

Computer Communications

CSE 1302

Today's Outline

- Communicating between PC and Arduino
 - Streams in Java
 - SerialComm and observability
- Representing text
 - ASCII, control codes, Unicode
- Moving bytes in both directions
 - Marking the end of a message
- Images

Computer Communications

- Link that provides byte-level data delivery
 - Network
 - Serial port
- Ability to send and receive on each endpoint
- Must use a protocol to understand anything other than individual bytes
 - Individual data elements (ints, chars, strings, etc.)
 - Higher-level, application-specific messages
 - The user just pressed button “X”
 - The pressure in vessel X is Y psi at time Z
- Needs to work across platforms
 - E.g., Java on PC and C on Arduino

Java Communications uses Streams

- Upstream writer, downstream reader



- Source writes to stream
- Destination reads from stream
- Either endpoint might be a file or some other input/output device, e.g.,
 - Dest. could be Arduino connected via serial port
 - Source could be the keyboard

Stream Conventions

- FIFO ordering (First-In-First-Out)
- Protocol must be same at both ends of stream for effective communication to take place
 - Stream of bytes? chars? integers? what is a char?
- Streams can be "wrapped" by other streams that add properties, e.g.,

```
InputStreamReader isr = new InputStreamReader(System.in);  
BufferedReader in = new BufferedReader(isr);
```

 - System.in gives you bytes; the wrappers give you chars, then whole lines

Wrapping Streams

- A stream can take another stream as a parameter to its constructor
- The outer stream adds functionality to the wrapped stream
- This is called the "decorator" pattern
- SerialComm decorates SerialPort the same way – but you hand it a port name and it opens the port for you:

```
SerialComm sc = new SerialComm("/dev/cu.usbmodem1101");  
sc.setDebug(true);
```

- We will use SerialComm and SerialPort in modules 4-6

Reading the Console in Java

- System.out is a stream going out; System.in is a stream coming in
- System.in hands you raw bytes – wrap it to get something useful

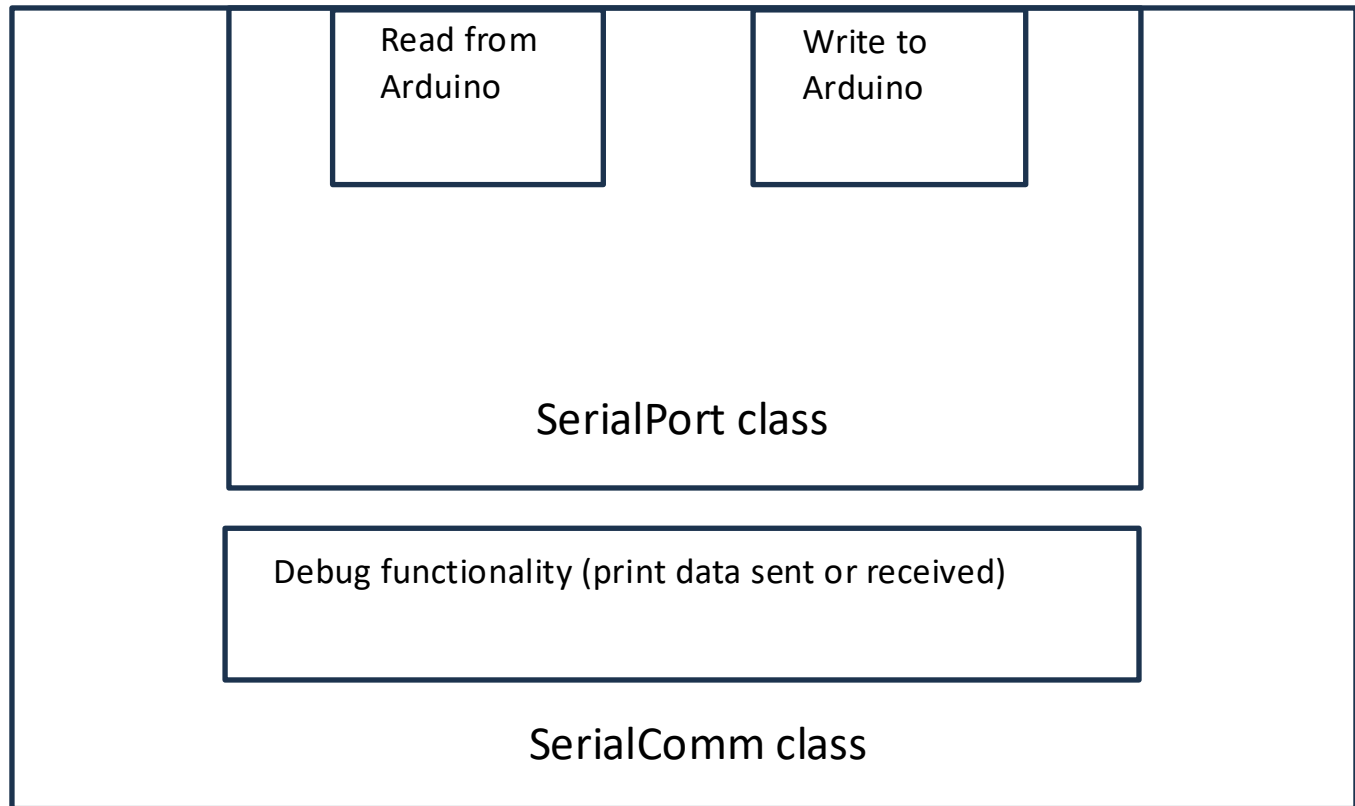
```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(System.in));  
String line = in.readLine();
```

