TDT4113 (PLAB) Project 5: Keypad

— Assignment Sheet —

Contents

1 Introduction 2

2 The Finite State Machine (FSM) 4
2.1 Interactions between the Agent and FSM . . . 6
2.2 Implementation Details of the FSM . . . 7
2.3 Additional Design and Debugging Tips . . . 10

3 Setup of The Keypad Controller 11

4 The Keypad 11
4.1 The physical keypad . . . 11
4.2 The simulated keypad . . . 11
4.3 Programming interface to the simulated Keypad . . . 12
4.4 Polling the keypad . . . 13

5 The Charlieplexed Breadboard 14
5.1 The physical Charlieplexing circuit . . . 14
5.2 Implementing the Charlieplexer . . . 14

6 Primary Python Classes, Instance Variables, and Methods 16
6.1 Finite State Machine . . . 16
6.2 Keypad . . . 16
6.3 LED Board . . . 16
6.4 KPC Agent . . . 17

7 A Typical Run of the Keypad Controller 18

8 Demonstration of the Keypad Controller 19

Purpose:

- Learn to properly interpret signals from a simple keypad.
- Learn about Charlieplexed circuits.
- Learn the basics of a Finite State Machine (FSM) and its use as a controller.
- Gain experience integrating the FSM and simulated hardware.

Practical Information

- This project will be done in groups of size four.
- You must use the provided simulator for keypad and Charlieplexed circuits
- Your code must be uploaded to BLACKBOARD prior to 8:00am on March 10, 2021 and demonstrated before 8pm on March 10, 2021.
- Your code needs to have a Pylint level of at least 8.0

1 Introduction

As shown in Figure 1, your system, the *Keypad Controller*, will contain 3 key functional components: a finite state machine (FSM), the keypad and a *'Charlieplexed'* LED board.

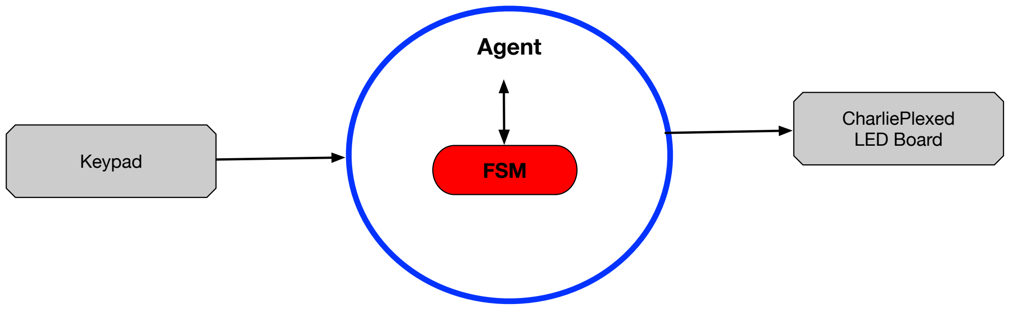

Figure 1: Basic functional architecture of the keypad controller.

There is one central software object, the *agent*, which coordinates activity among them. The agent uses the keypad for input from the (human) user, and the LED board for output communication with the user. Within the agent, the FSM controls much of its decision making. However, as described later, the FSM will work at a reasonably abstract level, leaving many of the straightforward, but detailed, computations (and their related data structures) to the agent.

You will implement the FSM as a simple rule-based system (RBS). Each behavioral rule consists of a condition and a consequent. During rule application, the condition is compared to the current situation (a.k.a. *context*) of the FSM. The FSM context consists of its internal state and its current signal. When *context* matches *condition*, the rule *fires* by performing the operations specified in the consequent. These operations are simply changes to the FSM's state and requests for the agent to perform particular actions, such as verifying a password, lighting a particular LED, etc.

The keypad device does not come fully programmed to transfer simple symbolic signals (i.e. keystrokes) to your Python program, running on the Raspberry-Pi (R-Pi). Instead, you will need to explicitly probe the keypad's button array and discern digits (0-9) and two special symbols ("#" and "*") from the results of those probes.

As an added bonus, you will learn a creative trick for getting the most out of a breadboard that has limited number of connection pins to the Raspberry-Pi. The technique, known as *Charlieplexing*, enables you to control many actuators (e.g. lights, buzzers, etc.) using only a few input/output ports. The trick lies in the combination of wiring and Python programming.

Due to the pandemic situation, we cannot stay together in Datasal. This makes PLAB with real hardware impossible because not every student has access to the hardware. Even if some students have, we cannot do the demonstration together. Alternatively we simulate the above hardware with software, where the keypad input is replaced with your computer keyboard input, and LED lighting is replaced with screen printing.

The general operation of the agent is quite straightforward, with a complexity level similar to that of an advanced garage-door opener: a password-protected system enabling a few basic actions, including changing the password.

Via the keypad, the user will be able to enter sequences of symbols that will direct the agent to perform the following operations:

- Receive and validate — or reject — the password.
- Change the password.
- Select one of the 6 LEDs on the breadboard and turn it on for a time duration specified by the user via the keypad.
- Log out of the system.

We will now delve into the details of the FSM before discussing its interactions with the agent and the design of the keypad and Charlieplexed breadboard.

2 The Finite State Machine (FSM)

FSMs are often depicted as graphs: diagrams containing nodes with labeled, directed connections (a.k.a. arcs) between them. Each node represents a unique state of the system, while arc labels denote the external signal that triggers the transition from one state to another. As a simple example, Figure 2 provides two similar FSMs for identifying a particular password: 836. Presumably, the external signals, the password digits, come from a keypad.

