EEE 158: Electrical and Electronics Engineering Laboratory V

The USART Microcontroller Peripheral

Introduction

Communications – and their means – are about as ancient as the universe has existed; we only need a quick look at the various languages history has borne witness to, for confirmation. Computing systems, including microcontrollers, are no exception – since the early days communications in one form or another has been a central theme in allowing computers to talk with other devices, ranging from on-site equipment to other systems located half a world away.

The development of computer communications dates back to the days of the wired telegraph in the 19th century, where messages were transmitted over a pair of wires using a sequence[1] of ON/OFF signals; for this reason, this type of communication is also called **serial communication**, as only one bit can be transmitted at any given time. Each message is broken down into a sequence of *symbols*, or *characters*; in turn, each symbol has its own unique bit sequence. Like human languages, there are conventions (protocols) about what sequence represents what symbol, as well as the rate at which bits and symbols are sent (bit rate and baud rate, respectively); both parties have to agree to all such parameters in order to properly reconstruct the original message from the sequence of symbols.

Tracking all serial protocols that have come into existence over the course of modern computing would be a never-ending endeavor, as new protocol designs come out and existing ones get upgrades. Here is a list of few of them:

- Universal Serial Bus (USB)
- 16550-compatible UART and its descendants
- PCI Express
- Ethernet (10/100/1000-Base-T [copper] and their fiber-optic cousins)

This document focuses on the 16550-compatible UART, which has become a *de facto* standard for serial ports in both PCs and embedded systems alike.

1 Like human language, ordering does matter.

The USART

The **universal synchronous/asynchronous receiver/transmitter**, or **USART** for short, is a peripheral that enables the microcontroller to transmit multi-bit characters over a single pair of wires; as well as receive bits and reconstruct them into multi-bit characters. The modern USART descends from the PC-compatible 16550 family of UART ICs, which form the foundations of the serial ports present from the first PC up to the late 2000's. Even though it has since virtually disappeared from the latest PCs, the USART retains its place in most embedded systems due to its relative simplicity and ease-of-use.

Trivia: The *universal* in USART means that the peripheral can also process other serial signals like SPI, LIN, and IrDA; while *synchronous* means that the data bits can also be synchronized by a clock signal separate from the data. The ancestor of the USART does not have provisions for externally clocking the data signal, and recovers clocking information from the data signal; hence the term UART.

Electrical Characteristics

In its raw form, USART signals are digital, TTL-level; meaning, they are either LO (ground (V_{SS})), or HI (positive (V_{DD})). For example, a microcontroller operating at 3.3 Volts has the LO level at 0 Volts, while the HI level is 3.3 Volts.

The transmit (TX) and ready-to-send (RTS) signals are outputs, while receive (RX) and clear-to-send (CTS) signals are inputs. Thus, when connecting two USARTs **TX and RTS on one end must be connected to RX and CTS on the other end, respectively.** If two TX or RTS pins are wired together, an output-to-output short circuit *will* occur, which can damage the ICs involved.

Another important consideration is the standard used on the PHY[2] layer. Older serial ports use +12V for LO and -12V for HI levels (RS232/EIA-232), while another may use 4-20mA current-loop signaling. **Connecting incompatible PHY layers together can cause damage to the transceivers involved.** That is why the electrical characteristics of connectors must be fully-ascertained before connecting two pieces of equipment together.

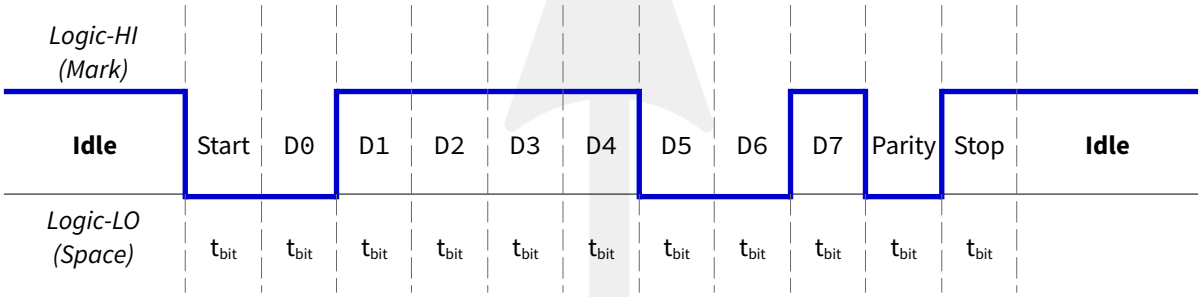
Yet another important consideration is the fact that voltages are always relative to a reference node. Connecting two systems without connecting their grounds together can cause ground loops to form, whose effects can range from annoying to destructive. One rule of thumb is to connect ground first, and disconnect it last.[3]

Timing

Figure 1 shows the timing diagram of a USART processing one character; as shown in the figure, each bit is spaced t_{bit} seconds apart, which comprises one bit period. In principle, one baud (symbol) would be made up of all bits between the two idle conditions, exclusive. However, in the early days of computer networking modems could only transfer one bit per symbol time; thus, **for USARTs one bit and one baud are the same**. The bit (baud) rate, in bps (bits per second) is the inverse of the bit period.