 - InputStreamReader: bytes → characters
 - BufferedReader: characters → whole lines
- Same decorator idea as SerialComm around SerialPort
- readLine() gives you a String – deciding which character to use is your job

Communications in Java

- Open COM port with SerialPort object
 - Use SerialPort class, which we provide
 - Works in Windows and Mac
- Wrap SerialPort with a SerialComm object
 - You will write SerialComm class (some of it, anyway)
 - What “properties” does SerialComm provide?
 - Fixes up a data type inconvenience on input bytes
 - Most importantly, provides a debugging capability

SerialComm class Functionality



What debug Output Looks Like

- debug is the whole point of SerialComm: it prints every byte that crosses the link
- `writeByte()` prints outgoing bytes in angle brackets
`<0x5f>`
- `readByte()` prints incoming bytes in square brackets
`[0x5f]`
- Two hex digits every time – `String.format("%02x", b)`, not `"%x"`
- One console, two directions: the brackets let you read the conversation
- Sending "Hi!" with debug on prints
`<0x48><0x69><0x21>`

Observability

- What is **really** going on?
- Option 1: stare at the code until inspired
 - When that doesn't work, make random change
- Option 2: don't assume the code you actually wrote does what you think it does!
 - Alter code so that you discover what it really does
 - On PC in Java, use the debugger!
 - Or use `System.out.print()` to display on console
 - On Arduino in C, use `Serial.print()`

Observability in Communications

- Need to know what is **really** going across the communication link
- On sender and receiver:
 - Display what is going out the output stream
 - Display what is coming in the input stream
 - Show the raw data (sequence of bytes)
- You will build these tools
 - This is the primary purpose of SerialComm class

Back to Communications

- Streams are sequences of bytes
- We need data at a higher level of abstraction
 - Integers
 - Floats, Doubles
 - Characters
 - Strings
 - More
- Protocols must be designed to enable this
 - Build bigger things out of streams of bytes

Individual Data Elements

- Byte – basic network element
 - `writeByte()`, `readByte()` in `SerialComm` class
 - `Serial.read()`, `Serial.write()` in Arduino C
- Character
 - Two bytes in Java
 - One byte in C
- Integer
 - Four bytes in Java
 - Two bytes on the Arduino Uno (a long is four)
- The two ends must agree on sizes before any of this means anything

Text – Characters and Strings

- ASCII – American Standard Code for Information Interchange
 - 7-bit codes representing basic Latin characters and numbers [A-Z, a-z, 0-9], some common punctuation, and control characters
 - There are a number of extensions to 8 bits, but only the 7-bit codes really standard.
- Unicode – 8- or 16-bit codes extending to a much wider set of languages
 - The first 128 codes are equivalent to the 7-bit ASCII standard

C Strings

- Strings are sequences of ASCII characters, stored one byte per character (8 bits), terminated by a NULL (zero) character
- E.g., “Hello!”