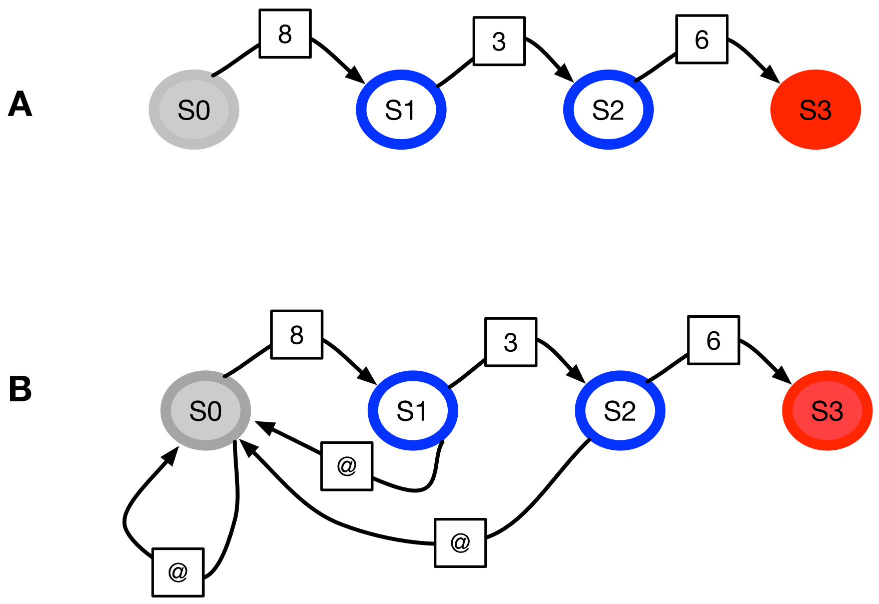

Figure 2: Two simple Finite State Machines (FSMs) for detecting the password 836, with gray start state (S0) and red final state (S3).

The FSM begins in state S0. If the user presses an "8", the FSM switches into state S1. It then moves to state S2 on a "3", and into the final state (S3) on a "6". Once in a final state, the FSM could then ask the agent to perform a password-protected action, such as unlocking a door, turning on a coffee machine, etc.

But what if the user makes a mistake? The upper FSM of Figure 2 specifies no particular actions for incorrect digits, which often implies that the FSM remains in the current state. But that is obviously a flawed design for a password reader, since the user could just keep guessing digits until she hits the proper one, thus advancing the FSM one state, and eventually pushing it to state S3.

A more reasonable design (Figure 2B) includes "@" such that the FSM switches back to the start state, S0. These apply to any situation in which the user enters an incorrect digit; they force the user to start over. This is still somewhat *forgiving* in that it allows the user an unlimited number of attempts, but for this assignment, that level of security (or lack thereof) is perfectly sufficient.

Many alternatives for programming an FSM exist, including those in which each node of the graph is a node object, with arc objects emanating from it. Conversely, this assignment uses a *rule-based system*'s (RBS) approach to modeling the FSM. In an RBS, a set of rules (often called "IF-THEN" rules) are applied to the current context (i.e. FSM state plus trigger signal) to determine the new FSM state and (optionally) the action(s) to be performed by the agent.

An IF-THEN rule has this basic structure: "IF condition THEN consequent": the consequent is performed only if the condition (a.k.a. *antecedent*) is true. In the parlance of RBS, a rule is *applied* when its condition is compared to the current context, while a rule is *fired* when its condition matches the context and its consequent is performed. When a sequence of such rules are applied to a context, the standard assumption is that only ONE of them will fire, which will change the context, and then initiate another round of rule application. Furthermore, the **only** rule that fires is the **first** applied rule whose condition matches the context. Hence, the **ordering** of rules within the RBS is an essential factor that strongly influences its performance.

Although we will **not** implement our RBS as a big IF-THEN-ELSE statement, it helps to use such a statement in order to understand the logic behind an RBS. Using the FSM of Figure 2B as our model, the pseudocode below realizes that FSM's overall behavior:

```
IF fsm.state == S0 AND fsm.signal == 8 THEN fsm.state ← S1
ELIF fsm.state == S0 AND fsm.signal != 8 THEN fsm.state ← S0
ELIF fsm.state == S1 AND fsm.signal == 3 THEN fsm.state ← S2
ELIF fsm.state == S1 AND fsm.signal != 3 THEN fsm.state ← S0
ELIF fsm.state == S2 AND fsm.signal == 6 THEN fsm.state ← S3
ELIF fsm.state == S2 AND fsm.signal != 6 THEN fsm.state ← S0
```

Although this is pseudocode, the intention is that EXACTLY ONE of the rules will be applied, and it will be the FIRST rule whose antecedent matches the current context. This code can be simplified in several ways. First, if we assume that the forward-transition rule always appears before the "return to S0" rule in the list, then the antecedent of the latter rule can be simplified (i.e. generalized) as follows:

```
IF fsm.state == S0 AND fsm.signal == 8 THEN fsm.state ← S1
ELIF fsm.state == S0 THEN fsm.state ← S0
ELIF fsm.state == S1 AND fsm.signal == 3 THEN fsm.state ← S2
ELIF fsm.state == S1 THEN fsm.state ← S0
ELIF fsm.state == S2 AND fsm.signal == 6 THEN fsm.state ← S3
ELIF fsm.state == S2 THEN fsm.state ← S0
```

The important principle here is *specific-before-general*: the specific condition (e.g. fsm.state == S0 AND fsm.signal == 8) appears before the more general condition (fsm.state == S0). This ensures that when the agent is in a particular state, the general rule fires only if the specific rule does NOT fire. Thus, the general rule effectively realizes the "everything but" logic signified by the "@" in Figure 2B.

