Chapter 11: Serial Interfacing
Embedded Systems - Shape The World
Jonathan Valvano and Ramesh Yerraballi

This chapter provides an introduction to serial interfacing, which means we send one bit at time. Serial communication is prevalent in both the computer industry in general and the embedded industry in specific. There are many serial protocols, but in this course we will show you one of the first and simplest protocols that transmit one bit at a time. We will show the theory and details of the universal asynchronous receiver/transmitter (UART) and then use it as an example for developing an I/O driver. We will use busy-wait to synchronize the software with the hardware.

Learning Objectives:
- I/O synchronization.
- Models of I/O devices (busy, done, off).
- Learn how to program the UART.
- Build a distributed system by connecting two systems together.
- Learn how to convert between numbers and ASCII strings

Video 11.0. Introduction to Serial Communication

11.1. I/O Synchronization

Before we begin define serial communication, let's begin by introducing some performance measures. As engineers and scientists we are constantly making choices as we design new product or upgrade existing systems. A **performance measure** is a quantitative metric that the goodness of the system. The metrics and synchronization algorithms presented in this section will apply to all I/O communication.

Latency is the time between when the I/O device indicated service is required and the time when service is initiated. Latency includes hardware delays in the digital hardware plus computer software delays. For an input device, software latency (or software response time) is the time between new input data ready and the software reading the data. For an output device, latency is the delay from output device idle and the software giving the device new data to output. In this book, we will also have periodic events. For example, in our data acquisition systems, we wish to invoke the analog to digital converter (ADC) at a fixed time interval. In this way we can collect a sequence of digital values that approximate the continuous analog signal. Software latency in this case is the time between when the ADC conversion is supposed to be started, and when it is actually started. The microcomputer-based control system also employs periodic software processing. Similar to the data acquisition system, the latency in a control system is the time between when the control software is supposed to be run, and when it is actually run. A **real-time** system is one that can guarantee a worst case latency. In other words, the software response time is small and bounded. Furthermore, this bound is small enough to satisfy overall specification of the system, such as no lost data. **Throughput** or **bandwidth** is the maximum data flow in bytes/second that can be processed by the system. Sometimes the bandwidth is limited by the I/O device, while other times it is limited by computer software. Bandwidth can be reported as an overall average or a short-term maximum. **Priority** determines the order of service when two or more requests are made simultaneously. Priority also determines if a high-priority request should be allowed to suspend a low priority request that is currently being processed. We may also wish to implement equal priority, so that no one device can monopolize the computer. In some computer literature, the term "soft-real-time" is used to describe a system that supports priority.

The purpose of our interface is to allow the microcontroller to interact with its external I/O device. One of the choices the designer must make is the algorithm for how the software synchronizes with the hardware. There are five mechanisms to synchronize the microcontroller with the I/O device. Each mechanism synchronizes the I/O data transfer to the busy to done transition. The methods are discussed in the following paragraphs.

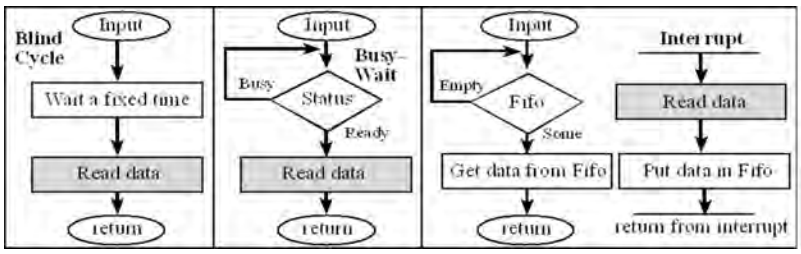


Figure 11.1. Synchronization Mechanisms

Video 11.1. Device Communication requires Synchronization

Blind cycle is a method where the software simply waits a fixed amount of time and assumes the I/O will complete before that fixed delay has elapsed. For an input device, the software triggers (starts) the external input hardware, waits a specified time, then reads data from device. Blind cycle synchronization for an input device is shown on the left part of Figure 11.1. For an output device, shown on the left part of Figure 11.2, the software writes data to the output device, triggers (starts) the device, then waits a specified time. We call this method **blind**, because there is no status information about the I/O device reported to the software. It is appropriate to use this method in situations where the I/O speed is short and predictable. We can ask the LCD to display an ASCII character, wait 37 µs, and then we are sure the operation is complete. This method works because the LCD speed is short and predictable. Another good example of blind-cycle synchronization is spinning a stepper motor. If we repeat this 8-step sequence over and over 1) output a 0x05, 2) wait 1ms, 3) output a 0x06, 4) wait 1ms, 5) output a 0x0A, 6) wait 1ms, 7) output a 0x09, 8) wait 1ms, the motor will spin at a constant speed.

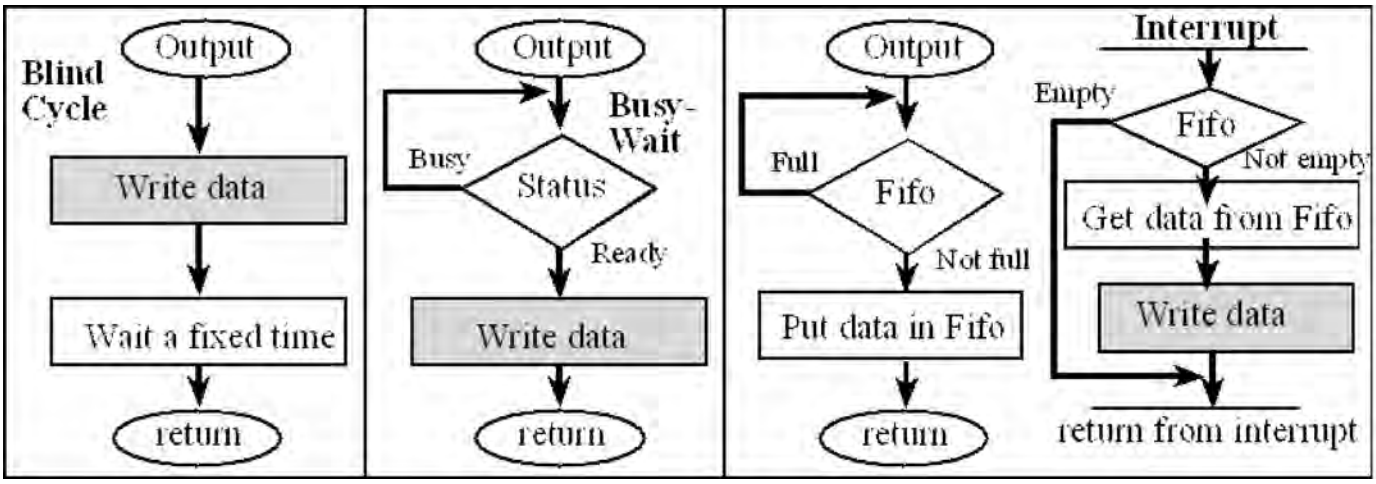


Figure 11.2. The output device sets a flag when it has finished outputting the last data.

Interactive Tool 11.1

Use the following tool to see how blind-cycle synchronization works. You will need to enter a number between 1-10 to simulate the timing behavior of the device.

Enter an amount of time to wait (1-10): Start

Blind Cycle

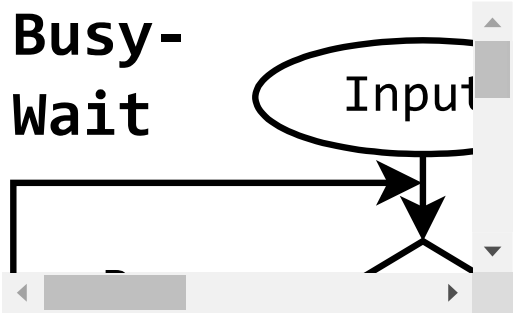
Inpu

Busy Wait is a software loop that checks the I/O status waiting for the done state. For an input device, the software waits until the input device has new data, and then reads it from the input device, see the middle parts of Figures 11.1 and 11.2. For an output device, the software writes data, triggers the output device then waits until the device is finished. Another approach to output device interfacing is for the software to wait until the output device has finished the previous output, write data, and then trigger the device. Busy-wait synchronization will be used in situations where the software system is relatively simple and real-time response is not important. The UART software in this chapter will use busy-wait synchronization.

Interactive Tool 11.2

Use the following tool to see how busy-wait synchronization works. You will press the "Ready" button to simulate the device being ready

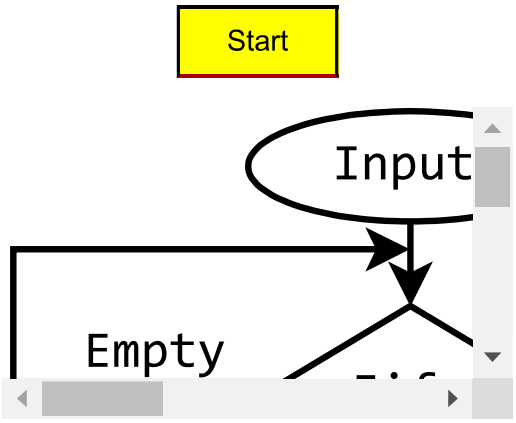
Start Ready Click to simulate I/O device becoming ready.



An **interrupt** uses hardware to cause special software execution. With an input device, the hardware will request an interrupt when input device has new data. The software interrupt service will read from the input device and save in global RAM, see the right parts of Figures 11.1 and 11.2. With an output device, the hardware will request an interrupt when the output device is idle. The software interrupt service will get data from a global structure, and then write to the device. Sometimes we configure the hardware timer to request interrupts on a periodic basis. The software interrupt service will perform a special function. A data acquisition system needs to read the ADC at a regular rate. Interrupt synchronization will be used in situations where the system is fairly complex (e.g., a lot of I/O devices) or when real-time response is important. Interrupts will be presented in Chapter 12.

