Experiment 8

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. 8.1. 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.

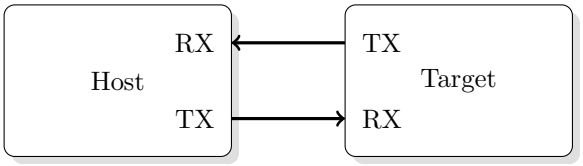

Figure 8.1: Basic Serial Communication

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. 8.2. 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. 8.2

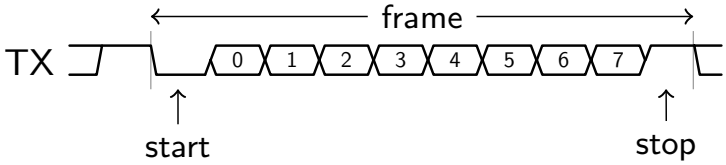

Figure 8.2: Transmission of a byte

There is no clock directly encoded in the signal (in contrast with signaling protocols such as Manchester encoding) – 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. 8.3), 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.

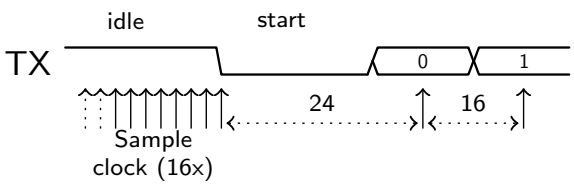

Figure 8.3: UART Signal Decoding

UART in TM4C123

The simplest form of UART communication is based upon polling the state of the UART device. The UART module in TM4C123 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., UART0_DR_R). When a new byte is written to UART0_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.

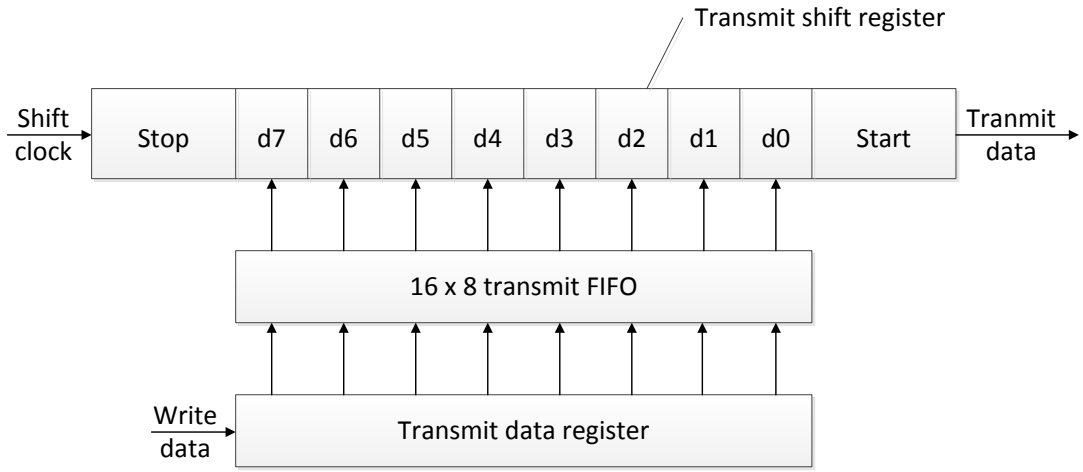

Figure 8.4: UART Data Transmission

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 UART0_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).

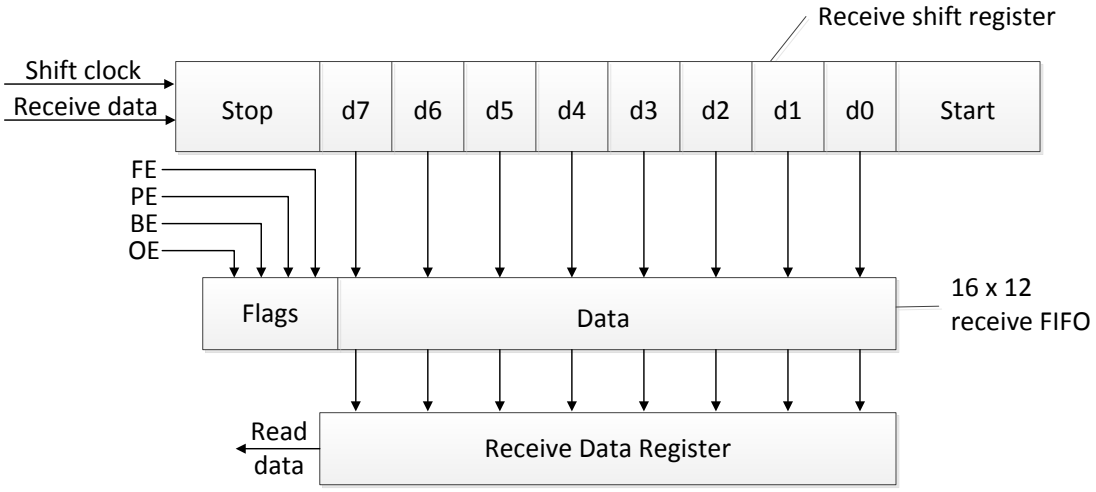

Figure 8.5: UART Data Reception

Initialization and Configuration

TivaC Launchpad has eight UART modules connected to different ports of the microcontroller. In this lab, we will be using UART module 0 connected to PA0 (U0Rx) and PA1 (U0Tx) of GPIO port A. 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). To enable UART0 module bit 0 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.

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

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 baudrate. 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. Following expression gives the relation of baud-rate divisor and system clock.

$$\boxed{BRD = BRDI + BRDF = UARTSysClk/(ClkDiv * BaudRate)}$$

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:

$$\boxed{BRD = 16,000,000/(16 * 115200)}$$

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.

$$\boxed{UARTFBRD[DIVFRAC] = integer(0.6805 * 64 + 0.5) = 44}$$

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

Note1: *The register map for System Control module (for RCGCUART) is to be consulted from article* ***5.4 - System Control - Register Map*** *of the controller datasheet.*

Note2: *The register map for UART is to be consulted from article* ***14.5 - Universal Asynchronous Receivers/Transmitters (UARTs) - Register Map*** *of the controller datasheet.*

Note5: *The register map for GPIO is to be consulted from article* ***10.4 - General-Purpose Input/Outputs (GPIOs) - Register Map*** *of the controller datasheet.*

Source Code

Template for source code is provided on piazza. It configures UART0 to communicate with the computer and display the echoed characters on hyperterminal (putty).
UART2 can also be configured. It requires to unlock PD7 and extension board for RS232 connector.