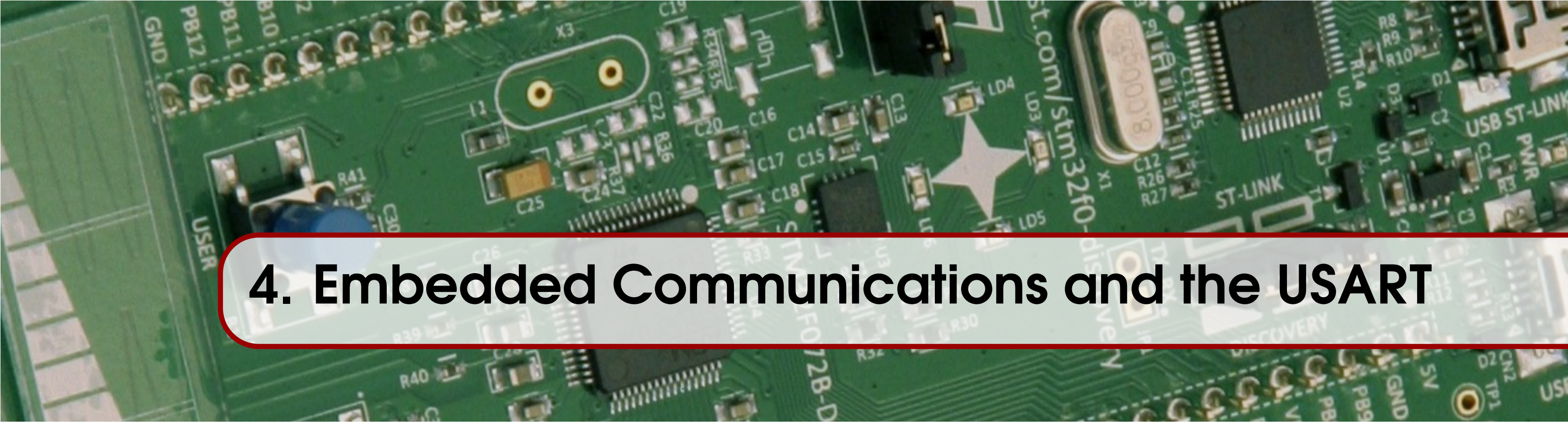


# 4. Embedded Communications and the USART

## 4.1 Communication Between Systems

All computing systems regardless of their size and power, are essentially useless if they do not have some way to communicate with the outside world. Even large supercomputers are unable to perform any meaningful work without some method of introducing new data into the systems, and another to output a result.

Naturally, this applies to embedded systems as well. In fact, the majority of embedded peripherals, as well as these labs, are directed at either capturing or sending information in and out of the system. In the first lab, we introduced the GPIO as the most basic method of data transfer. In the second we used the EXTI controller to allow efficient hardware monitoring of GPIO inputs. The third lab used the capture/compare units in a timer peripheral to generate pseudo-analog signals through PWM.

This lab introduces the fundamentals of data transfer through digital communication interfaces. These interfaces exist because information is only useful when it can be understood. Therefore there must be defined ways of transmitting and interpreting it.

### 4.1.1 Parallel vs Serial

There are two main schemes, parallel and serial, for moving binary information across electrical connections. These two schemes are opposites of each other and are demonstrated in figure 4.1.

#### Parallel

Parallel interfaces transmit entire blocks of data using multiple wires, with each wire representing the value of a single binary bit. In a parallel system, the transmitter sets the logical state of each wire, and the receiver samples all of the connections at a single instant. Parallel interfaces have a *bit-width* which represents how many wires are in the connection and indicates how many bits are sent at one time. Common bit-widths are powers of 2 to make converting into bytes simple.

#### Serial

Serial interfaces use a single wire and stream a block of data over time by lining up the bits behind each other. To properly transmit data, both the transmitter and receiver must agree on the time duration between data bits, known as the interface's *bit/data rate*. A serial transmitter produces periodic transitions on the single data line corresponding to the data to be sent. The receiver samples this data line on a similar period and appends the sampled value to the end of the received data.

#### Interface Bandwidth and Limitations

Because parallel connections move entire groups of bits at a time they have a much higher *bandwidth* (throughput) than an equivalent serial connection. Some examples of high-speed parallel connections are the STM32F0's internal device busses (AHB and APB) which connect the ARM processor core to the SRAM, flash memory and other peripherals.