Interactive Tool 11.3

Use the following tool to see how interrupt-based synchronization works. The foreground thread and background thread (the Interrupt Service Routine or ISR) communicate using a buffer called a first in first out queue (FIFO)



Periodic Polling uses a clock interrupt to periodically check the I/O status. At the time of the interrupt the software will check the I/O status, performing actions as needed. With an input device, a ready flag is set when the input device has new data. At the next periodic interrupt after an input flag is set, the software will read the data and save them in global RAM. With an output device, a ready flag is set when the output device is idle. At the next periodic interrupt after an output flag is set, the software will get data from a global structure, and write it. Periodic polling will be used in situations that require interrupts, but the I/O device does not support interrupt requests directly.

DMA, or direct memory access, is an interfacing approach that transfers data directly to/from memory. With an input device, the hardware will request a DMA transfer when the input device has new data. Without the software's knowledge or permission the DMA controller will read data from the input device and save it in memory. With an output device, the hardware will request a DMA transfer when the output device is idle. The DMA controller will get data from memory, and then write it to the device. Sometimes we configure the hardware timer to request DMA transfers on a periodic basis. DMA can be used to implement a high-speed data acquisition system. DMA synchronization will be used in situations where high bandwidth and low latency are important. DMA will not be covered in this introductory class. For details on how to implement DMA on the LM4F120/TM4C123, see Embedded Systems: Real-Time Operating Systems for ARM Cortex-M Microcontrollers, ISBN: 978-1466468863.

One can think of the hardware being in one of three states. The **idle** state is when the device is disabled or inactive. No I/O occurs in the idle state. When active (not idle) the hardware toggles between the **busy** and **ready** states. The interface includes a **flag** specifying either busy (0) or ready (1) status. Hardware-software synchronization revolves around this flag:

- The hardware will set the flag when the hardware component is complete.
- The software can read the flag to determine if the device is busy or ready.
- The software can clear the flag, signifying the software component is complete.
- This flag serves as the hardware triggering event for an interrupt.

For an input device, a status flag is set when new input data is available. The "busy to ready" state transition will cause a busy-wait loop to complete, see middle of Figure 11.1. Once the software recognizes the input device has new data, it will read the data and ask the input device to create more data. It is the **busy to ready** state transition that signals to the software that the hardware task is complete, and now software service is required. When the hardware is in the ready state the I/O transaction is complete. Often the simple process of reading the data will clear the flag and request another input.

The problem with I/O devices is that they are usually much slower than software execution. Therefore, we need synchronization, which is the process of the hardware and software waiting for each other in a manner such that data is properly transmitted. A way to visualize this synchronization is to draw a

state versus time plot of the activities of the hardware and software. For an input device, the software begins by waiting for new input. When the input device is busy it is in the process of creating new input. When the input device is ready, new data is available. When the input device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software accepts the input, it can release the input device hardware. The arrows in Figure 11.3 represent the synchronizing events. In this example, the time for the software to read and process the data is less than the time for the input device to create new input. This situation is called **I/O bound**, meaning the bandwidth is limited by the speed of the I/O hardware.

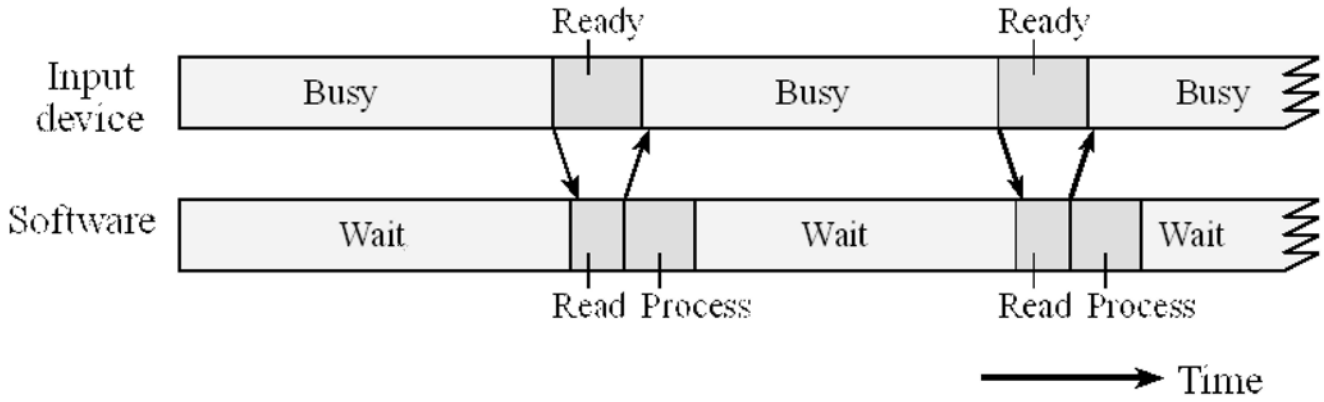


Figure 11.3. The software must wait for the input device to be ready (I/O bound input interface).

If the input device were faster than the software, then the software waiting time would be zero. This situation is called **CPU bound** (meaning the bandwidth is limited by the speed of the executing software). In real systems the bandwidth depends on both the hardware and the software. Another characteristic of real systems is the data can vary over time, like car traffic arriving and leaving a road intersection. In other words, the same I/O channel can sometimes be I/O bound, but at other times the channel could be CPU bound.

We can store or buffer data in a **first in first out** (FIFO) queue, see Figure 11.4, while passing the data from one module to another. These modules may be input devices, output devices or software. Because the buffer separates the generation of data from the consumption of data, it is very efficient, and hence it is prevalent in I/O communication. In particular, it can handle situations where there is an increase or decrease in the rates at which data is produced or consumed. Other names for this important interfacing mechanism include **bounded buffer**, **producer-consumer**, and **buffered I/O**. Data are entered into the FIFO as they arrive; we call **Put** to store data in the FIFO. Data are removed as they leave; we call **Get** to remove data from the FIFO. The FIFO maintains the order of the data, as it passes through the buffer. We can think of a FIFO like a line at the post office. There is space in the lobby for a finite number of people to wait. As customers enter the post office they get in line at the end (put onto FIFO). As the postal worker services the customers, people at the front leave the line (get from the FIFO). It is bad situation (a serious error) if the waiting room becomes full and there is no room for people to wait (full FIFO). However, if there are no customers waiting (empty FIFO) the postal worker sits idle. An empty FIFO may be inefficient, but it is not considered an error.

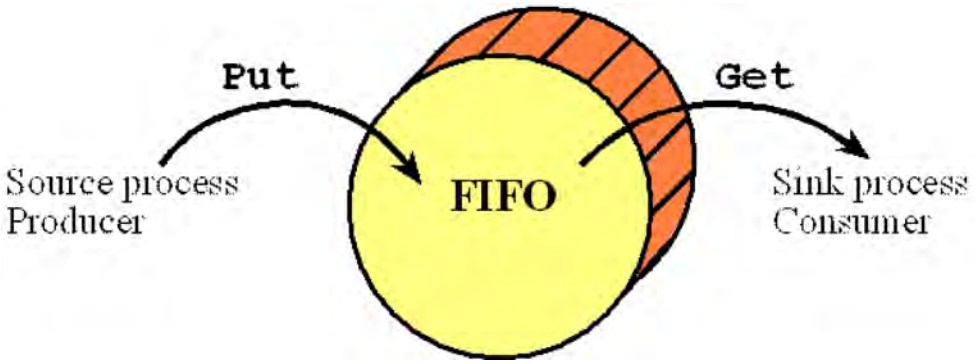


Figure 11.4. A FIFO queue can be used to pass data from a producer to a consumer. At any given time there can be a variable number of elements stored in the FIFO. The order in which data are removed is the same as the order the data are entered.

The busy-wait method is classified as unbuffered because the hardware and software must wait for each other during the transmission of each piece of data. The interrupt solution (shown in the right part of Figure 11.1) is classified as buffered, because the system allows the input device to run continuously, filling a FIFO with data as fast as it can. In the same way, the software can empty the buffer whenever it is ready and whenever there is data in the buffer. The buffering used in an interrupt interface may be a hardware FIFO, a software FIFO, or both hardware and software FIFOs. We will see the FIFO queues will allow the I/O interface to operate during both situations: I/O bound and CPU bound.

For an output device, a status flag is set when the output is idle and ready to accept more data. The “busy to ready” state transition causes a busy-wait loop to complete, see the middle part of Figure 11.2. Once the software recognizes the output is idle, it gives the output device another piece of data to output. It will be important to make sure the software clears the flag each time new output is started. Figure 11.5 contains a state versus time plot of the activities of the output device hardware and software. For an output device, the software begins by generating data then sending it to the output device. When the output device is busy it is processing the data. Normally when the software writes data to an output port, that only starts the output process. The time it takes an output device to process data is usually longer than the software execution time. When the output device is done, it is ready for new data. When the output device makes the transition from busy to ready, it releases the software to go forward. In a similar way, when the software writes data to the output, it releases the output device hardware. The output interface illustrated in Figure 11.5 is also I/O bound because the time for the output device to process data is longer than the time for the software to generate and write it. Again, I/O bound means the bandwidth is limited by the speed of the I/O hardware.