2 PHY = Physical; might have encountered this in EEE 157 (?)

3 More information can be found by searching for *Electrical hot swap*. By this time, you should already be familiar with this rule (eg. oscilloscope connections on powered-up circuits since EEE 118).



Byte = 158 = 0x9E, 8-bit mode, odd parity, 1 stop bit

Figure 1: USART Timing Diagram

The figure also shows several segments of transmission:

- **Idle.** This means its standard definition – nothing is being transmitted. This state is always a logic-HI, harking from the days of early telegraphy / telephony where the line was energized with a current at idle to enable detection of broken links.
- **Start bit.** This bit is used to denote the start of the frame. This is always logic-LO.
- **Data bits.** These contain the actual data being transmitted, **least-significant bit (LSB) first**. This means that, for example, the character `0x9E` will be transmitted as `01111001` rather than `10011110`. Sizes historically varied from 5 to 8 bits; but the vast majority of USARTs in operation use 8-bit, which correspond to the byte we are all familiar with (cue: EEE 111/121).
- **Parity.** This optional bit may be present to provide for error detection. There are five types of parity:
  - **None.** No parity bit is included; after the last data bit, a stop bit immediately follows.
  - **Even.** If the number of HI bits in the data are even, the parity is LO to maintain even count. Otherwise, the parity is HI to make the count even. For example, 0x9E has an odd number of HI bits; to satisfy even parity, the parity bit must be HI.
  - **Odd.** Works the same as even parity, but this time the number of HI bits must be odd. For the 0x9E example, since the number of HI bits is already odd the parity bit must remain LO to satisfy odd parity.
  - **Mark.** Forces the parity bit to be HI; not useful for error detection, but may be of use depending on the application.
  - **Space.** Same as mark parity, but forces the parity bit LO instead.
- **Stop bit/s.** These signify the end of the frame, and may either last 1, 1.5, or 2 bit periods. These are always logic-HI.

Shift Register

Internally, the USART peripheral is based on a *bit-shift register*, or BSR for short. For illustration purposes, say we have a 16-bit BSR whose initial contents ('pre-load') are 0xC09E; the BSR appears as shown in Figure 2.

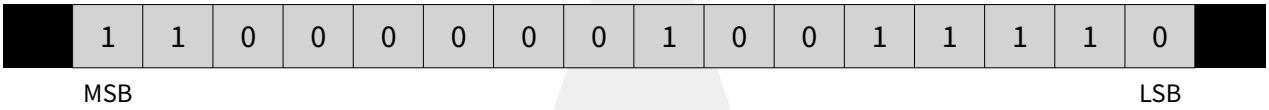


Figure 2: Bit-shift Register

As clock pulses are applied to the BSR ('clocked'), the bits get shifted either to the left (MSB), or to the right (LSB); the bit that occupied that position is either used somewhere or discarded (*shift-out*). As a result, a gap forms at the other end; the value filling the gap (*shifted-in*) depends on the application – it could be a fixed-HI, fixed-LO, or the bit just shifted-out. The last case results in a *circular shift register*, where bits keep going in circles.

Visually, the shifting operations can be seen in Figures 3 and 4; in either figure, the blue boxes represent bits being shifted out of the BSR, and the black boxes represent bits being shifted into the BSR.

