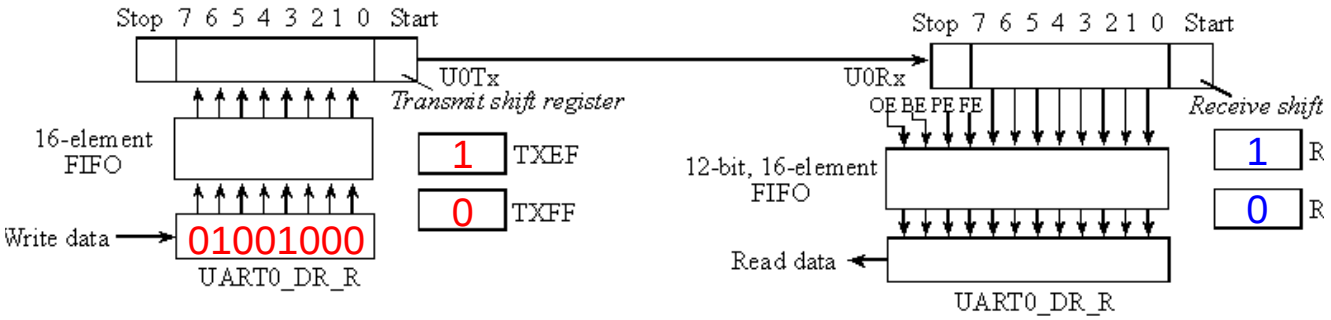
We will begin with transmission, because it is simple. The transmitter portion of the UART includes a data output pin, with digital logic levels as drawn in the following interactive tool. The transmitter has a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer. The FIFO and shift register in the transmitter are separate from the FIFO and shift register associated with the receiver. In other words each UART has a receiver and a transmitter, but the interactive tool just shows the transmitter on one microcontroller and the receiver on the other. To output data using the UART, the transmitter software will first check to make sure 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**). The bits are shifted out in this order: start, $\mathbf{b_0}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{b_3}$, $\mathbf{b_4}$, $\mathbf{b_5}$, $\mathbf{b_6}$, $\mathbf{b_7}$, and then stop, where $\mathbf{b_0}$ is the LSB and $\mathbf{b_7}$ is the MSB. The transmit data register is write only, which means the software can write to it (to start a new transmission) but cannot read from it. Even though the transmit data register is at the same address as the receive data register, the transmit and receive data registers are two separate registers. The transmission software can write to its data register if its TXFF (transmit FIFO full) flag is zero. TXFF equal to zero means the FIFO is not full and has room. The receiving software can read from its data register if its RXFE (receive FIFO empty) flag is zero. RXFE equal to zero means the FIFO is not empty and has some data. While playing the following interactive tool, watch the behavior of the TXFF and RXFE flags.

Next Click Next to Send 'H' to shift register

Run



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 them into the 10-bit transmit shift register. The 10-bit shift register includes a start bit, 8 data bits, and 1 stop bit. Then, the frame is shifted out one bit at a time at a rate specified by the baud rate register. If there are already data in

the FIFO or in the shift register when the **UART0_DR_R** is written, the new frame will wait until the previous frames have been transmitted, before it too is transmitted. The FIFO guarantees the data are transmitted in the order they were written. The serial port hardware is actually controlled by a clock that is 16 times faster than the baud rate, referred to in the datasheet as **Baud16**. When the data are being shifted out, the digital hardware in the UART counts 16 times in between changes to the **U0Tx** output line.

The software can actually write 16 bytes to the **UART0_DR_R**, and the hardware will send them all one at a time in the proper order. This FIFO reduces the software response time requirements of the operating system to service the serial port hardware. Unfortunately, it does complicate the hardware/software timing. At 9600 bits/sec, it takes 1.04 ms to send a frame. Therefore, there will be a delay ranging from 1.04 and 16.7 ms between writing to the data register and the completion of the data transmission. This delay depends on how much data are already in the FIFO at the time the software writes to **UART0_DR_R**.

Receiving data frames is a little trickier than transmission because we have to synchronize the receive shift register with the incoming data. The receiver portion of the UART includes a **U0Rx** data input pin with digital logic levels. At the input of the microcontroller, true is 3.3V and false is 0V. There is also a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer (shown on the right side of the interactive tool). The receive shift register is 10 bits wide, but the FIFO is 12 bits, 8 bits of data and 4 error flags. Again the receive shift register and receive FIFO are separate from those in the transmitter. The receive data register, **UART0_DR_R**, is read only, which means write operations to this address have no effect on this register (recall write operations activate the transmitter). The receiver obviously cannot start a transmission, but it recognizes a new frame by its start bit. The bits are shifted in using the same order as the transmitter shifted them out: start, $\mathbf{b_0}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{b_3}$, $\mathbf{b_4}$, $\mathbf{b_5}$, $\mathbf{b_6}$, $\mathbf{b_7}$, and then stop.

Help

There are six status bits generated by receiver activity. 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. When the FIFO becomes empty, the **RXFE** flag will be set, meaning there are no more input data. There are other flags associated with the receiver. There is a Receive FIFO full flag **RXFF**, which is set when the FIFO is full. There are four status bits associated with each byte of data. For this reason, the receive FIFO is 12 bits wide. The overrun error, **OE**, is set when input data are lost because the FIFO is full and more input frames are arriving at the receiver. An overrun error is caused when the receiver interface latency is too large. The break error, **BE**, is set when the input is held low for more than a frame. Parity is a mechanism to send one extra bit so the receiver can detect if there were any errors in transmission. With even parity the number of 1's in the data plus parity will be an even number. The **PE** bit is set on a parity error. Because the error rate is so low, most systems do not implement parity. We will not use parity in this class. The framing error, **FE**, is set when the stop bit is incorrect. Framing errors are probably caused by a mismatch in baud rate.

The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 bits of data one at a time from the **U0Rx** line. The internal clock is 16 times faster than the baud rate. After the 1 to 0 edge, the receiver waits 8 internal clocks and samples the start bit. 16 internal clocks later it samples $\mathbf{b_0}$. Every 16 internal clocks it samples another bit until it reaches the stop bit. The UART needs an internal clock faster than the baud rate so it can wait the half a bit time between the 1 to 0 edge beginning the start bit and the middle of the bit window needed for sampling. The start and stop bits are removed (checked for framing errors), the 8 bits of data and 4 bits of status are put into the receive FIFO. The hardware FIFO implements buffering so data is safely stored in the receiver hardware if the software is performing other tasks while data is arriving.

Observation: If the receiving UART device has a baud rate mismatch of more than 5%, then a framing error can occur when the stop bit is incorrectly captured.

An **overrun** occurs when there are 16 elements in the receive FIFO, and a 17th frame comes into the receiver. In order to avoid overrun, we can design a real-time system, i.e., one with a maximum latency. The latency of a UART receiver is the delay between the time when new data arrives in the receiver (**RXFE**=0) and the time the software reads the data register. If the latency is always less than 160 bit times, then overrun will never occur.

Observation: With a serial port that has a shift register and one data register (no FIFO buffering), the latency requirement of the input interface is the time it takes to transmit one data frame.