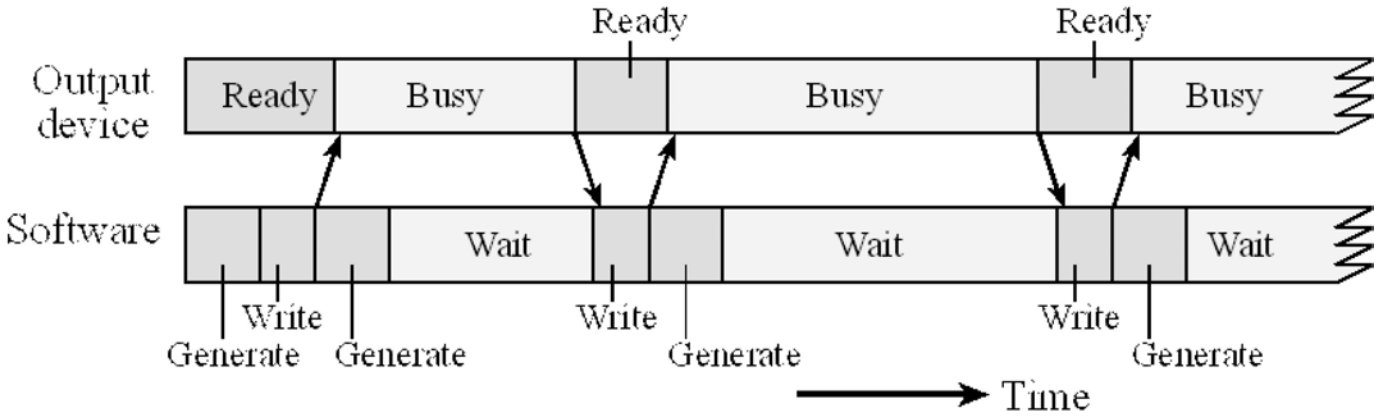


Figure 11.5. The software must wait for the output device to finish the previous operation (I/O bound).

The busy-wait solution for this output interface is also unbuffered, because when the hardware is done, it will wait for the software and after the software generates data, it waits for the hardware. On the other hand, the interrupt solution (shown as the right part of Figure 11.2) is buffered, because the system allows the software to run continuously, filling a FIFO as fast as it wishes. In the same way, the hardware can empty the buffer whenever it is ready and whenever there is data in the FIFO. Again, FIFO queues allow the I/O interface to operate during both situations: I/O bound and CPU bound.

On some systems an interrupt will be generated on a hardware failure. Examples include power failure, temperature too high, memory failure, and mechanical tampering of secure systems. Usually, these events are extremely important and require immediate attention. The Cortex™-M processor will execute special software (`fault`) when it tries to execute an illegal instruction, access an illegal memory location, or attempt an illegal I/O operation.

11.2. Universal Asynchronous Receiver Transmitter (UART)

Video 12.2a. UART Background and Launchpad Support *Video 12.2b. UART Operation*

In this section we will develop a simple device driver using the Universal Asynchronous Receiver/Transmitter (UART). This serial port allows the microcontroller to communicate with devices such as other computers, printers, input sensors, and LCDs. Serial transmission involves sending one bit at a time, such that the data is spread out over time. The total number of bits transmitted per second is called the **baud rate**. The reciprocal of the baud rate is the **bit time**, which is the time to send one bit. Most microcontrollers have at least one UART. The LM4F120/TM4C123 has 8 UARTs. Before discussing the detailed operation on the TM4C, we will begin with general features common to all devices. Each UART will have a baud rate control register, which we use to select the transmission rate. Each device is capable of creating its own serial clock with a transmission frequency approximately equal to the serial clock in the computer with which it is communicating. A **frame** is the smallest complete unit of serial transmission. Figure 11.6 plots the signal versus time on a serial port, showing a single frame, which includes a **start bit** (which is 0), 8 bits of data (least significant bit first), and a **stop bit** (which is 1). There is always only one start bit, but the Stellaris® UARTs allow us to select the 5 to 8 data bits and 1 or 2 stop bits. The UART can add even, odd, or no parity bit. However, we will employ the typical protocol of 1 start bit, 8 data bits, no parity, and 1 stop bit. This protocol is used for both transmitting and receiving. The information rate, or **bandwidth**, is defined as the amount of data or useful information transmitted per second. From Figure 11.6, we see that 10 bits are sent for every byte of usual data. Therefore, the bandwidth of the serial channel (in bytes/second) is the baud rate (in bits/sec) divided by 10.

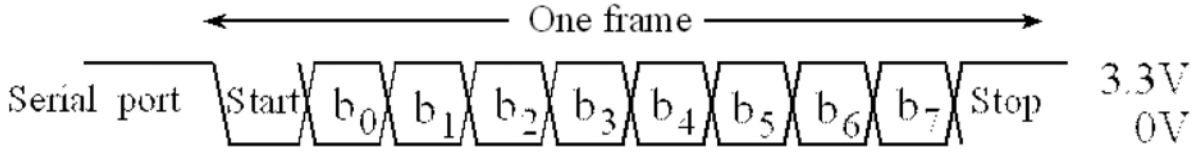


Figure 11.6. A serial data frame with 8-bit data, 1 start bit, 1 stop bit, and no parity bit.

Common Error: If you change the bus clock frequency without changing the baud rate register, the UART will operate at an incorrect baud rate.

Checkpoint 11.1: Assuming the protocol drawn in Figure 11.6 and a baud rate of 1000 bits/sec, what is the bandwidth in bytes/sec?

Table 11.1 shows the three most commonly used RS232 signals. The EIA-574 standard uses RS232 voltage levels and a DB9 connector that has only 9 pins. The most commonly used signals of the full RS232 standard are available with the EIA-574 protocols. Only **TxD**, **RxD**, and **SG** are required to implement a simple bidirectional serial channel, thus the other signals are not shown (Figure 11.7). We define the **data terminal equipment** (DTE) as the computer or a terminal and the **data communication equipment** (DCE) as the modem or printer.

DB9 Pin	EIA-574 Name	Signal	Description	True	DTE	DCE
3	103	TxD	Transmit Data	-5.5V	out	in
2	104	RxD	Receive Data	-5.5V	in	out
5	102	SG	Signal Ground			

Table 11.1. The commonly-used signals on the EIA-574 protocol.

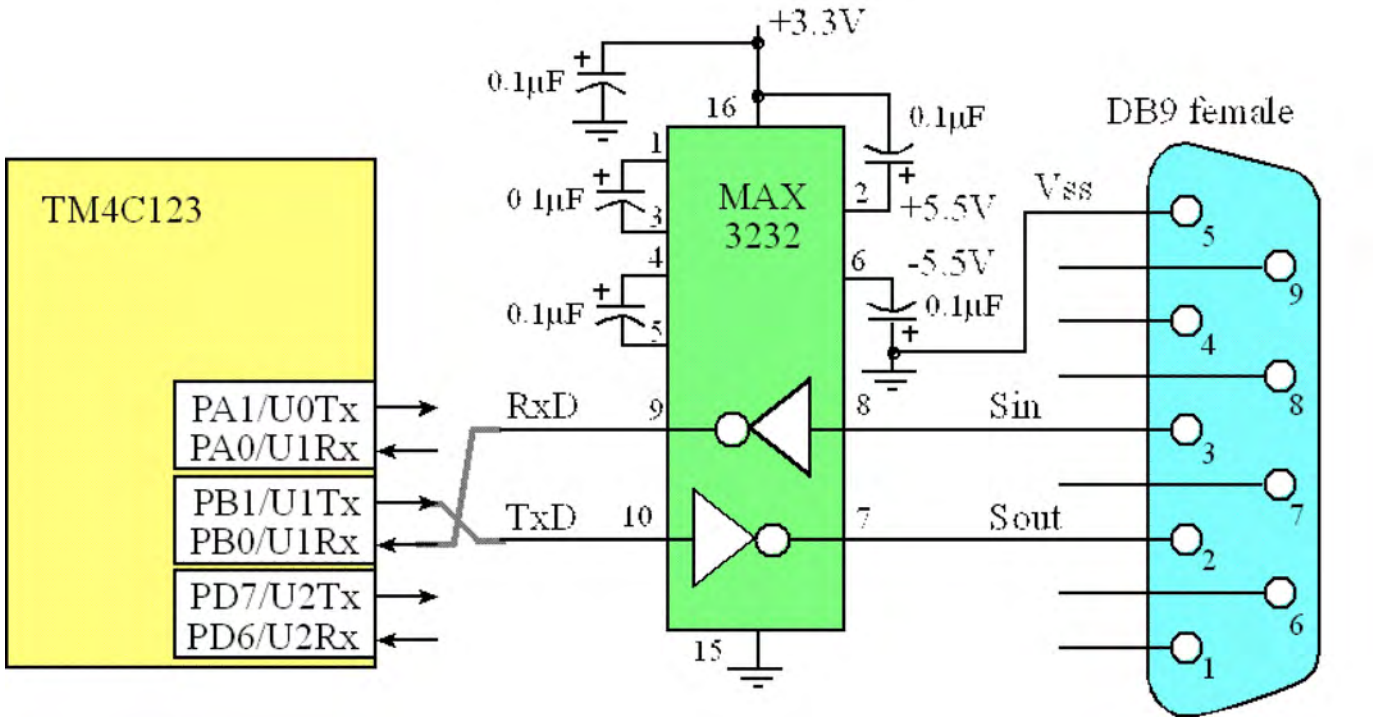


Figure 11.7. Hardware interface implementing an asynchronous RS232 channel. The TM4C123 has eight UART ports.

Observation: The LaunchPad sends UART0 channel through the USB cable, so the circuit shown in Figure 11.7 will not be needed. On the PC side of the cable, the serial channel becomes a virtual COM port.

