Hashing Difficulty Demonstrator

Description

Describe the Raspberry Pi project that is being proposed for Lab Exercise 6.

The Hashing Difficulty Demonstrator is designed to demonstrate the difficulty of generating a correct hash to “mine” a cryptocurrency block. The raspberry pi will simulate the process of trying to mine a block by generating hashes with a unique nonse, outputting them to the command line and showing the “closest” hash until a correct hash is generated.

An array of LEDs will be used to display the “progress” towards the correct hash. Each LED will represent how many zeroes are at the start of the hash, which themselves represent the difficulty of the hash. If at some point, a hash has been generated with a difficulty of three (ex: three leading zeroes), then three LEDs will be lit up until a hash with a higher level of difficulty has been generated, or until a valid hash has been found.

Once the correct hash is generated, the all of the LED lights will blink to show success. The hash itself, and the time taken to reach it, will be recorded.

Hardware

The raspberry pi will be connected to an array of LEDs via GPIO to show a progress to the correct hash.

Example

Difficulty Level being demonstrated = 6 //We need a hash with 6+ left aligned 0s
Hash generated: 0xabc3267890abcdef - no LEDs lit
Hash generated: 0x0000567890abcdef - 4 leftmost LEDs lit
Hash generated: 0x0034567890abcdef - 4 leftmost LEDs are still lit, progress remains
Hash generated: 0x0000007890abcdef - all LEDs blink to show success

Additionally, a UART device will be used to handle user input and display output to the user on a separate device. For demonstration purposes, a Windows laptop will be used.

Software

OS support will be provided by Raspbian and the program will be written in Python. Raspbian has been chosen primarily because this is the OS that we have the most experience with. We have chosen Python as our language of choice since we are both experienced with Python, and due to the fact that libraries for hashing with a wide variety of algorithms exist for Python.

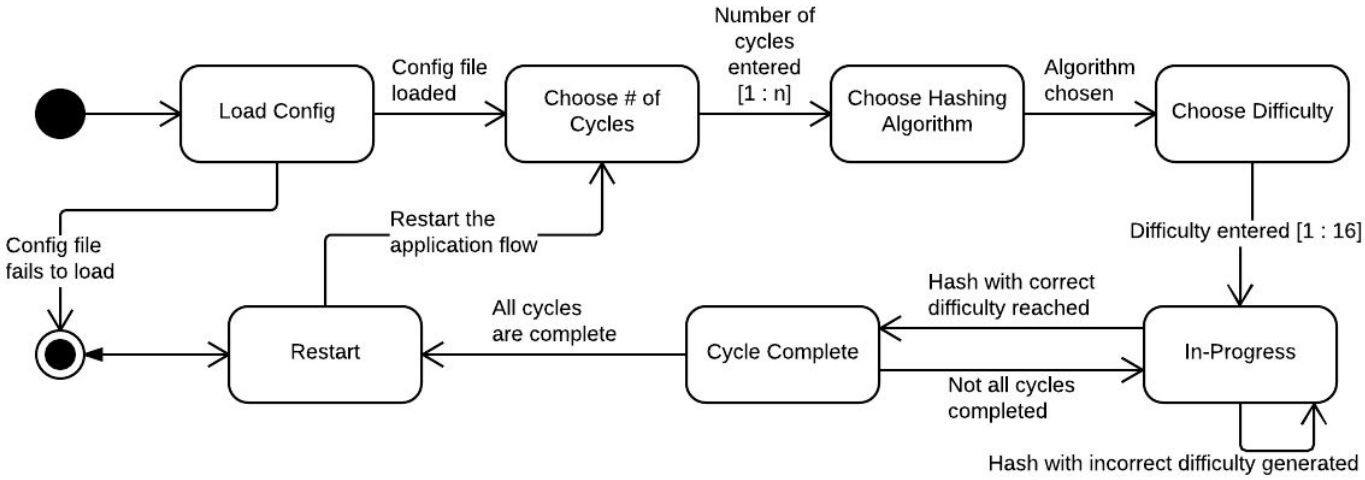
Prelab Work

Application Flow- *The application flow of the device will be as follows*

On startup, the application will attempt to load a configuration file containing data representing a mock cryptocurrency block. If the file cannot be loaded, the application will print out an error message and end execution; otherwise, the application will continue on to start accepting user input.

