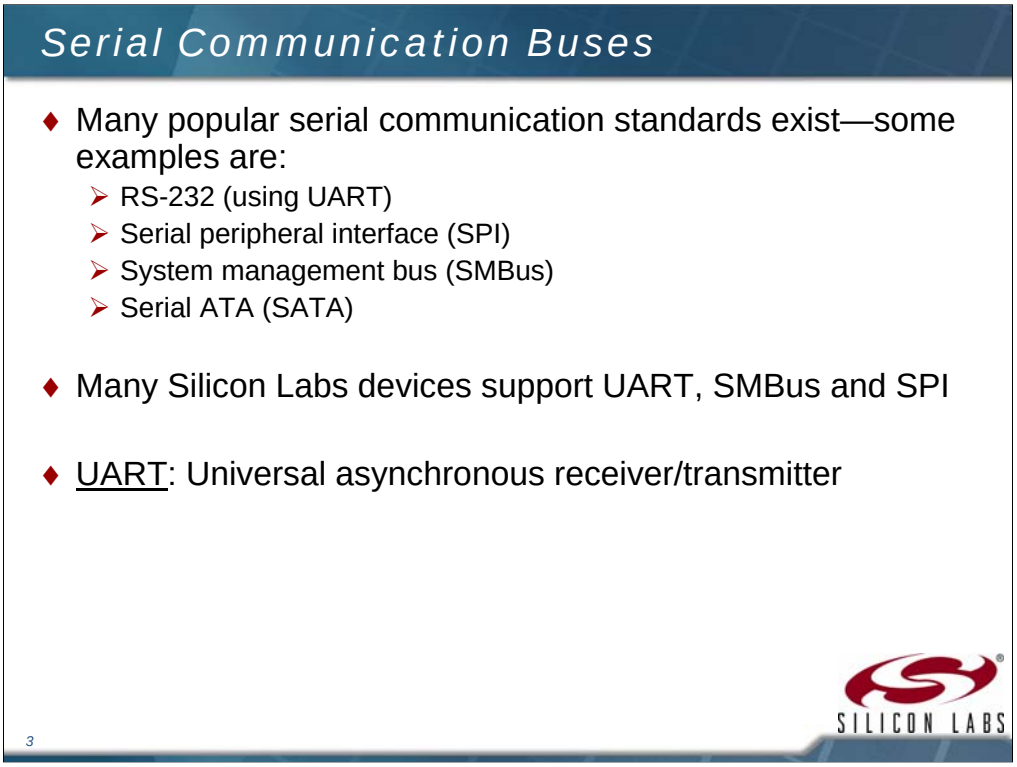

SILICON LABS
Serial Communications

Serial Communication

- Introduction
- Serial communication buses
- Asynchronous and synchronous communication
- UART block diagram
- UART clock requirements
- Programming the UARTs
- Operation modes
- Baud rate calculations—timer 1
- Initializing the UART—using timer 1
- Baud rate calculations—timer 2
- Initializing the UART—using timer 2
- UARTx interrupt flags—receiving data
- UARTx interrupt flags—sending data

2

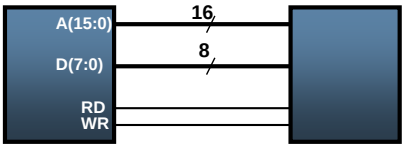
We start this lecture by looking at the functional block diagram of the UART. We will learn how to program the UARTs and the different modes in which UARTs may be configured. We will learn how to program Timer 1 and Timer 2 to generate the baud rate. The interrupt flags for receiving and sending data will be covered.

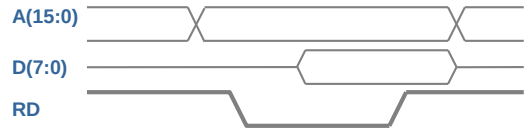


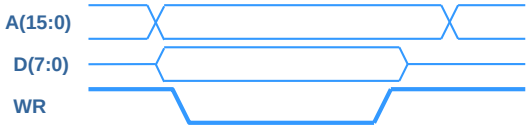
There are quite a few standards to choose from when considering a serial communications protocol and in many MCUs multiple standards are supported on the same IC. RS232 or UART is one of the most popular standards and is the one we are going to cover in this course. The Serial Peripheral interface is a synchronous serial interface that uses 4 (or 3) wires to communicate between what is considered a master device and a slave device. The data lines are for the data input, data output and the clock. The fourth wire is for the chip select such that multiple slave devices can be on the bus. The SMBus is a two wire bus that consists of one wire for data input and output and then one wire for the clock. Multiple devices can be added on the bus as the protocol provides an addressing scheme for data transfers. Serial ATA is a serial form of the parallel bus used in the PC for things like hard disk drive connections.

