TDT4113 (PLAB2) Project 1: Morse Decoder

— Assignment Sheet —

Purpose of this project:

- Gain a basic appreciation of temporal abstraction in computer systems.
- Learn to transfer information between the Arduino and a Python program running on a laptop.
- Understand the essentials of Morse code via hands-on experience.

Practical Information:

- This project must be worked on individually.
- It will be solved using a combination of object-oriented Python and Arduino C.
- You must upload your code for this project to BLACKBOARD by **10:00 (in the morning)** on Friday, August 30, 2019. Note that most other projects must be handed in by 8:00 in the morning.
- You must demonstrate (in lab room A4-100) a working version of this project by **16:00** on Friday, August 30, 2019).
- It is *your* responsibility to see that you get your results approved by that deadline, so deliver early, and be on site early.
- That means the lab hours on Friday Aug 30 (from 10-12 and 14-16) will only be used for demonstrations and *not* for coding assistance in project 1 or others.

1 Introduction

Temporal abstraction is the separation of modules in a computer system based on differences between the timescales on which they operate. Operations that occur at a high frequency, say 100 times per second (a.k.a.100 Hertz) are encapsulated in *low level* routines, most of whose details are unknown to *higher level* routines, which run at lower frequencies. The levels exchange information, typically in the form of relatively simple signals, but neither level has access to the details of the other. And in a well-designed system, the levels should be independent enough that changes to one level should not, in most cases, require changes in the other.

This project deals with two timescales, one being the 100-1000 Hz behavior of the Arduino as it receives button-press signals from the bread board, and the other being the 1-3 Hz behavior of a Python program (running on your laptop) that combines simple signals from the Arduino into more complex symbols: letters and words.

The goal is to produce a system wherein the user taps in the *dots* and *dashes* of a Morse code message at the breadboard, and the natural-language text appears on the laptop screen.

2 Basic Setup of the Arduino

Essentially, the human user and breadboard combine to encode a message, such as "*Send more supplies*," which the Arduino and Python programs then decode. Hopefully, this decoded version matches the original.

As shown in Figure 1, the basic configuration involves 4 components: the breadboard, Arduino, Python program, and computer screen. The breadboard, whose detailed design is for each student to decide, must contain a button (black circle in figure) and a simple visual display (the red square and blue pentagons). The display should give some visual indication of whether the most recent button press was short (known as a *dot* in Morse code) or of longer duration (known as a *dash*). For example, a dot may illuminate one red light, while a dash lights up 3 green ones. Audio indicators (e.g. buzzers) should not be used for this assignment simply because they can cause unwelcome disturbances in large labs full of hard-working students.

2.1 The Arduino Code

The Arduino receives a steady stream of input from the *button pin* of the breadboard, indicating the current state of the button. This input will arrive very frequently, at 100 to 1000 Hz (or more), and it will be a simple binary value indicating a HIGH or LOW state of the button.

Your Arduino code must keep track of these binary signals (or at least a few of them) to determine when a key press begins and ends. It can then use time functions, such as *millis* to determine how long a key was held down or remained up. Both types of information (the length of a press and the length of a pause) are essential to Morse code communication. The combination of the two determines each of the 5 key primitive signals that are the building blocks of messages: dot, dash, short pause, medium pause, and long pause. This is a slight simplification of the complete Morse code specification, which includes an additional (very) long pause to indicate the end of a sentence.

When the classification of the latest press or pause is known, it should be sent, via a very simple signal, to the Python program running on your laptop. For each classification, the Arduino should send one of these four signals to Python:

1. Dot - a short button press
2. Dash - a long button press
3. Medium pause - time gap between two LETTERS in a message
4. Long pause - time gap between two WORDS in a message

That is, the Arduino should just send the number *0* for a dot, *1* for a dash, etc. In the case of a short pause, the Arduino need not send a signal to the laptop, since your Python program can just assume that any neighboring dots or dashes in the signal stream were originally separated by a short pause. The Python progran needs to know about medium pauses, however, because it cannot figure out where letters end on its own since Morse code is not a *prefix code*. A prefix code is a code in which the code for a symbol is guaranteed to NOT be the prefix of the code for another symbol). And, of course, it cannot figure out where words end unless it has a lot more (dictionary) knowledge (which you will certainly **not** be incorporating into this assignment).