01001000	‘H’	0x48
01100101	‘e’	0x65
01101100	‘l’	0x6c
01101100	‘l’	0x6c
01101111	‘o’	0x6f
00100001	‘!’	0x21
00000000	NULL	0x00

ASCII Facts

- Numerical digits are assigned in order of increasing value
i.e., '0' = 0x30
 '1' = 0x31
 '2' = 0x32

 '9' = 0x39
- For single character, value conversion is simply a difference of 0x30

More ASCII Facts

- Letters are also assigned in lexicographical order:
 'A' = 0x41
 'B' = 0x42

 'Z' = 0x5a

 'a' = 0x61
 'b' = 0x62

 'z' = 0x7a
- Upper/lower case conversion is simply a difference of 0x20

Still More ASCII Facts

- First 32 characters (0-0x1f) are control codes:
 - 0x00 ^@ null (C string terminator)
 - 0x07 ^G bell
 - 0x0a ^J line feed
 - 0x0c ^L form feed
 - 0x0d ^M carriage return

Line breaks are not standardized

- End of line conventions differ by operating system:
 - In MS Windows: 0x0a, 0x0d is end of line
 - In Unix/Linux: 0x0a is end of line
 - 0x0a, linefeed, is sometimes called 'newline'
- In C, '\n' is mapped to OS end of line termination convention

Java Strings

- Strings are represented via the class “String”
- String objects are immutable
- The character encoding is system specific, e.g., either UTF-8 or UTF-16 (typical).
- The length is an instance variable in the object (in most implementations)
- The characters are stored in a `char[]` array (again, in most implementations)

Unicode

- Standard for character representation
 - Supports wide variety of languages, symbols
- UTF-8
 - Variable length code with 8-bit code units
 - U+0000 to U+007F are the same as ASCII
- UTF-16
 - Uses 16-bit code units, also variable length
 - Latin + Greek + Cyrillic + Coptic + Armenian + Hebrew + Arabic + Syrian + Tāna + N’Ko fit in 16 bits
- UTF-32
 - Uses 32-bit code units, fixed length

Using ASCII in Module 4

- Studio 4: one hex digit typed by the user → four LEDs
 - '0'–'9' are contiguous, so the value is $c - 0x30$
 - 'a'–'f' are contiguous, so the value is $c - 0x61 + 10$
 - `asciiToDecimalValue()` is provided – read it and see why it works
- Assignment 4: `toUpper()` on the Arduino
 - Case is a single bit: `'a' == 'A' | 0x20`, so clear 0x20 to go up
 - Only do it to letters – punctuation and digits must pass through unchanged
- Nothing "is" a character. A byte means 'A' because both ends agreed it does

Note: the Text Representation video says 'A' is 0x40 and 'a' is 0x60. It is wrong – they are 0x41 and 0x61.

Sending from Arduino

- Byte – basic network element
 - `Serial.write()` – send one byte out serial port
 - Only sends least significant bits of argument!
 - `Serial.write(0x1234)` will be received at PC as `0x34`
- Character – one byte
 - `Serial.write(char c)` works just fine
- `Serial.print()` is not `Serial.write()`
 - `print(50)` sends two characters, '5' and '0' (`0x35 0x30`)
 - `write(50)` sends one byte, `0x32`

Sending from Java

- Byte – basic network element
 - `s.writeByte()` – send byte through `SerialComm` object `s`
 - Takes Java byte type as an argument
- Character – two bytes
 - Only send out least significant byte (= ASCII)
 - `byte lowByte = (byte) (0x00ff & charValue);`
 - `s.writeByte(lowByte);`

Receiving

- First check to see if byte has arrived
 - Arduino
 - `Serial.available()` returns integer count of available bytes
 - Java
 - `s.available()` returns Boolean if a byte is available
- Next read one (and only one) byte
 - `Serial.read()` on Arduino (returns an int)
 - Return value is -1 if nothing received, byte value otherwise
 - `s.readByte()` on Java (returns a byte)
- Check `available()` prior to *each* read!

Where Does a Message End?

- A stream is bytes with no boundaries – the receiver never gets "a message"
- So the two ends must agree on where a message ends
- Assignment 4's answer: the newline character '\n' (0x0a)
 - Java: send every character of the line, then send '\n'
 - Arduino: on '\n', send the Morse back and finish with '\n'
 - Java: keep reading until '\n' comes back, then prompt again
- You may not use Thread.sleep() – nothing promises you a time, only the marker
- That agreement is a protocol, the simplest one there is
 - In Module 6 we replace the marker with a real header: magic number, key, value.