Parallel Communication

- Parallel communication implies sending a whole byte (or more) of data over multiple parallel wires
- Control bits used to determine the timing for reading and writing the data



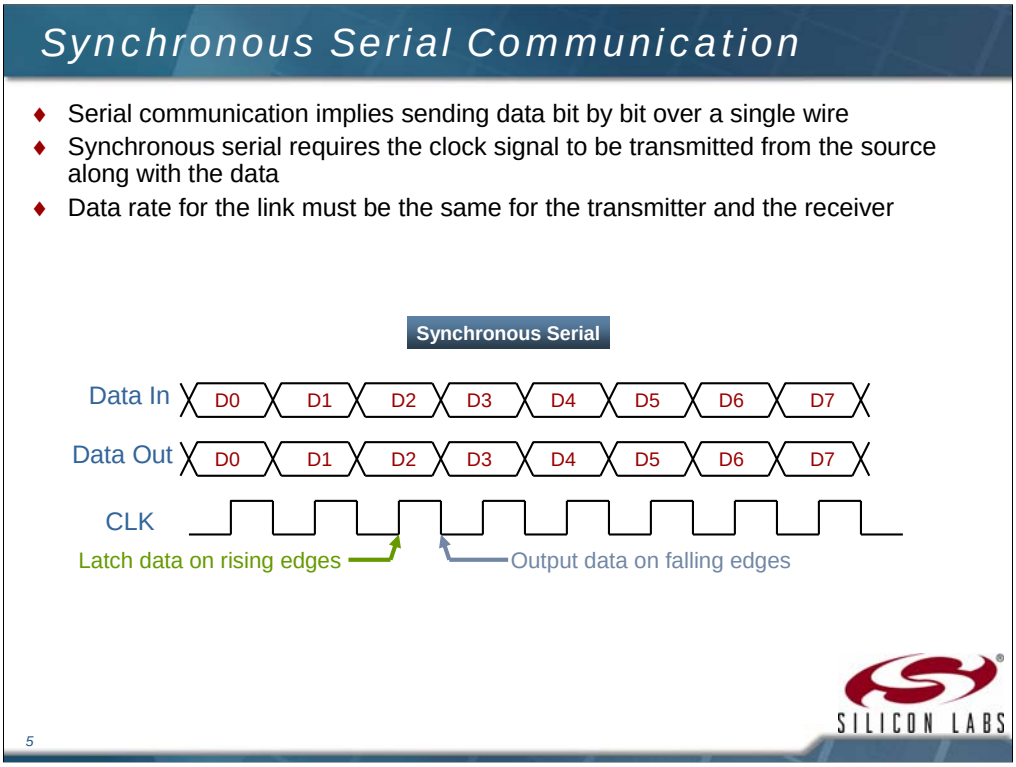




4

SILICON LABS

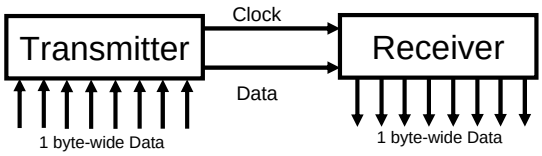
In this course we are going to cover how to transmit data between two relatively low speed devices. To accomplish this data transfer we really have two choices, transfer data in parallel or in series. Here is an example of a parallel communications using a standard External Memory Interface (EMIF) as a reference. Some Silicon Labs MCUs provide this interface to external memory or devices. In this communications method the address and data are sent as multiple wires. The number of pins required for the example shown above is 26 (address + data + control). By moving to serial we can reduce the pin count to the IC. Another aspect of parallel busses that becomes a limitation is the transmission speed. This really isn't a concern when considering interfaces in the 8 bit MCU space, however, as speeds increase the interfacing becomes much more of a challenge as the propagation delays between the data lines and their relationship with the clock sources is minimized.



We just saw how parallel communications has a wide data bus typically 8 bits or more. The data bits are sent out together in conjunction with a clock and control signaling and these interfaces require high pin count depending on the width of the data bus. With serial data we are referring to transmitting each bit one after the other over a single wire. These interfaces require low pin counts since only one pin is required to transmit or receive data. In this course we are covering very low data rate interfaces used in microcontrollers. The serial interfaces in these systems are asynchronous (clock is not sourced with the data) or synchronous (clock is sourced with the data). You can see from the diagram that the synchronous serial interface provides a clock with the data. The transmitter drives data out the pin on one edge of the clock and the receiver latches data on the following edge.

Synchronous Serial Communication

- In the **synchronous** mode, the transmitter and receiver share a common clock
- The transmitter typically provides the clock as a separate signal in addition to the serial data



Transmitter	Receiver
Shifts the data onto the serial line using its own clock	Extracts the data using the clock provided by the transmitter
Provides the clock as a separate signal	Converts the serial data back to the parallel form
No start, stop, or parity bits added to data	