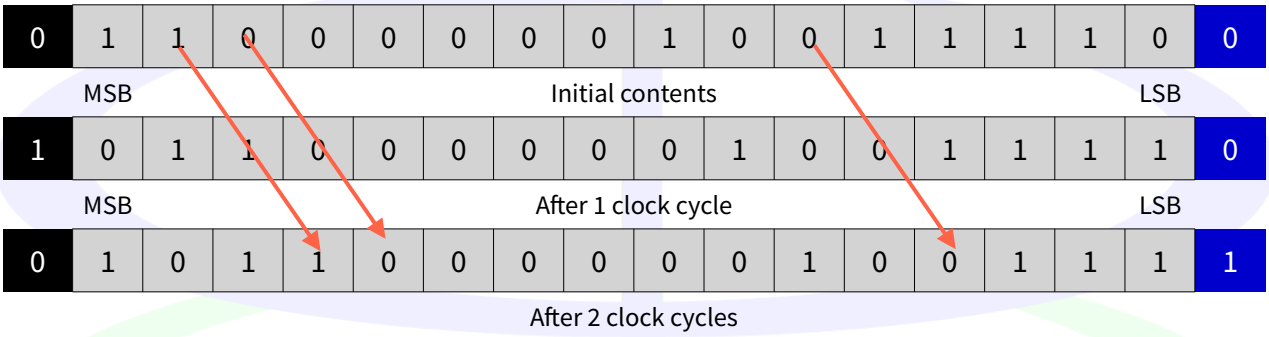


Figure 3: Right-shift Operation

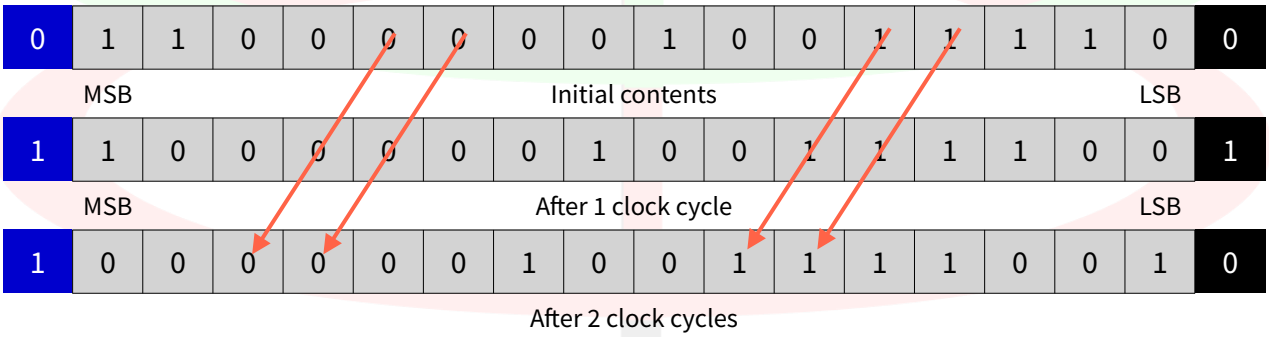


Figure 4: Left-shift Operation

Transmission

When transmitting data, the following sequence of events is carried out:

1. The BSR is pre-loaded with the character to be transmitted.
2. One space bit is transmitted, denoting the start of the frame ('start bit').

3. The contents of BSR are shifted out onto the TX ('transmission') line, starting from the least-significant position.
4. If enabled, the appropriate parity bit is sent out after all bits in BSR have been shifted out.
5. One or two mark bits are transmitted, depending on the configuration; these signify the end of the frame ('stop bit/s').
6. The above steps are repeated for other characters, if applicable.

The bits are clocked at a rate that is an integer divisor *N* of the physical clock; for instance, with N=4 the bits are clocked at 1/4 of the physical clock. The reason for this is to maintain synchronization with the receiver, which has more timing requirements as discussed in that section.

Reception

When receiving data, the following sequence of events is carried out:

1. Starting from idle, a HI→LO transition is detected on the RX ('receive') line. This signals the start of the frame, and lasts one bit period.
2. The next 8 bits are shifted into the BSR from the MSB position.
    1. Numbers other than 8 may be used depending on the application.
    2. The MSB position is always at position (n-1); the idea is that the first data bit will always end up at the LSB position. For example, 8-bit characters are shifted into the BSR at position 7.
3. If enabled, a parity bit is then received. If even or odd parity is configured, the receiver checks if the number of HI data bits plus parity is odd or even, respectively; if false, a *parity error* (PE) is signaled.
4. The last 1 or 2 bits must be HI, and denote the end of the frame ('stop bits').
5. A padding period of 1 or 2 bits of IDLE may be used. To receive another character, the process is repeated.

The receiver maintains a finite state machine (FSM) that indicates whether the bit being expected for a bit period is *idle*, *start*, *data*, *parity*, or *stop*. An error occurring during the *start* period causes the frame to be aborted, and the receiver to restart from the *idle* state. If the input is LO during the *stop* period, a *framing error* (FE) is signaled. If the LO state persists beyond the alloted time for the stop state a *break condition* is signaled; this can be used to detect disconnection from the remote terminal, as well as reset the communications state, among other actions.

