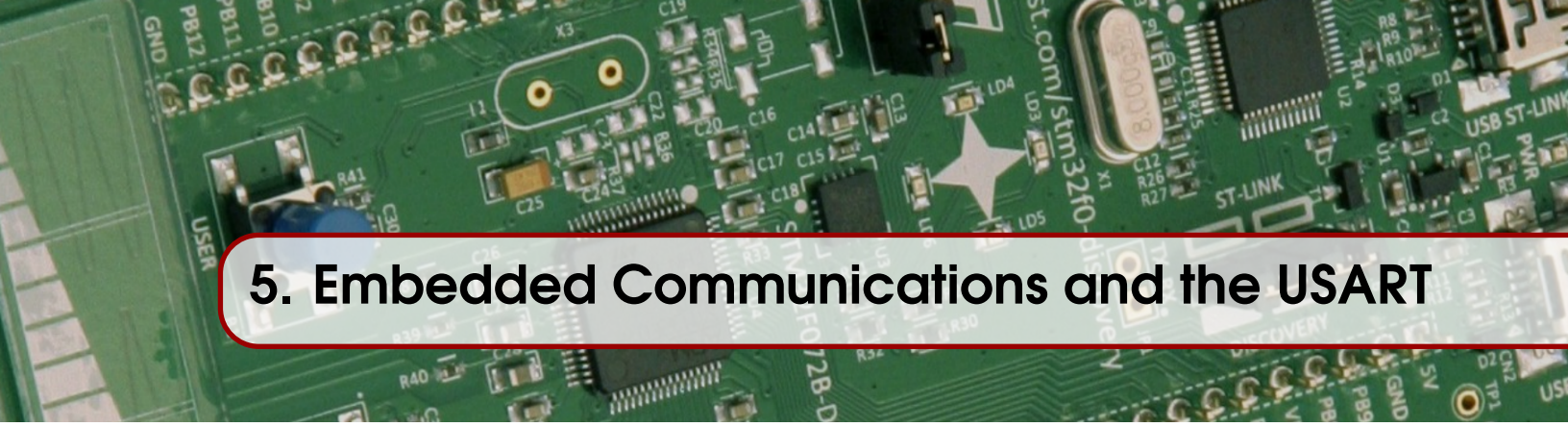

5. Embedded Communications and the USART

5.1 Communication Between Systems

All computing systems regardless of their size and power, are essentially useless if they don't have some way to communicate with the outside world. Even a 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.

5.1.1 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 5.1.

Parallel

Parallel interfaces transmit entire blocks of data using multiple wires, with each wire representing the state 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. In order 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.