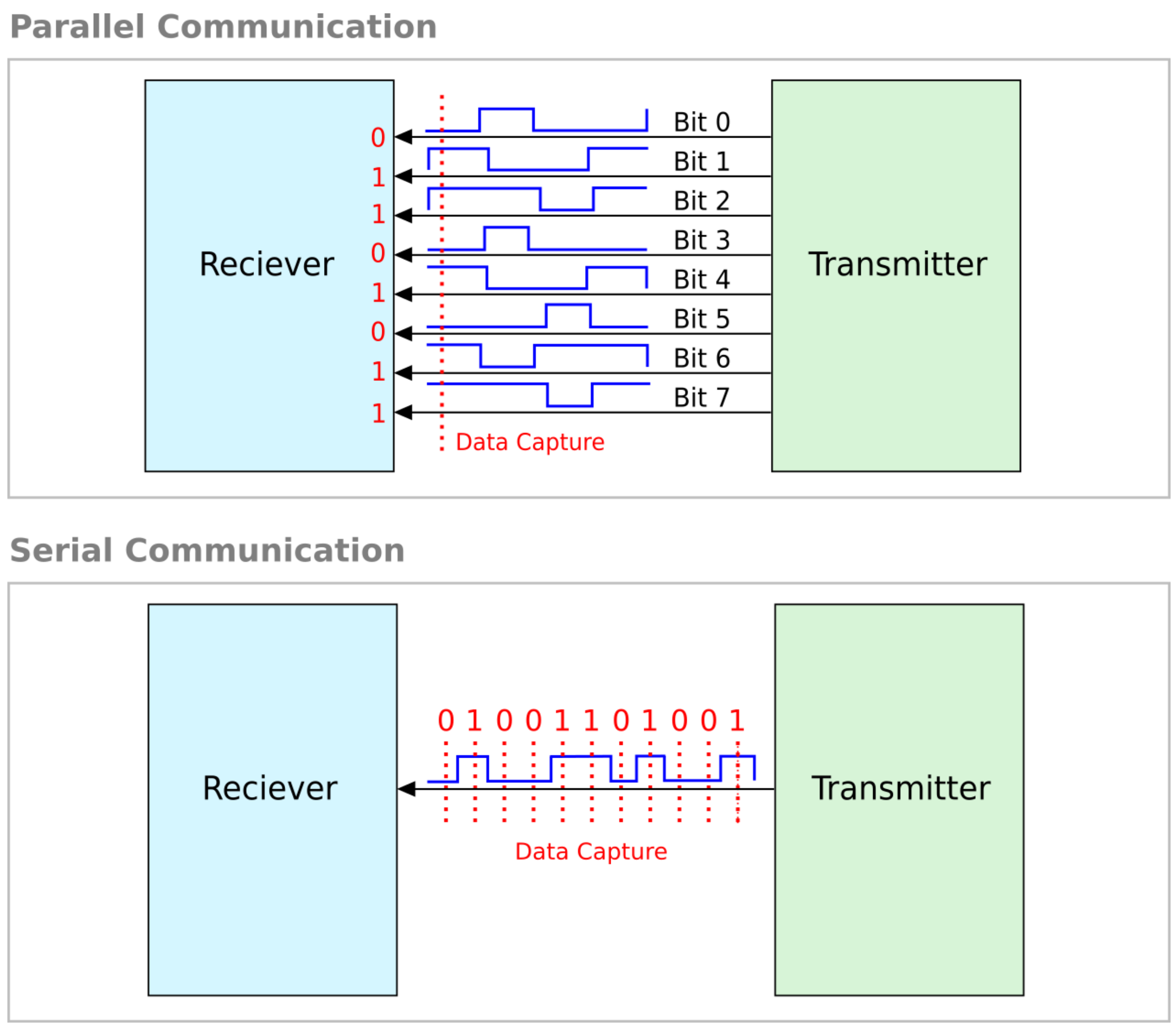


Figure 4.1: Comparison of parallel and serial communication.

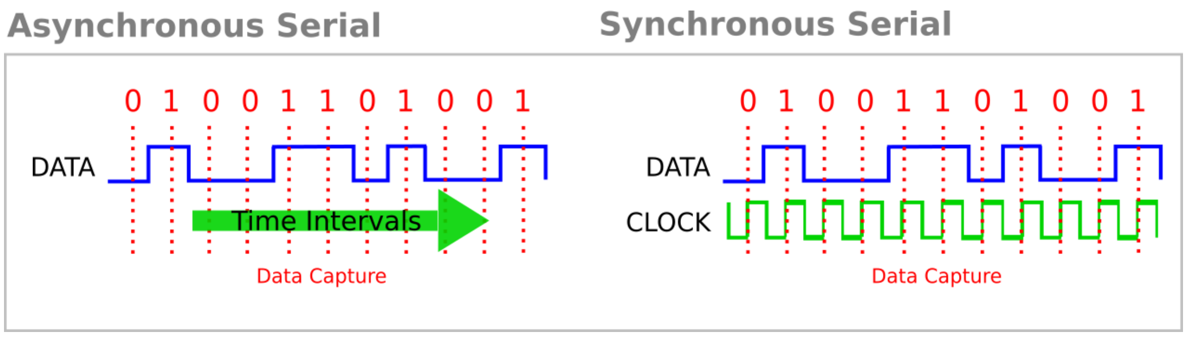


Figure 4.2: Comparison of synchronous and asynchronous serial communication.

However, despite that equivalent-speed parallel connections have a higher bandwidth, most high-speed device interfaces such as SATA, USB and Ethernet are serial interconnects. The reason for this is because parallel connections are far harder to design and operate as the distance between the receiver and transmitter increases. With the incredibly high speeds of today's communications, unless every wire in a connection is exactly the same length it is possible to have some bits arrive after the others. This means that unless the sampling rate of the receiver is slow enough to account for every bit of the connection, incorrect data will be captured.
Serial connections inherently don't have issues with wire-delay. Because bits are streamed one after another, any delays along the wire have a consistent effect on the data and can't cause corruption. This enables serial connections to have a much higher bit rate than parallel, and possibly even a higher total bandwidth.

### 4.1.2 Synchronous vs Asynchronous

Regardless of whether an interface is serial or parallel, there must be a mechanism that synchronizes when the transmitter and receiver updates or samples the connection. Similar to parallel and serial there are two methods, synchronous and asynchronous, of performing this task. These are demonstrated in figure 4.2.

#### Synchronous

Synchronous systems use a separate "clock" signal which notifies the receiver when to sample. Often the data capture is synchronized to a transition such as rising or falling edge. Synchronous systems are often simpler in design but require the extra clock connection.

#### Asynchronous

Asynchronous systems operate without a physical clock signal. Some methods of asynchronous communication encode a virtual clock within the transitions of the data while others estimate the time intervals that data should be expected to arrive. Because of the lack of a clock signal, asynchronous interconnects are typically more complex and have lower data rates than synchronous ones.

### 4.1.3 Connection Topologies

The topology of a communication interface is how the different devices (nodes) and the connections between them are arranged. Some interfaces are strictly point-to-point, which means that they connect only two devices with direct wires. Other interfaces have topologies which allow networks of devices to be connected. Figure 4.3 demonstrates a few simple topologies.

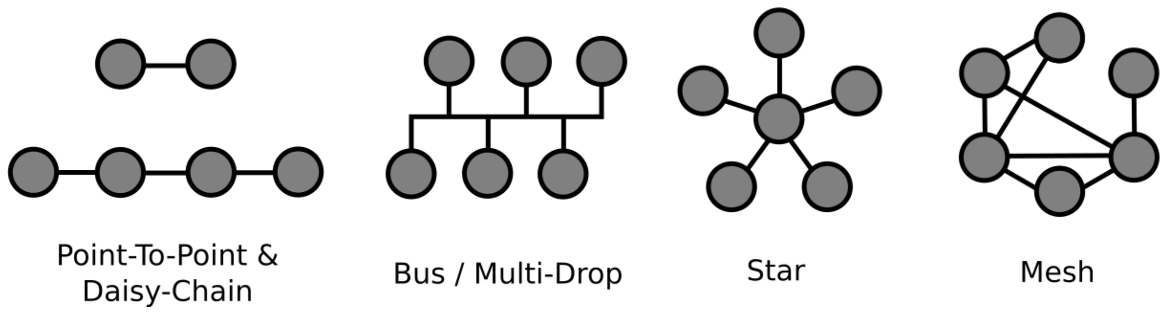


Figure 4.3: Simple network topologies.

