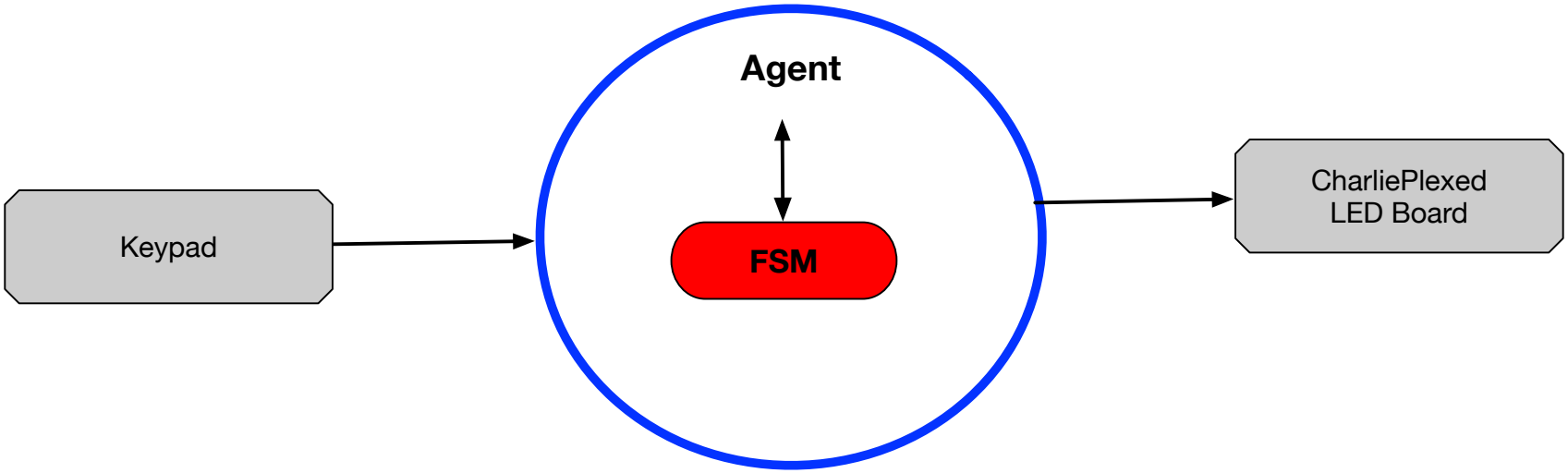
TDT4113 Project 5: Keypad Controller

Practicals of Project 5

- The 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

Architecture of the keypad controller



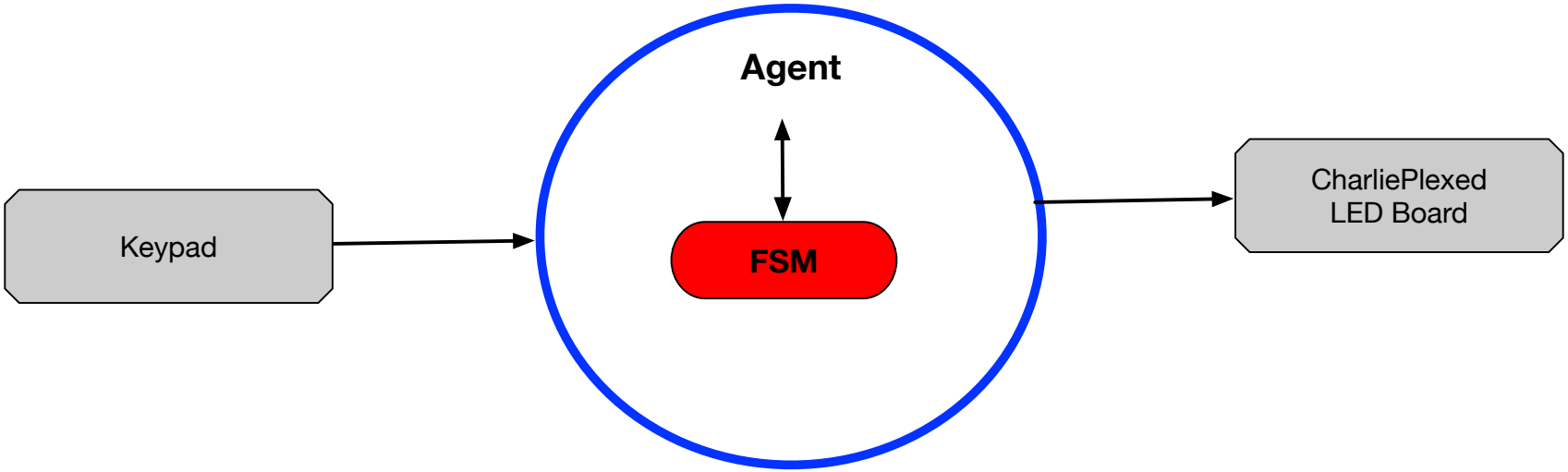
Setup with physical hardware

An example in real life

In your demo

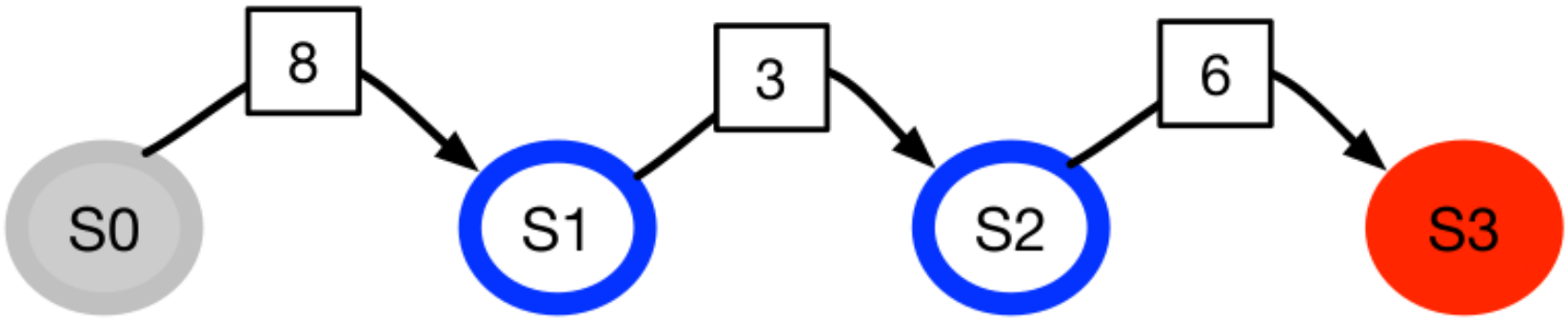
1. Press any key and a “powering up” light display begins.
2. Attempt to login, but fail, thus triggering the “failed login” light display.
3. Successfully login, thus triggering the “successful login” light display.
4. Change the password - your system must accept all-digit passwords of length 4 or 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.

Major steps



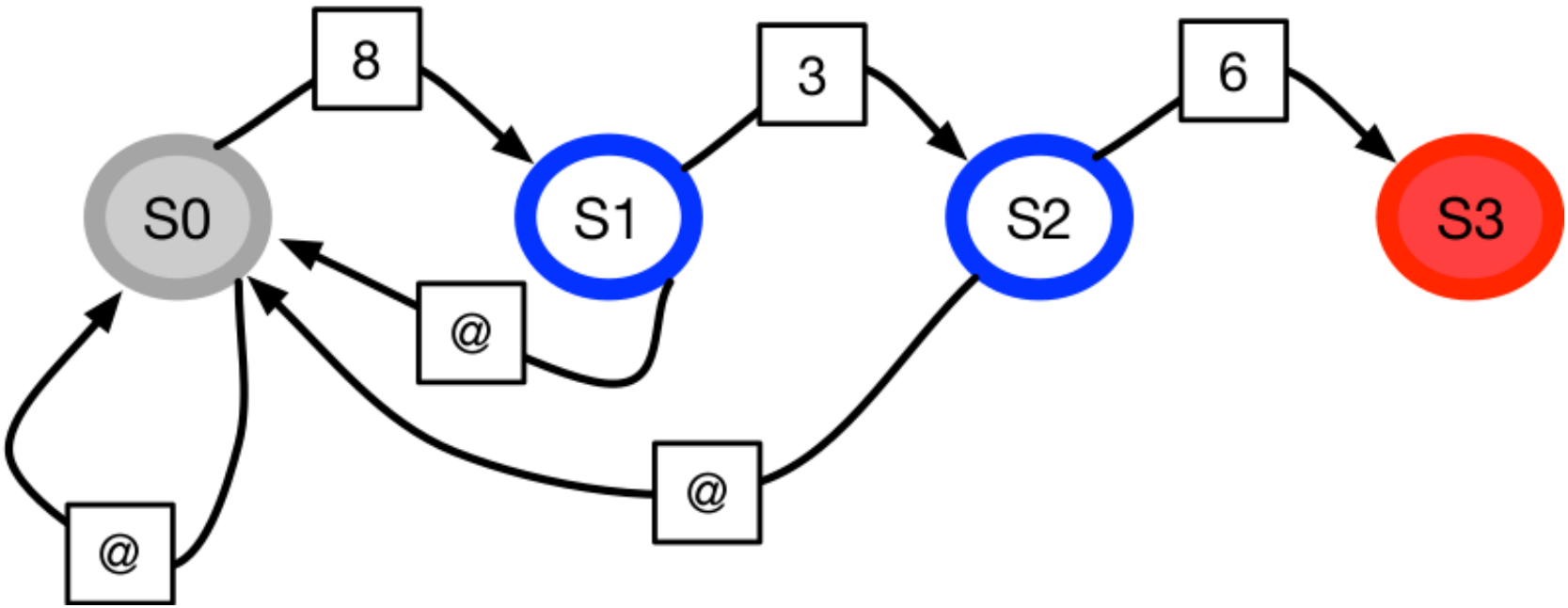
- **Finite State Machine (FSM)**
- Keypad
- Charlieplexed LED board
- Integration