RS232 is a non-return-to-zero (NRZ) protocol with true signified as a voltage between -5 and -15 V. False is signified by a voltage between +5 and +15 V. A MAX3232 converter chip is used to translate between the +5.5/-5.5 V RS232 levels and the 0/+3.3 V digital levels. The capacitors in this circuit are important, because they form a charge pump used to create the ±5.5 voltages from the +3.3 V supply. The RS232 timing is generated automatically by the UART. During transmission, the MAX3232 translates a digital high on microcontroller side to -5.5V on the RS232/EIA-574 cable, and a digital low is translated to +5.5V. During receiving, the MAX3232 translates negative voltages on RS232/EIA-574 cable to a digital high on the microcontroller side, and a positive voltage is translated to a digital low. The computer is classified as DTE, so its serial output is pin 3 in the EIA-574 cable, and its serial input is pin 2 in the EIA-574 cable. When connecting a DTE to another DTE, we use a cable with pins 2 and 3 crossed. I.e., pin 2 on one DTE is connected to pin 3 on the other DTE and pin 3 on one DTE is connected to pin 2 on the other DTE. When connecting a DTE to a DCE, then the cable passes the signals straight across. In all situations, the grounds are connected together using the SG wire in the cable. This channel is classified as **full-duplex**, because transmission can occur in both directions simultaneously.

11.2.1. Asynchronous Communication

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.

Interactive Tool 11.4

Use the following tool to watch the steps involved in Serial Communication of a simple two-byte message. Click Start/next over and over to single step the process, and click Run to run the entire sequence.

Start Click Start to Send 'H' to the direction 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.

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 17^{th} 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.

11.2.2. TM4C UART Details

Next we will overview the specific UART functions on the TM4C microcontroller. This section is intended to supplement rather than replace the Texas Instruments manuals. When designing systems with any I/O module, you must also refer to the reference manual of your specific microcontroller. It is also good design practice to review the errata for your microcontroller to see if any quirks (mistakes) exist in your microcontroller that might apply to the system you are designing.

Stellaris TM4C microcontrollers have eight UARTs. The specific port pins used to implement the UARTs vary from one chip to the next. To find which pins your microcontroller uses, you will need to consult its datasheet. Table 11.2 shows some of the registers for the UART0 and UART1. For the other UARTs, the register names will replace the 0 with a 1 – 7. For the exact register addresses, you should include the appropriate header file (e.g., **tm4c123gh6pm.h**). To activate a UART you will need to turn on the UART clock in the **RCGCUART** register. You should also turn on the clock for the digital port in the **RCGCGPIO** register. You need to enable the transmit and receive pins as digital signals. The alternative function for these pins must also be selected. In particular we set bits in both the AFSEL and PCTL registers.

The OE, BE, PE, and FE are error flags associated with the receiver. You can see these flags in two places: associated with each data byte in **UART0_DR_R** or as a separate error register in **UART0_RSR_R**. The overrun error (**OE**) is set if data has been lost because the input driver latency is too long. **BE** is a break error, meaning the other device has sent a break. **PE** is a parity error (however, we will not be using parity). The framing error (**FE**) will get set if the baud rates do not match. The software can clear these four error flags by writing any value to **UART0_RSR_R**.

The status of the two FIFOs can be seen in the **UART0_FR_R** register. The **BUSY** flag is set while the transmitter still has unsent bits, even if the transmitter is disabled. It will become zero when the transmit FIFO is empty and the last stop bit has been sent. If you implement busy-wait output by first outputting then waiting for **BUSY** to become 0 (right flowchart of Figure 11.10), then the routine will write new data and return after that particular data has been completely transmitted.

The **UART0_CTL_R** control register contains the bits that turn on the UART. **TXE** is the Transmitter Enable bit, and **RXE** is the Receiver Enable bit. We set **TXE**, **RXE**, and **UARTEN** equal to 1 in order to activate the UART device. However, we should clear **UARTEN** during the initialization sequence.

	31–12	11	10	9	8	7–0	Name
$4000.C000		OE	BE	PE	FE	DATA	UART0_DR_R

	31–3	3	2	1	0	Name
$4000.C004		OE	BE	PE	FE	UART0_RSR_R

	31–8	7	6	5	4	3	2–0	Name
$4000.C018		TXFE	RXFF	TXFF	RXFE	BUSY		UART0_FR_R

	31–16	15–0	Name
$4000.C024		DIVINT	UART0_IBRD_R

	31–6	5–0	Name
$4000.C028		DIVFRAC	UART0_FBRD_R

	7	6 – 5	4	3	2	1	0	Name
$4000.C02C	SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART0_LCRH_R

	31–10	9	8	7	6–3	2	1	0	Name
$4000.C030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART0_CTL_R

	31–12	11	10	9	8	7–0	Name
$4000.D000		OE	BE	PE	FE	DATA	UART1_DR_R

	31–3	3	2	1	0	Name
$4000.D004		OE	BE	PE	FE	UART1_RSR_R

	31–8	7	6	5	4	3	2–0	Name
$4000.D018		TXFE	RXFF	TXFF	RXFE	BUSY		UART1_FR_R

	31–16	15–0	Name
$4000.D024		DIVINT	UART1_IBRD_R

	31–6	5–0	Name
$4000.D028		DIVFRAC	UART1_FBRD_R

	31–8	7	6 – 5	4	3	2	1	0	Name
$4000.D02C		SPS	WPEN	FEN	STP2	EPS	PEN	BRK	UART1_LCRH_R

	31–10	9	8	7	6–3	2	1	0	Name
$4000.D030		RXE	TXE	LBE		SIRLP	SIREN	UARTEN	UART1_CTL_R

Table 11.2. Some UART registers. Each register is 32 bits wide. Shaded bits are zero.

The **IBRD** and **FBRD** registers specify the baud rate. The baud rate **divider** is a 22-bit binary fixed-point value with a resolution of 2^{-6}. The **Baud16** clock is created from the system bus clock, with a frequency of (Bus clock frequency)/**divider**. The baud rate is 16 times slower than **Baud16**

Baud rate = **Baud16**/**16** = (Bus clock frequency)/(16***divider**)

For example, if the bus clock is 80 MHz and the desired baud rate is 19200 bits/sec, then the **divider** should be 80,000,000/16/19200 or 260.4167. Let *m* be the integer part, without rounding. We store the integer part (*m*=260) in **IBRD**. For the fraction, we find an integer *n*, such that *n*/64 is about 0.4167. More simply, we multiply 0.4167*64 = 26.6688 and round to the closest integer, 27. We store this fraction part (*n*=27) in **FBRD**. We did approximate the divider, so it is interesting to determine the actual baud rate. Assume the bus clock is 80 MHz.

Baud rate = (80 MHz)/(16* (*m*+*n*/64)) = (80 MHz)/(16* (260+27/64)) = 19199.616 bits/sec

The baud rates in the transmitter and receiver must match within 5% for the channel to operate properly. The error for this example is 0.002%.

The three registers **LCRH**, **IBRD**, and **FBRD** form an internal 30-bit register. This internal register is only updated when a write operation to **LCRH** is performed, so any changes to the baud-rate divisor must be followed by a write to the **LCRH** register for the changes to take effect. Out of reset, both FIFOs are disabled and act as 1-byte-deep holding registers. The FIFOs are enabled by setting the **FEN** bit in **LCRH**.

Checkpoint 11.2: Assume the bus clock is 10 MHz. What is the baud rate if UART0_IBRD_R equals 2 and UART0_FBRD_R equals 32?

Checkpoint 11.3: Assume the bus clock is 50 MHz. What values should you put in UART0_IBRD_R and UART0_FBRD_R to make a baud rate of 38400 bits/sec?

11.2.3. UART1 Device Driver on PC5 and PC4

Software that sends and receives data must implement a mechanism to synchronize the software with the hardware. In particular, the software should read data from the input device only when data is indeed ready. Similarly, software should write data to an output device only when the device is ready to accept new data. With busy-wait synchronization, the software continuously checks the hardware status waiting for it to be ready. In this section, we will use busy-wait synchronization to write I/O programs that send and receive data using the UART. After a frame is received, the receive FIFO will be not empty (**RXFE** becomes 0) and the 8-bit data is available to be read. To get new data from the serial port, the software first waits for **RXFE** to be zero, then reads the result from **UART1_DR_R**. Recall that when the software reads **UART1_DR_R** it gets data from the receive FIFO. This operation is illustrated in Figure 11.8 and shown in Program 11.1. In a similar fashion, when the software wishes to output via the serial port, it first waits for **TXFF** to be clear, then performs the output. When the software writes **UART1_DR_R** it puts data into the transmit FIFO.

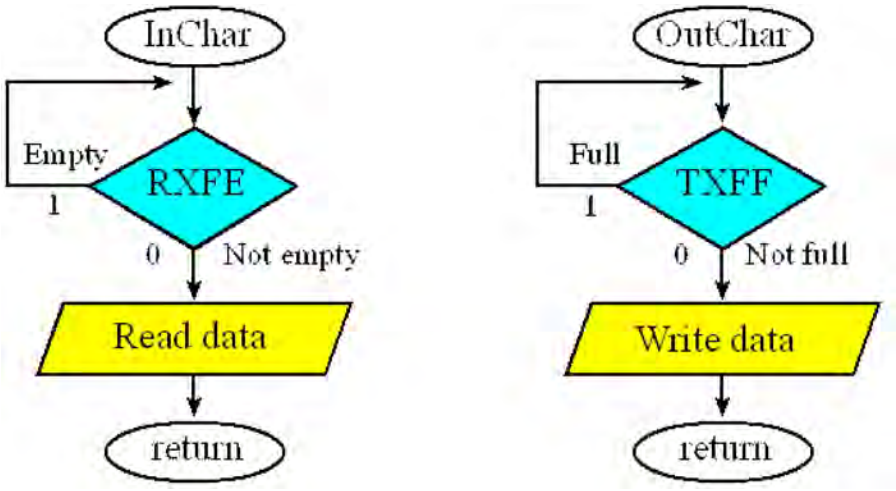


Figure 11.8. Flowcharts of InChar and OutChar using busy-wait synchronization.

