Chip Limeburner

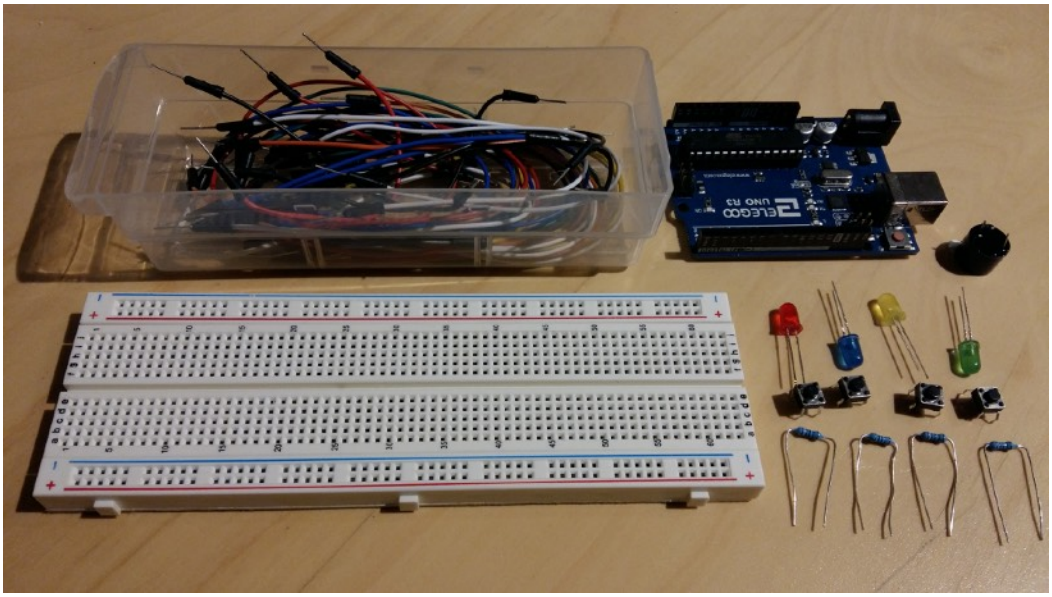
Student ID: 40177255

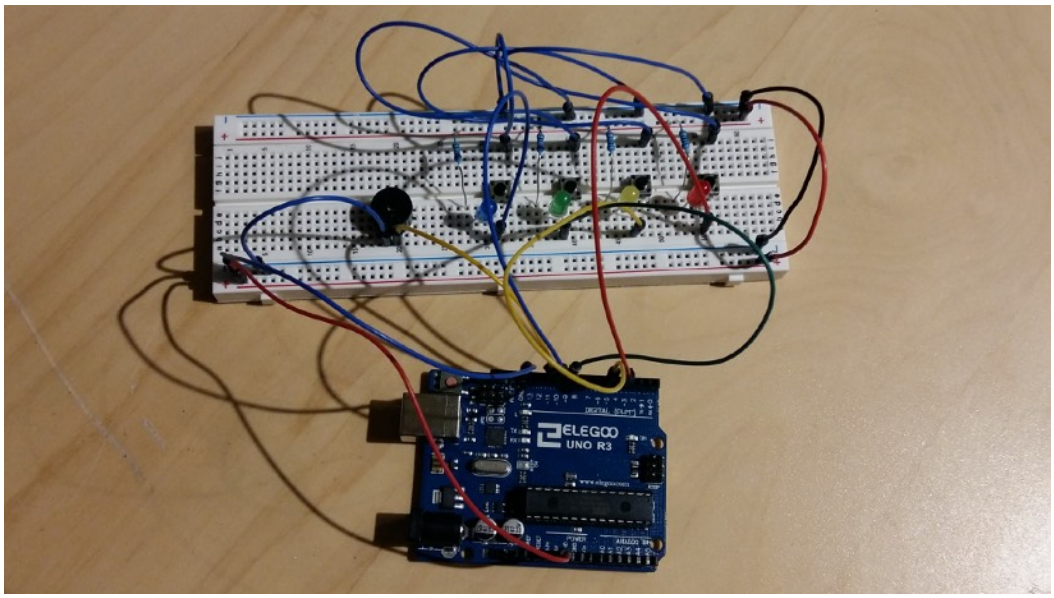
CART 360

Nov. 19, 2021

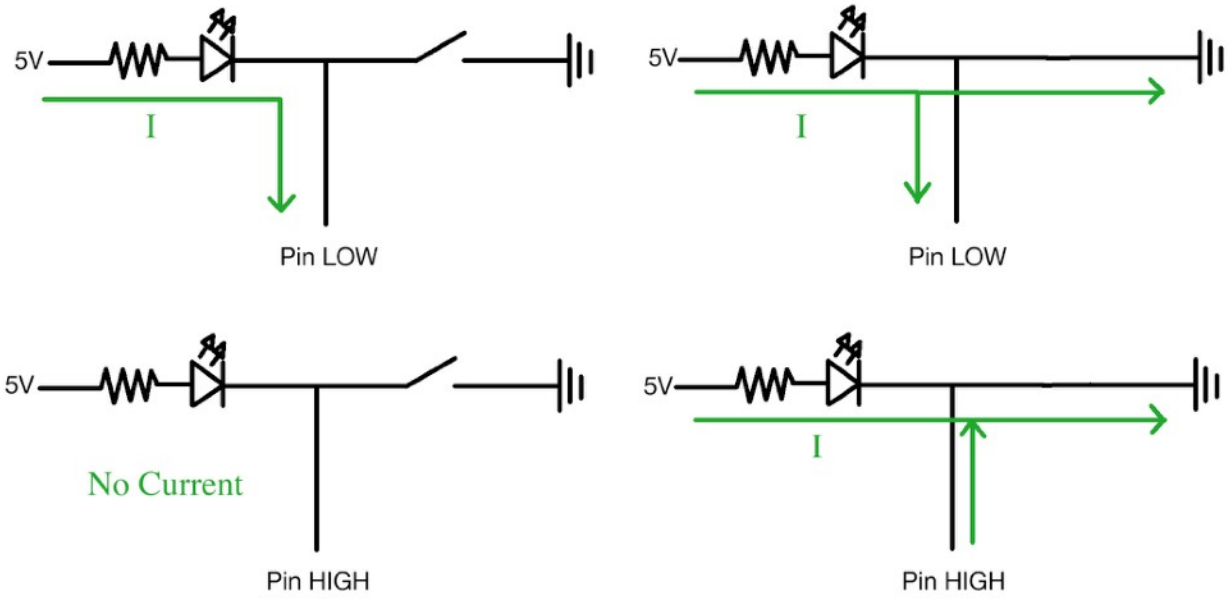
Etude 3

1) Due to the relative simplicity of the provided circuit, little time was needed to assemble it. The main steps were simply to pull the needed components from the course kit, and then assemble them on the board as shown below.





There was, however, one feature of the circuit that proved interesting, namely, the dual use of pins 3, 5, 9, and 10 each for an LED and a button. This seeming overlap of inputs and outputs on each pin is worth noting and the diagrams below outline several cases the circuit might find itself in.



As can be seen from this very brief analysis, in this configuration, current will always be passing into the pin if it's LOW, regardless of button activation, and so we should expect to see some use of the internal pull-up resistors on the Arduino board during periods when the button is being used as an input. However, since no current flows when the pin is HIGH and the button is unpunished, then we should also expect to see the code disable the internal pull-up resistors during any periods when it's flashing LEDs by itself. Ultimately, this is observed in the code provided, in both the checkButtonPush() and displayLightAndSound().

2)

(A)

i. - On setup, the program establishes initial pin modes, with the button pins set to Input and the buzzer pin to output.

- The program calls startUpLightsAndSounds(), which does some cosmetic display with the LEDs and buzzer, but then holds in a while-loop, calling getButtonPush() until any button is returned. This effectively returns the program back up to the setup routine.
- Continuing the setup routine, the program stores the time elapsed since the start of the program as the "game seed".
- Proceeding to the main loop, the program initializes a static integer sequenceLength of 1, to be used in starting the Simon Says game. Importantly, since it's a static integer, it is preserved across loops and will not be "re-initialized", avoiding overwriting.
- The program initializes the seed for the pseudo-random number generator, using the previously stored gameSeed.
- The program then iterates over a for-loop, counting down from the current sequenceLength variable. On each loop, it makes noise and flashes an LED based on the output of the pseudo-random generator initialized previously.
- Once it has counted down the length of he sequence, the program re-initializes the pseudo-random generator with the same seed, effectively returning the random() function to the beginning of the same sequence of pseudo-random numbers.
- The program then iterates over a for-loop, counting up from 0 to sequenceLength unless otherwise forced to break from the loop.
- The program iterates over a nested for-loop acting as a timer that counts down from 30, calling getButtonPushed() on each loop.