Finite State Machine (FSM)



- FSMs are often depicted as graphs
- Each node represents a unique state of the system
- Arc labels denote the external signal that triggers the transition from one state to another.

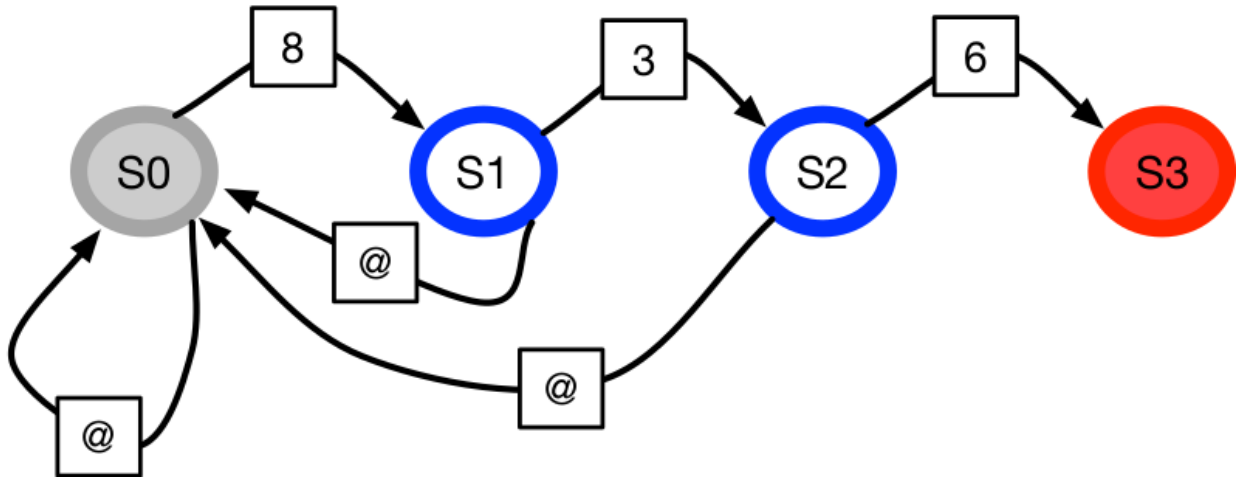
FSM with mistakes considered



Implementing FSM

- Directed graph is a good way to represent the concept
- We need to convert it to computer program
- There are many alternatives
- Rule-Based System (RBS) is a straightforward approach
  - RBS loops over a list of rules
  - Each rule has the structure:
    - **IF condition THEN consequent**
  - If the condition of a rule is not satisfied, run the next rule
  - The order of the rules matters

RBS for the simple FSM example



IF fsm.state == S0 AND fsm.signal == 8 THEN fsm.state ← S1

ELIF fsm.state == S0 AND fsm.signal != 8 THEN fsm.state ← S0

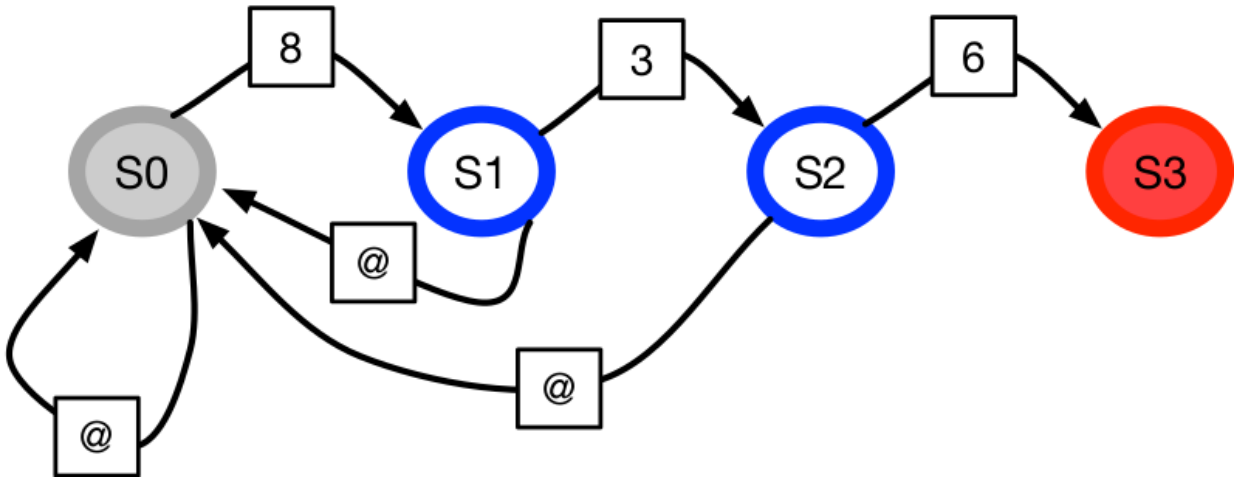
ELIF fsm.state == S1 AND fsm.signal == 3 THEN fsm.state ← S2

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

Simplify a bit



IF fsm.state == S0 AND fsm.signal == 8 THEN fsm.state ← S1

ELIF fsm.state == S0 THEN fsm.state ← S0

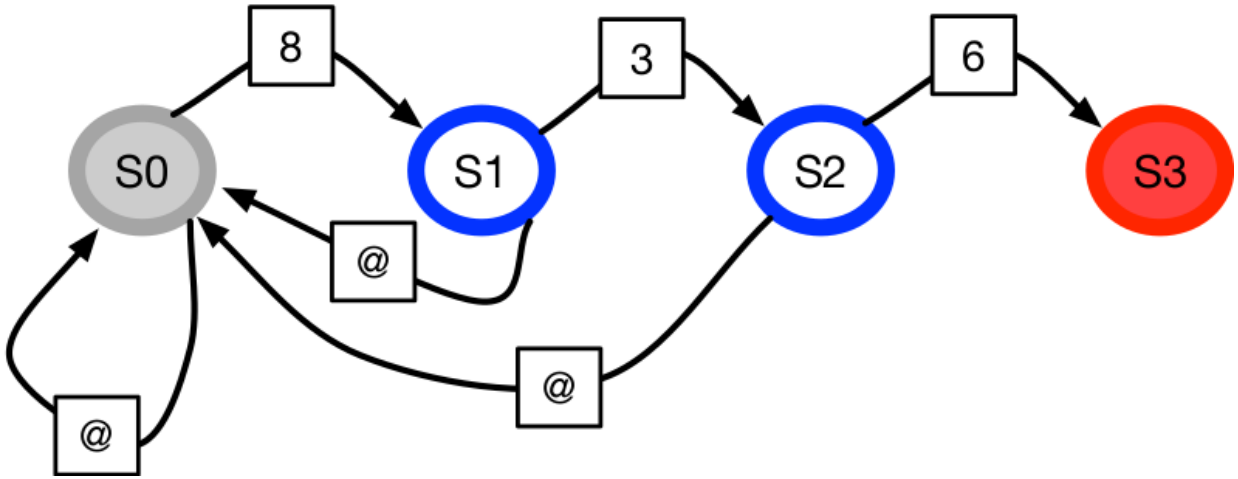
ELIF fsm.state == S1 AND fsm.signal == 3 THEN fsm.state ← S2