The initialization program, **`UART_Init`**, enables the UART1 device and selects the baud rate. The **PCTL** bits were defined back in Chapter 6, and repeated as Table 11.3. **PCTL** bits 5-4 are set to 0x22 to select U1Tx and U1Rx on PC5 and PC4. The input routine waits in a loop until **RXFE** is 0 (FIFO not empty), then reads the data register. The output routine first waits in a loop until **TXFF** is 0 (FIFO not full), then writes data to the data register. Polling before writing data is an efficient way to perform output. **UART2_xxx.zip** is the interrupt-driven version. Be careful when using Port C to be friendly; the pins PC3-PC0 are used by the debugger and you should not modify their configurations.

IO	Ain	0	1	2	3	4	5	6	7	8	9	14
PA0		*Port*	*U0Rx*							CAN1Rx		
PA1		*Port*	*U0Tx*							CAN1Tx		
PA2		Port		SSI0Clk								
PA3		Port		SSI0Fss								
PA4		Port		SSI0Rx								
PA5		Port		SSI0Tx								
PA6		Port			I_2C1SCL		M1PWM2					
PA7		Port			I_2C1SDA		M1PWM3					
PB0		Port	U1Rx						T2CCP0			
PB1		Port	U1Tx						T2CCP1			
PB2		Port			I_2C0SCL				T3CCP0			
PB3		Port			I_2C0SDA				T3CCP1			
PB4	Ain10	Port		SSI2Clk		M0PWM2			T1CCP0	CAN0Rx		
PB5	Ain11	Port		SSI2Fss		M0PWM3			T1CCP1	CAN0Tx		
PB6		Port		SSI2Rx		M0PWM0			T0CCP0			
PB7		Port		SSI2Tx		M0PWM1			T0CCP1			
PC4	C1-	Port	U4Rx	U1Rx		M0PWM6		IDX1	WT0CCP0	U1RTS		
PC5	C1+	Port	U4Tx	U1Tx		M0PWM7		PhA1	WT0CCP1	U1CTS		
PC6	C0+	Port	U3Rx					PhB1	WT1CCP0	USB0epen		
PC7	C0-	Port	U3Tx						WT1CCP1	USB0pflt		
PD0	Ain7	Port	SSI3Clk	SSI1Clk	I_2C3SCL	M0PWM6	M1PWM0		WT2CCP0			
PD1	Ain6	Port	SSI3Fss	SSI1Fss	I_2C3SDA	M0PWM7	M1PWM1		WT2CCP1			
PD2	Ain5	Port	SSI3Rx	SSI1Rx		M0Fault0			WT3CCP0	USB0epen		
PD3	Ain4	Port	SSI3Tx	SSI1Tx				IDX0	WT3CCP1	USB0pflt		
PD4	*USB0DM*	*Port*	*U6Rx*						*WT4CCP0*			
PD5	*USB0DP*	*Port*	*U6Tx*						*WT4CCP1*			
PD6		Port	U2Rx			M0Fault0		PhA0	WT5CCP0			
PD7		Port	U2Tx					PhB0	WT5CCP1	NMI		
PE0	Ain3	Port	U7Rx									
PE1	Ain2	Port	U7Tx									
PE2	Ain1	Port										
PE3	Ain0	Port										
PE4	Ain9	Port	U5Rx		I_2C2SCL	M0PWM4	M1PWM2			CAN0Rx		
PE5	Ain8	Port	U5Tx		I_2C2SDA	M0PWM5	M1PWM3			CAN0Tx		
PF0		Port	U1RTS	SSI1Rx	CAN0Rx		M1PWM4	PhA0	T0CCP0	NMI	C0o	
PF1		Port	U1CTS	SSI1Tx			M1PWM5	PhB0	T0CCP1		C1o	TRD1
PF2		Port		SSI1Clk		M0Fault0	M1PWM6		T1CCP0			TRD0
PF3		Port		SSI1Fss	CAN0Tx		M1PWM7		T1CCP1			TRCLK
PF4		Port					M1Fault0	IDX0	T2CCP0	USB0epen		

Table 11.3. PMCx bits in the GPIOPCTL register on the LM4F/TM4C specify alternate functions. *PD4* and *PD5* are hardwired to the USB device. *PA0* and *PA1* are hardwired to the serial port. PWM does not exist on LM4F120.

```
// Assumes a 80 MHz bus clock, creates 115200 baud rate
void UART_Init(void){           // should be called only once
  SYSCTL_RCGCUART_R |= 0x00000002;  // activate UART1
  SYSCTL_RCGCGPIO_R |= 0x00000004;  // activate port C
  UART1_CTL_R &= ~0x00000001;    // disable UART
   UART1_IBRD_R = 43;      // IBRD = int(80,000,000/(16*115,200)) =
int(43.40278)
  UART1_FBRD_R = 26;     // FBRD = round(0.40278 * 64) = 26
```

```
   UART1_LCRH_R = 0x00000070;  // 8 bit, no parity bits, one stop,
FIFOs
  UART1_CTL_R |= 0x00000001;     // enable UART
  GPIO_PORTC_AFSEL_R |= 0x30;    // enable alt funct on PC5-4
  GPIO_PORTC_DEN_R |= 0x30;      // configure PC5-4 as UART1
  GPIO_PORTC_PCTL_R = (GPIO_PORTC_PCTL_R&0xFF00FFFF)+0x00220000;
  GPIO_PORTC_AMSEL_R &= ~0x30;   // disable analog on PC5-4
}
// Wait for new input, then return ASCII code
char UART_InChar(void){
  while((UART1_FR_R&0x0010) != 0);      // wait until RXFE is 0
  return((char)(UART1_DR_R&0xFF));
}
// Wait for buffer to be not full, then output
void UART_OutChar(char data){
  while((UART1_FR_R&0x0020) != 0);      // wait until TXFF is 0
  UART1_DR_R = data;
}
// Immediately return input or 0 if no input
char UART_InCharNonBlocking(void){
  if((UART1_FR_R&UART_FR_RXFE) == 0){
    return((char)(UART1_DR_R&0xFF));
  } else{
    return 0;
  }
}
```

Video 11.3. UART Device Driver walk through

Program 11.1. Device driver functions that implement serial I/O (CC11_UART and C11_Network).

Checkpoint 11.4: How does the software clear RXFE?

Checkpoint 11.5: How does the software clear TXFF?

Checkpoint 11.6: Describe what happens if the receiving computer is operating on a baud rate that is twice as fast as the transmitting computer?

Checkpoint 11.7: Describe what happens if the transmitting computer is operating on a baud rate that is twice as fast as the receiving computer?

Checkpoint 11.8: How do you change Program 11.1 to run at the same baud rate, but the system clock is now 10 MHz.

11.3. Conversions

Video 11.4. Device Drivers, Successive Refinement, Number Conversions

In this section we will develop methods to convert between ASCII strings and binary numbers. Let's begin with a simple example. Let **Data** be a fixed length string of three ASCII characters. Each entry of **Data** is an ASCII character 0 to 9. Let **Data[0]** be the ASCII code for the hundred's digit, **Data[1]** be the ten's digit and **Data[2]** be the one's digit. Let **n** be an unsigned 32-bit integer. We will also need an index, **i**. The decimal digits 0 to 9 are encoded in ASCII as 0x30 to 0x39. So, to convert a single ASCII digit to a decimal number, we simply subtract 0x30. To convert this string of 3 decimal digits into binary we can simply calculate

```
n = 100*(Data[0]-0x30) + 10*(Data[1]-0x30) + (Data[2]-0x30);
```

Adding parentheses, we can convert this 3-digit ASCII string as

```
n = (Data[2]-0x30) + 10*((Data[1]-0x30) + 10*(Data[0]-0x30));
```

This second method could be used for converting any string of known and fixed length. If **Data** were a string of 9 decimal digits we could put the above function into a loop

```
n = 0;
for (i=0; i<9 ;i++){
```

```
    n = 10*n + (Data[i]-0x30);
  }
```

If the length were variable, we can replace the for-loop with a while-loop. If **Data** were a variable length string of ASCII characters terminated with a null character (0), we could convert it to binary using a while loop, as shown in Program 11.2. A pointer to the string is passed using call by reference. In the assembly version, the pointer R0 is incremented as the string is parsed. R1 contains the local variable **n**, R2 contains the data from the string, and R3 contains the constant 10.

```
// Convert ASCII string to
//    unsigned 32-bit decimal
// string is null-terminated
uint32_t Str2UDec(char string[]){
  uint32_t i = 0;  // index
  uint32_t n = 0;  // number
  while(string[i] != 0){
    n = 10*n +(string[i]-0x30);
    i++;
  }
  return n;
}
```

Program 11.2. Unsigned ASCII string to decimal conversion.

The example, shown in Program 11.3, uses an I/O device capable of sending and receiving ASCII characters. When using a development board, we can send serial data to/from the PC using the UART. The function **UART_InChar()** returns an ASCII character from the I/O device. The function **UART_OutChar()** sends an ASCII character to the I/O device. The function **UART_InUDec()** will accept number characters (0x30 to 0x39) from the device until any non-number is typed. All input characters are echoed.

```
#define CR 0x0D
// Accept ASCII input in unsigned decimal format, up to 4294967295
// If n> 4294967295, it will truncate without reporting the error
uint32_t UART_InUDec(void){
uint32_t long n=0;            // this will be the return value
char character;      // this is the input ASCII typed
  while(1){
   character = UART_InChar(); // accepts input
   UART_OutChar(character);   // echo this character
   if((character < '0') || (character > '9')){  // check for non-number
     return n;                // quit if not a number
   }
   n = 10*n+(character-0x30); // overflows if above 4294967295
  }
}
```

Program 11.3. Input an unsigned decimal number.