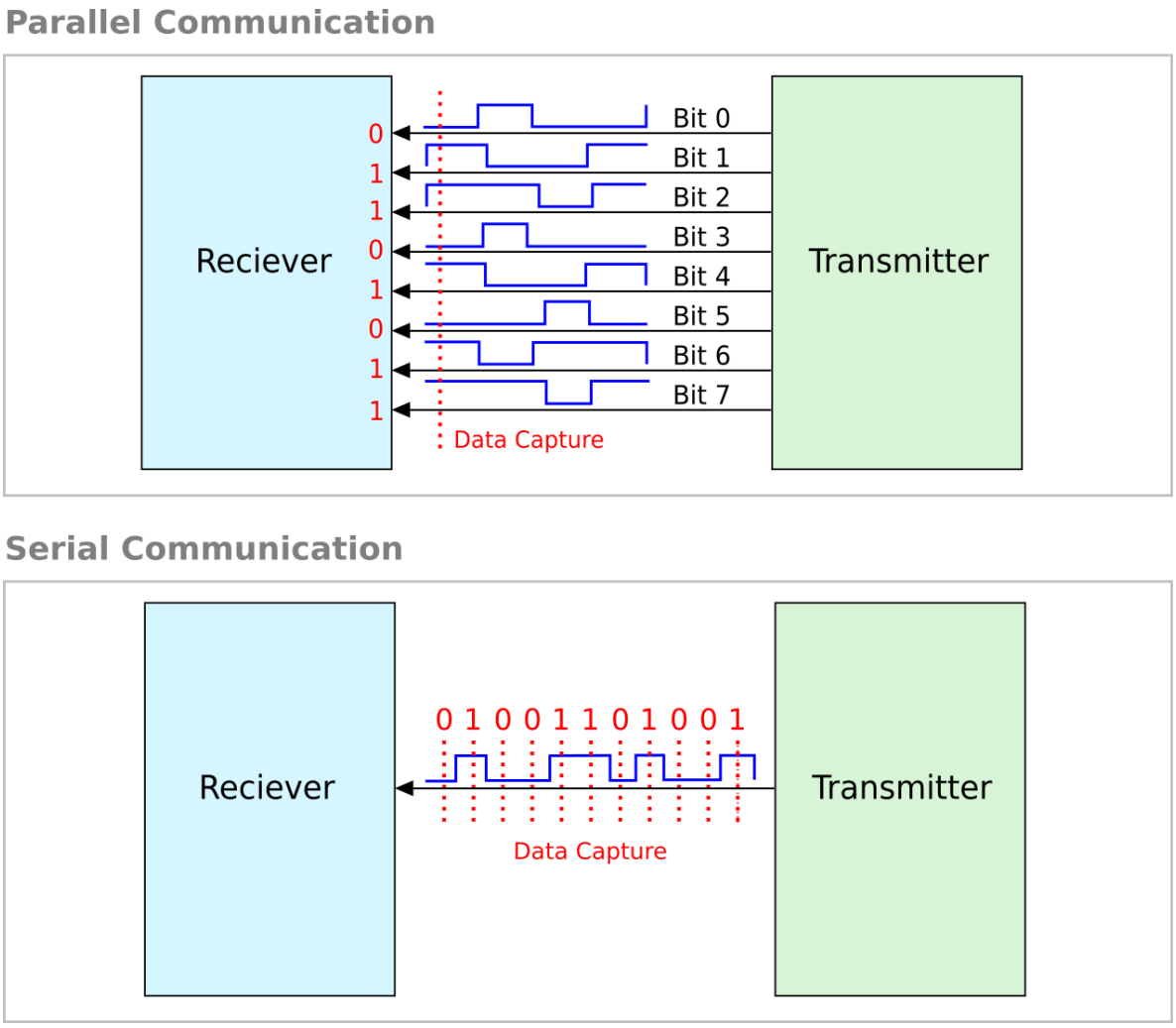

Figure 5.1: Comparison of parallel and serial communication.

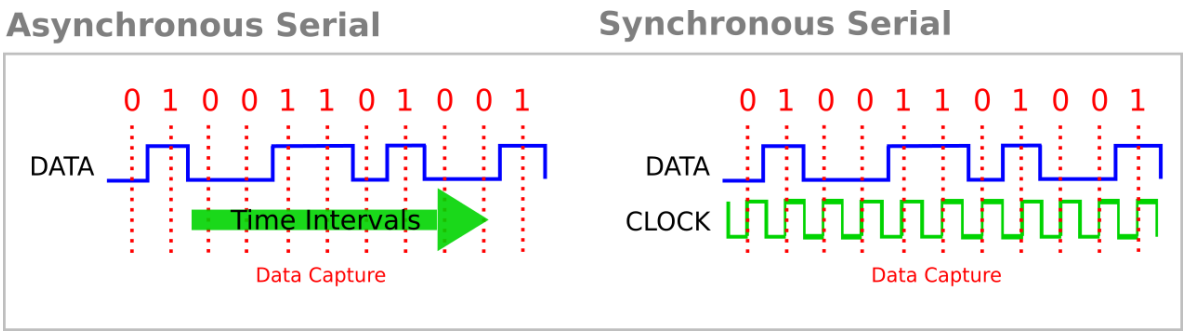

Figure 5.2: Comparison of synchronous and asynchronous serial communication.

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.

However, despite that equivalent-speed parallel connections have a higher bandwidth, most high-speed device interfaces such as SATA, USB and Ethernet are actually serial interconnects. The reason for this is because parallel connections are far more difficult 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 in 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 constant 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.

5.1.2 Synchronous vs Asynchronous

Regardless of whether an interface is serial or parallel, there must be 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 5.2.

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.