6

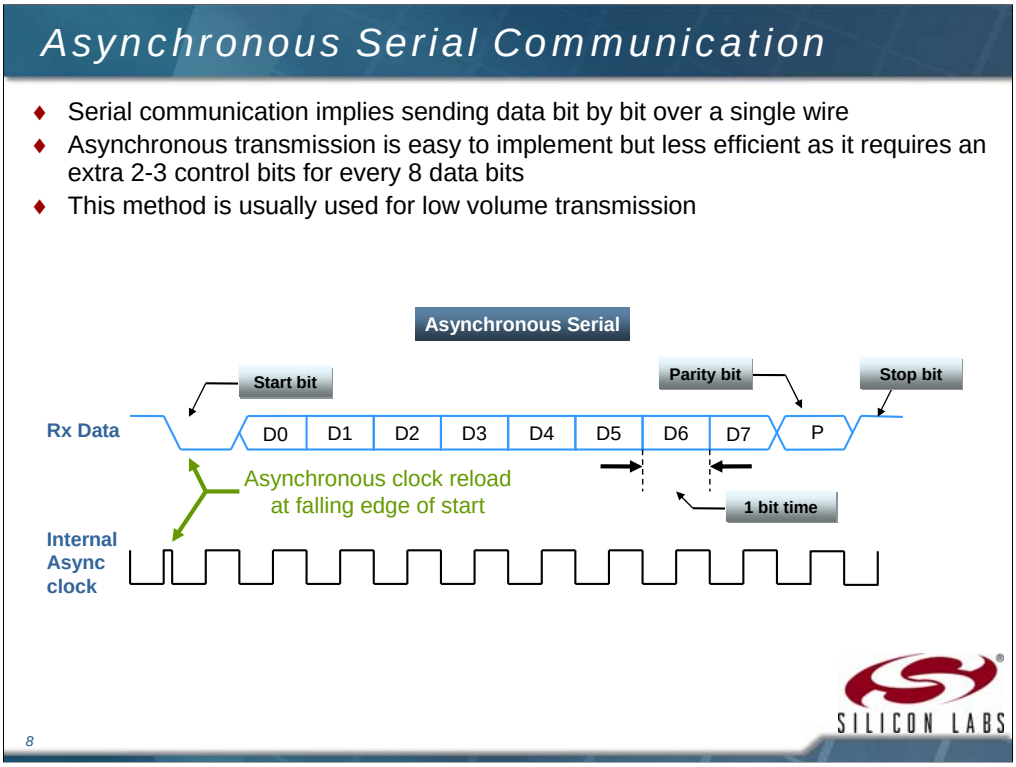
As mentioned earlier, in synchronous serial communications, the transmitter sends the data out the pin at a specific rate determined by the internal clock system. This clock is also output along with the data in order for a receiving device to know when to latch the incoming data bits. By sending the clock output along with the data there are no additional bits required to provide a synchronization time for a receiver. The receiving device just latches the data based on the received clock and converts the data internally to a parallel value for use by the CPU. Clock speed, clock polarity and data width are all established at design time. One aspect not shown here is the use of a chip enable (CE) control signal. In some synchronous serial interfaces the chip enable is used to allow multiple devices to be connected to the same bus. This is typical of Serial Peripheral Interface (SPI) devices.

Asynchronous Serial Communication Terms

- Start bit—indicates the beginning of the data word
- Stop bit—indicates the end of the data word
- Parity bit—added for error detection (optional)
- Data bits—the actual data to be transmitted
- Baud rate—the bit rate of the serial port
- Throughput—actual data transmitted per sec (total bits transmitted-overhead)
  - Example: 115200 baud = 115200 bits/sec
  - If using 8-bit data, 1 start, 1 stop, and no parity bits, the effective throughput is: 115200 * 8 / 10 = 92160 bits/sec

7

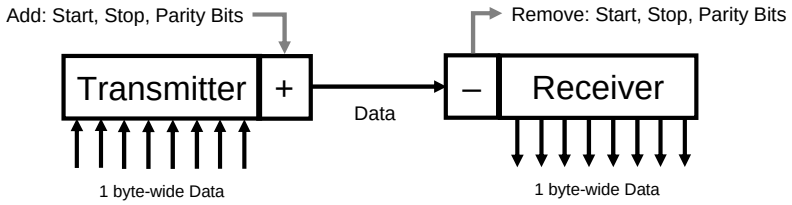
We are going to look at asynchronous serial communications. Here are some terms we hear when discussing this transmission type. Take a minute to familiarize yourself with these terms as we will be using them in subsequent slides.



In asynchronous serial communications, data is transmitted without the clock. Therefore, it is up the MCU to synchronize its internal baud rate clock to the incoming data. In the case of RS232 the start bit is used to start the process. The start bit tells the receiver the phase relationship required for the internal clock used to latch the data bits. The rate of this clock is typically determined at design time, however, there are some applications where devices are required to determine the baud rate from the incoming data stream. The example above shows one way for devices to align their internal clock with the incoming data. Once the start bit is received then the clock system restarts its count sequence allowing for the clock edges to line up according to the bit time specified.

Asynchronous Serial Communication

- With **asynchronous** communication, the transmitter and receiver do not share a common clock