Note that we could simplify the above pseudocode even more:

```
IF fsm.state == S0 AND fsm.signal == 8 THEN fsm.state ← S1
ELIF fsm.state == S1 AND fsm.signal == 3 THEN fsm.state ← S2
ELIF fsm.state == S2 AND fsm.signal == 6 THEN fsm.state ← S3
ELSE fsm.state ← S0
```

In this case, an equivalent RBS would house only 3 rules. Then, if none of them applied to the current context, the FSM's state would be reset to S0. Although this shortcut works fine, we normally want the contents of an RBS to mirror that of an FSM, and this is easiest if each arc of the FSM maps to a unique rule of the RBS. Hence, in Figure 2B, each "@" move should also have its own rule in the RBS. This ease of understanding becomes very important when large FSMs are to be implemented by an RBS.

2.1 Interactions between the Agent and FSM

As mentioned earlier, the FSM for this project will work at an abstract level, while the agent will do a lot of the routine computational work. Implementing all such activity within the FSM can be tedious, if not downright impossible. Essentially, we will rely on the agent to provide the FSM with a memory for everything other than the FSM's current state and signal.

We can begin with a simple practical aspect: a keypad controller system should allow the user to define and re-define their own password. For the time being, we will ignore exactly HOW the password gets defined but focus on how the FSM should be designed to check for ANY password, not just 836.

Assume that the agent has the current password (CP) stored in memory (or in a short text file). A more abstract FSM could then operate as follows:

- The FSM begins in state S-Init
- When the user hits any keypad button, the FSM switches to a "read password" state (S-Read).
- Each digit entered during this S-Read is simply passed to the agent, stored in a cumulative-password (CUMP) array or string.
- When the user enters a '*' (asterisk), this tells the FSM to instruct the agent to verify the password, which it does by comparing CUMP to CP. The FSM also moves into a new state (S-Verify).
- The next signal from the agent should then indicate whether verification succeeded or not. Success causes the FSM to change to state S-Active, while failure forces a return to state S-Init.

The exact details of this sequence may vary, but the key point is that the FSM is no longer bound to a single password but can handle any password as long as the user signals the end of the password-entry phase via "*".

To handle agent-FSM interactions, the *consequent* aspect of each FSM rule needs to be expanded to a pair: new state, action. This means that the labels on arcs in FSM diagrams must change, but to a different pair: trigger signal, action. Thus, the FSM updates its state and sends an action command to the agent. Figure 3 incorporates this enhancement in providing the FSM for a more flexible agent.

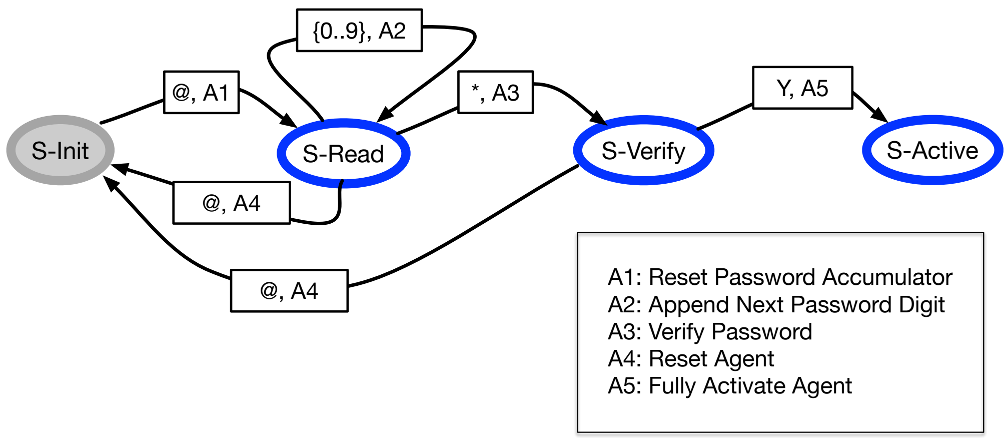

Figure 3: Portion of an FSM for a more abstract, and flexible, agent. Each arc indicates a pair: triggering signal, agent action, with actions described in lower-right box. The signal "@" still denotes "anything not specified as a trigger for the other outgoing arcs from this same state".

To extend this example to an FSM that accepts password changes, a few new FSM states and agent functions are required. Figure 4 illustrates this new FSM, which uses state S-active from Figure

3 as its starting point: once the user has successfully logged in via their (old) password, they are then in an FSM state (S-active) where it is possible to do many things, including changing to a new password.

The agent presumably has a few simple data structures for storing the current password (CP), the cumulative password (CUMP), and the last value of CUMP (CUMP-OLD). As shown in Figure 4, the user signals a desire to change the password by typing "*" (asterisk), which causes the FSM to change to state S-READ-2 and to instruct the agent to perform action A1, i.e. to reset CUMP. Each new digit that the FSM reads is then added to CUMP, via action A2. Upon reading a "*" in S-READ-2, the FSM initiates action A7, which will cache the current state of CUMP in CUMP-OLD and then reset/clear CUMP.

The FSM also switches to state S-READ-3, whose behavior is very similar to S-READ-2, with the only difference being that upon receiving a "*" in S-READ-3, the FSM induces action A8, which will compare CUMP to CUMP-OLD and change the current password (CP) if and only if CUMP == CUMP-OLD. Either way, once A8 is done, the FSM returns to state S-Active. If the password does change, then the agent will need to write the new password to more permanent storage, such as a file.

In this example, the difference between agent actions A1 (reset agent) and A6 (refresh agent) may be quite minor. Resetting essentially means putting the agent in a very restricted mode where it can only receive password login attempts, whereas refreshing means returning to a less-restricted active state from which many other actions (not shown in Figure 4) such as turning on lights, may be initiated.

