clocks are not synchronized and are not guaranteed to be exactly the same frequency, but they must be close enough in frequency (better than 2%) to recover the data. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud-rate (i.e., number of bits per second), the number of data bits and stop bits, and use of parity bit.

To understand how the UART's receiver extracts encoded data, assume it has a clock running at a multiple of the baud rate (e.g., 16x). Starting in the idle state (as shown in Fig. 10.3), the receiver samples its RX signal until it detects a high-low transition. Then, it waits 1.5 bit periods (24 clock periods) to sample its RX signal at what it estimates to be the center of data bit 0. The receiver then samples RX at bit-period intervals (16 clock periods) until it has read the remaining 7 data bits and the stop bit. From that point this process is repeated. Successful extraction of the data from a frame requires that, over 10.5 bit periods, the drift of the receiver clock relative to the transmitter clock be less than 0.5 periods in order to correctly detect the stop bit.

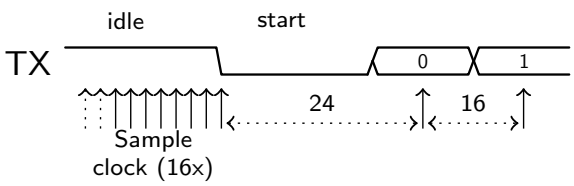

Figure 10.3: UART Signal Decoding

UART in LM4F120

The simplest form of UART communication is based upon polling the state of the UART device. The UART module in LM4F120 microntroller has 16-element FIFO and 10-bit shift register, which cannot be directly accessed by the programmer. The FIFO and shift register in the transmitter are separate from the FIFO and shift register associated with the receiver. While the data register occupies a single memory word, it is really two separate locations; when the data register is written, the written character is transmitted by the UART. When the data register is read, the character most recently received by the UART is returned. The UART Flag Register (UARTFR) contains a number of flags to determine the current UART state. The important flags are:

```
TXFE -- Transmit FIFO Empty
TXFF -- Transmit FIFO Full
RXFE -- Receive FIFO Empty
RXFF -- Receive FIFO Full
```

To transmit data using the UART, the application software must make sure that the transmit FIFO is not full (it will wait if TXFF is 1) and then write to the transmit data register(e.g., UART2_DR_R). When a new byte is written to UART1_DR_R, it is put into the transmit FIFO. Byte by byte, the UART gets data from the FIFO and loads into 10-bit shift register which transmits the frame one bit at a time at a specified baud rate. The FIFO guarentees that the data are transmitted in the order they were written.

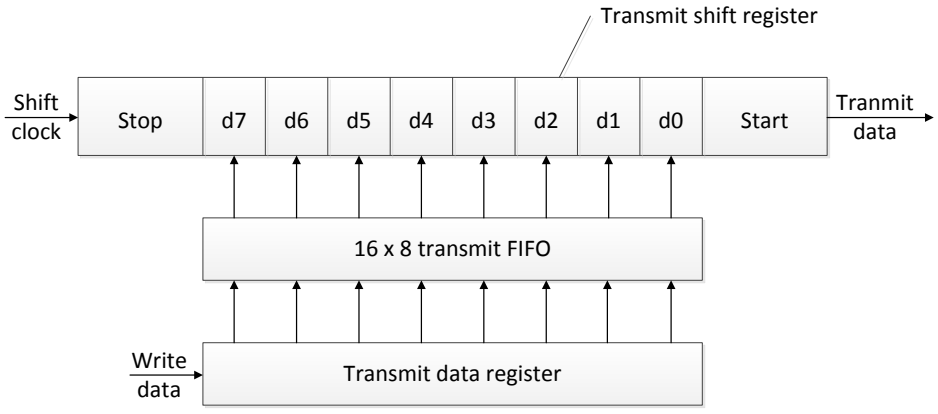

Figure 10.4: UART Data Transmission

Receiving frame is a little bit trickier than transmission as we have to synchronize the receive shift register with the data. The receive FIFO empty flag, RXFE, is clear when new input data are in the receive FIFO. When the software reads from UART1_DR_R, data are removed from the FIFO. The other flags associated with the receiver are RXFF (Receive FIFO Full). Four status bits are also associated with each byte of data. These status bits are Overrun Error(OE), Break Error(BE), Parity Error(PE) and Framing Error(FE). The status of these bits can be

checked using UART Receive Status/Error Clear Register(UARTRSR/UARTECR).