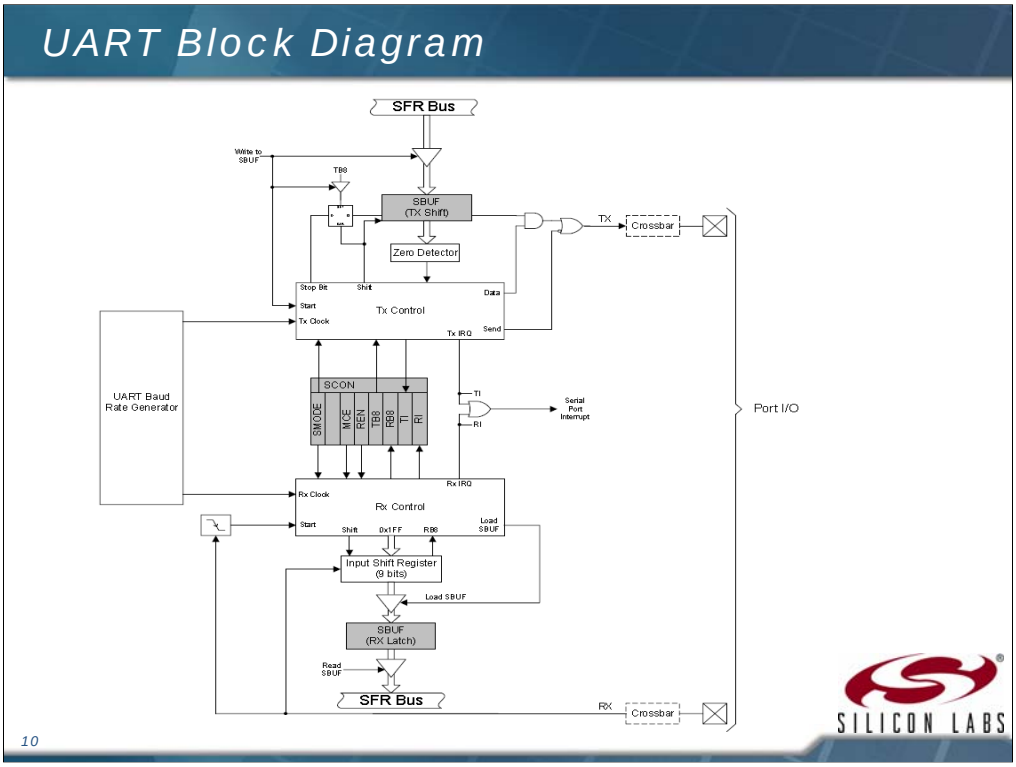


Transmitter	Receiver
Shifts the parallel data onto the serial line using its own clock	Extracts the data using its own clock
Also adds the start, stop, and parity check bits	Converts the serial data back to the parallel form after stripping off the start, stop, and parity bits

SILICON LABS

9

We mentioned earlier slides that the synchronous serial interface provides a clock with the data and we showed how an asynchronous interface receives the data on a pin and then synchronizes an internal clock to be able to accurately receive the data. It is asynchronous because the receiving device has no knowledge of the clock tolerance or jitter of the source port. The only information we have is the baud rate of the link. The baud rate is the bit rate of the link in bits/second. Some common rates for UART communications are 19200 baud, 57600 baud or 115200 baud. When the receiver starts receiving the data it will synchronize its baud clock such that the edges align with the rate of the incoming data stream. In doing so it can then latch the bits and convert them to the parallel value required by the CPU.



This diagram shows the various functional blocks of a UART peripheral found in the Silicon Labs C8051F900 family. There are two Special Function Registers (SFR) - **SBUFx** and **SCONx** – used to control and manage the serial communication. The UART peripheral is split between the transmit and receive sides. From the diagram you can see the entire interface is driven by a separate block called the UART Baud Rate Generator. This is the internal clock source used to recover the received data as well as time the transmitted data output. We will see in later slides how to set up the baud rates using the internal timers. The I/O is controlled via the crossbar which is a special Silicon Labs feature that enables peripheral usage even on the smallest pin count devices by moving the pin assignments based on utilization. Lastly is the interrupt generation which allows the CPU to be running other tasks while UART communications are active and only service the peripheral when the transmission is complete.

UART Block

- The UART0 peripheral is accessed by two SFRs—SBUFx and SCONx
- The serial port buffer (SBUFx) is two distinct buffers
  - Writing loads data to be transmitted to the write-only buffer
  - Reading accesses received data from the read-only buffer
- The serial port control register (SCONx) contains status and control bits
  - The control bits set the operating mode for the serial port, and status bits indicate the end of the character transmission or reception
  - The status bits are tested in software (polling) or programmed to cause an interrupt

11

Even though the '*programmer's model*" has only one Serial Port Buffer (SBUFx), there are essentially two separate and distinct hardware buffers (registers) - the transmit write-only buffer and the receive read-only register. Writing to SBUF0 sends the data out serially from the Transmit Shift register through the TX0 pin. The data received on RX0 pin is accumulated in the Receive Latch. A read of SBUF0 fetches the data from the Receive Latch.