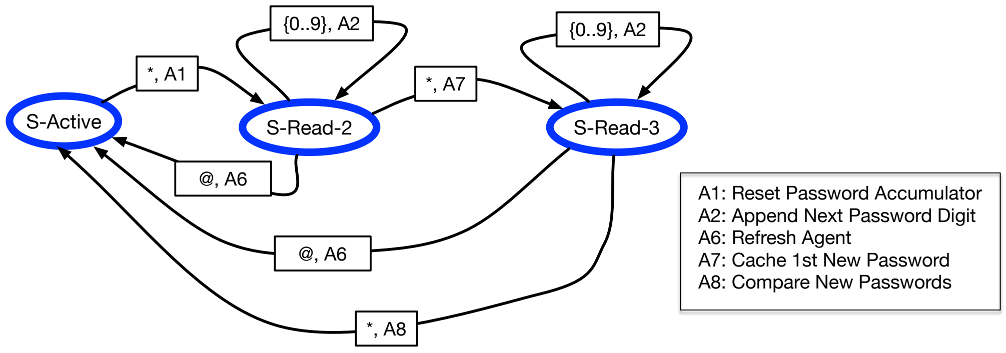
{0..9}, A2
{0..9}, A2
*, A1
*, A7
S-Active
S-Read-2
S-Read-3
@, A6
@, A6
*, A8
A1: Reset Password Accumulator
A2: Append Next Password Digit
A6: Refresh Agent
A7: Cache 1st New Password
A8: Compare New Passwords

Figure 4: An extension of the FSM in Figure 3 to handle password updates, wherein the user must enter the new password twice.

2.2 Implementation Details of the FSM

To realize the FSM as a rule-based system (RBS), you will need to implement each rule as an object and to then cycle through those rules to determine the next action of the system.

Remember that a rule represents one arc (from an FSM graph) along with the nodes at its head and tail.

Thus, a rule object should contain at least these four instance variables:

1. state1 - triggering state of the FSM
2. state2 - new state of the FSM if this rule fires
3. signal - triggering signal
4. action - the agent will be instructed to perform this action if this rule fires.

Each rule has the following meaning:

IF the FSM is in a state that matches rule.state1 and the current signal matches rule.signal THEN change the FSM's state to rule.state2 and call the method specified by rule.action (and give that method two arguments (the agent itself and the current signal).

In this description, note the word *matches*. This is not necessarily the same as *equals* but is a more general term.

Thus, rule.state1 may be a simple string, such as 's4', meaning that the FSM's actual state (S) must be s4. However, rule.state1 can also be a function that implicitly represents a collection of FSM states, any one of which might *match* S. Crucially, if rule.state1 is not a string, but a function, then that function needs to take exactly one argument: the FSM's actual state (S). The function then returns a boolean value (True/False) depending upon whether or not S matches its criteria. If the function accepts ANY state, then it should always return True, regardless of the value of S.

Similarly, rule.signal can *match* the current signal (X) by either being a symbol that is identical to X, or by being a function that accepts X as an argument, performs some test on X, and returns a boolean. Or, if rule.signal is meant to match anything, then it would simply accept X as an argument and output *True* regardless of X's value.

In another case, you may want rule.signal to be a simple function that only accepts the digits 0-9. To do this in Python:

- Define a general function (not a method), such as:

  def signal_is_digit(signal): return 48 <= ord(signal) <= 57

- When defining the rule, simply write:

  rule.signal = signal_is_digit

- Then, when checking whether the current signal (sig) matches rule.signal, make the call:

  rule.signal(sig)

  which should only return True if sig is a digit.

Note, however, that rule.state2 should never be a function; it should always be a string specifying a single, legal state of the FSM. Similarly, rule.action should be a specific method that your agent object understands.

Practically speaking, you are probably much more likely to use a function for rule.signal than for rule.state1. Still, you may have an FSM where the same transition can be taken FROM several different states, as exemplified by the "@" transitions in Figures 3 and 4; and in those cases, you may want to reduce the size of your rule base by having one rule (with rule.state1 as a function) that replaces several rules (with rule.state1 as different strings in each rule). As mentioned above, this shortcut does destroy the one-to-one correspondence between the arcs of an FSM graph and the rules of an RBS, but you are free to explore either option.

Since rule.state1 and rule.signal may be strings or functions, at some point in your code you will probably want to import the *isfunction* function provided in Python's *inspect* module via this command:

```
from inspect import isfunction
```

This allows you to quickly check whether rule.signal (or rule.state1) is a function and to take action accordingly.

The following pseudocode reflects the basic operation of your overall system, with focus on the FSM:

- Create all rule objects for the FSM - and ensure they are listed in the desired order.
- Initialize agent, keypad and LED board

```
• state ← fsm-start-state
• While state ≠ fsm-end-state do:
      # Get next signal, typically from the keypad
      signal ← agent.get_next_signal()
      # (Rule-Application Loop)
      For rule in fsm.rules:
          IF rule.match(state, signal):
              state ← rule.state2
              agent.do_action(rule.action, signal)
              GO TO (**)
          (**)
      End For
• End While
• Shutdown agent, keypad, LED board, etc.
```

As one useful option for coding up `agent.do_action()`:

- Assume that your agent is of type KPC (keypad controller) and that KPC has several methods, including *start_password_entry*.
- Assume rule R3 declares: when in state S0 and reading signal "*" (asterisk), switch to state S1 and invoke *agent.start_password_entry()*.
- R3 would then be defined as a rule with the following Python code used to set its action attribute:

  rule.action = KPC.start_password_entry

  This syntax tells Python to fetch the desired method from the class KPC and put it into the action slot of rule.
- Given this rule definition, the actual Python syntax for agent.do_action() becomes:

  rule.action(agent, signal)

  where the first argument (agent) gets bound to *self* in the method's argument list.
- Then, the action slot of **every** rule in your FSM should be set to KPC.zzzz, where zzzz is a different method name.

Remember that the order of rules is extremely important. As a simple example, Table 1 lists the appropriate rules and their ordering for the FSM of Figure 3. In that table, note the use of KPC as the agent class and the use of simple functions such as *all_signals* that return True for any signal.