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

Further simplify



- Principle: specific-before-general
- Unify to a more general "Everything but" rule in the end

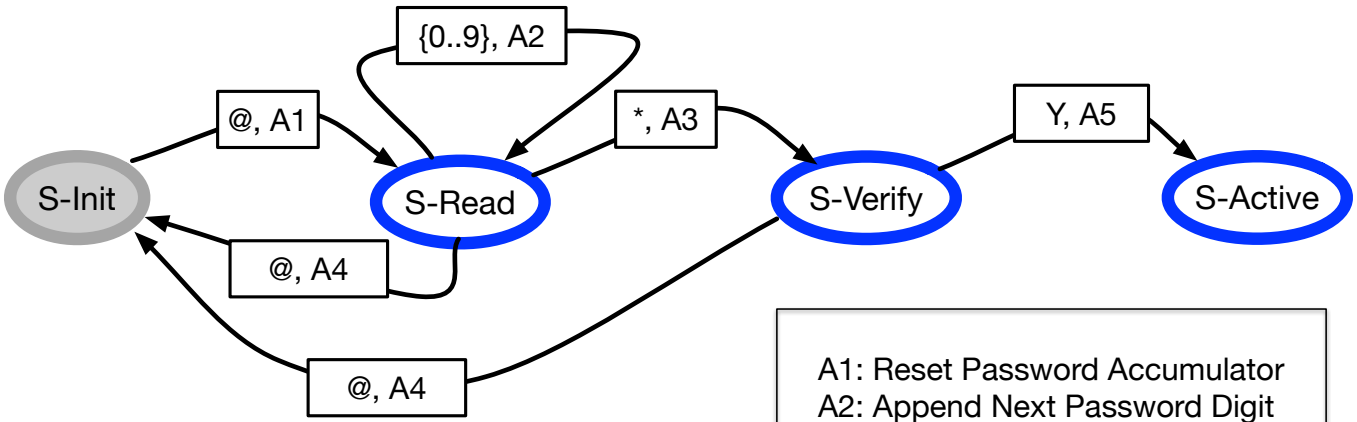
IF fsm.state == S0 AND fsm.signal == 8 THEN fsm.state ← S1
ELIF fsm.state == S1 AND fsm.signal == 3 THEN fsm.state ← S2
ELIF fsm.state == S2 AND fsm.signal == 6 THEN fsm.state ← S3
ELSE fsm.state ← S0

Interaction between the Agent and FSM

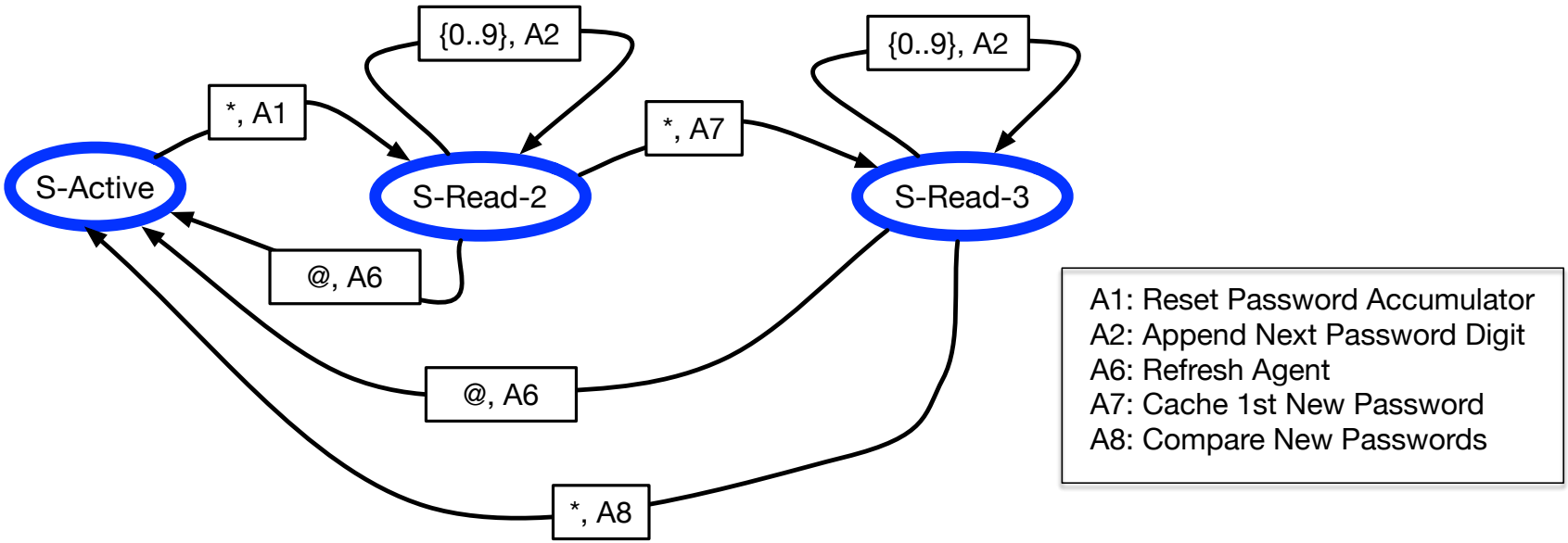
- The controller is not only about state jumping
- It should include some *actions*, e.g., verifying password and resetting the agent.
- The Keypad FSM may operate as follows
  - The FSM begins in state S-Init
  - When the user hits any keypad button, the FSM switches to a "read password" state (S-Read).
  - Each digit entered during this S-Read is passed to the agent, stored in a cumulative-password (CUMP) array or string.
  - When the user enters a "*" (asterisk)
    - The system should verify the password by comparing CUMP to CP
    - The FSM moves into a new state (S-Verify).
    - The next state depends on the verification succeeded or not: goto S-Active if succeeded or goto S-Init otherwise.

Include actions in rules

- An FSM rule: **IF condition THEN consequent**
  - In the "836" example: IF condition THEN new state
  - Now we need: IF condition THEN new state and action
- Directly code the actions in FSM is bad
  - would make the FSM code messy and difficult to maintain
- Better to separate the FSM and the agent
  - Code each action as in a function in the agent
  - Action in FSM becomes a call to the function



FSM for password changing



- The user must first login with old password (S-Read-2)
- and then supply the new password (S-Read-3)
- Need storage of
  - cumulative password (CUMP)
  - last value of cumulative password (CUMP-OLD)

Realize FSM as RBS

- A rule object has
  - state1 - triggering state of the FSM
  - state2 - new state of the FSM if this rule fires
  - signal - triggering signal
  - action - the agent will be instructed to perform this action if this rule fires.
- Meaning of a rule
  - IF the FSM is in rule.state1 and the current signal matches rule.signal THEN change the FSM's state to rule.state2 and call `rule.action(agent, signal)`.
- The RBS (i.e. the FSM) cycles through its list of rules

Implementation details

- IF the FSM is in rule.state1 and the current signal matches rule.signal THEN change the FSM's state to rule.state2 and call `rule.action(agent, signal)`.
- the red part is more subtle
  - "matches" is not simply "=="
    - e.g. rule.signal can be a set; "matches" means "$\in$"
  - rule.signal can be outside the 12 keys
    - e.g. it can be password verifying result ("Y" or "N")
- Better to implement the red part as a function
  - to accommodate various situations
  - returns True/False to indicate "matched" or not
- Use an additional variable `override_signal`
  - `get_signal()` first checks `override_signal`
  - and uses it if it is not None
  - otherwise gets signal from keypad
  - resets it to None before return

Pseudocode of overall system

- Create all rule objects for the FSM - and ensure they are listed in the desired order.
- Initialize agent, keypad and LED board
- state $\leftarrow$ fsm-start-state
- While state $\neq$ fsm-end-state do:

```
symbol ← agent.get_next_symbol()
For each rule in fsm.rules:
        IF rule.match(state,symbol):
                state ← rule.state2
                agent.do_action(rule.action, symbol)
                GO TO (**)
End For
(**)
```

- End While
- Shutdown agent, keypad, LED board, etc.