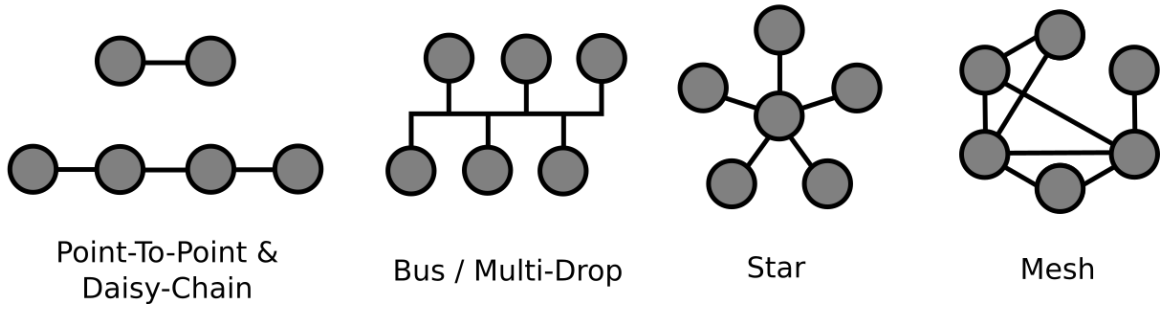

Figure 5.3: Simple network topologies.

5.1.3 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 together. Figure 5.3 demonstrates a few simple topologies.

- **Daisy-Chain** – Nodes in a daisy-chained network connect only to their adjacent neighbors. Some interfaces 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.

5.2 Communication Standards & Protocols

Aside from knowing whether an interface is parallel or serial, asynchronous or synchronous, communicating devices must know how to interpret the patterns of bits that make up the data. To manage this, all interfaces are combinations of hardware/physical standards and communication protocols.

5.2.1 Hardware Standards

A hardware or physical standard defines the physical characteristics of an interface. This involves things such as single-ended or differential signaling, multi-device topologies, optical, wireless frequency and modulation etc... Whether an interface is parallel and serial is something a hardware standard determines.

5.2.2 Communication Protocols

Protocols define the meaning of a communication signal.

Hardware Protocols

Low level or hardware protocols define how bits are extracted from a signal. This primarily involves how signals are sampled and timed. Whether an interface has an explicit clock signal or is asynchronous. Depending on the complexity, hardware protocols may also include higher-level features such as error-correction, start/stop signals, message acknowledgment, addressing, data packets and more.

Software Protocols

Application or software protocols interpret raw data. These are written by the end-user and define how the multiple systems on the communication interface interact.

5.2.3 Separating Interface and Protocol

important thing to note is that while the terms protocol and standard are sometimes used interchangeably there are differences between the two. A protocol defines the bit patterns, data interpretation and control flow of an interface. A standard typically includes a protocol, but also specifies physical characteristics of the system such as voltage, current, detection method (single-ended, differential) and possibly even the connectors used.
Some interfaces such as Ethernet have very clearly defined separations between the different parts and layers within them. Computer networks (which typically use Ethernet) in particular have the OSI model.
Many low-level interfaces such as those used in many embedded systems are much more fuzzy where the boundaries are.

5.3 Common Embedded Interfaces

In general the interfaces that we'll be using are hardware protocols. You will have to define what the raw data means using a software protocol.
Three of the most common (and simplest) interfaces used in embedded systems are: TTL RS-232 (UART) Discussed in detail in this lab SPI - Serial Peripheral Interface (Include very brief intro on basic details?) I2C - Inter-Integrated Circuit (Include very brief intro on basic details?) Others include: 1-wire, CAN, USB, Various wireless; Bluetooth, Zigbee, LoRa, WiFi etc...

5.4 Introducing RS-232

5.4.1 Conventional RS-232

Historical use as "serial" Origin, uses, old connector, voltage levels

5.4.2 Embedded TTL RS-232 (TTL-Serial)

Embedded system use differences to conventional RS-232 TTL RS-232 Standard Details BAUD rates and Oversampling Start and Stop Bits Parity USB-UART Cables Purpose and use, specifically mentioned in later section

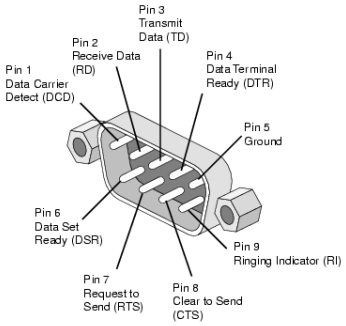

Figure 5.4: Conventional DB9 serial connector and pinout.