Table 1: A rule set corresponding to the FSM graph of Figure 3.

Index	State-1	Signal	State-2	Action
1	S-init	all_signals	S-Read	KPC.reset_password_accumulator
2	S-Read	all_digits	S-Read	KPC.append_next_password_digit
3	S-Read	*	S-Verify	KPC.verify_password
4	S-Read	all_signals	S-init	KPC.reset_agent
5	S-Verify	Y	S-Active	KPC.fully_activate_agent
6	S-Verify	all_signals	S-init	KPC.reset_agent

Note that rules 1, 4 and 6 use the *all_signals* test, but that rule has a different meaning (a.k.a. semantics) in each case, due to the effects of rule ordering and the fact that we have declared specialized rules before the general rules. In rule 1, it really does mean "all signals", but the effects of rule ordering are obvious when we get down to rule 4. Since rules 2 and 3 also apply in state S-Read, rule 4 will only be fired if rules 2 and 3 fail to match the current signal. Hence, the actual semantics of *all_signals* in rule 4 is "any signal except a digit or *". So if the FSM is in state S-Read with a current signal of "*", then rule 3 will fire, not rule 4, simply because the rule-ordering ensures that rule 3 is always applied before rule 4. Similarly, the semantics of *all_signals* in rule 6 is "all signals except a Y".

2.3 Additional Design and Debugging Tips

1. Although your FSM needs to be programmed as a rule-based system, it is **strongly advised** that you draw your complete FSM as a graph, and then convert the graph to a list of rules. It is much easier to debug your FSM logic in graph form than in rule form.
2. FSM states can be represented as simple strings or integers, while signals are probably easiest to model as strings (typically of length 1).
3. The method agent.get_next_signal() should normally cause the agent to query the keypad (as detailed later in this document), but you should also keep an `override_signal` as a slot in the agent for cases where the agent's decision (such as to accept or reject a password) is the next signal that the FSM should receive, instead of keypad input. Be sure to clear the `override_signal` immediately after using it. The agent should know that if the `override_signal` is empty, it should query the keypad. Setting of the `override_signal` should only occur via your agent's action methods, and only a small subset of those methods should ever need to do so.
4. The call to agent.do_action() can be implemented in many different ways, but all legal agent actions should be defined as methods for your agent class.
5. Debugging of the FSM is much easier if done on your laptop with the simulated keypad and LED board replaced by proxies. The proxy keypad can be a simple object that asks for user input from the laptop keyboard, while the proxy LED board can just print out messages such as "Turning light 3 on for 20 seconds".

3 Setup of The Keypad Controller

Figure 5 show the setup with physical hardware, where several components contribute to the Keypad Controller. The keypad itself has many wires, each of which connects to the Transition Breadboard, whose only real purpose (when combined with the T-Cobbler) is to provide a more understandable interface between the keypad and the Raspberry Pi (as explained below).

All of the Python code for your system will reside on the Raspberry Pi. That includes code for your agent, FSM and interfaces to the keypad and LED board. The Charlieplexed LED Breadboard serves as the primary output for the agent. Different on/off patterns of the LEDs will provide basic (but useful) information to the human user.

In simulation, everything is run on computer (e.g., your laptop). The keypad and LED breadboard are virtual and simulated by software, where the input is with computer keyboard and LED status is printed on computer screen. The simulator preserves a very similar programming interface as Raspberry Pi GPIO.

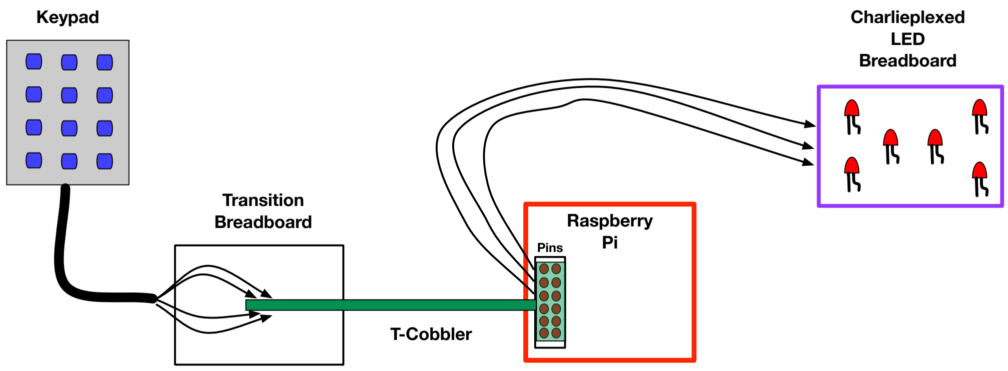

Figure 5: Overall Architecture of the Keypad Controller

4 The Keypad

4.1 The physical keypad

As shown in Figure 6, the keypad consists of 12 keys arranged in a grid, with each key having a unique location (row, column). Since these keypads do not come with code for converting grid activity into keystrokes, you need to write that code yourself, as explained below. First, however, you need to make sure that the keypad is properly connected to your Raspberry Pi.

4.2 The simulated keypad

Due to the COVID situation, we will use a simulated keypad to replace the physical one. The simulator requires the Python `pynput` package. You can install it by `pip install pynput`. No ROOT privilege is required.