- **Daisy-Chain** – Nodes in a daisy-chained network connect only to their adjacent neighbors. Some are designed to allow data to be passed along to its destination.
- **Bus/Multi-Drop** – All nodes on a bus share the same communication lines. This means that only a single device can be transmitting at a time, but all can receive
- **Star** – All peripheral nodes communicate with a central master node.
- **Mesh** – Nodes in a mesh network have arbitrary connections to each other. Some mesh networks are hierarchically ordered, others may be fully-connected.

## 4.2 Communication Standards & Protocols

Aside from knowing how to connect and when to sample the inputs, communicating devices must know how to interpret signals into bits and how the patterns of bits make up data. To manage this, all interfaces are combinations of hardware standards and communication protocols.

### 4.2.1 Hardware Standards

Hardware standards define the physical and signaling characteristics of an interface. A standard typically indicates the medium that the signals are transferred over such as conductive wires, optical, radio frequency or others. The standard defines the method that signals are interpreted into bits. This may be as simple as thresholding a voltage on a wire (single-ended), monitoring the voltage difference or current flow between a pair of wires (differential) or even a modulated waveform.
Standards often indicate whether an interface is parallel or serial.

### 4.2.2 Communication Protocols

Hardware standards make it possible to convert an input signal into a collection of bits. However, protocols define the meaning of bits such that they create useful data.

#### Hardware Protocols

Low level or hardware protocols define how bits are organized to form raw data. This primarily involves the rate that data is sampled and whether the interface has an explicit clock signal or is asynchronous. Depending on the complexity of the interface, hardware protocols usually include higher-level features such as start/stop signals, error-correction, control flow, message acknowledgment, addressing, data packets and more.

#### Software Protocols

At this point the communication peripheral's job is complete. Raw data can be converted into a communication signal and back on the receiving end. The last task is to interpret the raw data into useful instructions or information for the user's application.
A software protocol is a device driver that gives meaning to the binary data flowing into and out of the system. These drivers can be as simple as recognizing certain values as commands, or as complex as defining the organization of variable-length data packets. In order to communicate, the user's application must define a software protocol.

### 4.2.3 Separating Interface and Protocol

Some communication interfaces have clearly defined separations between the different standards, protocols, and application layers within them.
For example, IEEE 802.3 is the base standard for wired Ethernet and contains subclasses for the different speeds and modifications that have been made over the years. Built upon these standards are multiple layers of protocols (such as MAC, IP, and TCP) which eventually lead to network sockets that transfer raw data on conventional computer networks.
An example of a common software protocol used by computer networks is HTML. This format is used to give the browser application a known way to interpret raw text data into a displayed web page.
Unfortunately, unlike Ethernet, many low-level interfaces such as those used in embedded systems are less clear about where the boundaries are. Many of these interfaces have protocols and standards that were designed specifically for each other and are called by the same name. Because of this, many documents use the terms standard and protocol interchangeably.

## 4.3 Common Embedded Interfaces

The STM32F072 has a generous selection of interface peripherals. These can be seen on the first page summary of the chip datasheet. These interfaces differ widely in operation, complexity, and features; however, they all are serial.
The three most common serial communication interfaces used in embedded systems are TTL RS-232 (TTL-Serial/UART), Serial Peripheral Interface (SPI), and Inter-Integrated Circuit Bus (I2C).
This lab teaches the basics of TTL RS-232 in the next few sections. SPI and I2C will be covered in later labs; a brief introduction is included here.

- **Serial Peripheral Interface (SPI)**
  – SPI is a synchronous interface typically used for high-speed connections between micro-controllers and external memories or other fast-data devices.
  – It is a full-duplex interface, which means that both sides transmit and receive at the same time.
- **Inter-Integrated Circuit Bus (I2C)**
  – I2C is a low(er) speed synchronous interface designed to allow multiple master devices to share the same set of signal wires.
  – It was primarily intended for low-speed sensors, but newer standards have increased the speed where it is also used for other purposes.

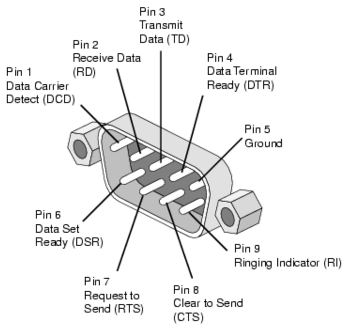


Figure 4.4: Conventional DE9 serial connector and pinout.

## 4.4 Introducing RS-232

### 4.4.1 Conventional RS-232

RS-232 was first introduced in 1962 by the Electronic Industries Alliance as a method of connecting teletype terminals to a central computer. It is a point-to-point, asynchronous, full-duplex (simultaneous bi-directional communication) interface with hardware flow control signals.

Despite changing its voltage levels and connectors over the years, RS-232 became the standard method of interconnection between computers and accessories until it began to be replaced in the late 1980's by PS2 and later USB. Because of its long and common use, RS-232 is often referred to simply as "Serial."

RS-232 contains features originally designed for slow dial-up connections. The flow control signals allow the transmitter and receiver to directly notify each other that they are ready with new data or that they need a pause to process what has been transmitted. To ensure that the data signal was able to be read across a potentially long cable, RS-232 uses a wide voltage swing with positive voltages representing a '1' and negative voltages for a '0.' Although the standard allows up to ±25V many devices use between ±12V and ±15V.

The most recent and common connector used for RS-232 is the 9-pin trapezoidal DE-9 (often mistakenly called DB-9). Figure 4.4 shows a representation and pinout of the connector. While most consumer equipment does not feature a physical serial port anymore, RS-232 is still commonly used for equipment control within science and industry.

### 4.4.2 Embedded TTL RS-232 (TTL-Serial)

Unlike the original legacy hardware, many embedded systems run at far lower voltages than the RS-232 ±25V specification. Additionally, many of these systems cannot generate or tolerate negative voltages. Because of this, most systems use a variant of RS-232 called TTL Serial.

TTL Serial is the same protocol as RS-232 but uses a more reasonable voltage range for signaling. Depending on the power supply that the embedded system operates from, TTL Serial often has a voltage range between 0V and 1.8V to 5V.