If the ASCII characters were to contain optional "+" and "-" signs, we could look for the presence of the sign character in the first position. If there is a minus sign, then set a flag. Next use our unsigned conversion routine to process the rest of the ASCII characters and generate the unsigned number, **n**. If the flag was previously set, we can negate the value **n**. Be careful to guarantee the + and – are only processed as the first character.

To convert an unsigned integer into a fixed length string of ASCII characters, we could use the integer divide. Assume **n** is an unsigned integer less than or equal to 999. In Program 11.4, the number 0 is converted to the string "000". The first program stores the conversion in an array and the second function outputs the conversion to the UART.

```
char Data[4]; // 4-byte empty buffer
// n is the input 0 to 999
void UDec2Str(uint16_t n){
  Data[0] = n/100 + 0x30; // hundreds digit
  n = n%100;              // n is now between 0 and 99
  Data[1] = n/10 + 0x30;  // tens digit
  n = n%10;               // n is now between 0 and 9
  Data[2] = n + 0x30;     // ones digit
  Data[3] = 0;            // null termination
}
// n is the input 0 to 999
void UART_OutUDec3(uint16_t n){
  UART_OutChar(0x30+n/100); // hundreds digit
  n = n%100;                // 0 to 99
  UART_OutChar(0x30+n/10);  // tens digit
  n = n%10;                 // 0 to 9
  UART_OutChar(0x30+n);     // ones digit
}
```

Program 11.4. Unsigned decimal to ASCII string conversion.

Sometimes we represent noninteger values using integers. For example the system could store the value 1.23 as the integer 123. In this example, the voltage ranges from 0.00 to 9.99V, but the values in the computer are stored as integers 0 to 999. If the system wishes to display those values in volts, we simply add a decimal point to the output while converting, as shown in Program 11.5. For example calling **OutVolt(123)** will output the string "1.23V". Instead of creating an output string, this function outputs each character to display device by calling **UART_OutChar**.

```
//----------------------OutVolt----------------------
// Output a voltage to the UART
```

```
// Input: n is an integer from 0 to 999 meaning 0.00 to 9.99V
// Output: none
// Fixed format: for example n=8 displayed as "0.08V"
void OutVolt(uint32_t n){  // each integer means 0.01V
  UART_OutChar(0x30+n/100);      // digit to the left of the decimal
  n = n%100;                     // 0 to 99
  UART_OutChar('.');             // decimal point
  UART_OutChar(0x30+n/10);       // tenths digit
  n = n%10;                      // 0 to 9
  UART_OutChar(0x30+n);          // hundredths digit
  UART_OutChar('V');}            // units
```

Program 11.5. Print the voltage value to an output device (0≤n≤999).

To convert an unsigned integer into a variable length string of ASCII characters, we convert the digits in reverse order, and then switch them.

```
//----------------------UART_OutUDec-----------------------
// Output a 32-bit number in unsigned decimal format
// Input: 32-bit number to be transferred
// Output: none
// Variable format 1-10 digits with no space before or after
void UART_OutUDec(uint32_t n){
  if(n >= 10){
    UART_OutUDec(n/10);
    n = n%10;
  }
  UART_OutChar(n+'0'); /* n is between 0 and 9 */
}
```

Program 11.6. Print unsigned 32-bit decimal number to an output device.

Example 11.1. You are given a subroutine, **`UART_OutChar`**, which outputs one ASCII character. Design a function that outputs a 32-bit unsigned integer.

Solution: We will solve this iteratively. As always, we ask "what is our starting point?", "how do we make progress?", and "when are we done?" The input, n, is a 32-bit unsigned number, and we are done when 1 to 10 ASCII characters are displayed, representing the value of n. Figure 11.9 demonstrates the successive refinement approach to solving this problem iteratively. The iterative solution has three phases: initialization, creation of digits, and output of the ASCII characters. The digits are created from the remainders occurring by dividing the input, n by 10. To get all the digits we divide by 10 until the quotient is 0. Because the digits are created in the opposite order, each digit will be saved in a buffer during the creation phase and retrieved from the buffer during the output stage. The counter is needed so the output stage knows how many digits are in the buffer.

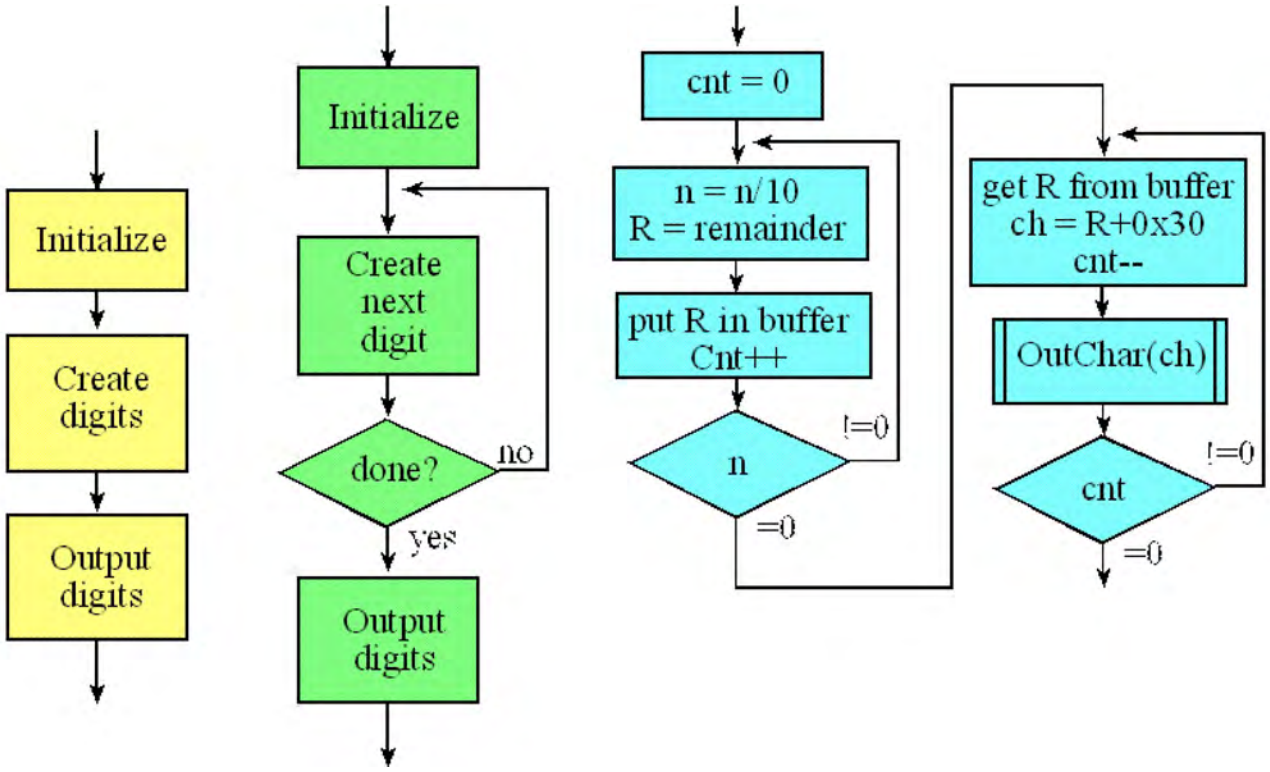


Figure 11.9. Successive refinement method for the iterative solution.

The iteration solution requires two loops; the first loop determines the digits in opposite order, and the second loop outputs the digits in proper order.

```
// iterative method
void OutUDec(uint32_t n){
```

```
uint32_t cnt=0;
char buffer[11];
  do{
    buffer[cnt] = n%10;// digit
    n = n/10;
    cnt++;
  }
  while(n);// repeat until n==0
  for(; cnt; cnt--){
    OutChar(buffer[cnt-1]+'0');
  }
}
```

Program 11.7. Iterative implementation of output decimal.

11.4. Distributed Systems

Animation (html5): two microcontrollers connected via serial port, communicating in full duplex, both main programs use busy wait, data collected on A is passed to B, data collected on B is passed to A

A **network** is a collection of interfaces that share a physical medium and a data protocol. A network allows software tasks in one computer to communicate and synchronize with software tasks running on another computer. For an embedded system, the network provides a means for distributed computing. The **topology** of a network defines how the components are interconnected. Examples topologies include rings, buses and multi-hop. Figure 11.10 shows a **ring** network of three microcontrollers. The advantage of this ring network is low cost and can be implemented on any microcontroller with a serial port. Notice that the microcontrollers need not be the same type or speed.

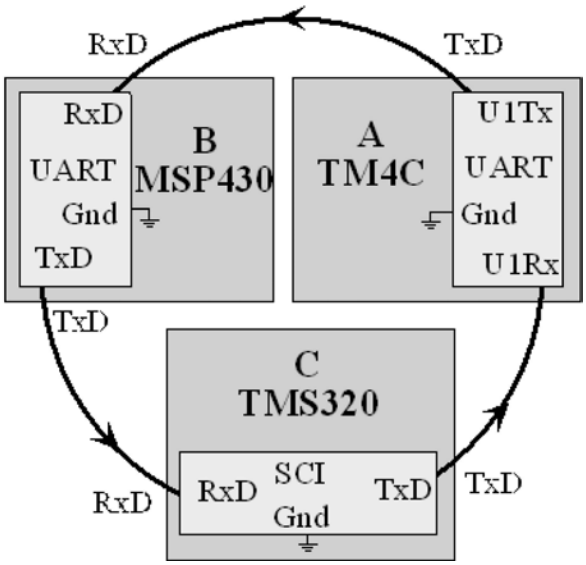


Figure 11.10. A simple ring network with three nodes, linked using the serial ports.