An option to code `agent.do_action()`

- Assume your agent is an instance of KPC (keypad controller)
- and it has a `start_password_entry` method.
- Assume rule R3 declares: when in state S0 and reading symbol "*" (asterisk), change to state S1 and invoke `agent.start_password_entry()`.
- R3 would then be defined as a rule with

  rule.action = KPC.start_password_entry

- The actual Python syntax for `agent.do_action()` becomes:

  rule.action(agent,symbol)

- The action slot of every rule in your FSM should be set to KPC.zzzz, where zzzz is a different method name.

A rule set corresponding to the FSM graph

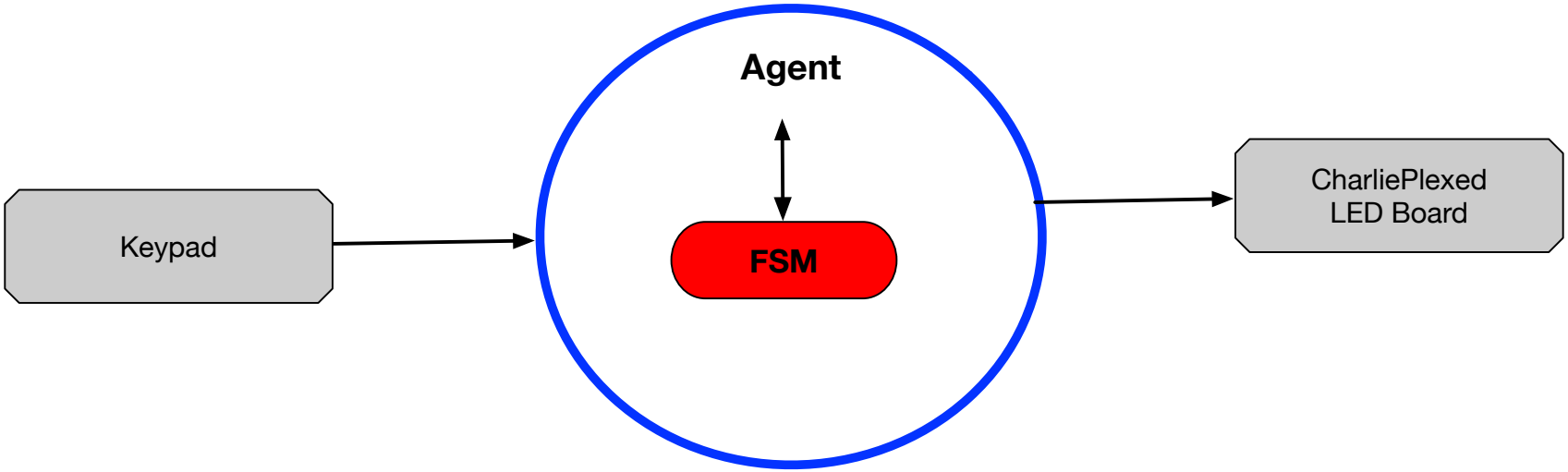
Index	State-1	Symbol	State-2	Action
1	S-init	all_symbols	S-Read	KPC.reset_password_accumulator
2	S-Read	all_digits	S-Read	KPC.append_next_password_digit
3	S-Read	*	S-Verify	KPC.verify_password
4	S-Read	all_symbols	S-init	KPC.reset_agent
5	S-Verify	Y	S-Active	KPC.fully_activate_agent
6	S-Verify	all_symbols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

The order of the rules is extremely important!

Design and debugging tips

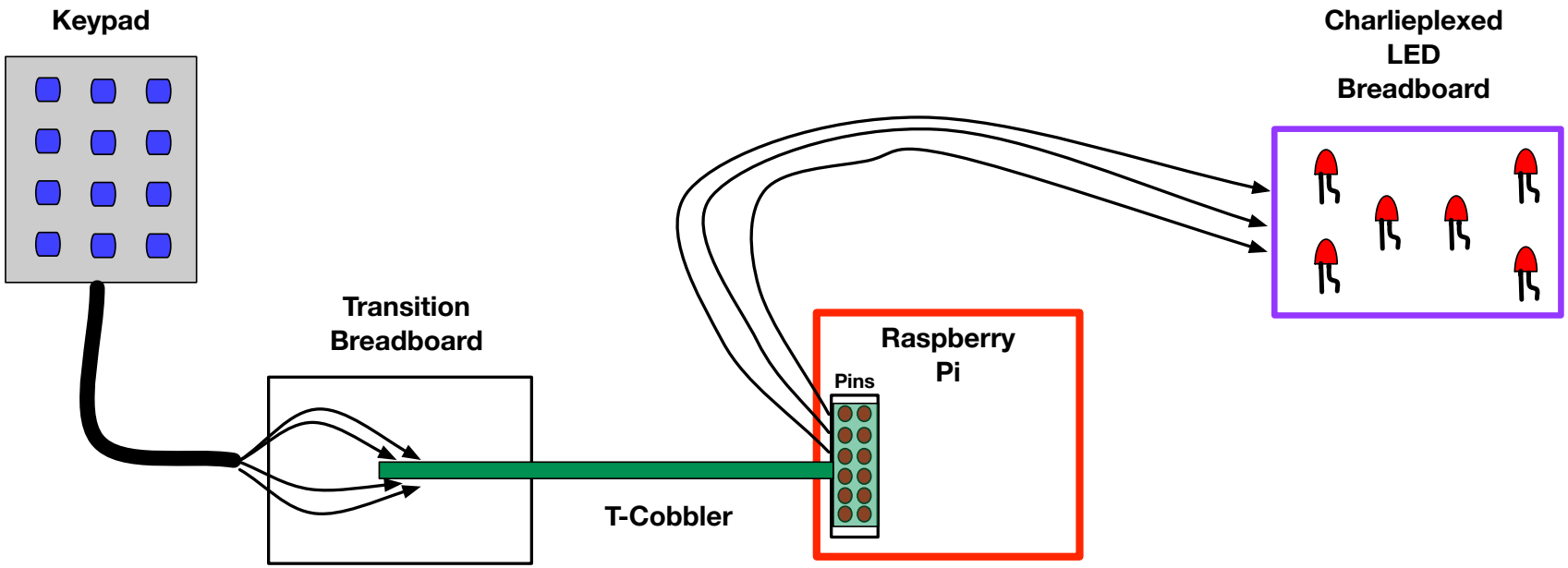
- Draw the graph first
- FSM states can be str./int.; symbols (signals) are str.
- Divide and Conquer!
  - The project comprises three major complex components
  - Isolate the potential issues in unit test
    - When testing a component, use simple fakes of the other two
    - e.g. when testing FSM, use a proxy (fake) keypad directly with Python `input()` function to get a key
    - and use a proxy (fake) LED breadboard with direct `print()`
  - Replace the fake with the real when each component is ready
  - and do the final integration test