After importing the simulator, you can imagine that the simulated keypad has already been connected to the "Raspberry Pi GPIO" with the following pins which correspond to the R0-R3 and C0-C2 in Figure 6 right:

- `PIN_KEYPAD_ROW_0`
- `PIN_KEYPAD_ROW_1`
- `PIN_KEYPAD_ROW_2`

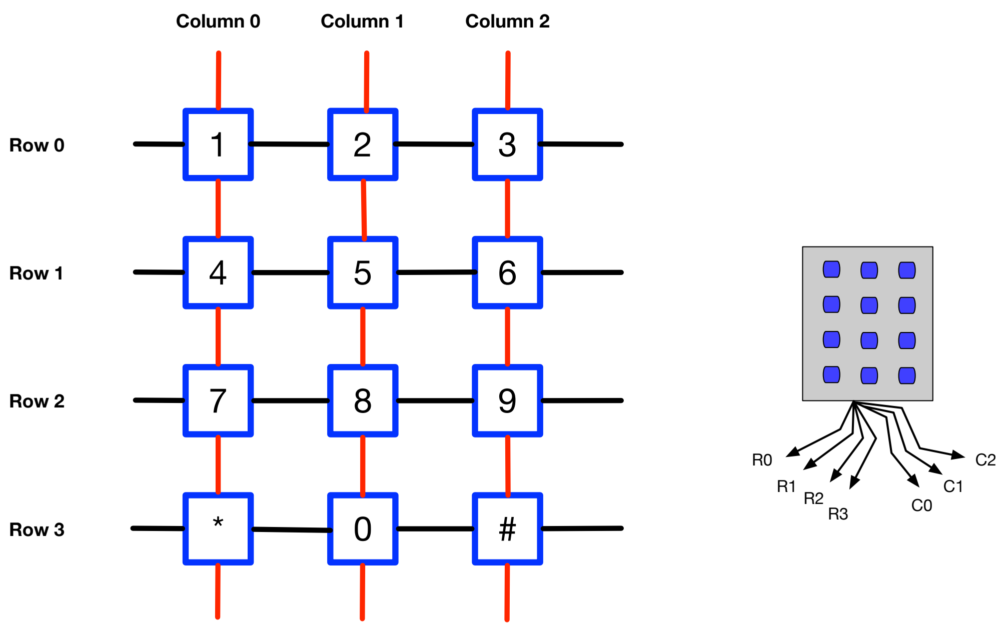

Figure 6: (Left) Internal circuitry of the keypad, with one wire for each row and column. (Right) External keypad circuitry, with all 7 wires exiting as a flat, transparent band from the bottom of the keypad. In that band, the ordering of row and column wires are as diagrammed: row 0's wire is leftmost, while column 2's is rightmost.

- `PIN_KEYPAD_ROW_3`
- `PIN_KEYPAD_COL_0`
- `PIN_KEYPAD_COL_1`
- `PIN_KEYPAD_COL_2`

4.3 Programming interface to the simulated Keypad

Your code should include a Keypad class that serves as an interface between the Keypad Controller agent and the simulated keypad. The code in the class should also include the all-important declarations of which the simulated Raspberry Pi pins serve as inputs and outputs to the keypad code.

With physical hardware, interactions between the R-Pi and keypad require the General Purpose Input/Output (GPIO) Python module for R-Pi. In the original GPIO programming you should

```
import RPi.GPIO as GPIO
```

When working with the simulator, you should do the following instead

```
from GPIOSimulator_v5 import *
GPIO = GPIOSimulator()
```

To initialize the row and column pins, these three operations are necessary:

1. For each row pin, rp, do:

   ```
   GPIO.setup(rp, GPIO.OUT)
   ```

2. For each column pin, cp, do:

   ```
   GPIO.setup(cp, GPIO.IN, state=GPIO.LOW)
   ```

4.4 Polling the keypad

It is important to notice that the (simulated) keypad cannot directly tell you which key is pressed. To determine which (if any) key is currently being pressed, we will use a *polling* process in which different combinations of rows and columns are stimulated and sampled. This is implemented as a pair of nested loops, where the outer loop cycles through the keypad rows, setting them HIGH one at a time. Once a row (R) is stimulated, the inner loop cycles through the columns and reads them as input, one at a time. If column C has a HIGH reading, then the key at location (R,C) is revealed as the one being pressed. Simple calculations, and/or a dictionary can be used to map (R,C) pairs to the 12 key symbols.

During the actual polling process, the command to set a row pin (rp) to high is:

```
GPIO.output(rp,GPIO.HIGH)
```

and the code to check whether a column pin (cp) is high is:

```
GPIO.input(cp) == GPIO.HIGH
```

Note that during polling, only one row pin should be high at a time. To reset a row pin to low, use:

```
GPIO.output(rp,GPIO.LOW)
```

To get the sequence of pressed keys, you must iteratively run the above polling to detect the pressed events. Import Python `time` package and use `time.sleep()` to control the delay between polls.

A simple loop is not enough for robust detection because it is sensitive to the polling frequency (controlled by the sleep duration). If the polling frequency is too low, you may miss some pressed keys. To overcome this, your program should be able to unify the duplicated pressing into a single one. This is could be done by taking pressing duration into account as well. The duration could, for example, be obtained by recording the beginning press and ending press (i.e. nothing is pressed).

In this project we do not add randomness to the simulated keypad. But if your computer keyboard gives noisy input, you may also consider repeated polling in high frequency (e.g. with a 10 millisecond delay between polls) and make a majority voting after the polls to remove some "ghost pressing".

5 The Charlieplexed Breadboard

A Charlieplexed circuit board allows individual on/off control of many devices (e.g. LEDs) using just a few pins, where each pin has adjustable input/output settings.

5.1 The physical Charlieplexing circuit

Consider the circuit diagram (without resistors) in Figure 7. If we declare pins 0 and 1 as output, and pin2 as input, then we are effectively allowing the voltages of pins 0 and 1 to drive the circuit, while pin2 nearly shuts down: not enough current flows through it to run devices along its portion of the circuit. This isolates board activity to LEDs 0 and 1.

Now, if we set pin0 high and pin1 low, then significant current will only flow through LED 0, lighting it.[1]. Conversely, if we set pin1 high and pin0 low, LED 1 lights up. Similar logic enables us to selectively light LEDs 2 and 3 by declaring pin 0 as input (neutralizing it) and pins 1 and 2 as output. Finally, with pin1 as input and pins 0 and 2 as output, LEDs 4 and 5 can be lit.

