**Day 5 Notes**

Yes, a \*\*byte\*\* variable is perfect for storing the value of a 3-position DIP switch in binary! A `byte` can hold values from `0` to `255` (or `00000000` to `11111111` in binary), which is more than enough for this setup. With a 3-position DIP switch, you’re working with values from `000` (0 in decimal) to `111` (7 in decimal), which fits comfortably within a `byte`.

### Here’s how to use a byte to store and read the DIP switch state:

1. \*\*Declare a `byte` Variable\*\*:

```cpp

byte switchState = 0;

```

2. \*\*Read Each Switch and Assign Values\*\*:

- Assign each DIP switch to a pin, then read each pin and combine their values to store a binary number in `switchState`.

- For example, if you have your DIP switches connected to pins 2, 3, and 4, you can shift each input into place.

3. \*\*Example Code for a Binary Counter\*\*:

```cpp

byte switchState = 0;

void setup() {

pinMode(2, INPUT);

pinMode(3, INPUT);

pinMode(4, INPUT);

}

void loop() {

// Read each switch and build the binary number

switchState = (digitalRead(2) << 2) | (digitalRead(3) << 1) | digitalRead(4);

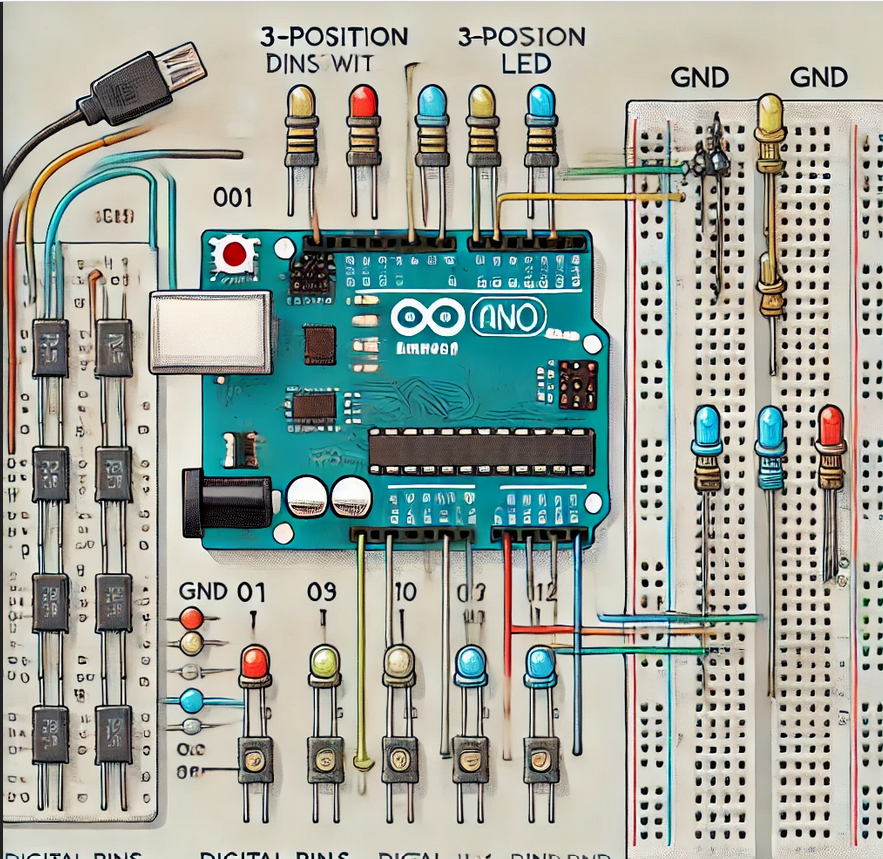
// Now, switchState holds the binary value of the DIP switch

// Use switchState to control LEDs or other outputs

}

```

Each switch position is read as either `0` or `1` and shifted into place to create a binary number representing all three switches. This allows you to control LEDs in a binary count sequence or create different patterns. Let me know if you want to explore this further or see it in action with some LEDs!



*~~Here’s the visual circuit diagram you requested! The setup includes a 3-position DIP switch connected to digital pins 2, 3, and 4 on the HERO board, each with a 10k Ohm pull-down resistor to GND. Three LEDs are connected to pins 9, 10, and 11 with 220 Ohm resistors.~~*

Reuse Board from Day 4 instead

A circuit board with wires

Description automatically generated

### Code Example

This code reads the DIP switch position and turns on LEDs based on the binary values:

```cpp

byte switchState = 0;

void setup() {

pinMode(2, INPUT);

pinMode(3, INPUT);

pinMode(4, INPUT);

pinMode(9, OUTPUT);

pinMode(10, OUTPUT);

pinMode(11, OUTPUT);

}

void loop() {

// Read the DIP switch state and create a binary number

switchState = (digitalRead(2) << 2) | (digitalRead(3) << 1) | digitalRead(4);

// Turn LEDs on or off based on the switchState value

digitalWrite(9, switchState & 0b001 ? HIGH : LOW); // LED on pin 9

digitalWrite(10, switchState & 0b010 ? HIGH : LOW); // LED on pin 10

digitalWrite(11, switchState & 0b100 ? HIGH : LOW); // LED on pin 11

}

```

Each switch position corresponds to binary numbers from `000` to `111` (0 to 7), selectively controlling which LEDs turn on based on the DIP switch configuration. Enjoy experimenting, and let me know how it goes!