Figure 4.5 shows a detailed look into an RS-232 and TTL Serial data frame (packet). As shown, a TTL Serial signal not only has a smaller voltage range, but the waveform is also inverted.

RS-232 is a very flexible protocol in that many of the communication parameters are adjustable by the user. These next few sections discuss important characteristics of an RS-232 frame.

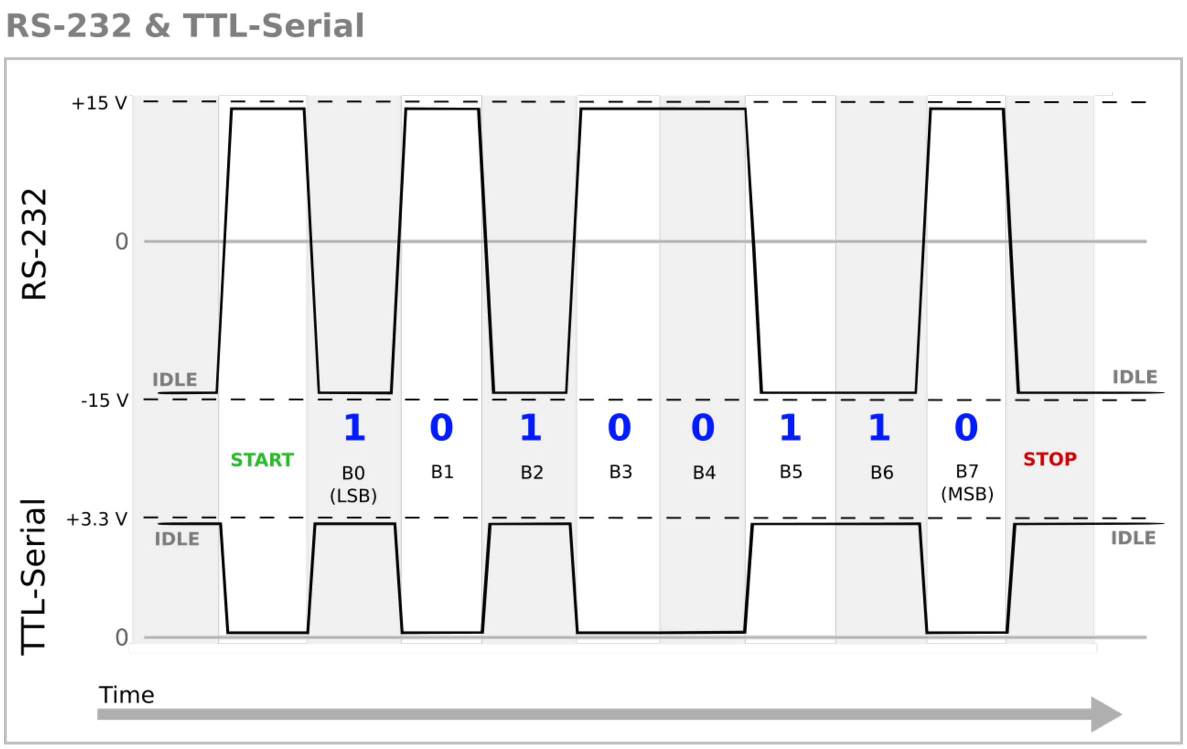


Figure 4.5: RS-232 and TTL-Serial logic levels and polarity.

### Baud Rate

Because RS-232 is asynchronous, both the transmitter and receiver are required to operate on a pre-agreed period between bits. This frequency is known as the *Baud Rate* and represents the number of bits per second that is transmitted.
RS-232 has no checking mechanism to ensure that both devices are tuned to the same baud. This means that it is possible to receive corrupt data if the baud settings are not configured properly.
In figure 4.5 the bit periods are indicated by alternating white and gray shading.

### Start and Stop Bits

Although the official RS-232 protocol does not have a mechanism to detect the receiving Baud rate, it does include *framing bits* which alert the hardware about newly arriving data.
The first signal transition in figure 4.5 is the *start bit* of a data frame. The start bit is used to signal a new data frame and synchronize the clock of the receiver to the transmitter.
The last transition is the *stop bit* which serves as a spacer to give a minimum delay time before the next frame can start. Originally this was designed to allow the mechanical hardware in the teletype machines to move to the next position in the document.
The number of start and stop bits within a data frame is configurable. Many serial peripherals allow the user to select the number of framing bits. However, similar to the baud rate, both devices need to agree to interpret the data correctly.

#### Data Bits

Typically each frame in RS-232 contains a single data byte. Although uncommon, the standard allows non-byte data sizes between 5 and 9 bits. The baud rate of the signal includes the framing bits within its calculation of total bits per second. This means that the actual useful data transmitted is always less than the theoretical throughput indicated by the baud.

#### Parity & Error Checking

Most hardware peripherals support appending a *parity* bit to the end of the data. Parity is a method of error checking, where the parity bit is set such that the number of ‘1’ bits in each data frame is always even or odd. If the number of received ‘1’ bits fails to match the expected parity, then the hardware can determine that the transmission has been corrupted.

Parity can only determine if a single bit in the data has been corrupted, it does not provide for any error correction and can fail if an even number of bits are corrupt. Parity is no longer used often because of this limitation. Many devices contain hardware for performing more sophisticated *cyclic redundancy checks* (CRC).

### 4.4.3 ASCII Text Encoding

Unlike more sophisticated interfaces, RS-232 does not require any device enumeration or addressing. This enables us to interface with serially connected devices by simply typing characters into a terminal. Regardless of whether a device is attached to the other end is irrelevant, data will be streamed out the transmit line, and any data received will be printed into the terminal console.

Streaming binary data across a serial link is straightforward. Ideally, the same binary values will be received as was transmitted. However, how is text-based data transfered considering that a serial interface only moves raw bytes?

Over the years, organizations have created standardized ways to encode text characters as numerical values. One of the oldest of these is the *American Standard Code for Information Interchange* (ASCII). ASCII originally assigned a 7-bit value to 128 text characters in the English language. These included upper and lower case letters, numbers, common punctuation marks, and a selection of control codes which modify where and how text was printed in the original teletype machines. Initially, the 8th bit of the byte-wide ASCII encoding was reserved for parity checking. However, this was later changed to allow an additional 128 characters to be added to the extended encoding.

Only a few of the original control codes still have meaning in modern systems. Examples of these are the ‘\t’ (Tab) and ‘\n’ (Newline) characters. Figure 4.6 shows some of the more commonly used English characters and their ASCII numerical values.

