MTRE 2610 Engineering Algorithms and Visualization – Dr. Kevin McFall

Laboratory – Serial communication between Python and Arduino

Introduction

The goal of this laboratory is to program two-way communication between Python on a PC and the Arduino Uno.

Running Python scripts from the command line

The python scripts described in this exercise are intended to be run from the command prompt rather than inside Idle, Spyder, or other IDEs. Such IDEs introduce overhead costs and reduce the speed of executed code. Additionally, the `msvcrt` module does not work completely inside these IDEs. Open a command window with cmd and type `python`. This opens a Python console similar to the IDEs where Python commands can be entered. The command `quit()` returns to the command line. In order to execute a script from the command line, call the `python` command but pass to it the filename of the Python script to run, for example `python filename.py`. This will result in an error unless `filename.py` is in the cmd current directory or in a system path directory. To solve this problem, use the `cd` (change directory) command to move the current directory to where the source file is. The directory may be excessively long, and as a short cut, navigate to the directory in a Windows Explorer window, right click on the folder directory at the top and "Copy address as text". In the command window, type `cd`, space, and right click to paste in the folder directory rather than typing it in.

Reading and writing serial data on an Arduino

It may be surprising, but serial communication from Arduino to PC was already performed in earlier laboratory exercises with the `Serial.println` command. The Arduino IDE is configured to read serial data sent by the Arduino in the serial monitor. In this laboratory exercise, a Python program will be written to capture this data instead, and even send data back to the Arduino.

Recall the `println` command requires a preceding `Serial.begin(9600)`, which opens serial communication at a rate of 9600 bits per second (otherwise known as baud). Both sending and receiving machines must be set to the same baud rate or sent data will be garbled (notice in the serial monitor how baud rate can be adjusted).

The Arduino serial communication library[1] has many different functions, one of which is `Serial.read()`, returning a single byte of data, assuming at least one had been sent to the Arduino since any previous reads, i.e. the input buffer is not empty. Each byte consists of 8 bits, allowing for $2^8 = 256$ different values ranging in value from 0 to 255. Exactly what data type is used for serial communication depends on the programming language. Sometimes, serial data are considered characters according to ASCII codes[2], but often they are interpreted instead as 8-bit unsigned integers (UINT-8) when data rather than text is transmitted.

On Arduino, `Serial.read()` returns the incoming byte as an integer. However, if this result is instead stored in a `char` variable, the integer is automatically type cast by assigning the character corresponding to the ASCII code of the integer. `Serial.println` and `Serial.print` are heavily overloaded to accept input arguments of various data types. Passing an integer such as `Serial.print(38)`, actually sends the two bytes 51 and 56 corresponding to ASCII codes of the characters `'3'` and `'8'`. `Serial.println(38)`, sends a total of four bytes including integers 13 and 10 at the end which are carriage return `\r` and new line `\n`, respectively. In order to send the ASCII

[1] https://www.arduino.cc/en/reference/serial
[2] http://www.asciitable.com/

code 38 rather than the digits 3 and 8, instead typecast the integer `38` as a `char` as in the command `Serial.println((char)38)`.

Reading and writing serial data in Python

Including `import serial` allows access to the serial read/write class and its methods. The command `ser = serial.Serial('com15', timeout=0.001)` opens communication at the default 9600 baud to a device on the COM15 port where any read requests will stop waiting after 1 ms if no data is available. This function returns an instance of the serial class, which has methods for reading and writing, among others. Both `ser.write` and `ser.read` handle data of the bytes data type[3]. This data type is like a list of UINT-8 integers, in the same way a string is like a list of characters (both are immutable). Bytes are displayed as `b'???'` where the `b` stands for bytes and the various byte values in the list appear between the single quotes (much like a string without the preceding `b`). Unprintable bytes as displayed as `\x??` where `??` is the hexadecimal value of the integer (two hexadecimal digits represent numbers between 0 and 255). Most commonly, programs deal with integers or strings rather than bytes so conversion between these different data types is necessary. Below are some examples of converting between bytes and integers/strings.

```
>>> bytes([97,98,99]) # accepts a list of integers to convert to bytes
b'abc'
>>> bytes([97]) # Single integers to be converted must be placed in a list
b'a'
>>> bytes([10]) # New line character
b'\n'
>>> bytes([200]) # Unprintable character, 200 decimal equals C8 hexidecimal
b'\xc8'
>>> ord(b'a') # accepts a single byte and converts to an integer ASCII value
97
>>> ord(b'abc') # does not work on multiple bytes
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    ord(b'abc')
TypeError: ord() expected a character, but string of length 3 found
>>> ord('a') # can also convert characters to their ASCII values
97
>>> str.encode('abc') # Converts string of characters to a string of bytes
b'abc'
>>> bytes.decode(b'abc') # Converts string of bytes to a string of characters
'abc'
>>> x = b'abc'
>>> x[0] # Individual elements in a bytes object are already integers
97
>>> x[1] # ASCII 97 is a and ASCII 98 is b
98
>>> x = b'a'
>>> x[0] # Works even if the bytes object contains only a single element
97
```

[3] https://docs.python.org/3/library/stdtypes.html

The `ser.write` command expects a `bytes` object as an input argument and writes it to the connection associated with `ser`. Conversely, `ser.read` accepts an integer as an input argument for the number of bytes to read, and returns a `bytes` object containing the requested number of elements, assuming data exists in the buffer to be read. An empty `bytes` object `b''` is returned if no data is available.