In short, the main job of your Arduino code is to continuously sample the button state, register when it changes (from HIGH to LOW or LOW to HIGH), and then, when such a change is detected, use the time between the current change and the last change to determine the duration of the intervening press or pause. That duration will then permit identification of one of the 5 signal

classes; and whenever the signal is anything other than a short pause, that signal should be sent to the Python program running on your laptop. [1]

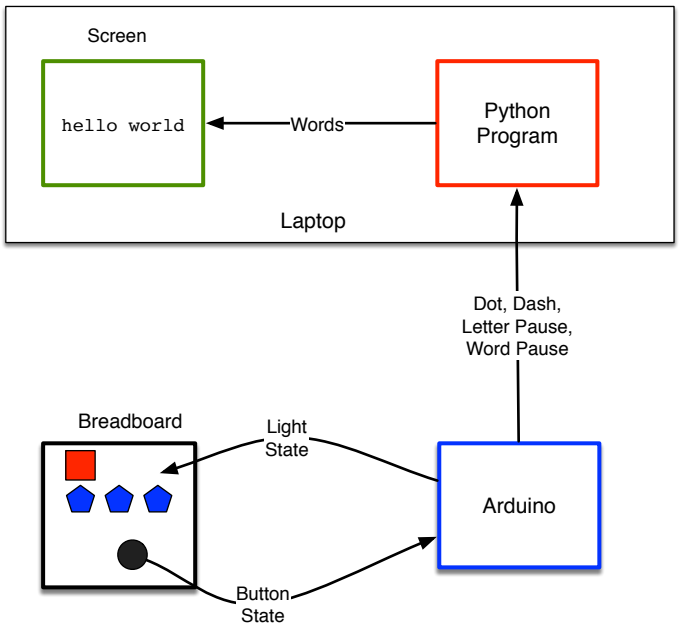

Figure 1: Basic setup of the Morse code system.

The official Morse code definitions of the dot, dash and three pauses are as follows (where T is a fixed time interval agreed upon by the sender and receiver):

Signal	Dot	Dash	Short (signal) pause	Medium (letter) pause	Long (word) pause
Duration	T	3T	T	3T	7T

You need not employ these exact timing differences between the 5 signals in your own Morse coder. Working with the standard Arduino buttons can be a bit tricky and may require a little *finessing* of the 5 durations in the above table. For instance, a reasonable value for T is around 300 milliseconds. It is pretty easy to press and hold a button for only 300 ms or less. However, it seems to be harder to create pauses of 300 ms or less. So you may need to declare the short pause as 1.5*T, for example, and the medium pause as 4.5*T, while keeping the dot and dash at T and 3T, respectively.

In short, feel free to adjust the 5 entries in the table to suit your breadboard, Arduino and personal button-pushing preferences, reaction time, etc. The important point is that your button presses and releases can encode the 5 basic signals with as little ambiguity as possible.

When working with Arduino buttons, the need to **debounce** an input often arises due to the noise occuring in the milliseconds surrounding a push or release action. This assignment is no exception. For more on debouncing, see any of a wide range of online tips, including `https://www.arduino.cc/en/tutorial/debounce`. As an alternative solution to noisy input, you may want to use a pull-down resistor on your breadboard, as described here: `https://playground.arduino.cc/CommonTopics/PullUpDownResistor/`.

[1] You may want to add an additional *reset* signal (for example, a 5) that the Arduino sends in the case of a very long pause (or long press). This allows you to avoid restarting the Arduino (and Python) everytime you want to try again to encode and decode a message.

2.2 The Serial Port

The Arduino and Python will communicate via a serial port on your laptop. This connection involves a few simple commands in both systems. On the Arduino side, you will need to do the following:

1. Include the command **Serial.begin(9600)** at the beginning of your Arduino code, preferably as part of your *setup* routine.
2. To write the value of some variable, x, to the serial port, use the command **Serial.print(x)**.

On the Python side, you will need to download the module called **pySerial**. In addition, we will provide you with the module called **arduino_connect**, which imports the proper module from pySerial via the statement **import serial**. Once in place, this supports easy reading from the serial port in your Python code. See instructions on installing pySerial in the supporting materials for this course. The code template that you will receive for this project includes the call to **arduino_connect**. You will need to modify that call (slightly) to use the proper device for your serial port (e.g. **COM23** or **/dev/cu.usbmodem1411**). This device name can be found in the bottom right corner of your Arduino window. Windows users can simply call **pc_connect()** – instead of **basic_connect(device)** – and **arduino_connect** will find the correct serial port.

2.3 The Python Code

In Python, you must use (very basic) object-oriented programming to convert the steady stream of signals (from the Arduino) into symbols (i.e., letters and digits), which your python code will aggregate into words. You will need one class, **mocoder** – give it a *reasonable* name of your own choice – that will have (at least) the following instance variables:

1. **serial_port** - the port that the Arduino writes to and Python reads from.
2. **current_symbol** - the symbol (i.e. letter or digit) currently under construction.
3. **current_word** - the word currently under construction. This may also be a number or hybrid (i.e. john3017). It is essentially any group of consecutive symbols.