One Port, One Program

- Only one program can hold the serial port at a time
 - Serial Monitor or your Java program – never both, or `PortInUseException`
 - Stop the Java program (red square in the Console) before uploading a sketch
- The port name must match the one in the Arduino IDE
 - `/dev/cu.usbmodem...` on a Mac, COM3 or similar on Windows
 - Read it carefully: that is a zero, not an O
- Both ends must agree on the baud rate
 - `Serial.begin(9600)` on the Arduino, 9600 in the `SerialComm` constructor
- jSSC prints a warning about missing logs – harmless, ignore it

Quiz Time

- Go to Canvas and answer the single question on Quiz 4A
- A stream delivers bytes in a "first-in, first-out" order:
 - True
 - False

Images

- Consider the following bits:

0x002400081881423c

0000 0000 0010 0100 0000 0000 0000 1000

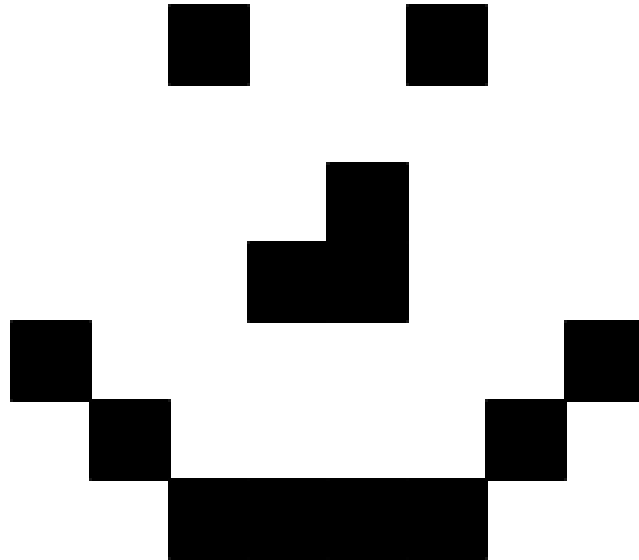
0001 1000 1000 0001 0100 0010 0011 1100

- Make 1 dark and 0 light:



Images

- Arrange in rows, one byte per row:



- Each bit is a “pixel” in the image

Bits per Pixel

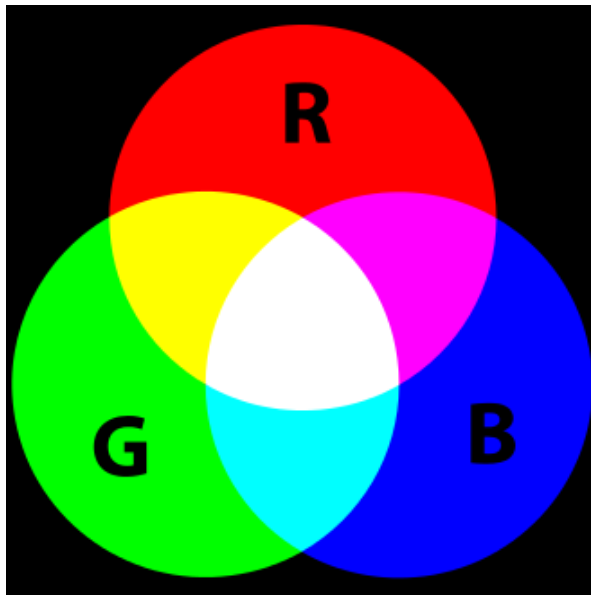
- One bit per pixel gives two colors – that is the face we just drew
- 8 bits per pixel → 256 levels of gray
- 24 bits per pixel → 8 bits each of red, green, blue
 - Written in hex as 0xRRGGBB, which is where web colors come from
 - 0xff0000 pure red, 0x00ff00 pure green, 0xffffffff white, 0x000000 black
 - 16.7 million colors, and your eye runs out before the encoding does
- The pattern never changes: agree on the encoding, and it is just bytes

Add color and more pixels



Color

- Additive color – primaries Red, Green, Blue



- Position close together and put diffuser above
 - This builds one pixel

How Many Bytes Is That?

- Count the bytes: $\text{pixels} \times \text{bits per pixel} \div 8$
 - 8×8 face at 1 bit per pixel = 8 bytes
 - 640×480 Earth at 24 bits per pixel = 921,600 bytes $\approx$ 0.9 MB
 - 1920×1080 photo at 24 bits per pixel = 6,220,800 bytes $\approx$ 6.2 MB
- The JPEG on your phone is a fraction of that – because it is compressed
 - Lossless (PNG): throw away nothing, exploit repetition
 - Lossy (JPEG): throw away what the eye will not miss
- And the file needs a header: width, height, bits per pixel

A receiver cannot decode the pixels without it – the same key-and-value idea as a message protocol.