In conventional computing, ASCII has largely been replaced by variable-length text encodings such as Unicode. These attempt to include accents and non-Latin characters used by other languages. Although these encodings are far superior to ASCII, they typically keep the old values for the original ASCII characters for backward compatibility. This means that we can successfully use a serial terminal with our embedded systems while only dealing with decoding the original character set.

While it is possible to include a Unicode decoder in an embedded system, the size and complexity may outweigh the potential benefit. Because of this, all text encodings used in the assignments are assumed to be ASCII.

| Common Other | | | | Upper Case | | | | Lower Case | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | **\t** (Tab) | 48 | **0** | 65 | **A** | 78 | **N** | 97 | **a** | 110 | **n** |
| 10 | **\n** (NL) | 49 | **1** | 66 | **B** | 79 | **O** | 98 | **b** | 111 | **o** |
| 13 | **\r** (CR) | 50 | **2** | 67 | **C** | 80 | **P** | 99 | **c** | 112 | **p** |
| 32 | *Space* | 51 | **3** | 68 | **D** | 81 | **Q** | 100 | **d** | 113 | **q** |
| 33 | **!** | 52 | **4** | 69 | **E** | 82 | **R** | 101 | **e** | 114 | **r** |
| 44 | **,** | 53 | **5** | 70 | **F** | 83 | **S** | 102 | **f** | 115 | **s** |
| 46 | **.** | 54 | **6** | 71 | **G** | 84 | **T** | 103 | **g** | 116 | **t** |
| | | 55 | **7** | 72 | **H** | 85 | **U** | 104 | **h** | 117 | **u** |
| | | 56 | **8** | 73 | **I** | 86 | **V** | 105 | **i** | 118 | **v** |
| | | 57 | **9** | 74 | **J** | 87 | **W** | 106 | **j** | 119 | **w** |
| | | | | 75 | **K** | 88 | **X** | 107 | **k** | 120 | **x** |
| | | | | 76 | **L** | 89 | **Y** | 108 | **l** | 121 | **y** |
| | | | | 77 | **M** | 90 | **Z** | 109 | **m** | 122 | **z** |

Figure 4.6: Subset of the ASCII text encoding standard.

## 4.5 Introducing the USART

The *Universal Synchronous Asynchronous Receiver Transmitter* (USART) as suggested by its name, is one of the most flexible communications peripherals available. Its use is common enough that most embedded devices regardless of their manufacturer have multiple USARTs or at the least a simpler Universal Asynchronous Receiver Transmitter (UART).
Although a USART/UART can be configured into a variety of custom communication modes, they are often used to perform basic TTL serial. Sections 27.2 through 27.4 of the peripheral reference manual document the main features of the USARTs found within the STM32F072. From these sections, it is possible to see that the flexibility of the USART makes it appear as a complex device to configure.
Luckily, due to the popularity of using the USART as a TTL serial device, many of the options that we would normally need to enable are configured by default. The following sections of this lab manual will attempt to provide an overview of each control register used for TTL serial and the options they configure.

### 4.5.1 USART Registers

#### Control register 1 (USART_CR1)

Control register 1 enables/disables interrupt conditions and portions of the USART peripheral. It sets basic timing and sampling modes as well as the number of data bits in a frame. (The manual calls serial frames “words”.)
Note that many bits within the register have a note in their description stating that they are only editable when the USART peripheral has not yet been enabled for communication.

#### Control register 2 (USART_CR2)

Control register 2 controls signal polarity and routing for the USART. A common and irritating occurrence when designing circuits is the accidental swap of the receive and transmit lines. Through the control registers, it is possible to swap the signals within the USART peripheral itself.

Other functions of the control register 2 involve the LIN interface mode and stop bit configuration

#### Baud rate register (USART_BRR)

The baud rate register is the prescaler for the USART's baud rate frequency. The value written here can be derived from the processor clock and target communication rate. This will be discussed in the next section of the lab.

#### Interrupt and status register (USART_ISR)

The interrupt and status register is read-only and indicates different operational states of the USART. Unlike the previously used peripherals, the USART self-clears many of its condition flags once their conditions have been resolved.
Read the bit descriptions carefully to determine if a flag needs manual attention.

#### Interrupt flag clear register (USART_ICR)

For the flags that need manual clearing, or if a certain condition needs to be bypassed, the flag clear register allows the user to clear status flags.

#### Receive data register (USART_RDR)

The receive data register holds the last completely received byte of data. The USART has an internal buffer which holds in-progress receptions until they have completed successfully. If new data is available and the receive register contains unread data, the USART discards the new information and sets an overrun error condition.

#### Transmit data register (USART_TDR)

The transmit data register holds the next byte that is waiting to be transmitted. The USART has an internal buffer which holds data currently in the process of transmission. This means that the transmit data register can be safely filled with new data while the previous is still being processed. The USART will signal an empty transmit register with a flag in the status register.

#### Other Control Registers

These registers contain advanced configuration options primarily for alternate modes of the USART peripheral. While recommended that you browse through them, you will not need to use these for these lab assignments.

- **Control register 3 (USART_CR3)** – Manages hardware flow control, DMA, and Smartcard-interface.
- **Guard time and prescaler register (USART_GTPR)** – Used in low-power and Smartcard mode.
- **Receiver timeout register (USART_RTOR)** – Used by Smartcard mode.
- **Request register (USART_RQR)** – Manually triggers USART events/interrupts.

### 4.5.2 Configuring the Baud Rate

The baud rate of the USART depends on three factors: the target frequency, the processor clock speed, and the oversampling mode.

Because it is impossible to completely synchronize the clocks between the transmitter and receiver the USART compensates for the inaccuracy by sampling the input much faster than the original baud rate. In the default oversampling mode, the USART samples the input 16-times faster than the bit-rate. It then uses the average of the multiple samples to decide if the bit should be a one or a zero. The USART can reduce the oversampling to 8x to allow more rapid baud rates to be chosen. This mode should only be selected when signal integrity is known to be good.
Section 27.5.4 documents in detail the process of calculating the value to be written in the baud rate register. The simple equation used is as follows:

$$Baud_{TX/RX} = \frac{f_{CLK}}{USART\_BRR}$$

■ **Example 4.1 — Calculating Baud Rate.** Consider a system with a processor clock prequency of 8 MHz, 16x Oversampling, and a target baud rate of 9600.

$$Baud\_Divider = 8000000/9600 = 833.33\overline{33} \qquad USART\_BRR = 833$$

$$Actual\_Baud = 9604 \qquad Error = 0.05\%$$

■

Most baud rates will not be exactly achievable using the frequencies that the STM32F0's internal oscillator can produce. In general the higher the baud rate in relation to the processor's clock, the greater the error due to clock cycle granularity. When choosing a baud rate, aim for the highest possible standard value that gives reasonable accuracy. Typically, most manufacturers recommend an error of less than 2%.