Major steps

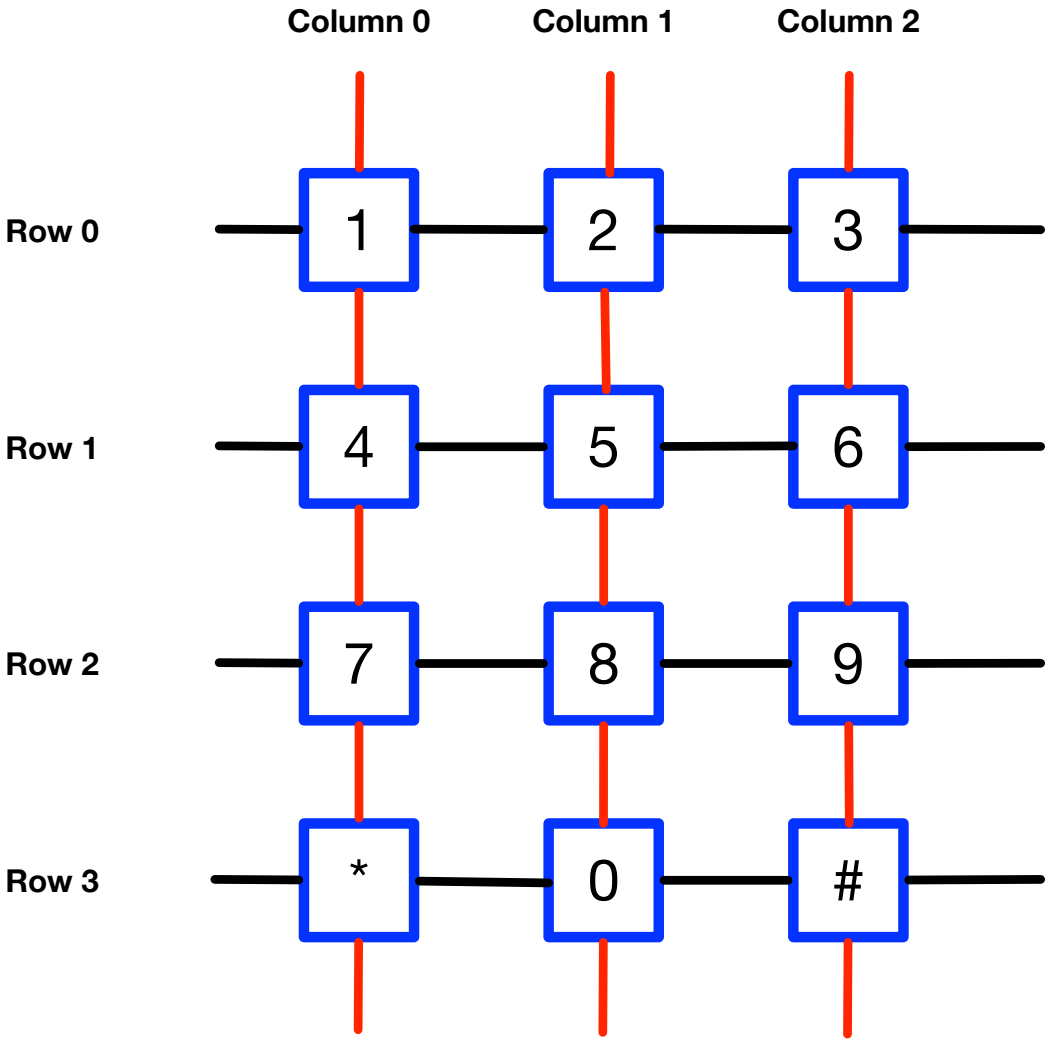


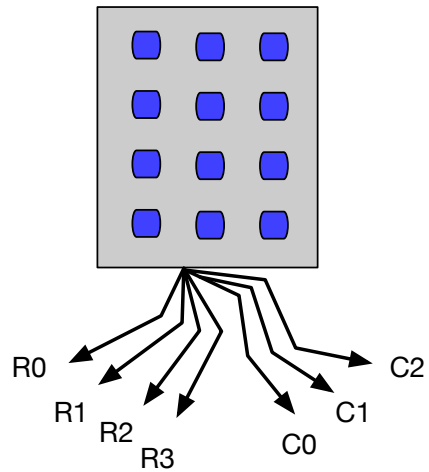
- Finite State Machine (FSM)
- **Keypad**
- Charlieplexed LED board
- Integration

Setup of The Keypad Controller



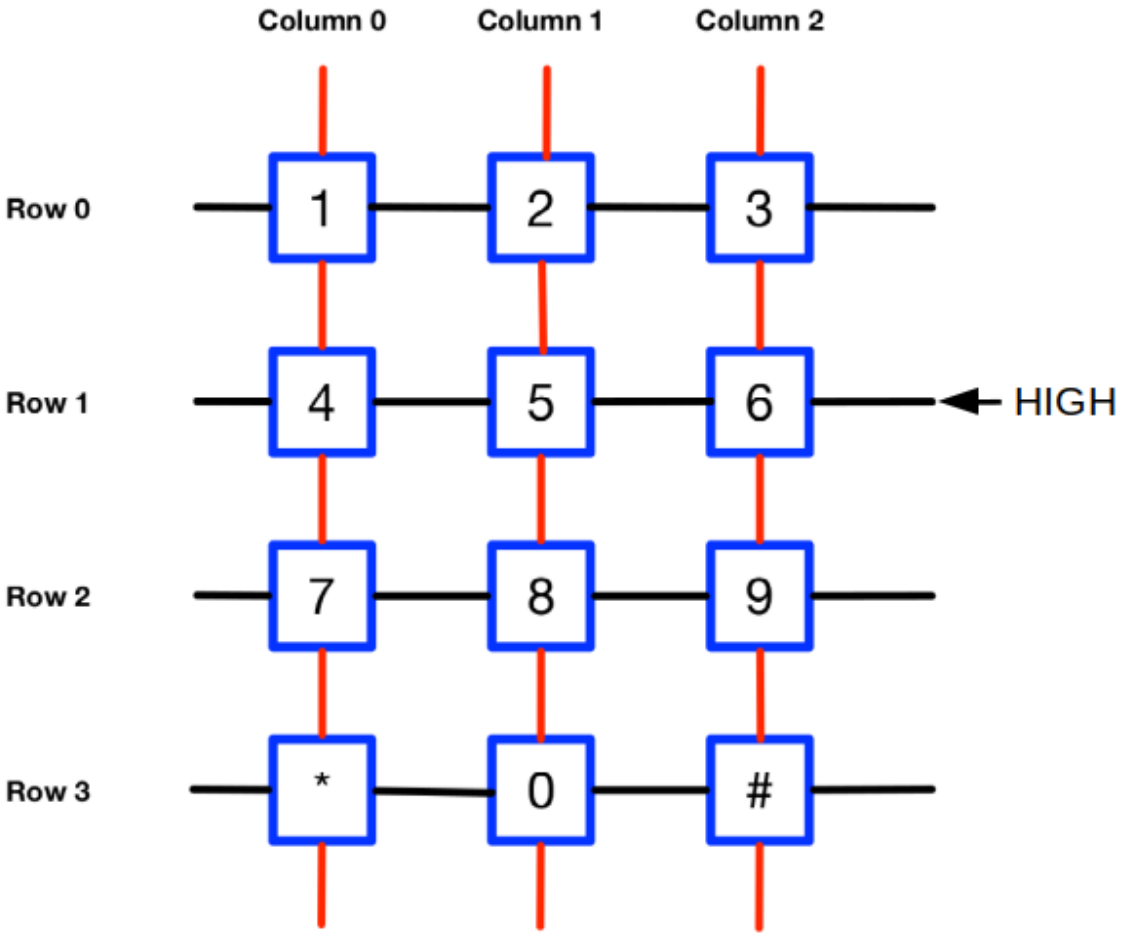
The physical keypad

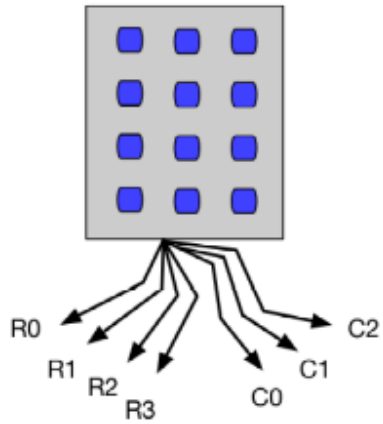




NB! The keypad cannot directly tell the pressed key!