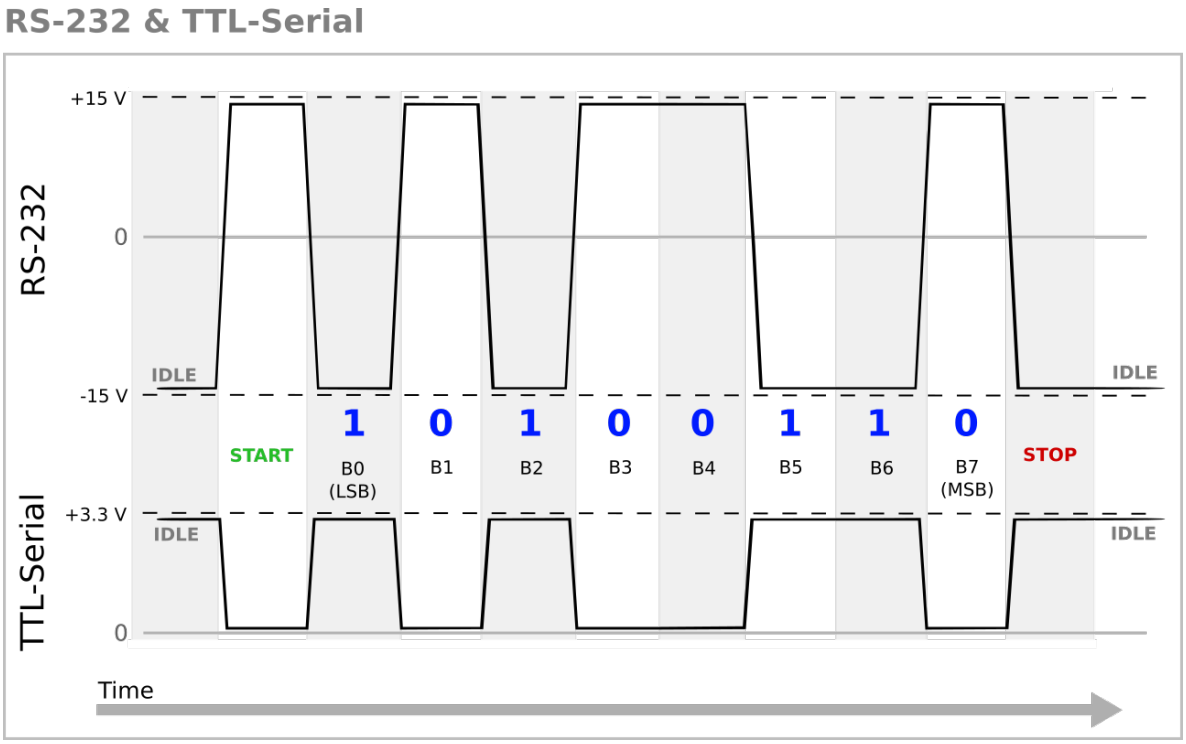

Figure 5.5: RS-232 and TTL-Serial logic levels and polarity.

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 5.6: Subset of the ASCII text encoding standard.

5.4.3 ASCII Text Encoding

Old standard for text exchange based on 8-bit characters Has 128 characters in 7-bits, the last bit is used for parity Some characters have system meanings such as a newline char others are non-printing Some system chars no longer have much meaning in modern systems Was extended to use all 8-bits The original ASCII has since been expanded into the Unicode character standard Many different encodings for unicode such as UTF8 Will be using base ASCII for embedded communication

5.5 Introducing the USART

What the USART is and why it has its name Typically used for TTL RS-232 Modes of operation mainly to mention that it has other modes, but we won't discuss them

5.5.1 USART Registers

Overview of all registers used for RS-232 Give short description and what each register does (what settings it manages) Mention other registers and indicate that they won't need them

5.5.2 Configuring the BAUD Rate

Configuring the BAUD rate registers Basic equations of calculating baud rate How to load into the mantissa and fractional baud registers

5.5.3 Blocking vs Non-blocking Operation

mainly mention that you can use buffers and interrupts to make code that doesn't wait around until transmission is complete. mention why it's bad to wait sometimes won't be doing non-blocking drivers in this lab (later lab perhaps?)