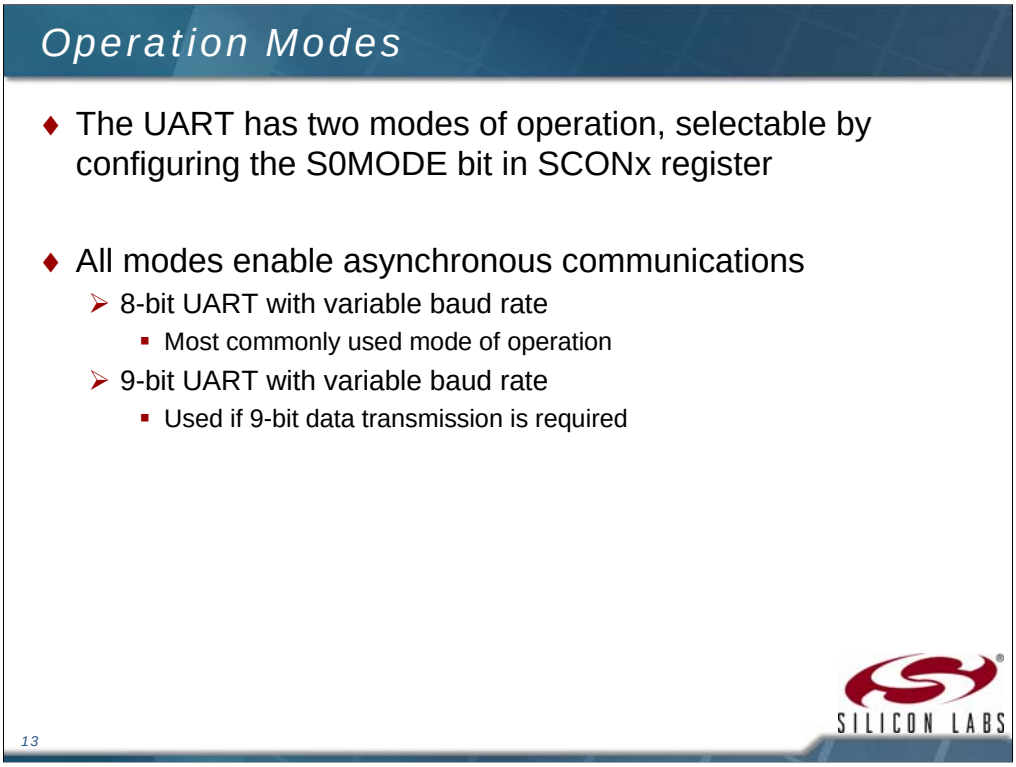
The Serial Port Control register (SCONx) contains the status and control bits. The control bits set the operating mode for the serial port, and status bits indicate the end of the character transmission or reception. The status bits are tested in software (polling) or programmed to cause an interrupt.

UART Clock Requirements

- A UART needs a clock input for bit timing
- UART baud rates are usually much lower than the MCU system clock, so the system clock cannot be directly used as the UART clock
- Timers are used to generate the UART baud rate by dividing down the system clock
  - Example: MCU system clock—24.5 MHz; UART baud rate—57600
- A bit time accuracy of ±2.5% or better is required at both the transmitter and receiver ends to be able to communicate without errors
  - Silicon Labs' internal oscillators can be used to generate baud rates as they provide +/-2% tolerance over temperature

12

This slide discusses the need for timers to be used with UARTs. Clock accuracy is important for proper UART communication , with a worst case scenario of Tx being +2% and Rx being –2%, the difference is 4%. The absolute timing error between the transmitter and receiver to avoid a bit error is 5%. We split that and give half to the TX and half to the RX, which leads to the required timing accuracy of +/- 2.5% for each side. Because the oscillator is guaranteed to +/-2 %, that leaves 0.5% to handle baud rate generation error (error due to a non-integer number of timer clocks per UART bit). If baud rates differ more than this, then the received bytes can have bit errors. The Silicon Labs internal oscillators have accuracy specifications over temperature of 2% that allow them to be used to generate the baud rate clocks without the use of an external crystal. The internal clock used for the baud rate clock is divided down to a lower rate for use as the baud rate clock. The example provided shows the 24.5 MHz clock source relative to the UART baud rate required for the application.



The S0MODE bit in the SCON0 register is used to configure the mode of operation. There are two modes available in this particular peripheral. The 8 bit mode and the 9 bit mode. In this lecture we have covered the 8-Bit UART with Variable Baud Rate (Mode 1) and will look at programming the UART peripheral for this mode in more detail so we will take a minute here to discuss the 9 bit mode. There are some applications that require multiple UARTs to be connected as a bus. The 9 bit mode is used to signal multiprocessor communications such that the ninth bit gets set to indicate that the data received represents an address. A device with a matching address can accept the data and provide the appropriate action or response. All other devices that don't have a matching address simply discard the data. The 8 bit mode is the most common mode and is used as the data representation for the RS232 standard.

Programming the UARTs

The UARTs can be programmed through the following sequence:

- Step 1: configure the digital crossbar (XBR0 and XBR2) to enable UART operation
  - Set the TXx pin to be push-pull by setting the corresponding PnMDOUT bit (PnMDOUT.n)
  - The digital crossbar has to be configured to enable TXx and RXx as external I/O pins (XBR0.0 for UART0 Tx on P0.4 and Rx on P0.5)
  - In addition, XBARE (XBR2.6) must be set to 1 to enable the crossbar
- Step 2: initialize the appropriate timers for desired baud rate generation
  - Timer 1 can be used to generate baud rate for UART0
- Step 4: select the serial port operation mode and enable/disable UART reception (SCONx register)
- Step 5: enable UART interrupts and set priority (if desired)

14

All peripherals must have their functions allocated by the crossbar in order for them to be available at the pins of the device. In many devices there are no dedicated pins for TX and RX, however some TX and RX signals are allocated by the crossbar to port 0 pin 4 for Tx and port 0 pin 5 for Rx for all cases. The XBRx registers define what peripherals get assigned to I/O through the crossbar. In addition, the I/O pins need to be configured appropriately for the direction of the signal we map to the pin. For example, the Tx signal requires the pin to be configured as a push pull output whereas the Rx signal requires the pin to be configured as an input. We can configure these through the PnMDOUT register. A suitable timer needs to be programmed to generate the baud rate. We will take a look at the configuration of the timer in later slides. The SCON0 register is used to select the serial port operation mode and enable/disable UART reception. We then enable the UART interrupts and set the priorities (if desired).