Charlieplexing permits the individual control of $N(N-1)$ devices using only N pins. That relationship derives from a straightforward analysis: If you have N pins, then any pair of them can be used to control 2 different devices. There are "N choose 2" possible pairs of pins, which is:

$$\frac{N!}{(N-2)!(2)!} = \frac{N(N-1)}{2} \tag{1}$$

Multiply that by 2 (for 2 devices per pair), and you get $N(N-1)$ total devices controllable by N pins. Thus, 4 pins are enough to control 12 LEDs, while 5 can control 20, etc. Of course, the circuit layouts for these larger configurations probably look like spaghetti; the nice planar layout of the 6-LED case cannot be maintained as N increases[2].

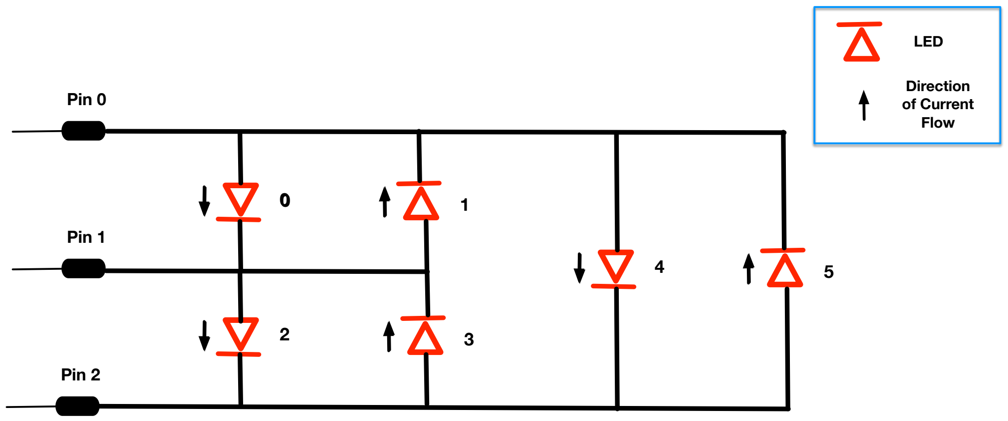

Figure 7: A Charlieplexed circuit with 3 pins and 6 LEDs. No resistors are drawn for ease of illustration. Arrows indicate the only legal direction of current flow for each LED, but these flows only occur under the proper pin settings; e.g. lighting LED 0 requires pin 0 to be high and pin 1 to be low.

5.2 Implementing the Charlieplexer

As with the keypad, the Charlieplexer should have an interface object that links it to the agent. Thus, all requests to perform different lighting sequences on the LED Board should go from the agent to the board via this interface object.

[1]LEDs permit current flow in only one direction

[2]If you want to know more about the circuit, you can check `http://razzpisampler.oreilly.com/ch04.html` for more details.

The simulator mimics the wiring in Figure 7. Your program controls Pins 0 to 2 through the following predefined pin constants, respectively

- `PIN_CHARLIEPLEXING_0`
- `PIN_CHARLIEPLEXING_1`
- `PIN_CHARLIEPLEXING_2`

We provide an extra function `GPIO.show_leds_states()` in the simulator. You can call the function whenever you want to see the LED states. For example, when only the 5th LED is on, a call to the function will print the following on screen:

```
LEDs[ 0:  OFF, 1:  OFF, 2:  OFF, 3:  OFF, 4:  OFF, 5:  ON ]
```

Similar to RPi.GPIO, you can use the simulated GPIO to set the pin modes and states back and forth to control the simulated LEDs So whichever of the 6 LEDs we want to light, there will be two pins chosen as output and one for input; and one of the output pins will be set HIGH, the other LOW. The pin settings corresponding to each LED should be saved in a dictionary or array such that when the agent requests lighting of the kth LED, your code can fetch the kth setting set and send it to a method that performs the proper IN/OUT and HIGH/LOW assignments. All of this can be done with 10 or 15 lines of Python code.

To turn on or turn off a specific LED, we need a correct combination of pin modes and pin states. This can be done by one or more of the following commands:

- GPIO.setup(pin, GPIO.IN)
- GPIO.setup(pin, GPIO.OUT)
- GPIO.output(outpin,GPIO.HIGH)
- GPIO.output(outpin,GPIO.LOW)

Note that the Charlieplexer here is only for output. The command `GPIO.setup(pin, GPIO.IN)` actually says we are not using the specific pin in the lighting action.

After you implement basic turn-on/turn-off actions, you should also implement the following mandatory behaviors for your simulated LED board:

1. A display of lights (of your choosing) that indicates "powering up". This should be performed when the user does the very first keystroke of a session.
2. Flashing of all lights "in synchrony" when the user enters the wrong password during login.
3. Twinkling the lights (in any sequence you choose) when the user successfully logs in.
4. A fourth type of light display (of your choosing) that indicates "powering down". This should be performed immediately after the user logs out.
5. Turn one user-specified LED on for a user-specified number of seconds, where information about the particular LED and duration are entered via the simulated keypad.

Each of these activities should be initiated by a different method, callable by the agent.

6 Primary Python Classes, Instance Variables, and Methods

This section provides a basic class structure and functionality, but you will almost certainly implement more methods (and possibly more classes) as you see fit. Here only some method names are listed. The actual arguments to these methods may vary depending upon your exact implementation.

The four main classes should be:

1. Finite State Machine (FSM)
2. Keypad - interface to the simulated keypad
3. Led Board - interface to the simulated Charlieplexed LED board.
4. KPC - the keypad controller agent that coordinates activity between the other 3 classes along with verifying and changing passwords.

6.1 Finite State Machine

An FSM object should house a pointer back to the agent, since it will make many requests to the agent (KPC) object.