You will also need one class variable, **morse_codes**, which is a Python dictionary whose key is a string of signals and whose value is the corresponding symbol. For instance, the beginning of the dictionary might look like this:

morse_codes = {'01':'a', '1000': 'b', '1010': 'c', '100': 'd', '0': 'e', ... }

where the binary strings represent sequences of dots (0's) and dashes(1's). Hence, the beginning of morse codes indicates that the symbol 'a' is a dot-dash sequence (01), whereas 'd' is dash-dot-dot (100). Your dictionary must include all lower-case letters of the English alphabet along with the digits 0 - 9; no punctuation, capital letters nor other special symbols are necessary for this project. As the basis for your dictionary, be sure to use the International Morse Code table, as found on the top right of the Wikipedia page for 'Morse code'.

Mocoder will have (at least) the following methods:

1. **read_one_signal()**: Read the next signal from the serial port and return it.
2. **process_signal(signal)**: Examine the recently-read signal and call one of several methods, depending upon the signal type. If it is a dot or dash, then call **update_current_symbol**; if it is a pause, then call **handle_symbol_end** or **handle_word_end** depending upon the type of pause.
3. **update_current_symbol(signal)**: Append the current dot or dash onto the end of **current_symbol**.

4. **handle_symbol_end()**: When the code for a symbol ends, use that code as a key into **morse_codes** to find the appropriate symbol, which is then used as the argument in a call to **update_current_word**. Finally, reset **current_symbol** to the empty string.
5. **update_current_word(symbol)**: Add the most recently completed symbol onto **current_word**.
6. **handle_word_end()**: This should begin by calling **handle_symbol_end** ; it should then print **current_word** to the screen and finally, reset **current_word** to the empty string.

Figure 2 shows the state of a mocoder and laptop screen while processing a 3-word expression. Note that most of the message has been processed, so the first two words have already been written to the screen. Half of the final word has been decoded, and the system is currently assembling the code for the letter 'r'.

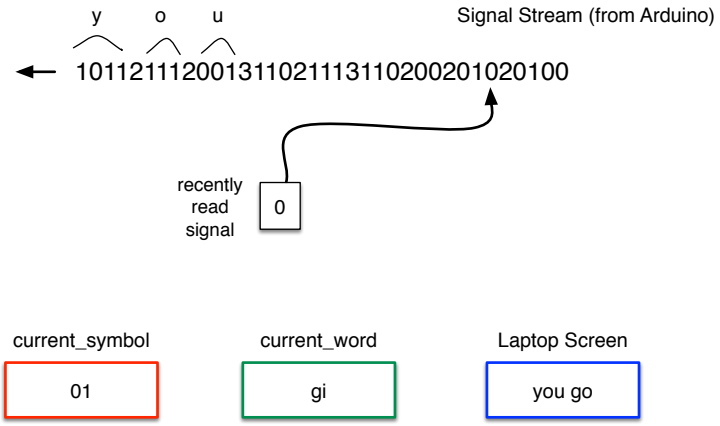

Figure 2: Intermediate state of a mocoder during the processing of the message, *you go girl*. Signals in the stream (read from the serial port) are dot (0), dash(1), symbol pause(2) and word pause(3).

2.4 General Comments

During a run of the complete system, both the Arduino and Python programs will run continuously. You are free to use additional signals to indicate the end of a message, for example, but that is not necessary for the project. In the basic version outlined above, any extremely long break from button pressing will simply be recorded as a 'long pause', nothing more.

During debugging of the Arduino, you will probably want to use the 'Serial Monitor' option from the 'Tools' menu. This can help you decide whether or not your chosen durations (for dots, dashes and pauses) are practical. However, once you begin running your Python code, the serial monitor needs to be disabled (by simply closing the monitor window). Otherwise, Python will probably signal an error.

3 Demonstration

To receive a passing mark for this project, you must provide a demonstration of your system to an instructor or course assistant. At the demo:

1. Your system must produce decoded words that are (in the eyes of the instructor or assistant) very similar to the original words. For example, if the original word is 'boat', then 'bozt' is close enough, but 'frzt' is (clearly) not. You will receive several words and thus several chances to show that you and your system perform reasonably well.

2. You must show well-structured and well-commented Arduino and Python code.
3. You must be able to answer any questions that the instructor/assistant asks concerning your system.

Extra Challenge: Fine-tune the timing parameters of your Arduino, and polish your own telegraph-operator skills to the point where you can type in a message, without error, at the rate of 3 seconds per letter, or faster. So, for example, *'Watson come quick'*, should take no more than 45 seconds to enter. You never know when this 19th-century skill could come in handy.

— o —