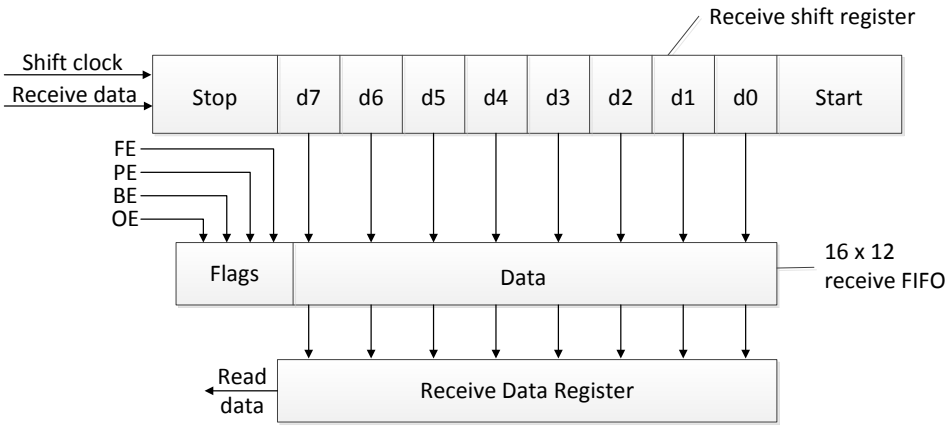

Figure 10.5: UART Data Reception

Initialization and Configuration

Stellaris Launchpad has eight UART modules connected to different ports of the microcontroller.[Table 14-1] In this lab, we will be using UART module 2 connected to PD6 (U2Rx) and PD7 (U2Tx) of GPIO port D. As with all other peripherals, UART must be initialized before it can be used. This initialization includes pin configuration, clock distribution, and device initialization. The steps required to initialize the UART are stated below:

1. The first initialization step is to enable the clock signal to the respective UART module in Run Mode Clock Gating Control UART register (RCGCUART).[pg. 318] To enable UART2 module bit 2 of this register should be asserted.

2. Enable the clock for the appropriate GPIO port to which UART is connected. The clock can be enabled using the RCGCGPIO register [pg. 314]. Table 14-1 [pg. 851] can be used to find out the port to which UART is connected.

3. To enable the alternate functionality set the appropriate bits of GPIO Alternate Function Select (GPIOAFSEL) register.[pg. 630] Now, write the value in GPIO Port Control (GPIOPCTL) register to enable UART signals for the appropriate pins.[pg. 647]

4. PD7 is also connected to Non-maskable interrupt (NMI) which is protected from accidental programming.[Table 10-2] In order to use PD7 as Tx of UART2 we must disable its write protection by unlocking GPIO Commit (GPIOCR) register and set its appropriate bits.[pg. 644] To unlock GPIO Commit we must write 0x4C4F.434B to GPIO Lock (GPIOLOCK) register.[pg. 643]

Now, we will discuss the steps required to configure the UART module.