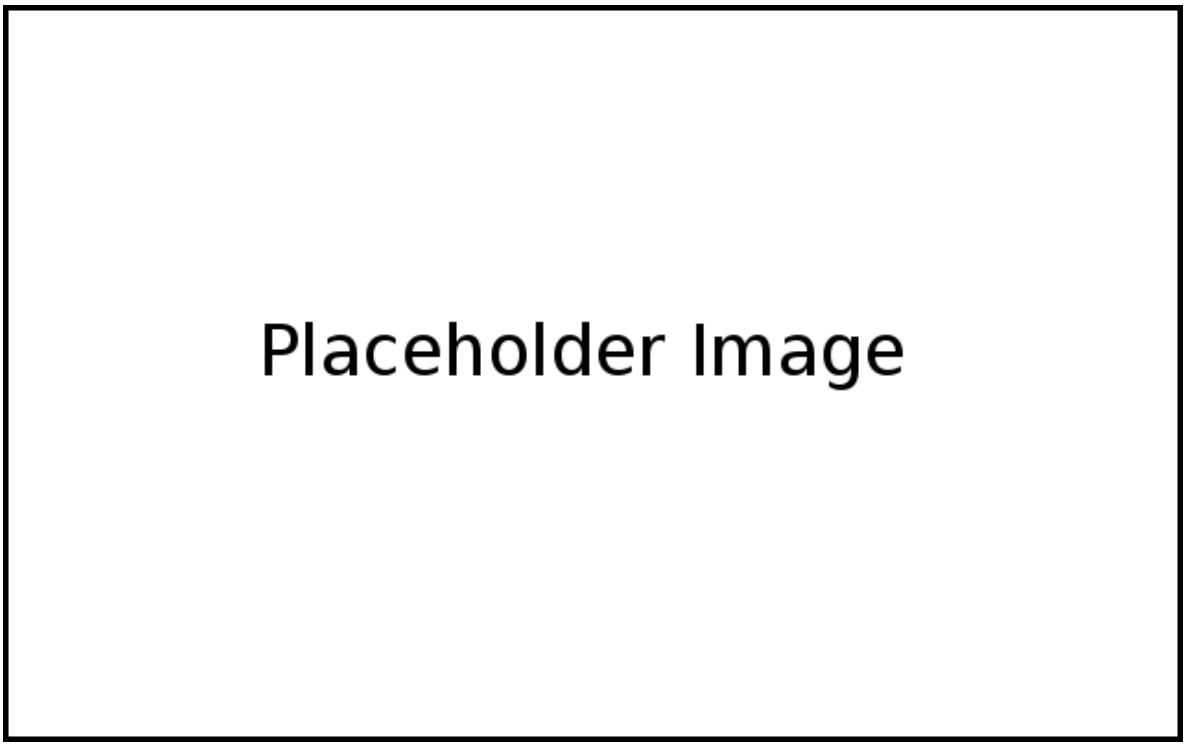

Figure 5.7: Blocking and non-blocking drivers.

5.5.4 Performing Basic Transmits and Receives

Give flowcharts for blocking receive and transmit Explicitly say when they need to wait/check a flag for this condition before moving onward. Don't give actual bit names, but they should be able to use the register descriptions to know where to look Give pretty obvious hints as to what they should look for when searching for bits.

Text Formatting

No printf yet... need to build own string transmit, string compare and eventually numeric value to ASCII character functions Kiel provides a miniature version of the c standard library called microlib Can configure and use printf with our USART, but won't in this lab. (later labs?)

5.6 Using a USB-UART Cable and the Terminal

5.6.1 The USB-UART Cable

Using the USB-USART cable and Putty Terminal program Using and connecting the cable

5.6.2 Finding Installed Ports on Windows

Where to find on windows windows calls "COM#" located in device manager need steps to find on lab machines.