- If the return button is either the wrong button, or an indication of a timeout, the game resets, running through similar boot up behaviour, and storing a new gameSeed based on elapsed time since the start of the program.
- If the for-loop is fully iterated without any faults, then the program increments sequenceLength, and repeats the main loop.

ii. There are seven functions in the program, including the setup and main loop, and of these, checkButtonPush(), displayLightAndSound(), startUpLightsAndSound(), and the main loop() make significant contributions to flow control and the maintenance of game state. StartUpLightsAndSound() is the simplest of these, sweeping through the LEDs and associated sounds and registering an initial gameSeed when called by the setup() function on boot, but importantly it contains a while loop that repeatedly checks for any button to be pushed before advancing the game state from “idle” to “playing”. DisplayLightAndSound() and checkButtonPush() are similar in their structure, toggling the pin mode and internal pull-up resistors of the LED/button pins as described in question (1), and then either outputting an LED/sound combination, or reading a button push and outputting the corresponding sound respectively. These two are important to flow control in that they maintain the direction of current across the LED and button to allow the pins to work as both inputs and outputs as needed for either playback or input game states. Finally, the main loop carries out the primary outline of the game, calling the above mentioned playback and input state functions as their turn arises in the course of the game. It does this with for- and if-loops to maintain timeout countdowns and to check return button pushes. Finally, it implicitly tracks the game’s colour sequence using a repeated initialization of the random number generator from a consistent known seed.

(B)

i. Given the game of Simon Says is fundamentally based on sequence matching, this program does not have an array, linked list or similar data structure it uses to store new randomly generated values, leaving it with no way to explicitly check user input against the previous values it generated.

ii. On each iteration of the main loop, the program could randomly generate its next value and add it to the sequence data structure for later comparison against user input.

3)

i. The program addresses the above issue of not explicitly tracking its Simon Says sequence by leveraging the technical properties of the random() and the randomSeed() functions when used with a known seed.

ii. The random() function, when initialized with a specific seed, will always proceed in the same order on each subsequent call, so by re-initializing with randomSeed() and a consistent seed value, the order of random() will always be the same, giving an arbitrarily long and constant sequence without need for explicit tracking.