Key methods for the FSM include:

- `add_rule` - add a new rule to the end of the FSM's rule list.
- `get_next_signal` - query the agent for the next signal
- `run` - begin in the FSM's default initial state and then repeatedly call `get_next_signal` and run the rules one by one until reaching the final state.

There are two key methods of a FSM rule

- `match` - check whether the rule condition is fulfilled
- `fire` - use the consequent of a rule to a) set the next state of the FSM, and b) call the appropriate agent action method.

6.2 Keypad

The keypad has a few essential methods:

- `setup` - initialize the row pins as outputs and the column pins as inputs.
- `do_polling` - Use nested loops (discussed above) to determine the key currently being pressed on the keypad.
- `get_next_signal` - This is the main interface between the agent and the keypad. It should initiate repeated calls to do_polling until a key press is detected.

6.3 LED Board

The Charlieplexed LED Board has these methods:

- `light_led` - Turn on one of the 6 LEDs by making the appropriate combination of input and output declarations, and then making the appropriate HIGH / LOW settings on the output pins.

- `flash_all_leds` - Flash all 6 LEDs on and off for k seconds, where k is an argument of the method.
- `twinkle_all_leds` - Turn all LEDs on and off in sequence for k seconds, where k is an argument of the method.

In addition, you will need methods for the lighting patterns associated with powering up (and down) the system.

6.4 KPC Agent

A KPC agent is the main object in your system, so it will have many methods and a few important instance variables. These variables include:

- a keypad instance,
- an LED Board instance,
- the complete pathname to the file holding the KPC's password.
- the `override_signal` (discussed earlier in this document)

Some of the most important KPC methods are the following:

- `reset_passcode_entry` - Clear the passcode-buffer and initiate a "power up" lighting sequence on the LED Board.
- `get_next_signal` - Return the `override_signal`, if it is non-blank; otherwise query the keypad for the next pressed key.
- `verify_login` - Check that the password just entered via the keypad matches that in the password file. Store the result (Y or N) in the `override_signal`. Also, this should call the LED Board to initiate the appropriate lighting pattern for login success or failure.
- `validate_passcode_change` - Check that the new password is *legal*. If so, write the new password in the password file. A legal password should be at least 4 digits long and should contain no symbols other than the digits 0-9. As in verify_login, this should use the LED Board to signal success or failure in changing the password.[3]
- `light_one_led` - Using values stored in the Lid and Ldur slots, call the LED Board and request that LED # Lid be turned on for Ldur seconds.
- `flash_leds` - Call the LED Board and request the flashing of all LEDs.
- `twinkle_leds` - Call the LED Board and request the twinkling of all LEDs.
- `exit_action` - Call the LED Board to initiate the "power down" lighting sequence.

[3]For this assignment, it is sufficient to ask the user to type the new password just ONCE, although twice, as shown by the FSM in Figure 4, is more realistic.

7 A Typical Run of the Keypad Controller

This section walks through one sample session with the Keypad Controller, showing what the user types and how the system responds.

The session begins with the main controller program started up in Python. Once started, the controller will wait for input from the agent. Table 2 shows a sequence of agent inputs (most of which will come from the keypad) and the corresponding LED-board actions and FSM states. These are the states that the FSM moves into after reading input from the agent. For inputs of length 2 or longer, it is assumed that the FSM repeats the corresponding state (such as S_{read} during password entry). Note that this is only an example: the FSM that you design may involve different states and connections between them.

Table 2: Example sequence of agent′ inputs (primarily from the keypad) and system responses.

Activity	Input from Agent	Example	LED Display	FSM State
Wake up System	Any Key Press	8	Power-up light show	S_{init}
Enter Password	Digits of Password	12345	None	S_{read}
Completing Password Entry	*	*	None	S_{verify}
Password Accepted	Y	Y	Twinkling lights	S_{active}
Choose a LED	Digit in 0-5	4	None	S_{led}
Begin Duration Entry	*	*	None	S_{time}
Choose a Duration	Digits	29	None	S_{time}
Complete duration	*	*	LED 4 ON for 29 secs	S_{active}
Choose a LED	Digit in 0-5	2	None	S_{led}
Begin Duration Entry	*	*	None	S_{time}
Choose a Duration	Digits	14	None	S_{time}
Complete duration	*	*	LED 2 ON for 14 secs	S_{active}
Begin Logout	#	#	None	S_{logout}
Confirm Logout	#	#	Power-down light show	S_{done}

In this example, the complete sequence of keypad inputs is:

8 12345 * 4 * 29 * 2 * 14 * # #

whereas one signal "Y" (4th table entry) was provided as an override signal from the agent, thus signaling password acceptance.

In Table 2, the FSM state S_{active} is the main start state for most activities, but successful login is required to reach S_{active}. Also note that all of the agent's inputs come from the keyboard **except** the "Y", which is the agent's *override signal* to the FSM that the password has been accepted. Similarly, an "N" should indicate password rejection and send the FSM back to state S_{init}. Finally, notice that logging out requires two pushes of the # key, one to move from state S_{active} to S_{logout}, and a second to confirm the desire to logout.

8 Demonstration of the Keypad Controller

During the demonstration session for this project, you will be asked to perform the following sequence of actions using the keypad:

1. Press any key and a "powering up" light display begins.
2. Attempt to login, but fail, thus triggering the "failed login" light display.
3. Successfully login, thus triggering the "successful login" light display.
4. Change the password - your system must accept all-digit passwords of length 4 or greater.
5. Logout, thus triggering the "powering down" light display.
6. Press any key to begin the "powering up" light display.
7. Login again, using the new password.
8. Turn on a user-specified LED for a user-specified number of seconds.
9. Turn on a different LED for a different duration.
10. Logout, thus triggering the "powering down" light display.

— o —

References