SCON0 Register

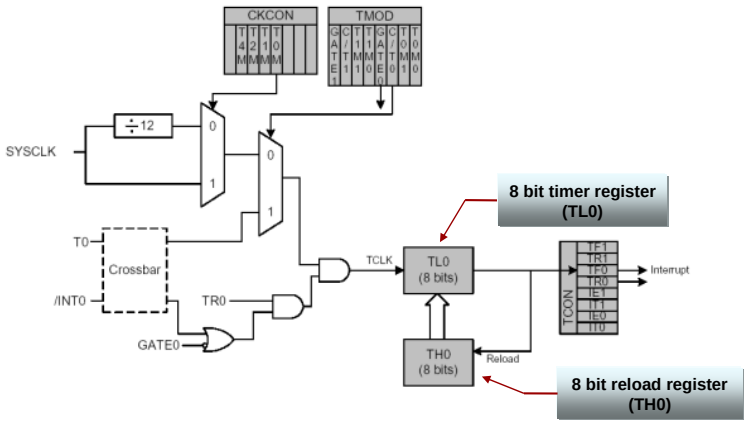
Bit	Symbol	Description
7	S0MODE	Serial Port Operation Mode **0: 8 Bit UART, Variable Baud Rate** 1: 9 Bit UART, Variable Baud Rate
6	Unused	**Unused** Read = 1b. Write = Don't care.
5	MCE0	**Multiprocessor Communication Enable** **For Mode 0 (8-bit UART): Checks for valid stop bit.** 0: Logic level of stop bit is ignored. 1: RI0 will only be activated if stop bit is logic level 1. **For Mode 1 (9-bit UART): Multiprocessor Communications Enable.** 0: Logic level of ninth bit is ignored. 1: RI0 is set and an interrupt is generated only when the ninth bit is logic 1.
4	REN0	**Receive Enable** 0: UART0 reception disabled 1: UART0 reception enabled
3	TB80	**9th Transmission Bit** The logic level of this bit will be sent as the ninth transmission bit in 9-bit UART Mode (Mode 1). Unused in 8-bit mode (Mode 0).
2	RB80	**9th Receive Bit** RB80 is assigned the value of the STOP bit in Mode 0; it is assigned the value of the 9th data bit in Mode 1.
1	TI0	**Transmit Interrupt Flag** Set by hardware when a byte of data has been transmitted by UART0 (after the 8th bit in 8-bit UART Mode, or at the beginning of the STOP bit in 9-bit UART Mode). When the UART0 interrupt is enabled, setting this bit causes the CPU to vector to the UART0 interrupt service routine. This bit must be cleared manually by software.
0	RI0	**Receive Interrupt Flag** Set to 1 by hardware when a byte of data has been received by UART0 (set at the STOP bit sampling time). When the UART0 interrupt is enabled, setting this bit to 1 causes the CPU to vector to the UART0 interrupt service routine. This bit must be cleared manually by software.

15

The **SCON0** register is used to select the serial port operation mode and enable/disable UART reception. The mode of operation is configured by programming the **S0MODE** bit. For 8 bit UART bode, set the bit to 0. **TI0** and **RI0** are the Transmit Interrupt Flag and Receive Interrupt Flag respectively. The REN0 bit is used to enable the receiver to start accepting data from another UART peripheral.

Using Timer 1 to Generate Baud Rate

- Timer 1 in mode 2 (8-bit auto-reload mode) can be used to generate the baud rate for UART0



Block diagram of Timer 0 in Mode 2 (8-bit Auto-reload mode)

We discussed the use of a timer in order to generate the baud rates required for the application. Timer 1 can be used to generate the baud rate for UART0 for this peripheral. For a detailed overview of timer operations you can check out the course that covers the different operating modes for the timers. In order to generate the baud rate to the UART we will use the timer in the 8 bit auto reload mode. As an overview to the timers course, this mode allows the timer to be used as an 8 bit timer and provides a register that is capable of reloading a count value upon overflow of the timer. The number of counts it takes the timer to overflow using this method is 256 – TH0. For example, if our TH0 value is 0xF0 we would have 16 counts on the timer before it reached the overflow since we start at 0xF0 each time. In this way we can generate different baud rates based on the system clock.

Baud Rate Calculations—Timer 1

- The Baud Rate and Timer 1 reload value (for **TH1** register) are related by the following equation:

$$\text{UartBaudRate} = \frac{1}{2} \times \text{T1\_Overflow\_Rate}$$

$$\text{T1\_Overflow\_Rate} = \frac{\text{T1}_{\text{CLK}}}{256 - \text{TH1}}$$

Baud Rate Example:
Desired baud rate = 57600 baud
Clock input to Timer 1 = System clock = 24.5 MHz

Changing equation 1:
T1_Overflow_Rate = 2 x Baud rate
T1_Overflow_Rate = 2 x 57600 = 115200

