TDT4113 (PLAB) Project 1: Morse Decoder

— Assignment Sheet —

Purpose of this project:

- Gain a basic appreciation of temporal abstraction in computer systems.
- Learn to transfer information through a simulated hardware to a Python program
- Understand the essentials of Morse code via hands-on experience.

Practical Information:

- This project must be worked on individually.
- It will be solved using object-oriented Python.
- You must upload your code for this project to BLACKBOARD by **10:00 (in the morning)** on Friday, January 22, 2021. Note that most other projects must be handed in by 8:00 in the morning.
- You must demonstrate a working version of this project by **16:00** on Friday, January 22, 2021.
- The demonstration is done through Microsoft Teams.
- It is *your* responsibility to see that you get your results approved by that deadline, so deliver early, and be present in demo on time.
- That means the lab hours on Friday Jan 22 (from 10-16) will only be used for demonstrations and *not* for coding assistance in project 1 or others.

1 Introduction

Morse code is a simple method for telecommunication, where text symbols are represented by standardized binary code. The method is still useful under certain circumstances because it requires only one button as the input device. Implementing such a simple but useful tool gives you nice practice to start your journey in this programming course.

The goal is to produce a software system wherein the user (sender) taps in the *dots* and *dashes* of a Morse code message through simulated hardware. Your program should be able to decode the message and display it on the computer screen.

In the next section, we first describe the setup with external hardware for the project. However, due to the pandemic situation we have to use software simulators for the hardware so that we can work remotely without physical contact. The simulator will be described in Section 3. After that we give some hints on the programming task in Section 4. Finally the checklist for passing the project is given in Section 5.

2 Raspberry Pi

In the previous PLAB courses we had used Arduino hardware and software. This year we planned to replace Arduino with a more recent device called Raspberry Pi, which provides simplified programming interface to Python.

Raspberry Pi (RPi) is a series of small computers which are suitable for education and embedded programming. An RPi 4 Model B is shown in Figure 1. It contains all the basic computer elements such as CPU, RAM, external storage (a SD card), and various ports. A highlighted component is the I/O pins shown on the top right, where can be used to connected with various external devices, for example a button and LEDs in this project.

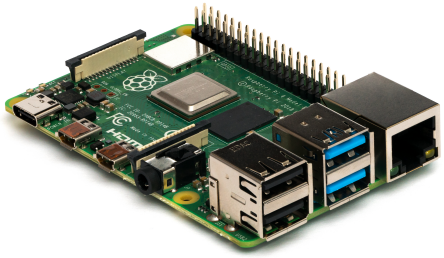

Figure 1: Raspberry Pi 4 Model B

2.1 Configuration

The project design with physical hardware involves the following components: a button, a red LED, three blue LEDs, an RPi, and a computer screen. The button and the LEDs are connected with the RPi I/O pins and the RPi is connected to the screen. The connection diagram is shown in Figure 2.

The sender uses the button to send message. Your Python program, which runs on the RPi, receives the button states, calculates the intervals between the presses, and determines the signal type. If it is a Morse dot, then set the red LED state(s) to blink it. If it is a Morse dash, then set the blue LED states to blink them. Your program then decodes the sequence of signals to text message according to the Morse code standard and prints the text on the screen.

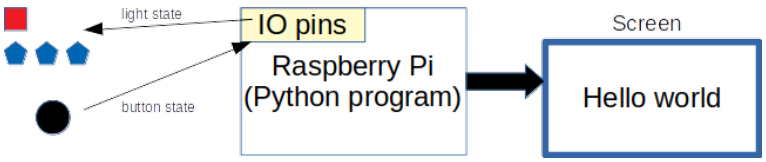

Figure 2: Configuration of the project with hardware: (red square) the red LED, (blue pentagons) the blue LEDs, and (black circle) the button.

2.2 Programming with GPIO

Raspberry Pi provides a package RPi.GPIO as programming interface with the devices connected to RPi I/O pins. There are three GPIO functions for basic operations

- `GPIO.setup(pin, pin_mode, pin_initial_state)`: the function setups a pin (specified by `pin`) to a mode and its initial state. The pin mode can be `GPIO.IN` or GPIO.OUT. The pin state can be for example `GPIO.LOW` or `GPIO.HIGH`.

- `state = GPIO.input(pin)`: the function reads the current state of the device connected to the specified pin.
- `GPIO.output(pin, state)`: the function set the state of the specified pin.

For example, the following commands lights up an LED at `PIN_LED`:

```
GPIO.setup(PIN_LED, GPIO.OUT, GPIO.LOW)
GPIO.output(PIN_LED, GPIO.HIGH)
```

2.3 Get the stream of signals

The Raspberry Pi receives a steady stream of input from the *button pin*, 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 program must keep track of these binary signals (or at least a few of them) to determine when a key press begins and ends. You can frequently check the button state and use the `time()` function in the `time` package 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:

1. Dot - a short button press
2. Dash - a long button press
3. Short pause - time gap between two SIGNALS
4. Medium pause - time gap between two SYMBOLS
5. Long pause - time gap between two WORDS

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. See Figure 3 for a illustration example.

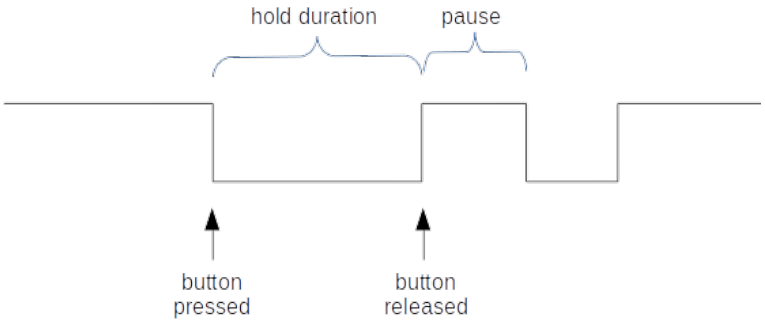

Figure 3: An illustration example of the key pressed signals.