1. The first step is calculate the baud-rate divisor (BRD) for setting the required baud-rate. The baud-rate divisor is a 22-bit number consisting of a 16-bit integer and a 6-bit fractional part. UART Integer Baud-Rate Divisor (UARTIBRD) register specifies the

integer part and UART Fractional Baud-Rate Divisor (UARTFBRD) [] register specifies the fractional part of baud-rate. [pg. 871, 872] Following expression gives the relation of baud-rate divisor and system clock [pg. 851]

$$BRD = BRDI + BRDF = UARTSysClk/(ClkDiv * BaudRate)$$

As the system clock (SysClk) is 16MHz and the desired baud-rate is 115200 bits/sec, then the baud-rate divisor can be calculated as:

$$BRD = 16,000,000/(16 * 115,200) = 8.6805$$

So, the value 8 should be written in DIVINT field of UARTIBRD register. Fractional part of baud-rate divisor is calculated in the following equation and the result should be written to the DIVFRAC field of UARTFBRD register.

$$UARTFBRD[DIVFRAC] = integer(0.6805 * 64 + 0.5) = 44$$

2. After calculating the baud rate divisor we must disable the UART by asserting UARTEN bit in UART Control (UARTCTL) register. [pg. 875]

3. Integer and fractional values of baud rate divisor, calculated previously, should be written to the appropriate bits of UARTIBRD and UARTFRD registers respectively.

4. Write the desired parameters for the serial communication you want to configure in UART Line Control (LCRH) register.[pg. 873] In this experiment, we will be using a word length of 8, one stop bit and enable the FIFOs.

5. To configure the clock source for UART configure the appropriate bit of UART Clock Configuration (UARTCC) register.[pg. 899] We will be using system clock for our experiment.

6. After configuring all the parameters, now enable the UART by asserting the UARTEN bit in UART Control (UARTCTL) register.