### 4.5.3 Blocking vs Non-blocking Operation

There are a number of choices to be made when writing code that interfaces with a communication peripheral. Typically you will be sending more data than will fit in a single transmission frame and you will need to wait while the interface alternates between transmitting and requesting more data.
A simple implementation of such a driver could simply poll the condition flags within the USART peripheral until the device becomes ready to continue. This style of driver is called *blocking,* because it stops the progression of the application thread until the transmission has completed. Depending on the timing of the main application, this may not be a problem. However, a blocking driver cannot be used in an interrupt because of the delays it causes.
*Non-blocking* drivers use interrupts and buffers to store and move data into the peripheral. A non-blocking driver executes rapidly because the data to transmit is only moved into a storage buffer. When there is data within the storage buffer, an interrupt is enabled which triggers whenever the transmit register within the USART becomes available.
The exercises in this lab will mostly use blocking-style drivers. Although non-blocking drivers for USARTs are simpler than those for most other interfaces, designing one is outside the time scope of a single lab.

#### Text Formatting

This lab requires a few functions that perform basic string operations. To implement these, you will need to how the c-language represents and terminates character strings.

The c-standard libraries, which provide conventional string operations, are not available for the STM32F0. These expect a more traditional environment with a freely available heap and plenty of stack and code space. However, there are a few versions of the standard libraries intended for embedded system use. Kiel provides a miniature version of the c-standard library called *microlib* within the ARM:MDK toolchain.
If configured properly it becomes possible to use standard printing functions such as `printf()` with the USART. This may be discussed in later labs.

## 4.6 Using a USB-UART Cable and the Terminal

### 4.6.1 The USB-UART Cable

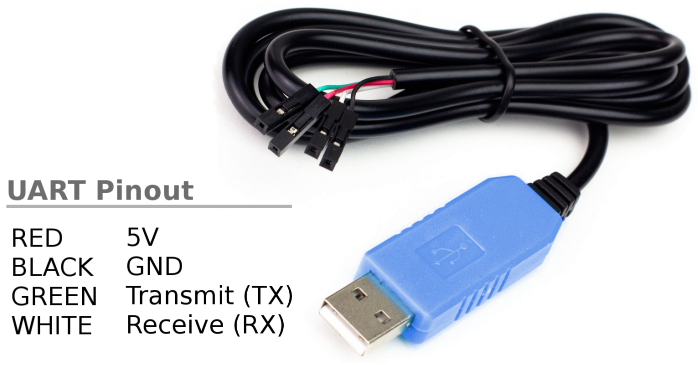
Because it is unlikely that consumer computers have native TTL Serial interfaces, many chip vendors sell devices that act as a protocol bridge between USB and TTL Serial. These are built into cables or small circuit boards and often called USB-Serial or USB-UART converters.
Although these devices are almost universal, this lab suggests using the Adafruit 954 USB-UART cable. This cable directly connects to the USB port of a desktop computer and outputs 3.3V TTL Serial on a 4-pin wire connector.
Figure 4.7 shows an image of the Adafruit 954 cable along with a pinout of the wire headers. To connect the USB-UART cable to the discovery board, first, attach the black GND wire on the cable to one of the matching pins on the board.
Both the USB-UART cable and the USART on the STM32F0 have receive (RX) and transmit (TX) lines. To communicate, the transmitter of one device must be connected to the receiver of the other. If necessary, refer to the following when connecting the cable to the board.

**USB-UART Transmit (TX) →STM32F0 Receive (RX)**
**USB-UART Receive (RX) →STM32F0 Transmit (TX)**

When connecting the USB-UART cable to the Discovery board, do **not** attach the 5V output while debugging.
This is because the ST-Link already provides 5V power and unless the two sources have equal voltages, current will flow through the supplies, possibly causing damage.
If both cables are sourced from the same computer, then it is unlikely that there will be issues. However, the extra connection is unnecessary.

### 4.6.2 Finding Installed Ports on Windows

When connected to a desktop or laptop computer, USB-UART cables usually appear as virtual RS-232 serial ports to the operating system. Although most adapters work with all common operating systems, because the Kiel toolchain operates on Windows, this lab does not document the steps to find the virtual port on others.
Whenever new hardware is installed on Windows, the operating system assigns a device name indicating its type and a unique identifier. Serial ports (virtual or physical) are assigned names beginning with “COM” and ending with a number. For example, the third serial port installed on a Windows system would be named “COM3.”
Unfortunately, it is not always easy to determine what name the computer has assigned to the connected cable. Windows creates a new device ID for each unique cable that is connected and has an odd habit of occasionally creating new identifiers for known ones.



**UART Pinout**

| | |
|---|---|
| RED | 5V |
| BLACK | GND |
| GREEN | Transmit (TX) |
| WHITE | Receive (RX) |

Figure 4.7: Pinout of the Adafruit 954 USB-UART cable.

The most reliable method of determining what name has been given to a connected USB-UART cable is to check the Windows, device manager. Figure 4.8 shows the device manager with a connected cable and the assigned name circled.
The following steps can be used to open the device manager.

1. **Open the Windows Control Panel** – The control panel can be accessed from the start menu or all applications view of the home screen.
   - If using Windows 8/10 you will need to access the original control panel not the simplified settings app.
2. **Select the Appropriate Category**
   - Windows 8/10 – Select *Devices and Printers* Category
   - Windows 7 – Select *System* Category
3. **Select the Device Manager** – The device manager should be located within the selected category.

Once opening the device manager, expand the *Ports (COM & LPT)* category. If the category is not visible then Windows has not recognized the device, the drivers may not have automatically installed. In this event visit the product page on Adafruit.com and manually install the cable driver.

### 4.6.3 Using the Putty Terminal Program

Putty is a multi-protocol terminal primarily used for accessing remote secure-shell (SSH) connections on Unix/Linux based operating systems. However, it among with many other terminal applications can be used for serial communication.
After launching Putty, the main settings window should open as shown in figure 4.10. There are three steps to configure Putty into the proper mode to communicate with the Discovery board.