5.6.3 Using the Putty Terminal Program

How to set up Putty

Placeholder Image

Figure 5.8: Flowchart for simple RX/TX communications on the USART.

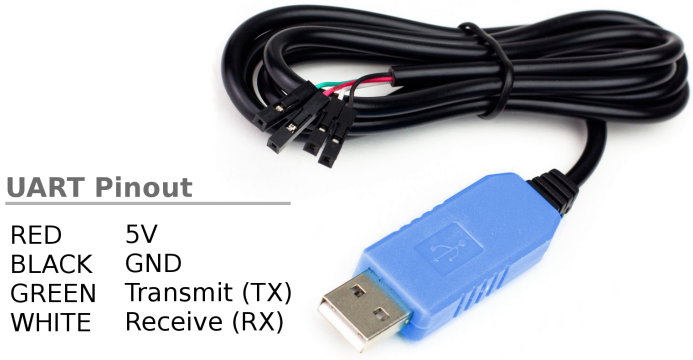

Figure 5.9: Pinout of the Adafruit 954 USB-UART cable.

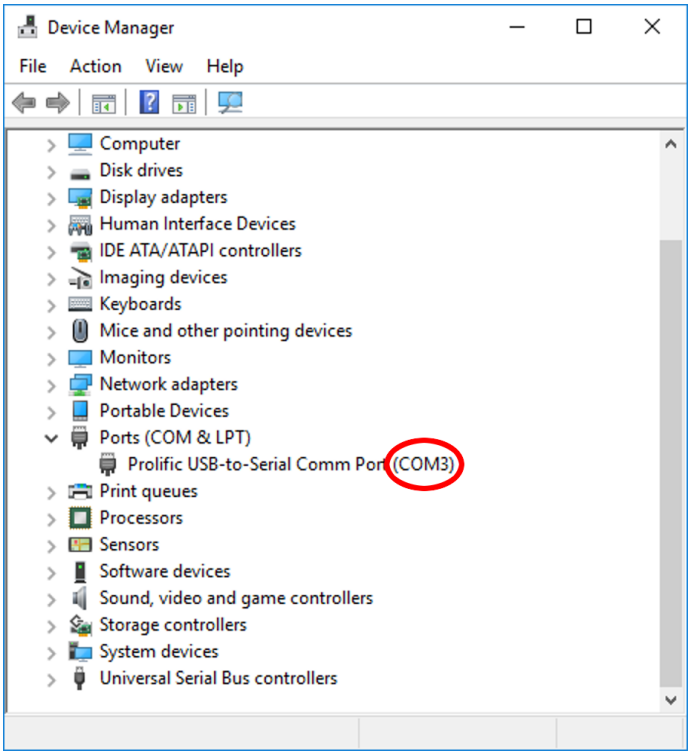

Figure 5.10: Windows device manager with the ports tab expanded.

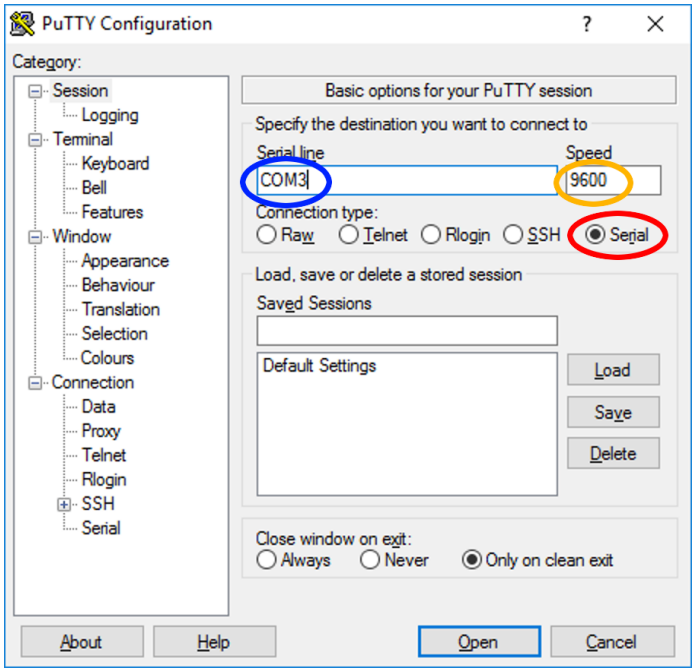

Figure 5.11: Putty serial terminal

mention typing in terminal, no response how to test cable/terminal with loopback (wire between RX/TX)