In this chapter we presented the hardware and software interfaces for the UART channel. We connected the TM4C to an I/O device and used the UART to communicate with the human. In this section, we will build on those ideas and introduce the concepts of networks by investigating a couple of simple networks. In particular, we will use the UART channel to connect multiple microcontrollers together, creating a network. A communication network includes both the physical channel (hardware) and the logical procedures (software) that allow users or software processes to communicate with each other. The network provides the transfer of information as well as the mechanisms for process synchronization.

When faced with a complex problem, one could develop a solution on one powerful and **centralized** computer system. Alternatively a **distributed** solution could be employed using multiple computers connected by a network. The processing elements in Figure 11.11 may be a powerful computer, a microcontroller, an application-specific integrated circuit (ASIC), or a smart sensor/actuator.

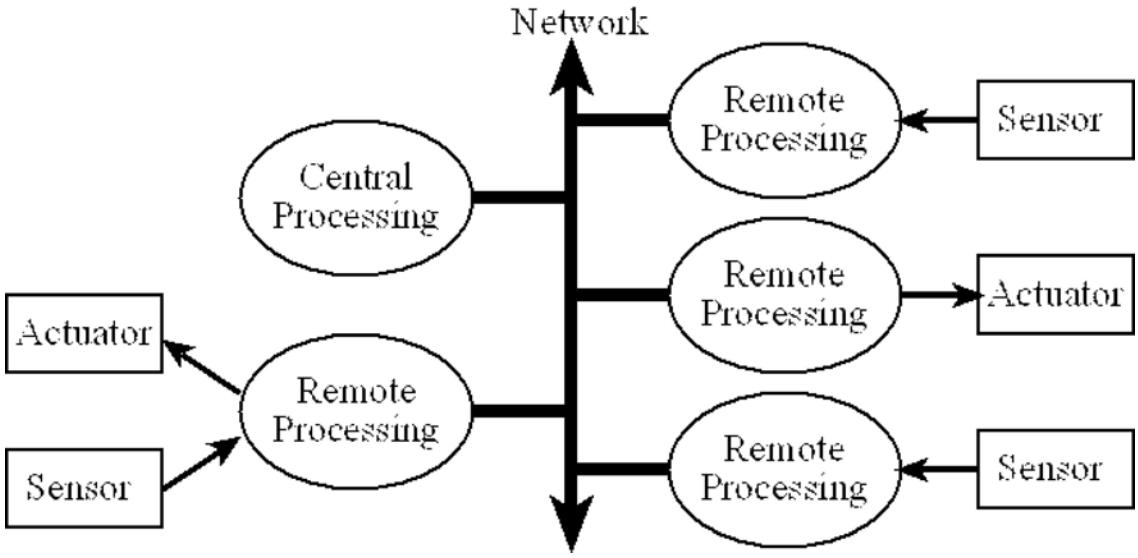


Figure 11.11. Distributed processing places input, output and processing at multiple locations connected together with a network.

There are many reasons to consider a distributed solution (network) over a centralized solution. Often multiple simple microcontrollers can provide a higher performance at lower cost compared to one computer powerful enough to run the entire system. Some embedded applications require input/output activities that are physically distributed. For real-time operation there may not be enough time to allow communication a remote sensor and a central computer. Another advantage of distributed system is improved debugging. For example, we could use one node in a network to monitor and debug the others. Often, we do not know the level of complexity of our problem at design time. Similarly, over time the complexity may increase or decrease. A distributed system can often be deployed that can be scaled. For example, as the complexity increases more nodes can be added, and if the complexity were to decrease nodes could be removed.

Example 11.2. Develop a communication network between two LaunchPads. The switches are inputs, LED is output, and the UART is used to communicate. There will be five questions and three responses. The information is encoded as colors on the LED. The five questions are

Red: Are you there in your office?
Yellow: Are you happy?
Green: Are you hungry, want to have lunch?
Blue: Are you thirsty, want to meet for a beverage?
LightBlue: Shall I come to your office to talk?

The three answers are

White: Yes
Purple: Maybe
Dark: No

Video 11.5a. Chat Tool Protocol

Solution: The operator selects the message to send by pushing the SW1/PF4 switch. While selecting the message the LED displays the message to be sent. Each time the operator pushes the SW1/PF4 switch, the system will cycle through the 8 possible colors on the LED. When the operator pushes the SW2/PF0 the message is sent. The message content is encoded as an ASCII character '0' to '7' (0x30 to 0x37) and send via the UART. The driver from Program 11.8 is used. When a UART frame is received, the data is encoded and the'0' to '7' data is displayed as the corresponding color on the LED. Figure 11.12 shows the hardware, which involves a 3-wire cable connecting the two LaunchPads.

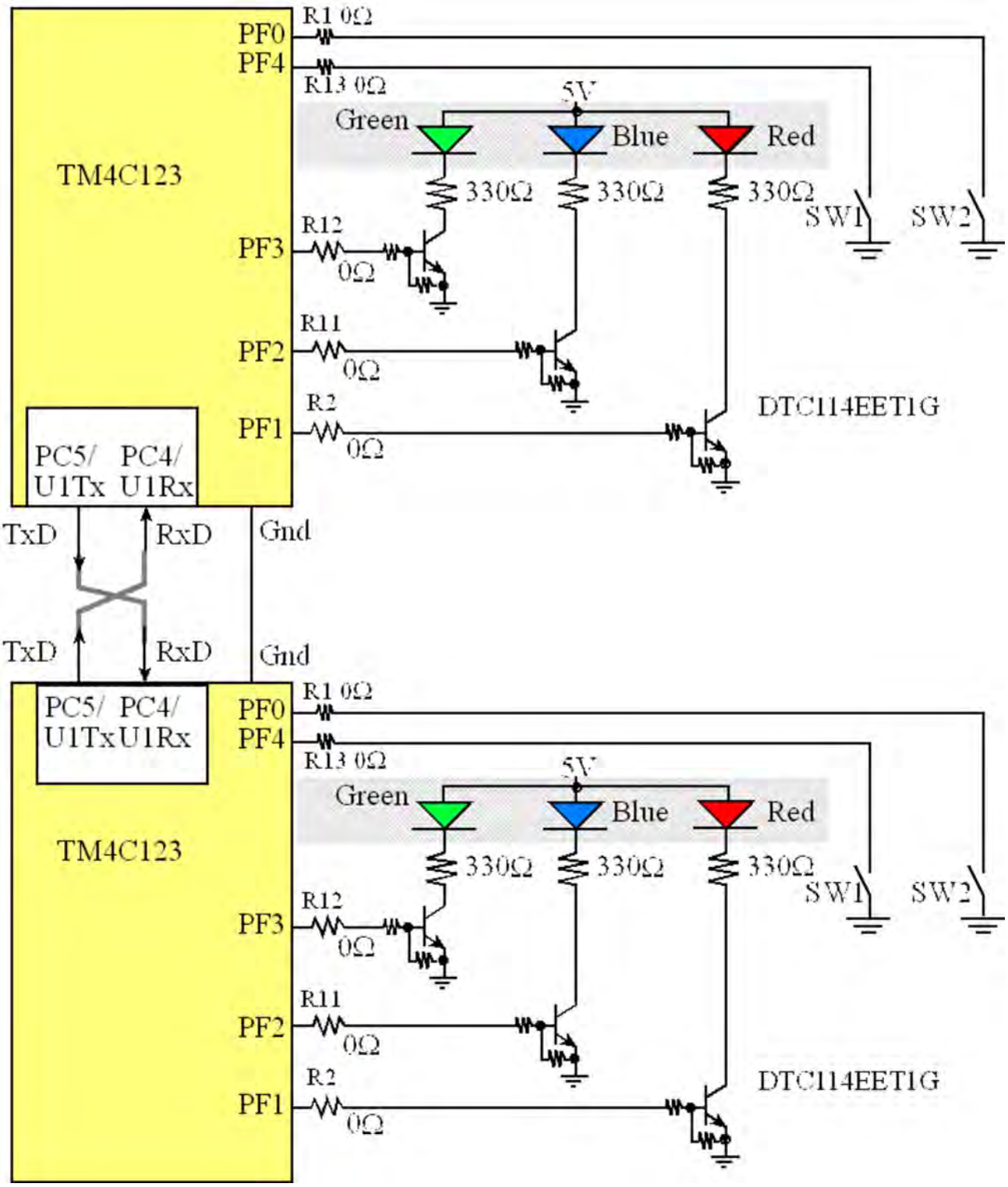


Figure 11.12. Distributed using two LaunchPads connected together by the UARTs.

```
//  red, yellow, green, light blue, blue, purple,  white,  dark
const uint32_t ColorWheel[8] =
{0x02,0x0A,0x08,0x0C,0x04,0x06,0x0E,0x00};
int main(void){ uint32_t SW1,SW2;
  uint32_t prevSW1 = 0;    // previous value of SW1
  uint32_t prevSW2 = 0;    // previous value of SW2
  uint32_t inColor;   // color value from other microcontroller
  uint32_t color = 0; // this microcontroller's color value
  PLL_Init();              // set system clock to 80 MHz
  SysTick_Init();          // initialize SysTick
  UART_Init();             // initialize UART
  PortF_Init();            // initialize buttons and LEDs on Port
F
  while(1){
    SW1 = GPIO_PORTF_DATA_R&0x10; // Read SW1
```

```
    if((SW1 == 0) && prevSW1){    // falling of SW1?
      color = (color+1)&0x07;     // step to next color
    }
    prevSW1 = SW1; // current value of SW1
    SW2 = GPIO_PORTF_DATA_R&0x01; // Read SW2
    if((SW2 == 0) && prevSW2){    // falling of SW2?
      UART_OutChar(color+0x30);   // send color as '0' - '7'
    }
    prevSW2 = SW2; // current value of SW2
    inColor = UART_InCharNonBlocking();
    if(inColor){ // new data have come in from the UART??
      color = inColor&0x07;     // update this computer's color
    }
    GPIO_PORTF_DATA_R = ColorWheel[color];  // update LEDs
    SysTick_Wait10ms(2);        // debounce switch
  }
}
```

Program 11.8. High-level communication network (C11_Network).