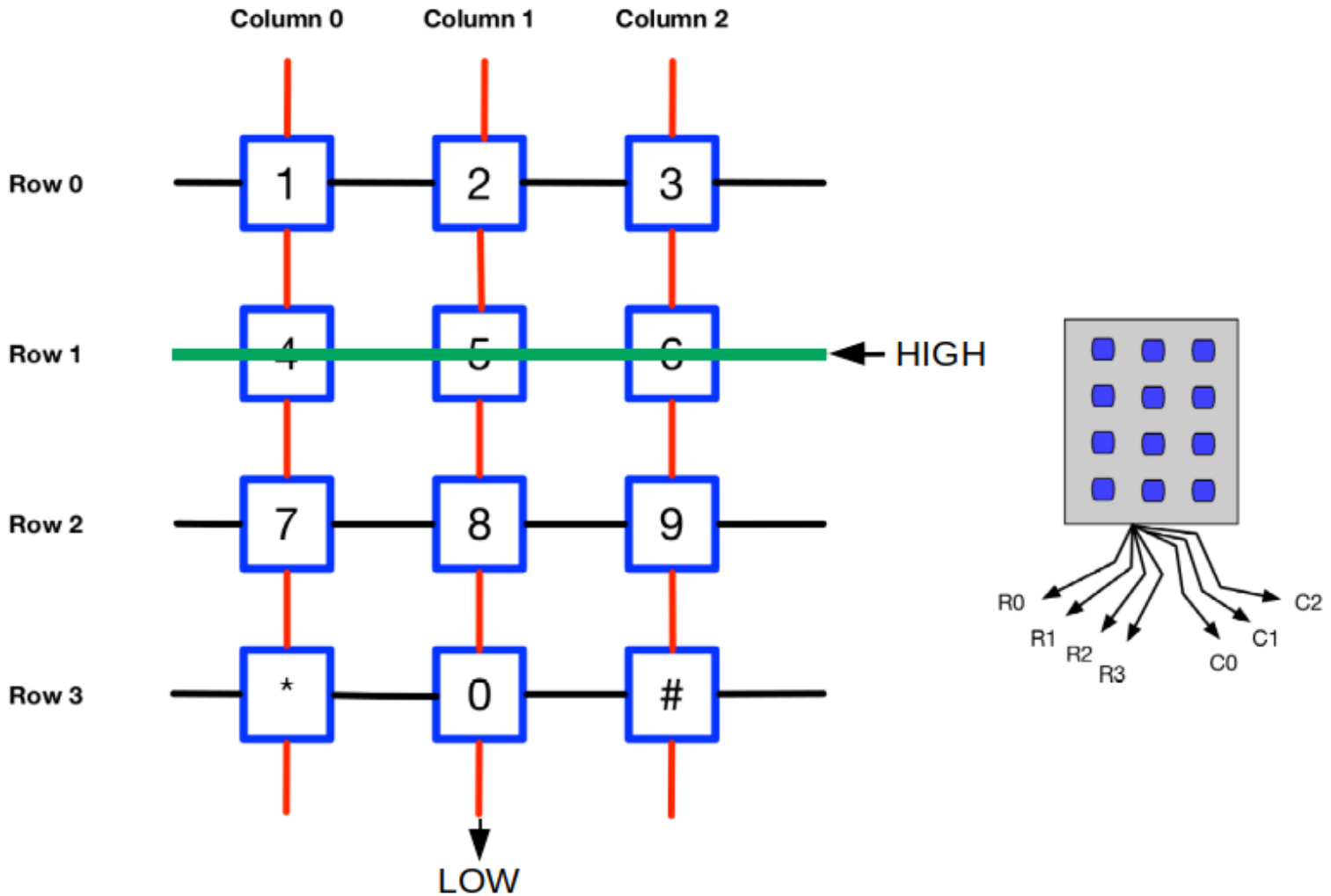
Polling the keypad





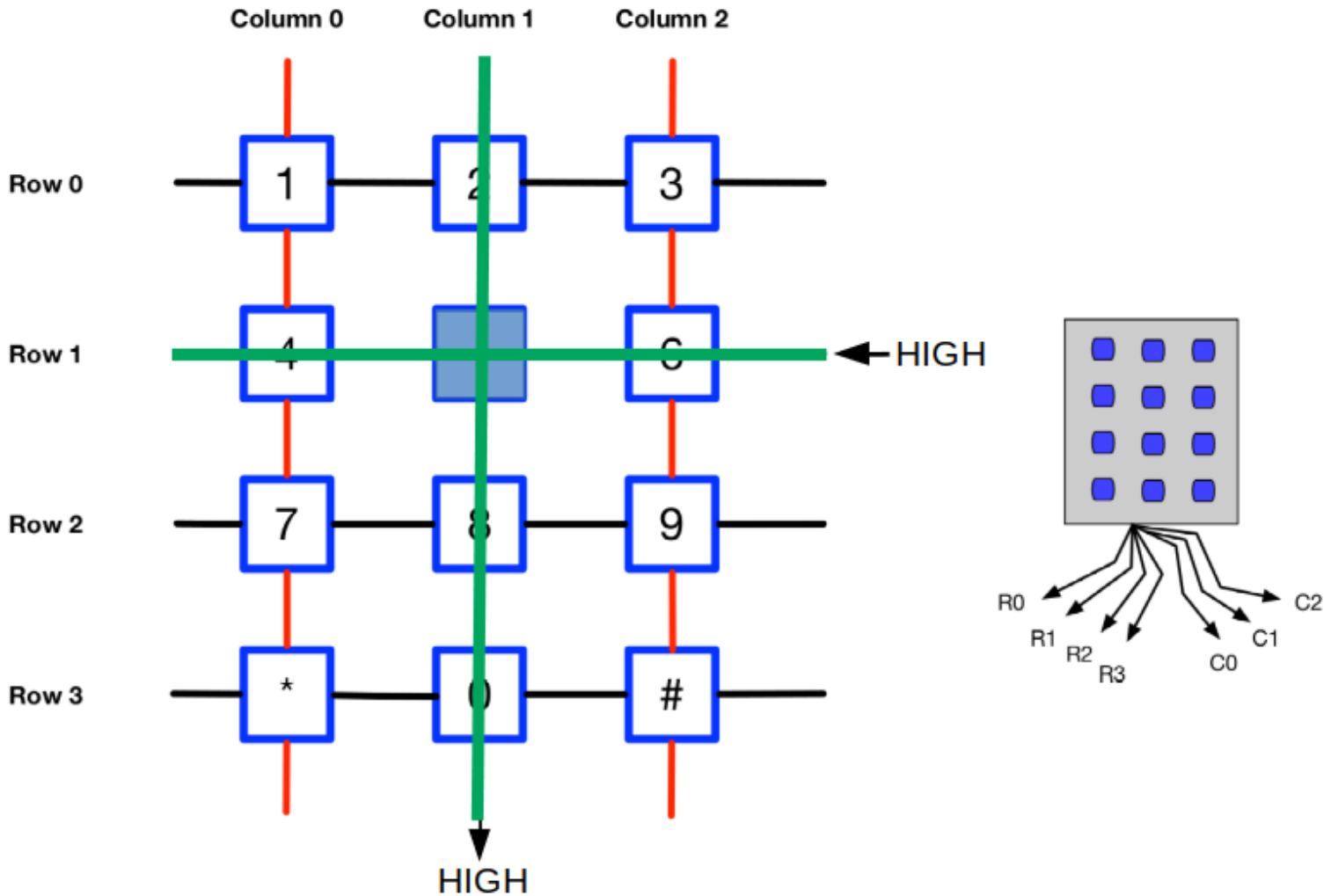
Polling the keypad

Green line shows the current when no key is pressed



Polling the keypad

Green lines show the current when "5" is pressed



The simulator

- Due to COVID, we use simulated hardware in the project
- The simulator is in `GPIOSimulator_v5.py`
- Replace `import RPi.GPIO as GPIO`
- with

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

- Then interact with the simulated keypad with one or more of the following functions
  - `GPIO.setup(pin, mode, state)`
  - `state = GPIO.input(pin)`
  - `GPIO.output(pin, state)`
- At the end call `GPIO.cleanup()`

Pins for the simulated keypad

The following pin constants correspond to the R0-R3 and C0-C2 in Figure

- `PIN_KEYPAD_ROW_0`
- `PIN_KEYPAD_ROW_1`
- `PIN_KEYPAD_ROW_2`
- `PIN_KEYPAD_ROW_3`
- `PIN_KEYPAD_COL_0`
- `PIN_KEYPAD_COL_1`
- `PIN_KEYPAD_COL_2`

Example usage:

- `GPIO.setup(PIN_KEYPAD_ROW_1, GPIO.OUT)`
- `GPIO.output(PIN_KEYPAD_ROW_1, GPIO.HIGH)`

Pseudocode for the polling function

Set row pins for output and column pins for input

Loop over each row pin `rp`

    Output high state to `rp`
    Loop over each column pin `cp`

        Read state from `cp`
        If the read state is high then return (rp, cp)