Source Code

```
1
2 // Register definitions for Clock Enable
3 #define SYSCTL_RCGCUART_R       (*((volatile unsigned long *)0x400FE618))
4 #define SYSCTL_RCGCGPIO_R       (*((volatile unsigned long *)0x400FE608))
5
6 // Register definitions for GPIO PortD
7 #define GPIO_PORTD_AFSEL_R      (*((volatile unsigned long *)0x40007420))
8 #define GPIO_PORTD_PCTL_R       (*((volatile unsigned long *)0x4000752C))
9 #define GPIO_PORTD_DEN_R        (*((volatile unsigned long *)0x4000751C))
10 #define GPIO_PORTD_DIR_R       (*((volatile unsigned long *)0x40007400))
11 #define GPIO_PORTD_LOCK_R      (*((volatile unsigned long *)0x40007520))
```

```
12 #define GPIO_PORTD_CR_R          (*((volatile unsigned long *)0x40007524))
13
14 // Register definitions for UART2 module
15 #define UART2_CTL_R              (*((volatile unsigned long *)0x4000E030))
16 #define UART2_IBRD_R             (*((volatile unsigned long *)0x4000E024))
17 #define UART2_FBRD_R             (*((volatile unsigned long *)0x4000E028))
18 #define UART2_LCRH_R             (*((volatile unsigned long *)0x4000E02C))
19 #define UART2_CC_R               (*((volatile unsigned long *)0x4000EFC8))
20 #define UART2_FR_R               (*((volatile unsigned long *)0x4000E018))
21 #define UART2_DR_R               (*((volatile unsigned long *)0x4000E000))
22
23 // Macros for initialization and configuration of UART2
24 #define UART2_CLK_EN             0x00000004  // Enable clock for UART2
25 #define GPIO_PORTD_CLK_EN        0x00000008  // Enable clock for GPIO_PORTD
26
27 #define GPIO_PORTD_UART2_CFG     0x000000C0  // Digital enable
28                                              // Activate alternate function
                                                    for PD6 and PD7
29 #define GPIO_PCTL_PD6_U2RX       0x01000000  // Configure PD6 as U2RX
30 #define GPIO_PCTL_PD7_U2TX       0x10000000  // Configure PD7 as U2TX
31 #define GPIO_PORTD_UNLOCK_CR     0x4C4F434B  // Unlock Commit register
32 #define GPIO_PORTD_CR_EN         0x000000FF  // Disable write protection
33
34 #define UART_CS_SysClk           0x00000000  // Use system as UART clock
35 #define UART_CS_PIOSC            0x00000005  // Use PIOSC as UART clock
36 #define UART_LCRH_WLEN_8         0x00000060  // 8 bit word length
37 #define UART_LCRH_FEN            0x00000010  // Enable UART FIFOs
38 #define UART_FR_TXFF             0x00000020  // UART Transmit FIFO Full
39 #define UART_FR_RXFE             0x00000010  // UART Receive FIFO Empty
40 #define UART_CTL_UARTEN          0x00000001  // Enable UART
41 #define UART_LB_EN               0x00000080  // Use UART in Loopback mode
42
43 // Function definitions
44 unsigned char UART_Rx(void);
45 void UART_Tx(unsigned char data);
46 void UART_Tx_String(char *pt);
47 void UART_Rx_String(char *bufPt, unsigned short max);
48
49 //Intialize and configure UART
50 void UART_Init(void){
51
52   // Enable clock for UART2 and GPIO Port D
53   SYSCTL_RCGCUART_R |= UART2_CLK_EN;          // Activate UART2
54   SYSCTL_RCGCGPIO_R |= GPIO_PORTD_CLK_EN;     // Activate Port D
55
56   // Configuration to use PD6 and PD7 as UART
57   GPIO_PORTD_LOCK_R = GPIO_PORTD_UNLOCK_CR;   // Unlock commit register
58   GPIO_PORTD_CR_R |= GPIO_PORTD_CR_EN;        // Enable U2Tx on PD7
```

```
59   GPIO_PORTD_DEN_R |= GPIO_PORTD_UART2_CFG;    // Enable digital I/O on PD6
       -7
60   GPIO_PORTD_AFSEL_R |= GPIO_PORTD_UART2_CFG; // Enable alt. func. on PD6-7
61   GPIO_PORTD_PCTL_R |= (GPIO_PCTL_PD6_U2RX | GPIO_PCTL_PD7_U2TX);
62
63   //Configuration of UART2 module
64   UART2_CTL_R &= ~UART_CTL_UARTEN;    // Disable UART
65     // IBRD = int(16,000,000 / (16 * 115,200)) = int(8.6805)
66   UART2_IBRD_R = 8;
67   // FBRD = int(0.6805 * 64 + 0.5) = 44
68   UART2_FBRD_R = 44;
69   // 8-bit word length, no parity bit, one stop bit, FIFOs enable
70   UART2_LCRH_R = (UART_LCRH_WLEN_8|UART_LCRH_FEN);
71   UART2_CC_R = UART_CS_SysClk;     // Use system clock as UART clock
72   //UART2_CTL_R |= UART_LB_EN;     // Enable loopback mode
73   UART2_CTL_R |= UART_CTL_UARTEN; // Enable UART2
74
75 }
76
77 // Wait for input and returns its ASCII value
78 unsigned char UART_Rx(void){
79   while((UART2_FR_R & UART_FR_RXFE) != 0);
80   return((unsigned char)(UART2_DR_R & 0xFF));
81 }
82
83 /* Accepts ASCII characters from the serial port and
84 adds them to a string. It echoes each character  as it
85 is inputted. */
86 void UART_Rx_String(char *pt, unsigned short max) {
87   int length=0;
88   char character;
89
90   character = UART_Rx();
91   if(length < max){
92     *pt = character;
93     pt++;
94     length++;
95     UART_Tx(character);
96   }
97
98   *pt = 0;
99 }
100
101 // Output 8-bit to serial port
102 void UART_Tx(unsigned char data){
103   while((UART2_FR_R & UART_FR_TXFF) != 0);
104   UART2_DR_R = data;
105 }
```

```
106
107 // Output a character string to serial port
108 void UART_Tx_String(char *pt){
109   while(*pt){
110     UART_Tx(*pt);
111     pt++;
112   }
113 }
114
115 int main(void){
116
117   char string[17];
118
119   UART_Init();
120
121   // The input given using keyboard is displayed on hyperterminal
122   // .i.e., data is echoed
123   UART_Tx_String("Enter Text: ");
124
125   while (1){
126     UART_Rx_String(string,16);
127   }
128 }
```