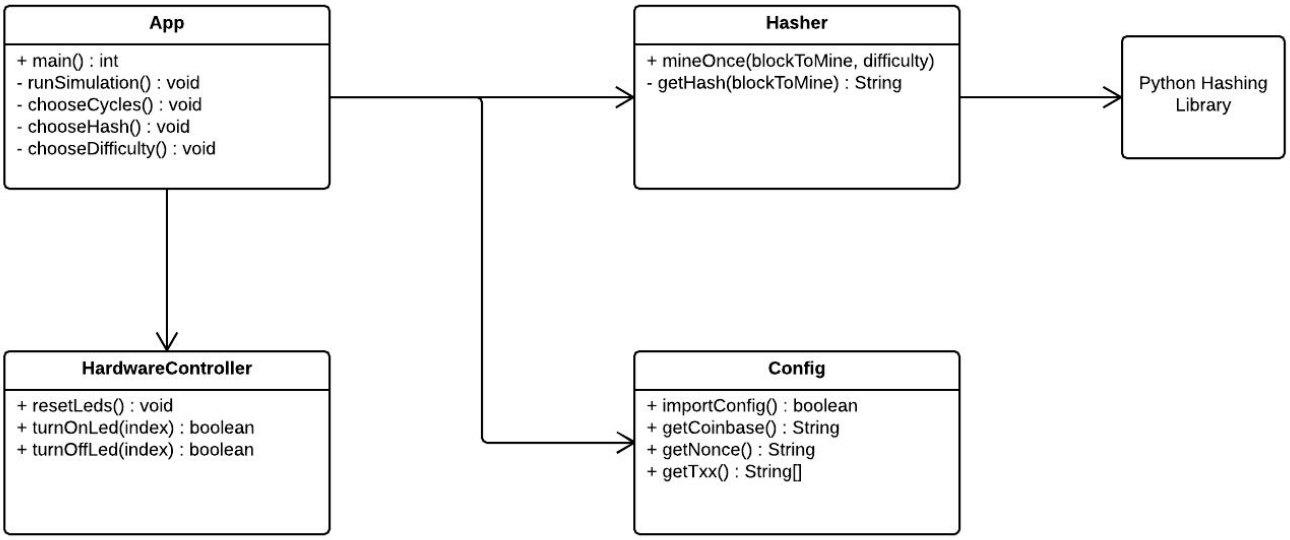
The user chooses a difficulty, hashing algorithm and the number of times that the application will simulate the process of mining a block (referred to in this documentation as **cycles**). After all three (3) of these variables have been accepted, the application will begin generating hashes. The difficulty of each generated hash is compared to the highest generated difficulty (-1 at the beginning) and then with the difficulty that needs to be achieved. Every time a hash is generated with a hash greater than the previous highest difficulty hash, the hash is printed out.

Once the required difficulty has been reached, either (1) another cycle begins or (2) all of the cycles have finished running. If the latter, then the application will print out the total number of cycles, the hashing algorithm chosen, the difficulty chosen and the average time to find an appropriate hash.



After this has been printed out, the user will be prompted to either enter values for difficulty/algorithm/cycles again, or to exit the application.

A simple class structure has been chosen, revolving around the **App** class, which is responsible for managing the primary flow of the application. Additional modules have been designed for the purpose of managing hardware, wrapping around the Python module to be used for generating hashes, and for managing file I/O.



Psuedocode has been developed for a few major functions

App.runSimulation()

1. Prompt the user for a number of cycles
    a. Continue prompting until a valid value is inputted
2. Print hashing algorithm options
3. Prompt the user for a hashing algorithm
    a. Continue prompting until a valid value is inputted
4. Prompt the user for difficulty
    a. Continue prompting until a valid value is inputted
5. While there are still cycles left to complete
    a. Call Hasher.mineOnce() to perform a single cycle

App.main()

1. Import the configuration file data through Config
    a. If the file could not be loaded, exit the application with an error message
2. Call runSimulation()
3. Prompt the user to run another simulation, or to exit
    a. If the user wants to exit, close the application
4. While the user wants to run another simulation, keep calling runSimulation()

Config.importConfig()

1. Check if the config file exists
2. If it does
    a. Load the file
    b. Check for correct formatting
    c. If formatting is correct, parse the data
    d. Else, return false
3. If it doesn't return false

Specifications

The following specifications have been identified for our project:

ID	Description
1	A difficulty exceeding the range of [1 : 16] cannot be entered.
2	No less than [1] can be entered for the number of cycles.
3	Strings will not be accepted for numeric input.
4	Error messages for invalid input will be provided, and the user will be allowed to input a value as many times as is necessary.
5	Users should be able to run multiple simulations.
6	Users should see LEDs light up as each cycle is executed.
7	LEDs should flash several times after one cycle is successfully executed.
8	User I/O should be handled through a UART device.