Changing equation 2:

$$TH1 = 256 - \left(\frac{T1_{CLK}}{T1\_Overflow\_Rate}\right) \text{ or } -\left(\frac{T1_{CLK}}{T1\_Overflow\_Rate}\right)$$

$$TH1 = 256 - \left(\frac{24500000}{115200}\right)$$

$$TH1 = 0x2B$$

17

SILICON LABS

This slide shows the equation used to calculate the reload value of TH1 register when Timer 1 is used to generate the baud rate. The baud rate is dependent on the value of T1M bit in the CKCON register. T1M defines whether the timer is sourced by the system clock directly or the system clock divided by 4, 12 or 48. The example provided illustrates the use of the timer to set the baud rate to 57600. The reload value we write to the TH1 register would be 0x2B in this case. Therefore, whenever the timer overflows the value 0x2B gets reloaded to the timer and the count begins again from there. 0xFD, 0xFE, 0xFF, 0x2B, 0x2C, 0x2D etc…

Initializing the UART—Using Timer 1

```
#define SYSCLK  24500000
#define BAUDRATE  57600

void UART0_Init (void)
{
   SCON0 = 0x10;                        // SCON0: 8-bit variable bit rate
                                        //        level of STOP bit is ignored
                                        //        RX enabled
                                        //        ninth bits are zeros
                                        //        clear RI0 and TI0 bits
   TH1 = -(SYSCLK/BAUDRATE/2);
   CKCON &= ~0x0B;                      // T0 = 1 (use system clock)
   CKCON |=  0x08;
   TL1 = TH1;                           // init Timer1
   TMOD &= ~0xf0;                       // TMOD: timer 1 in 8-bit autoreload
   TMOD |=  0x20;
   TR1 = 1;                             // START Timer1
   IP |= 0x10;                          // Make UART high priority
   ES0 = 1;                             // Enable UART0 interrupts

}
```

18

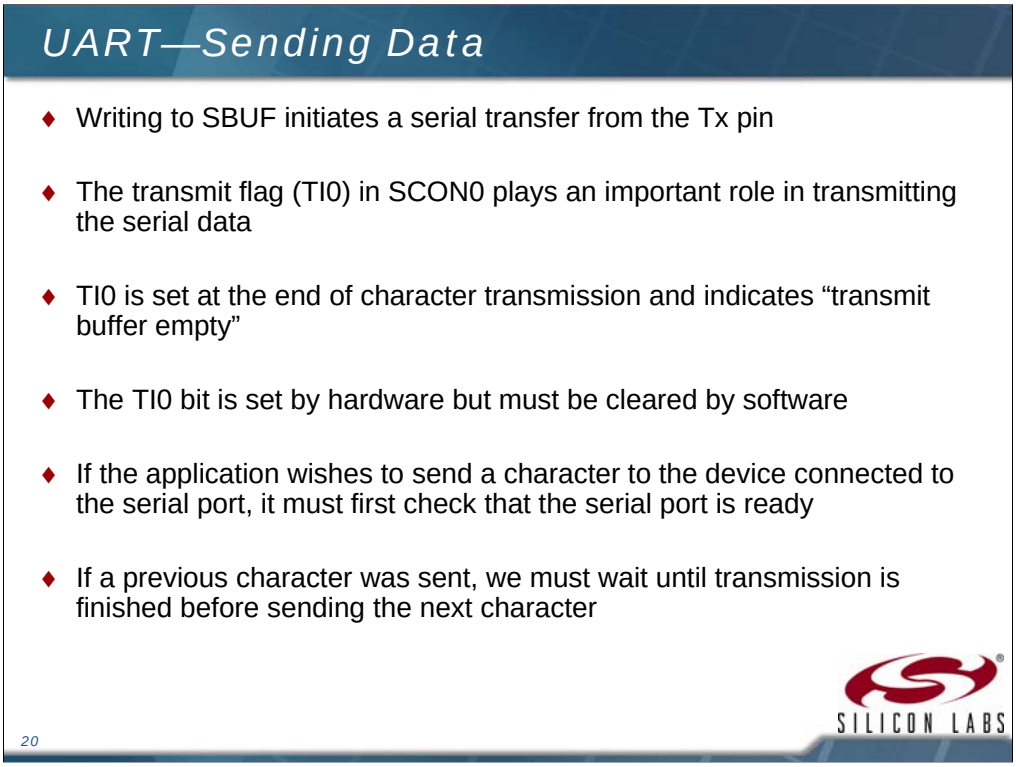
Here is a code segment to initialize the UART0 peripheral. It uses Timer 1 to generate the baud rate at 57600 using the 8 bit auto reload mode. The reload value is stored in TH1 and that value is also used to initialize the timer register itself so that the first count sequence is the correct length. The CKCON register sets the clock source for the timer to be the system clock at 24500000Hz and the SCON register enables the receiver and sets the UART mode to 8 bit.

UART—Receiving Data

- The receive flag (RI0) in SCON0 plays an important role in receiving the serial data
- The RI0 bit is set by hardware but must be cleared by software
- RI0 is set at the end of character reception and indicates “receive buffer full”
- This condition is tested in software (polled) or programmed to cause an interrupt
- If the application wishes to input (i.e. read) a character from the device connected to the serial port (e.g. COM1 port of PC), it must wait until RI0 is set, then clear RI0 and read the character from SBUF0

19

The previous slide showed that the receiver was enabled by the command SCON0 = 0x10 which sets the REN0 bit. Once enabled, received data will set the RI0 bit at the end of transmission (when the peripheral is sampling the stop bit time). If interrupts are enabled the setting of RI0 will generate an interrupt and the interrupt service routine (ISR) will be executed. The ISR code will have to clear the flags and read the data in the receive buffer SBUF0.