5.7 Lab Assignment

5.7.1 Preparing to use the USART

Set up pins and connect USB-USART cable Have them test terminal/cable with loopback Provided code (HAL library?) to enable and test USART, should see string printed in terminal

5.7.2 Blocking Transmission

Set up and enable USART to baud rate using registers Transmit single char in loop to terminal Write simple string transmit function and send strings to terminal mention C-strings, but tell them to look elsewhere for help with those Capture and decode the string using the logic analyzer include screenshot in lab report

5.7.3 Blocking Reception

Receive single chars in loop from terminal toggle LEDs on char match (r for red etc...)

5.7.4 Interrupt-Based Reception and Command Parsing

Set up USART RX interrupt set up static buffer (with basic over/underflow protection) buffer word until newline and set flag Use simple state machine and string match (they can write) in main program to recognize simple command strings (example: "red
n on
n") Provide text feedback "ready for command", "buffer overflow", "unknown command", "invalid command", "command accepted"

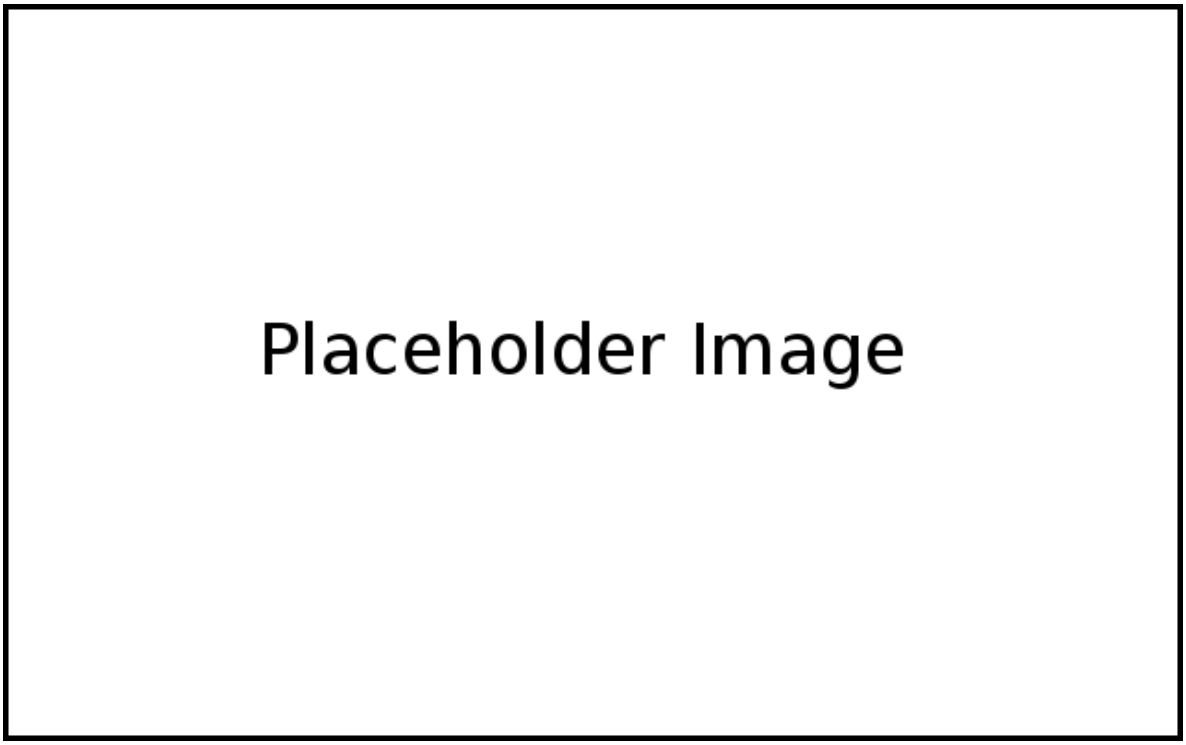

Figure 5.12: Flowchart for command parsing.