Figure 4.9: Putty terminal icon

1. Select the serial mode using the *Connection Type* option.
2. Type the port name for the USB-UART cable in the *Serial Line* box.
3. Select a baud rate to communicate at in the *Speed* box.

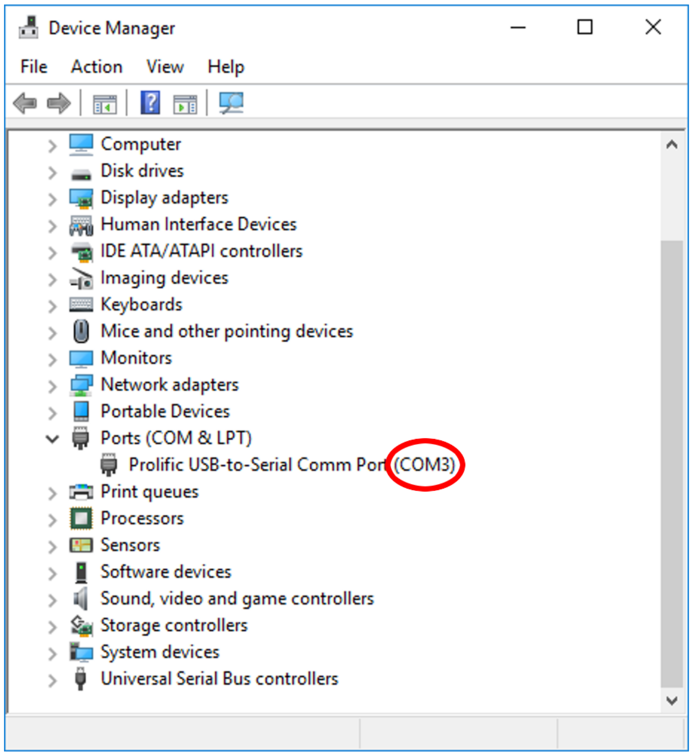
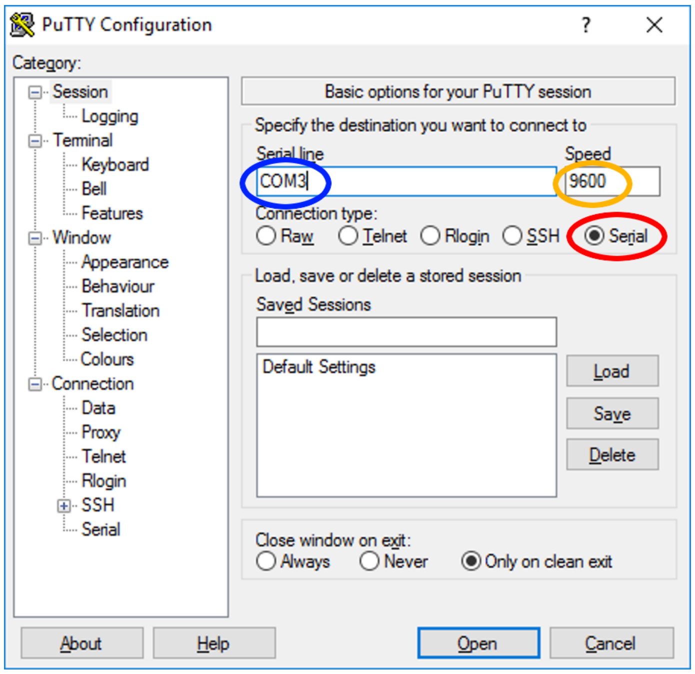


Figure 4.8: Windows device manager with the ports tab expanded.

After configuring Putty, click the *open* button. If an error appears, it is likely you have selected a serial name that doesn't currently exist. In this event open the device manager and check for the connected cable's ID.
If there aren't any errors, a blank terminal window should open. If you type text in the terminal, you should not see any text appearing. This is because the default behavior of the terminal is only to display data that is received from the remove device. (Discovery Board)
Connecting the transmit and receive lines of the USB-UART cable together using a jumper wire forms a loopback interface. A loopback interface sends data on the transmit and receives it immediately back because the input and outputs are directly connected. In this event, you will see the typed text in the terminal.

## 4.7 Lab Assignment

Within this lab's exercises you will be connecting the Discovery board to a PC through a USB-UART cable. The first few are designed to familiarize you with the basic concepts of transmitting and receiving data using blocking methods. The final exercise implements a simple character-based command parser which you can use to modify the LEDs on the board.



Specify the destination you want to connect to

Serial line: COM3

Speed: 9600

Connection type: Raw Telnet Rlogin SSH [✓ Serial]

Close window on exit: Always Never [✓ Only on clean exit]

Figure 4.10: Putty terminal settings screen

### 4.7.1 Preparing to use the USART

Unlike the previous labs these exercises involve multiple devices. This makes debugging slightly more complicated because there are now possible connection and software errors due to the cable and PC. Before writing the embedded application, you may want to perform a loopback test by connecting the transmit and receive lines on the adapter cable to ensure that the serial terminal is communicating with the correct virtual port.

The STM32F072 has four USART peripherals available; you will need to select one of these to use. Each USART has a small selection of GPIO pins that can be used as its transmit (TX) and receive (RX) signals. Because the split connections on the end of the Adafruit USB-USART cable are fairly short, you will want to choose USART output/input pins that are relatively near a GND pin.

1. Using the chip datasheet, locate pins that connect to TX/RX signals on USART peripherals.
2. Choose a set of RX/TX pins that are near enough to a GND connection such that the USB-USART cable's wire ends can reach.
3. Connect the USB-UART transmit (TX) line to the STM32F0's receive (RX) pin. Likewise, connect the STM32F0's transmit (TX) pin to the USB-UART receive (RX) line.
4. Set the selected pins into alternate function mode and program the correct alternate function number into the GPIO AFR registers.
5. Note the USART that the selected pins connect to, this is the peripheral you will be using in the lab exercises.

Ensure that both the Discovery board and USB-UART cable areunpowered when connecting them to each other. If one device is connected to the PC and the other is not, the transmit line on the powered device will feed voltage into the unpowered device through its input pin.

### 4.7.2 Blocking Transmission

This exercise transmits single characters to the serial terminal on the PC. The character transmission will be handled in the main loop of the application using a blocking method.