Video 11.5b. Chat Tool Program walk through

Video 11.5bLA. Logic Analyzer used as a Network Sniffer

Video 11.5c. Demonstration of the Chat Tool

11.5. Interfacing the Nokia 5110 Using a Synchronous Serial Port

Microcontrollers employ multiple approaches to communicate synchronously with peripheral devices and other microcontrollers. The synchronous serial interface (SSI) system can operate as a master or as a slave. The channel can have one master and one slave, or it can have one master and multiple slaves. With multiple slaves, the configuration can be a star (centralized master connected to each slave), or a ring (each node has one receiver and one transmitter, where the nodes are connected in a circle.) The master initiates all data communication. The Nokia5110 is an optional LCD display as shown in Figure 11.13. The interface uses one of the synchronous serial ports on the TM4C123.

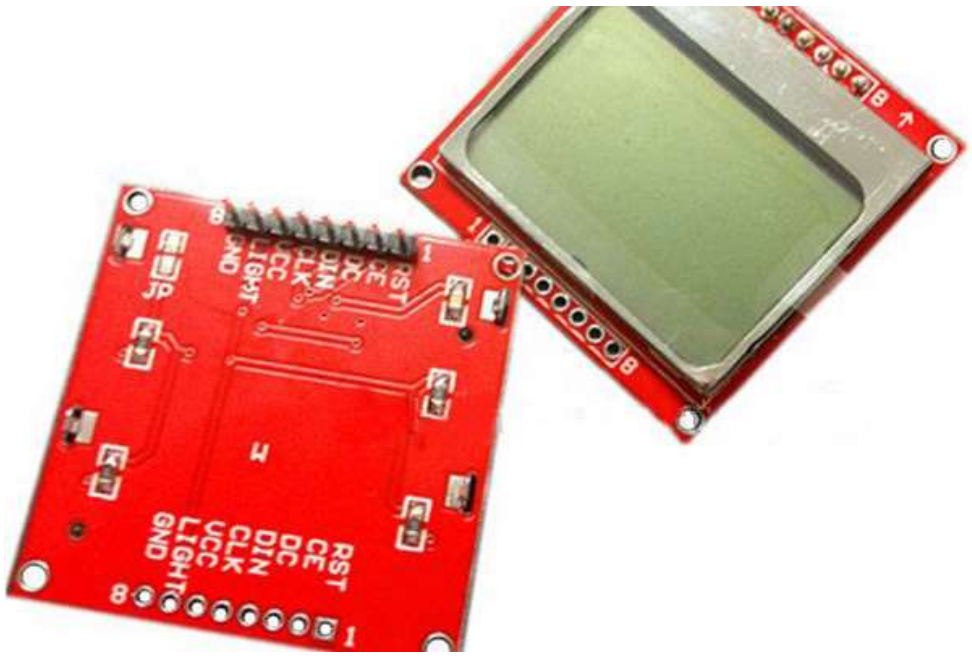


Figure 11.13. Optional Nokia 5110 LCD. Notice the PCB gives the signal names. Use the signal names not the numbers when connecting.

The TM4C123 microcontrollers has 4 **Synchronous Serial Interface** or **SSI** modules. Another name for this protocol is Serial Peripheral Interface or SPI. The fundamental difference between a UART, which implements an asynchronous protocol, and a SSI, which implements a synchronous protocol, is the manner in which the clock is implemented. Two devices communicating with asynchronous serial interfaces (UART) operate at the same frequency (baud rate) but have two separate clocks. With a UART protocol, the clock signal is not included in the interface cable between devices. Two UART devices can communicate with each other as long as the two clocks have frequencies within ±5% of each other. Two devices communicating with synchronous serial interfaces (SSI) operate from the same clock (synchronized). With a SSI protocol, the clock signal is included in the interface cable between devices. Typically, the master device creates the clock, and the slave device(s) uses the clock to latch the data (in or out.) The SSI protocol includes four I/O lines. The slave select SSI0Fss is a negative logic control signal from master to slave signal signifying the channel is active. The second line, SCK, is a 50% duty cycle clock generated by the master. The SSI0Tx (master out slave in, MOSI) is a data line driven by the master and received by the slave. The SSI0Rx (master in slave out, MISO) is a data line driven by the slave and received by the master. In order to work properly, the transmitting device uses one edge of the clock to change its output, and the receiving device uses the other edge to accept the data. SSI allows data to flow both directions, but the Nokia5110 interface only transmits data from the TM4C123. Notice the pin PA4 is not used, which would have allowed for receiving data from the device. The Nokia5110 interface does not use PA4.

Program 11.9 shows the I/O port connections and the Nokia display. Be careful, there are multiple displays for sale on the market with the same LCD but different pin locations for the signals. Please look on your actual display for the pin name and not the pin number. Program 11.9 also lists some of the prototypes for public functions available in the software stater project. If you have ordered and received the Nokia5110 display, open the C11_Nokia5110 starter project, connect the display to PortA. Be careful when connecting the backlight, at 3.3V, the back light draws 80 mA. If you want a dimmer back light connect 3.3V to a 100 ohm resistor, and the other end of the resistor to the **BL** pin.

```
// Blue Nokia 5110
// ---------------
// Signal        (Nokia 5110) LaunchPad pin
// Reset         (RST, pin 1) connected to PA7
// SSI0Fss       (CE,  pin 2) connected to PA3
// Data/Command  (DC,  pin 3) connected to PA6
// SSI0Tx        (Din, pin 4) connected to PA5
// SSI0Clk       (Clk, pin 5) connected to PA2
// 3.3V          (Vcc, pin 6) power
// back light    (BL,  pin 7) not connected, consists of 4 white LEDs which draw ~80mA total
// Ground        (Gnd, pin 8) ground

// Red SparkFun Nokia 5110 (LCD-10168)
// -----------------------------------
// Signal        (Nokia 5110) LaunchPad pin
// 3.3V          (VCC, pin 1) power
// Ground        (GND, pin 2) ground
// SSI0Fss       (SCE, pin 3) connected to PA3
// Reset         (RST, pin 4) connected to PA7
// Data/Command  (D/C, pin 5) connected to PA6
// SSI0Tx        (DN,  pin 6) connected to PA5
// SSI0Clk       (SCLK, pin 7) connected to PA2
// back light    (LED, pin 8) not connected, consists of 4 white LEDs which draw ~80mA
total//********Nokia5110_Init*****************
// Initialize Nokia 5110 48x84 LCD by sending the proper
// commands to the PCD8544 driver.
// inputs: none
// outputs: none
// assumes: system clock rate of 50 MHz or less
void Nokia5110_Init(void);

//********Nokia5110_OutChar*****************
// Print a character to the Nokia 5110 48x84 LCD.  The
// character will be printed at the current cursor position,
// the cursor will automatically be updated, and it will
// wrap to the next row or back to the top if necessary.
// One blank column of pixels will be printed on either side
// of the character for readability.  Since characters are 8
// pixels tall and 5 pixels wide, 12 characters fit per row,
// and there are six rows.
// inputs: data  character to print
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutChar(char data);

//********Nokia5110_OutString*****************
// Print a string of characters to the Nokia 5110 48x84 LCD.
// The string will automatically wrap, so padding spaces may
// be needed to make the output look optimal.
// inputs: ptr  pointer to NULL-terminated ASCII string
// outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
void Nokia5110_OutString(char *ptr);

//********Nokia5110_OutUDec*****************
// Output a 16-bit number in unsigned decimal format with a
// fixed size of five right-justified digits of output.
// Inputs: n  16-bit unsigned number
// Outputs: none
// assumes: LCD is in default horizontal addressing mode (V = 0)
```

```
void Nokia5110_OutUDec(uint16_t n);

//********Nokia5110_SetCursor*****************
// Move the cursor to the desired X- and Y-position.  The
// next character will be printed here.  X=0 is the leftmost
// column.  Y=0 is the top row.
// inputs: newX  new X-position of the cursor (0<=newX<=11)
//         newY  new Y-position of the cursor (0<=newY<=5)
// outputs: none
void Nokia5110_SetCursor(uint8_t newX, uint8_t newY);

//********Nokia5110_Clear*****************
// Clear the LCD by writing zeros to the entire screen and
// reset the cursor to (0,0) (top left corner of screen).
// inputs: none
// outputs: none
void Nokia5110_Clear(void);
```

Program 11.9. Wiring connections and high-level connection between the LaunchPad and the Nokia5110 LCD display (C11_Nokia).

For more information about SSI, see Section 8.3 in Embedded Systems: Introduction to ARM Cortex-M Microcontrollers, 2013, ISBN: 978-1477508992 or Section 7.5 in Embedded Systems: Real-Time Interfacing to ARM® CortexTM-M Microcontrollers, ISBN: 978-1463590154