Encoding short pauses could be optional because 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 program 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).

So your program needs encoding at least four classes of signals and convert them to numbers, for example, 0 for dot, 1 for dash, 2 for medium pause and 3 for long pause. After the encoding, the signals become a stream of numbers and ready for decoding.

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. It 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, the five entries in the table could depend on the hardware and the sender and are adjustable. The important point is that your button presses and releases can encode the 5 basic signals with as little ambiguity as possible.

3 Simulated GPIO

Unfortunately due to the COVID situation we cannot stay in Datasal together. This actually turns down the possibility of PLAB with real hardware because not every student has access to the hardware. Even if some students have, we cannot do the demonstration together.

Luckily the devices used in the project are simple and we can simulate them with software. The physical button is replaced with the SPACE key on computer keyboard, and LED lighting is replaced with screen printing.

We provide simulator code in a Python file `GPIOSimulator_v1.py`. After importing the file, you can program with the simulated GPIO as similarly as with the original GPIO. For example, the following lines can get the state of the simulated button:

```
from GPIOSimulator_v1 import *
GPIO = GPIOSimulator()
GPIO.setup(PIN_BTN, GPIO.IN, GPIO.PUD_DOWN)
state = GPIO.input(PIN_BTN)
```

Similarly, `GPIO.output(PIN_BLUE_LED, GPIO.HIGH)` will print "The blue LED becomes ON", which simulates lighting the blue LED.

NOTICE. Button presses on a physical button are not always accurately detected. To simulate this, we intentionally set the return of `GPIO.input` function only 99% reliable. That is, it has 99% probability to return the true state of compute keyboard SPACE key, while with 1% probability to return randomly either `GPIO.PUD_DOWN` or `GPIO.PUD_UP`.

Since 1% of the `GPIO.input(GPIO.PIN_BTN)` result is random, 0.5% of the function return can be wrong. Then for example, a symbol contains 4 dots and/or dashes has failed chance up to $1 - 0.995^4 = 2.0\%$. Consequently, if the message contains 6 such symbols, the failed chance is up to $1 - 0.995^{24} = 11.3\%$. The failed rate could be even worse if pauses are also counted. *So you must consider how to reduce the failed possibility.*

Note: the simulator requires ROOT privilege on Linux systems. Suppose your Python is installed in `/home/username/anaconda3/`. You may need to run the following commands before running your program:

- `su` (then input the ROOT password)
- `export PATH="/home/username/anaconda3/bin:$PATH"`

4 The main program

Repeatedly calling `GPIO.input(GPIO.PIN_BTN)` you can get the a steady stream of pin states. Then in the main program, you must use (very basic) object-oriented programming to convert the stream of pin states into symbols (i.e., letters and digits), which your 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. **current_symbol** - the symbol (i.e. letter or digit) currently under construction.
2. **current_word** - the word currently under construction. This may also be a number or hybrid (e.g. 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', ... }.
```

Alternative you can use the signal '.' and '-' to replace '0' and '1', respectively. The dictionary will be used to map a string of dot/dash to the corresponding symbol, for example, the symbol 'a' is a dot-dash sequence (or '01'), whereas 'd' is dash-dot-dot (or '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:

- **read_one_signal()**: Read the next signal from the serial port and return it.
- **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.
- **update_current_symbol(signal)**: Append the current dot or dash onto the end of **current_symbol**.
- **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.
- **update_current_word(symbol)**: Add the most recently completed symbol onto **current_word**.
- **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.

See `template.py` for some other optional functions.

Figure 4 shows the state of a Mocoder and computer 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'.

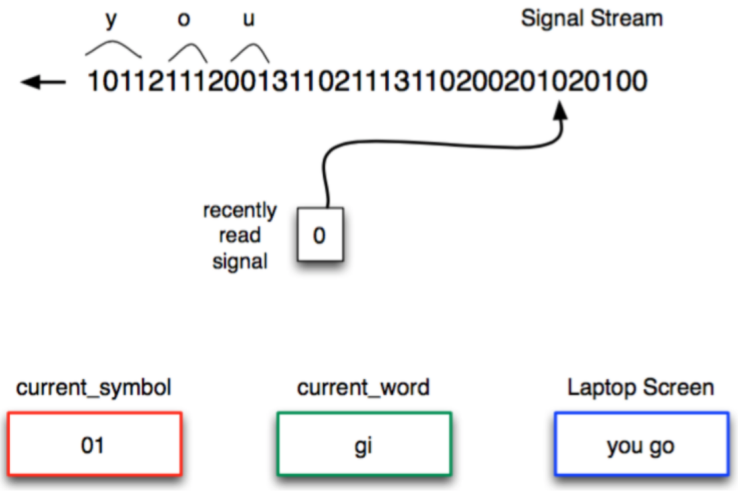

Figure 4: Intermediate state of a Mocoder during the processing of the message, *you go girl*. Signals in the stream (read from the serial port) are dot (0), dash(1), symbol pause(2) and word pause(3).

During a run of the complete system, the Python program 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, it is wise to isolate the factors and do modular tests. For example, you can write a small program which is dedicated for testing the button input, or you can test the decoding part using predefined sequence of signals without the simulated hardware.

5 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 code.
3. You must be able to answer any questions that the instructor/assistant asks concerning your system.

— o —

References