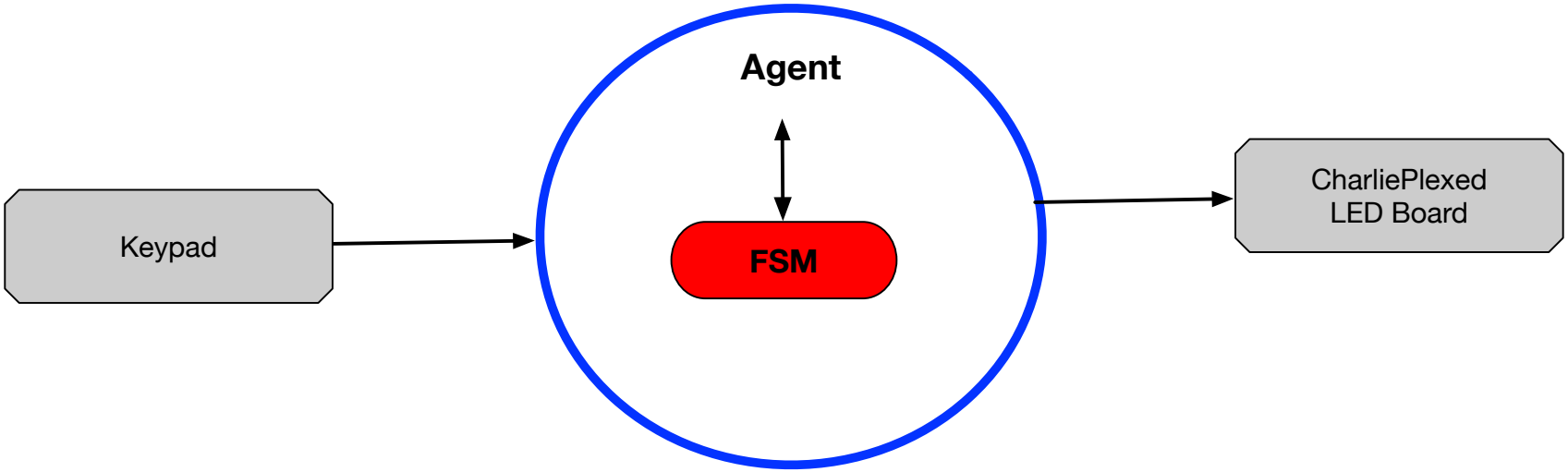
    Output low state to `rp`

Return a tuple which indicates "no pressed keys"

Some more details about key detection

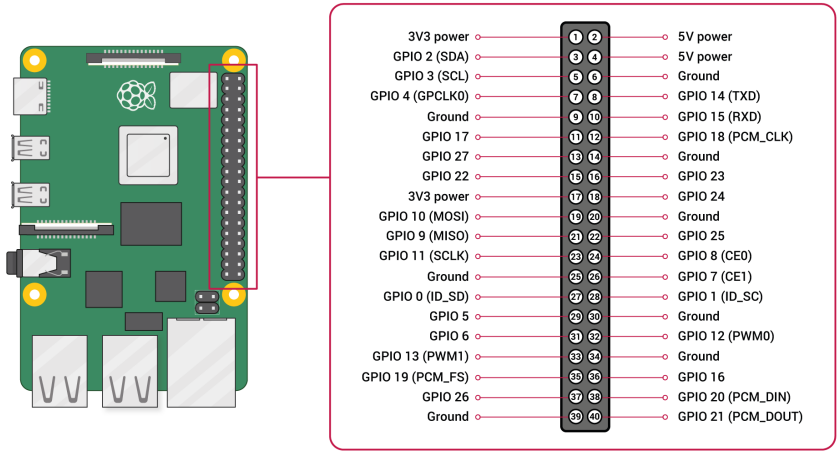
- Iteratively run the polling to detect the pressed sequence
  - Import Python `time` package
  - Use `time.sleep()` to control the detection frequency
- A simple loop is not enough for robust detection
  - polling frequency too low $\Rightarrow$ miss some pressed keys.
  - polling frequency too high $\Rightarrow$ get repeated pressing duplicates
    - Unify the duplicated pressing into a single one.
    - Could be done by taking pressing duration into account
- No randomness added to the simulated keypad
  - but you still need to make your detection as robust as possible

Major steps



- Finite State Machine (FSM)
- Keypad
- **Charlieplexed LED board**
- Integration

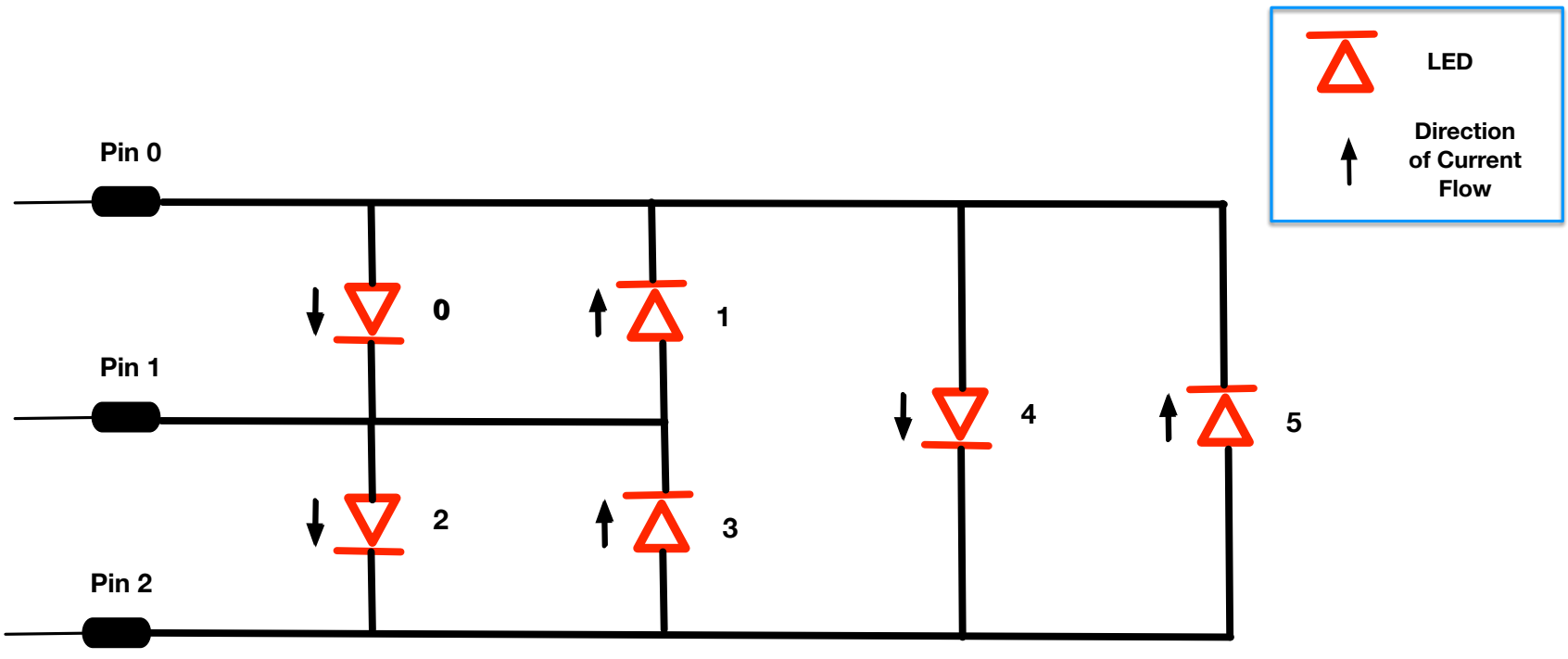
Charlieplexing (CP)



- Charlieplexing is a technique to control many devices with a few I/O pins.
  - naive wiring: use N pins for only $N/2$ devices
  - with CP: use N pins for $N(N-1)$ devices
- Note CP is not a device
- CP = special wiring + some programming

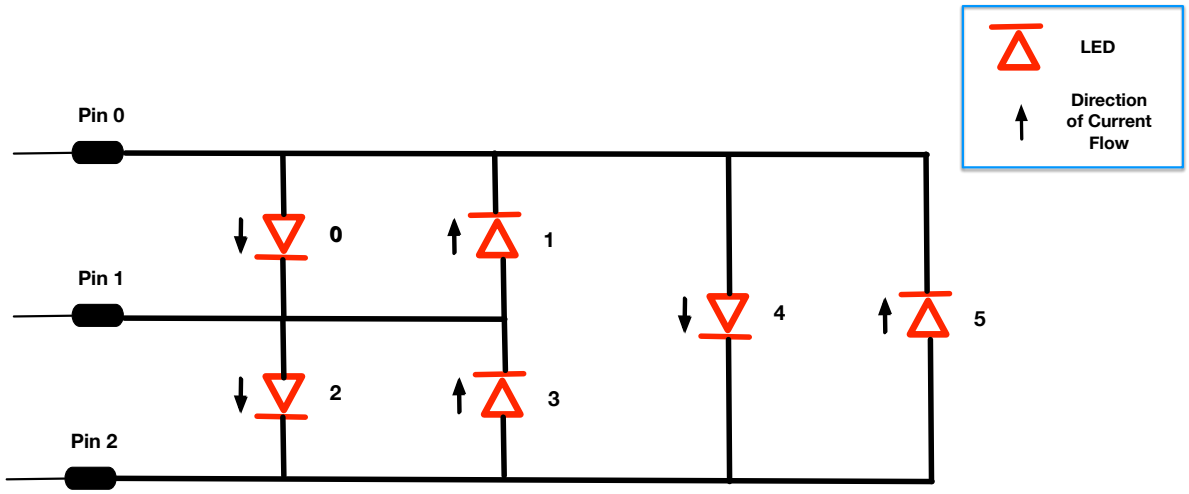
CP in this project

- You don't need physical wiring in this project
- Just fous on the following circuit diagram
- A Charlieplexed circuit with 3 pins and 6 LEDs