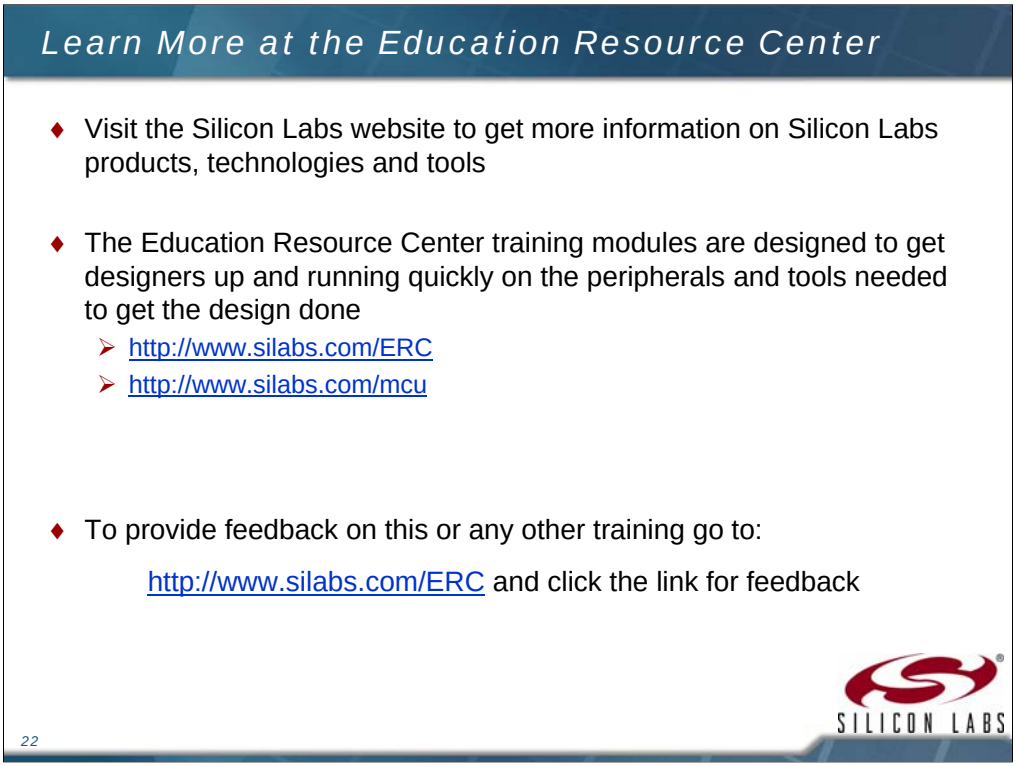
In order to initiate a transmit cycle the processor must first check to see if the peripheral is busy. The TI0 flag can be used to accomplish this as it indicates when the transmit buffer is empty. Once the processor verifies the transmit buffer is empty it writes to SBUF0 to start the transfer. As soon as data is written to the buffer the transmission begins. The TI0 flag is set at the end of character transmission and indicates “transmit buffer empty.” This condition occurs after the 8th bit is shifted out of the peripheral. In some applications where a buffer of data is required to be transferred, it is useful to set up the firmware such that writing a 1 to the TI0 bit would initiate an ISR cycle at which point the ISR can buffer pointers the flags in order to cycle through all of data.

UART Interrupts—Sending/Receiving Data

- RIO is set by hardware to 1 at the end of the STOP sampling time for a received byte
- Must be cleared by software
- TI0 is set by hardware to 1 after the eighth bit has been transmitted
- Must be cleared by software

```
INTERRUPT(UART0_ISR, INTERRUPT_UART0)
{
   //-- Pending flags RI0 (SCON0.0) and TI0(SCON0.1)
   if (RI0 == 1)                        // If interrupt generated from the
                                        // receive flag
   {
      Byte = SBUF0;                     // Read a character from UART
      RI0 = 0;                          // Clear interrupt flag
}
   if (TI0 == 1)                        // Check if transmit flag is set
   {
      TI0 = 0;                          // Clear interrupt flag
   }
}
```


This is the Interrupt Service Routine (ISR) for the UART peripheral. Keep in mind that this is just a very high level routine for illustration purposes. On slide 10 we saw the block diagram of the UART peripheral. An output of the module is the interrupt flag that is generated by the receive interrupt flag (RI0) or the transmit interrupt flag (TI0). In the ISR it is necessary to check which condition caused the CPU to vector to the ISR and in the example shown above we execute only the code based on which flag was set using the conditional statement. Each conditional tests the flag to see if it is set. If the condition is true then the flag is cleared. The receive ISR pulls the data from the buffer so that it can be used by the application. The transmit side clears the flag. The application is generally responsible for placing the next byte to be transmitted into the transmit buffer (SBUF0).



Visit the Silicon Labs Education Resource Center to learn more about the MCU products.


SILICON LABS
www.silabs.com/MCU