Reading keyboard presses in Python

The `input` command in Python is useful for entering data from the keyboard, although it requires pressing enter. Unfortunately, Python has no simple method for immediately reading keystrokes, and solutions to the problem are specific to the computer's operating system. In Windows, the `msvcrt` module[4] is one solution. After `import msvcrt`, The command `msvcrt.getch()` returns a `bytes` object containing the pressed character. However, program execution will wait until a key is pressed. This presents problems, however, if a loop is to continually run and take a specific action only on keypress. The `msvcrt.kbhit()` function returns `True` if a key has already been pressed, and `False` otherwise. In this way, executing `getch` only if `kbhit` returns true will allow execution to continue even if no key is pressed.

Laboratory exercise procedure

To confirm serial communication, create an Arduino program to increment a counter each time through `loop`. If the counter is under 2000, write `HIGH` to pin 13 (the built-in LED) and `Serial.print` ASCII code 100, otherwise write `LOW` to PIN 13 and `Serial.print` the ASCII code 200 instead. Reset the counter to 0 if it exceeds 4000. In this manner, the LED and serial output alternate values every 2000 iterations through `loop`. Upload the program and confirm the LED blinks as desired and serial monitor shows alternation between 100 and 200. Now write a Python script like the one below to open a serial connection with the Arduino and execute an infinite loop where a single byte is read and its integer ASCII value repeatedly displayed on the screen.

```
import serial
ser = serial.Serial('com15', timeout=0.001)
while True:
    Use ser.read() to get a value and store it in x
    print(type(x))
    print(len(x))
    Print the integer value, i.e. 100 or 200
ser.close()
```

Executing this script from the command line should result in the Python displayed values changing in sync with the Arduino LED (although running the script in Idle or Spyder may exhibit some lag). Now import the `time` module and add the command `time.sleep(0.002)` to add a 2 ms pause each time through the infinite loop. Running now, even at the command line, should show a lag between the LED and displayed values because the Python program is too slow to read in every value sent by the Arduino. Depending on the complexity of both Arduino and Python programs, one will always be slower than the other and all waiting values should be read in and only the most recent used. Modify the Python script to use the `ser.inWaiting()` command which returns the number of bytes waiting in the input buffer. Rather than reading one byte at a time, read all available bytes but display only the most recent to the screen. Demonstrate the display and LED are in sync even when increasing to a 200 ms delay.

Write a loop in Python continually checking for a keypress and writing the pressed key value to the Arduino serial line when one occurs. Program the Arduino to continually read the sent values, turning on the LED if `'q'` is pressed

[4] https://docs.python.org/2/library/msvcrt.html

and turning it off on `'w'`. Alternately pressing the two buttons should toggle the LED without any lag. Introduce a `delay` of 200 ms into the Arduino `loop` and run again. Perhaps surprisingly, the LED still toggles without lag. This will be true as long as the buttons are not pressed more quickly than every 200 ms. With this same program, hold down the `'q'` button for about 1 s and then immediately press `'w'`. Notice the Arduino takes some time before turning the LED off because holding down the button sends numerous `'q'` bytes, which each take 200 ms to read. To fix this, read in every available byte each time through the loop and only use the last one. Like, `ser.inWaiting()` in Python, `Serial.available()` on Arduino returns the number of bytes waiting in the input buffer. Unfortunately `Serial.read()` can only read a single byte. Other functions like `Serial.readBytes` could be used to read in everything, but a simpler approach is:

```
while(Serial.available())
    val = Serial.read();
```

The loop executes if at least one byte is available and stops when nothing is left, leaving the last byte read in the variable `val`. Add this to the program and demonstrate the LED toggles without lag even with the 200 ms delay.

Now combine both programs to simultaneously send information in both directions. Connect a potentiometer to one of the Arduino analog inputs. When pressing `'q'` on the keyboard, the Arduino reacts by continually sending the potentiometer value, which Python should display on the screen, until `'w'` is pressed. Recall that `analogRead` returns values between 0 and 1024, whereas bytes are between 0 and 255.

As a final exercise, connect the IR rangefinder to another analog input and send both potentiometer and rangefinder values in the same manner as previously. Sending more than one byte of information requires a protocol so the receiver knows how to handle streams of data sent by the transmitter. One method would be to `Serial.print` the potentiometer reading and then `Serial.println` the rangefinder value. The receiver would therefore know a new sequence of data begins after the new line. However, recall that `Serial.println` sends both `\r` and `\n`, when only one byte signaling the stream end is necessary. Consider instead printing the potentiometer, rangefinder, and ASCII 0 values all with `Serial.print` where the 0 signals a new data sequence. Then three rather than four bytes need be sent for each data sequence. In the receiving Python script, only read data if sufficiently many bytes are waiting, and then use the `bytes.find` method to locate the index of the ASCII 0 value. For example:

```
>>> x = b'abcabcabc'
>>> x.find(b'c')
2
```

The 2 is returned since the first appearance of `'c'` is in the second index of `x`. Once the location of a complete data sequence is found, extract and display the adjacent two values corresponding to the potentiometer and rangefinder values. Note that sent potentiometer or rangefinder values of zero could garble the read data, so it would be ideal (although not necessary for this exercise) to limit the sent data to values greater than zero.

Grading rubric

1. (25 points) Demonstrate synchronous reading in Python of values sent by Arduino despite delays in the Python script.
2. (25 points) Demonstrate synchronous reading in Arduino of values sent by Python despite delays in the Arduino script.
3. (25 points) Pressing `'q'` and `'w'` on the keyboard control sending potentiometer readings, which display in Python.
4. (25 points) Pressing `'q'` and `'w'` on the keyboard control sending both potentiometer and rangefinder readings, which display in Python.