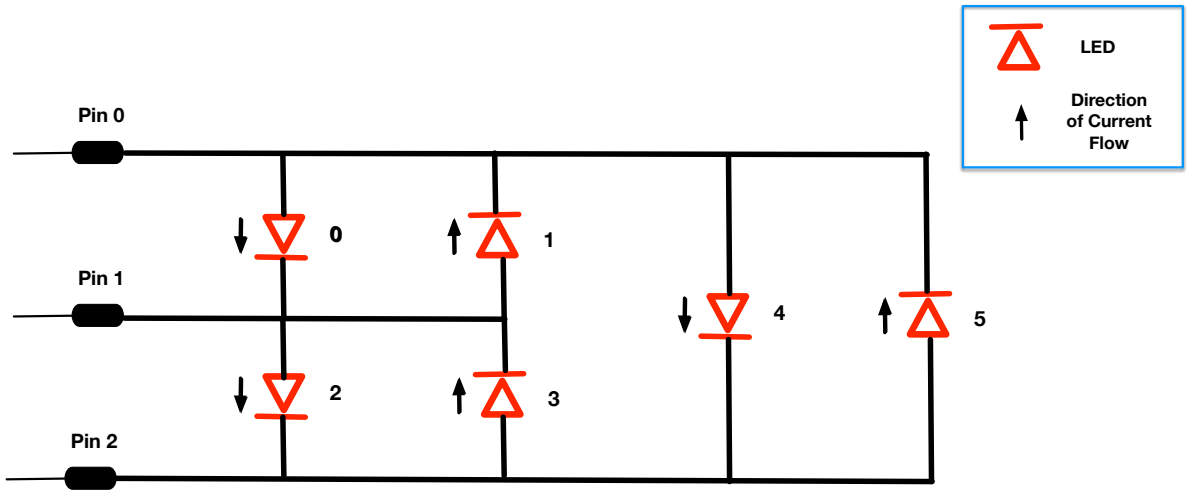
Note: the current cannot light more than one LED in a row.

Programming the CP



- Set pin modes/states to turn on/off a specific LED
- For example to light LED 1
  - Pin 0: mode← OUT, state←LOW
  - Pin 1: mode← OUT, state←HIGH
  - Pin 2: mode← IN, state←LOW/HIGH
  - Note: setting a LED breadboard pin mode to IN means we do not use it for the specific output operation

Programming the CP



- Set pin modes/states to turn on/off a specific LED
- For example to light LED 1
  - Pin 0: mode$\leftarrow$ OUT, state$\leftarrow$LOW
  - Pin 1: mode$\leftarrow$ OUT, state$\leftarrow$HIGH
  - Pin 2: mode$\leftarrow$ IN, state$\leftarrow$LOW/HIGH
  - Note: setting a LED breadboard pin mode to IN means we do not use it for the specific output operation
- Question: How can we turn on LED 4?

Simulator interface

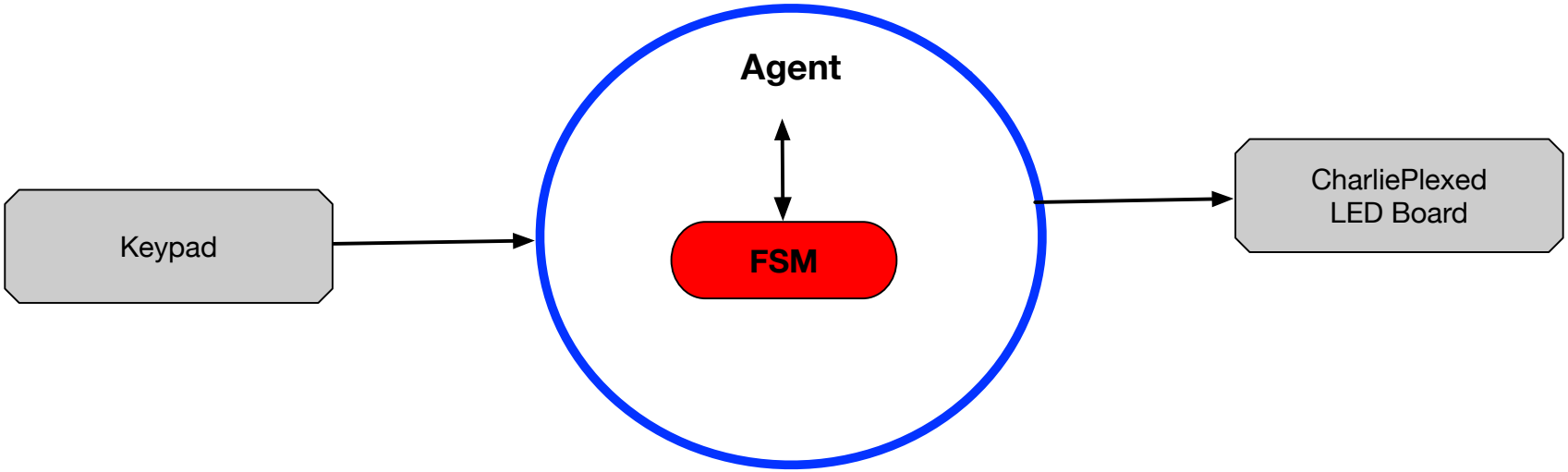
- The constants correspond to the pins in the figure
  - `PIN_CHARLIEPLEXING_0`
  - `PIN_CHARLIEPLEXING_1`
  - `PIN_CHARLIEPLEXING_2`
- Then the mode or state setting goes like
  - `GPIO.setup(PIN_CHARLIEPLEXING_1, GPIO.OUT)`
  - `GPIO.output(PIN_CHARLIEPLEXING_1, GPIO.HIGH)`
- We provide `GPIO.show_leds_states()`
  - You 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 ]
    ```

Your "LED board" should support

- A display of lights (of your choosing) for "powering up".
- Flashing of all lights in synchrony for failed login
  - e.g. duration=0.01
- Twinkling the lights (in any sequence) for successful login
  - e.g. duration=0.1
  - A type of light display of your choosing for "powering down"
  - 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.
- Some "of your choosing" options: flash/twinkle some (not all) of the lights, or progressively flash/twinkle more and more LEDs, etc.
- Each of these activities should be initiated by a different method, callable by the agent.

Major steps



- Finite State Machine (FSM)
- Keypad
- Charlieplexed LED board
- **Integration**

The major classes

- Finite State Machine (FSM)
- Keypad - interface to the simulated keypad
- Led Board - interface to the simulated Charlieplexed LED board.
- KPC - the keypad controller agent that coordinates activity between the other 3 classes along with verifying and changing passwords.
- The project description summarizes some major functions in each class

Finite State Machine

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.

Key methods of an FSM

- `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.

Keypad

Key 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` - main interface between the agent and the keypad. It should initiate repeated calls to `do_polling` until a key press is detected.

LED Board

Key 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.
- Other lighting patterns (e.g. powering up and powering down)

KPC Agent - the main object in your system

Some member variables

- a keypad instance,
- an LED Board instance,
- the complete pathname to the file holding the KPC's password.
- the `override_signal`

Some member methods

- `reset_passcode_entry`
- `get_next_signal` - Return the `override_signal`, if it is non-blank; otherwise get the next pressed key.
- `verify_login`
  - Check whether the entered password matches that in the password file
  - Store the result (Y or N) in the `override_signal`
  - Show appropriate lighting pattern for login success or failure.

KPC Agent (continued)

Some other member functions

- `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.
  - Show appropriate LED pattern for success or failure in password changing.
- `light_one_led`
- `flash_leds`
- `twinkle_leds`
- `exit_action`
- etc.

A typical run of the Keypad Controller

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}

- The complete sequence of keypad inputs is:

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

  whereas one symbol "Y" (4th table entry) was provided as an override signal from the agent, thus signaling password acceptance.
- Note: failed login and changing password are not in the run, but should be included in your demo