#### Initializing the USART

The lab manual mentions all of the registers that you may be required to modify throughout these exercises. Because TTL Serial is the default mode of the USART, many of the configuration bits will already be at their desired state.

1. Enable the system clock to the desired USART in the RCC peripheral.
2. Set the Baud rate for communication to be 115200 bits/second.
   - You may use the `HAL_RCC_GetHCLKFreq()` function to get the system clock frequency.
3. The USART starts with portions of the peripheral disabled for low-power use. You will need to enable the transmitter and receiver hardware.
4. The USART has a peripheral enable/disable bit in its control register. Once the USART is enabled, many of the configuration bits become read-only.

#### Transmitting a Character

Write a function that transmits a single character on the USART. Start with a function declaration that accepts a single character-type variable and returns nothing. Within this function implement the following.

1. Check and wait on the USART status flag that indicates the transmit register is empty.
   - You can use an empty `while` loop which exits once the flag is set.
   - Don't use the USART "BUSY" status bit. Although this will appear to work properly, this bit depends on multiple conditions and will slow down your transmission.
2. Write the character into the transmit data register.
   - Remember that c-language characters are really numerical values. (ASCII) You can modify and write them to registers without special syntax or type casting.
3. There is no need to manually clear the status bit, it will be automatically modified by the peripheral when you write into the transmit register.

Implement some code in the infinite `while` loop of the main function that calls the character transmit function with a character constant. Feel free to simply loop with delay, or use the button to trigger the transmit. Remember that C-language character constants are surrounded by single quote marks.

#### Transmitting a String

Unlike many other languages, C has no built-in string data type. Instead, text strings are built from arrays of characters. Because C arrays don't have any method of determining how long they are, string constants that are converted by the compiler into arrays are terminated with a null character. The null character '\0' has the literal ASCII value of 0 and is used as a sentinel (guard) value to let string processing functions know they've reached the end of the array.
To transmit strings, begin with a function declaration that accepts an array of characters, either in direct array form or as a pointer. This function should loop over each character in the array and call your character transmit function.

1. Loop over each element in the character array
2. If the current element is not the null character use your character transmit function.
   - You can increment over the array by using a counter and array index or by incrementing the pointer.
   - You can test for the null character by comparing the value against '\0' or the numerical value 0. (remember characters are numbers in C)
   - If you use the pointer method, remember to dereference when testing for the null value and when calling the character transmit function.
3. Return when the null character is encountered.

Once your string transmit function is complete, change your main application to transmit a short phrase instead of a single character. If you see large amounts of garbage printed in the terminal instead of your phrase, you probably have run off the end of your character array and are displaying the contents of the rest of the STM32F0's memory.

### 4.7.3 Using a Protocol Analyzer

The true benefit of using logic analyzers is when debugging communications between multiple devices. Connect one of the inputs of a logic analyzer to the transmit pin of the USART. You can do this by using the other side of the pin as the USB-UART cable on the Discovery board.
Once connected, set the trigger for falling-edge and capture a section of the transmission. Using the online user guide linked in the lab Canvas page, apply the *Async Serial* protocol analyzer to the captured trace.
When configuring the protocol analyzer you should be able to leave all of the settings except the baud rate as default. Once the analyzer has been applied, zoom onto the captured trace until you can see the decoded values above the signal transitions. **Save a screenshot of the decoded signal for your postlab.**
If the analyzer indicates multiple framing errors, you may have the incorrect baud selected. Click the small gear button on the analyzer tab and change the settings to use Autobaud. Hopefully this will detect the baud rate of your signal and decode properly.

### 4.7.4 Blocking Reception

For this exercise you will use the infinite `while` loop in the main function to read single-character commands typed at the serial terminal on the PC. Your goal is to develop an application that toggles the correct LED whenever the character matching the first letter of the color is pressed. For example, typing an 'r' into the terminal would toggle the red LED.

1. Check and wait on the USART status flag that indicates the receive (read) register is not empty.
   - You can use an empty `while` loop which exits once the flag is set or simply check each iteration of the main infinite loop.
   - It may be helpful to carefully read the bit descriptions in the register map.
2. Test the received data and toggle the appropriate LED
   - The receive register can be read like an ordinary variable. However, the data isn't guaranteed to remain in the register after it has been read once.

    - Unless you use a switch statement you may want to save the value into a local variable and test against that.
3. Whenever a key is pressed that doesn't match an LED color, print an error message to the console.
4. You will probably want to comment or remove any old transmit code and delay statements from the infinite loop.
    - The possible reception rate at 115200 Baud is faster than the minimum delay the HAL library functions are designed to provide.
    - It is possible to lose received data while waiting for a blocking transmit to complete. Where we are only receiving single bytes it is unlikely to cause problems.

### 4.7.5 Interrupt-Based Reception

In this final exercise you will use an interrupt to save the received data when it arrives. Additionally you will expand your simple command parser with more complex behavior.

1. In the USART initialization, enable the receive register not empty interrupt.
2. Enable and set the USART interrupt priority in the NVIC.
3. Setup the blank interrupt handler.
4. Within the handler, save the receive register's value into a global variable.
5. Within the handler set a global variable as a flag indicating new data.

Examine the status flag documentation for the receive register not empty interrupt. You should notice that the USART peripheral automatically clears the flag whenever the receive data register is read. Because of this there is no need to manually clear the bit within the interrupt handler.

In this exercise we are saving the contents of the receive register to a global variable. Technically since we are only operating on a single byte, using the interrupt doesn't offer us much of a benefit. This is because saving the data to a simple variable doesn't protect us from losing data if the main loop takes too long and the USART overwrites it with a new value. In better non-blocking implementations a buffer is used; possibly raising the new data flag only after a certain character such as a newline is received.

Now that the interrupt is ready, you will need to rewrite your main application to check and act on the flag and data variables you set in the interrupt handler. This portion is an open-ended exercise, you are free to implement it in any method you choose as long as the following requirements are fulfilled.

1. Your command parser must now accept two character commands.
    - The first character is a letter matching the one of the LED colors.
    - The second character is a number between 0 and 2.
        - '0' turns off the LED
        - '1' turns on the LED
        - '2' toggles the LED
2. Print a command prompt such as "CMD?" when waiting for user input.
3. Entering an unknown character prints an error message and restarts back to the beginning.
4. On a successful command, print a message about which command was recognized.