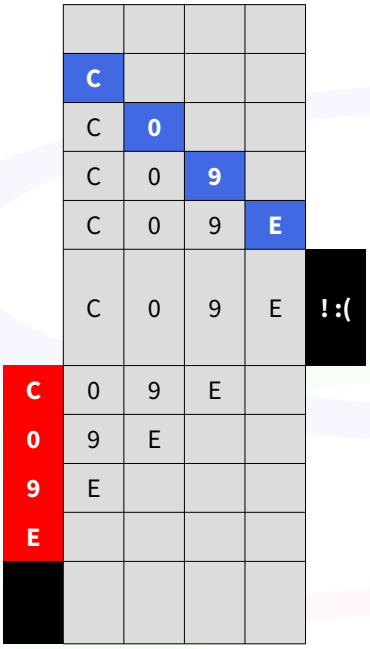
Digital Filtering

If bits were sampled naïvely by the receiver, the received data will almost-always be corrupted in one way or another; possible sources of corruption include noise (*always present in any real-world environment*) and configuration mismatches (eg. 38400 bps vs. 1200 bps). To avoid this, the receiver applies digital filtering such that one logical bit corresponds to a sequence of $N > 1$ consecutive physical-bit samples; as a result, the required physical clock rate is at least *N* times the target bit rate. Typical values for *N* are N=8 and N=16; lower values of *N* allow for higher speeds, but makes communications more susceptible to noise. On the other hand, higher values of *N* make communications more-resistant to noise, at the cost of lower data rates.

Buffering

Now that we have discussed physical considerations when using USARTs, we also need to pay attention to the logical side. While the BSR is shifting bits around, we cannot pre-load new data into it without corrupting the existing transmission; for the same reason, we cannot expect to get clean data out of the BSR while bits are being shifted in. For this reason, USARTs are required to have a FIFO (*first-in, first-out*) queue with space for at least one character in each direction. As characters are received, they are placed at the tail of the RX queue; the application reads them from the head end. For transmission, the peripheral plucks the character from the head end of the TX queue, while the application places bytes at the tail of this TX queue.

A visual representation is shown in the Figure 5 using a 4-element queue.



(1) Queue is empty.

(2) One element is inserted into the queue (blue).

(3) Another element is inserted.

(4) Another element is inserted.

(5) Another element is inserted. The queue is now full.

(6) Another element comes in, but can't be accomodated because the queue is full; an *overrun* has occurred and the character must be discarded.

(7) An element is popped from the queue (red).

(8) Another element is popped.

(9) Another element is popped.

(10) Another element is popped. The queue is now empty.

(11) The reader tries to pop again, but no more elements are available in the queue. An *underrun* has occurred.

Figure 5: Four-element Queue

Another way to visualize FIFO operation would be to imagine a water tank with one inlet pipe (producer) at the top, and one output pipe (consumer) at the bottom; flow rates of water through the pipes represent data flow in a FIFO. There are three cases to consider:

- **Large inflow, small outflow:** In this case, the tank will eventually fill up and overflow; some water would be irretrievably wasted as spillage. In a FIFO, this is referred to as *overrun*.
- **Small inflow, large outflow:** Here, the tank will eventually be completely emptied of water; when it happens, either the consumer must wait for a fresh supply of water to the tank, or do something else. In a FIFO, this is referred to as *underrun*.
- **Inflow = Outflow:** The consumer is using the same amount as the producer is outputting; on average, the tank's water level remains steady.

Circular Buffering

If we try implementing Figure 5 as-is, each time an element is dequeued (popped) all other elements need to be shifted to the left, which is an $O(n)$ operation. In other words, the time and complexity scale linearly with the number of elements in the queue, the worst occurring when the queue starts full.

For most systems, in a full communications FIFO the oldest element is usually no longer of any relevance, as its timeliness has expired. Thus, instead of discarding the newest data as in a standard FIFO, it instead *overwrites* the oldest element in the FIFO. Significant performance gains are realized because only 2 array indices – one for the producer, the other for the consumer – need to be updated, each having the cost of an increment and a modulo operation. A further optimization is possible if the FIFO size is a power of two – AND-bitmasking can be used instead, which is generally faster than modulo. Logically, the FIFO operates in circular, or ring fashion; hence the name.

A corner case can occur when the indices are numerically equal – the same condition represents either a full queue, or an empty one. One way to break the ambiguity is by designating one value as the NULL value – meaning if this value is encountered the participants act as if no data was there. Another would be the use of a counter, which must be clamped to avoid mis-counting.

Flow Control

In some applications, explicit indication of stopping or resuming flow of data is important. Termed flow control, it takes one of two forms:

- **Software (XON/XOFF):** Two special characters are sent which causes transmission to resume (XON) or pause (XOFF). The values are dependent on the application, but are most-commonly the ASCII values 0x11 and 0x13, respectively. When this flow control method is used, escape sequences must be designed to represent 0x11 and 0x13; they cannot be sent in any manner without causing either a pause or resumption.
- **Hardware (RTS/CTS):** Two physical lines are used, in addition to the TX and RX lines. When a USART is ready to receive more data, it asserts (HI→LO) its RTS (ready-to-send) output, informing the other end via its CTS (clear-to-send) input that data can now be sent again to the first USART. With out-of-band control lines, there is no need for inherently escaping special characters for transmission.

In applications where this is not necessary, neither method is activated (“None”); this is by far the most-common flow-control configuration.