That works, but the circuit you gave me was pretty cluttered and I already had one built so I just reused the one fro Day 4. Had to change the output pins from 9, 10, and 11 to 10, 11, 12 to match the circuit I had built.

Then I noted that the LEDs were reversed left to right compared to the switches from my viewing point. So, I switched the line of code

switchState = (digitalRead(2) << 2) | (digitalRead(3) << 1) | digitalRead(4);

to

switchState = (digitalRead(4) << 2) | (digitalRead(3) << 1) | digitalRead(2);

Now it works the way I wanted it to.

Lets take some time to clarify a couple of commands and what they do:

First command: digitalWrite(10, switchState & 0b010 ? HIGH : LOW); // LED on pin 10

Second Command: switchState = (digitalRead(4) << 2) | (digitalRead(3) << 1) | digitalRead(2);

Sure! Let’s break down these commands in detail:

### Command 1: `digitalWrite(10, switchState & 0b010 ? HIGH : LOW);`

This line uses \*\*bitwise operations\*\* and a \*\*ternary conditional operator\*\* to control an LED based on a specific bit in the `switchState` variable.

1. \*\*`switchState & 0b010`\*\*:

- Here, `0b010` is the binary representation of `2` in decimal, which in binary form is `010`. The `&` operator (bitwise AND) compares each bit of `switchState` with `0b010`:

- If `switchState` has the middle bit set to `1` (like `010`), this expression will return a non-zero result.

- If the middle bit is `0`, the expression returns `0`.

2. \*\*`? HIGH : LOW`\*\*:

- This is the \*\*ternary operator\*\*, which functions like an if-else statement. If the result of `switchState & 0b010` is non-zero (meaning the middle bit is `1`), it evaluates to `HIGH`; otherwise, it evaluates to `LOW`.

3. \*\*Overall Command\*\*:

- `digitalWrite(10, switchState & 0b010 ? HIGH : LOW);` sets the LED on pin 10 to `HIGH` if the middle bit in `switchState` is `1`, and to `LOW` if it is `0`. This means pin 10’s LED will turn on only when that specific bit is active.

### Command 2: `switchState = (digitalRead(4) << 2) | (digitalRead(3) << 1) | digitalRead(2);`

This command reads multiple digital inputs from DIP switches and combines them into a single \*\*binary number\*\*.

1. \*\*`digitalRead(4)`, `digitalRead(3)`, and `digitalRead(2)`\*\*:

- Each `digitalRead(pin)` reads the state (HIGH or LOW) of the specified pin.

- The result is `1` if the pin reads HIGH and `0` if the pin reads LOW.

2. \*\*Bit Shifting (`<<`)\*\*:

- \*\*`digitalRead(4) << 2`\*\*: This shifts the bit read from pin 4 two places to the left, setting it as the \*\*third bit\*\*.

- \*\*`digitalRead(3) << 1`\*\*: This shifts the bit read from pin 3 one place to the left, setting it as the \*\*second bit\*\*.

- \*\*`digitalRead(2)`\*\*: This is used as-is for the \*\*first bit\*\*.

3. \*\*Bitwise OR (`|`)\*\*:

- The \*\*`|`\*\* (bitwise OR) operator combines each bit into a single number.

- For example, if pins 4, 3, and 2 read as HIGH, LOW, and HIGH, respectively:

- `digitalRead(4) << 2` results in `100`.

- `digitalRead(3) << 1` results in `000`.

- `digitalRead(2)` results in `001`.

- Combined, `100 | 000 | 001` results in `101` (or `5` in decimal).

4. \*\*Overall Command\*\*:

- `switchState = (digitalRead(4) << 2) | (digitalRead(3) << 1) | digitalRead(2);` stores a 3-bit binary number in `switchState` based on the states of pins 4, 3, and 2. This lets you represent up to eight (0–7) unique values depending on the DIP switch configuration.

This approach enables efficient control over multiple LEDs or other outputs using a single variable to represent multiple binary states.

**Embellished Code**

I was thinking of two things: 1) Explore the syntax and useage of the new commands, and 2) A different upgrade than you suggested -- How about keeping the existing LED and switch function for the first 15 seconds (both at the beginning and whenever a switch is flipped AKA switch state changes) and then after 15 seconds shift over to using the LEDs as a binary counter for 0 - 7 with the LED on Pin 10 representing 1, the LED on 11 representing 2, and the LED on PIN 12 representing 4. Hold each state for 1 second before iterating. Loop back to zero when the 3 bit count overflows.

I love this idea, Explorer! Let’s break it down into manageable steps and explore the syntax for implementing both parts of your plan. We’ll first create a \*\*15-second timer\*\* that activates whenever the switches are flipped. During those 15 seconds, the LEDs reflect the DIP switch state. After the timer, we’ll start a \*\*3-bit binary counter\*\* with each LED representing a different binary value.

Here’s a guide to setting it up:

### 1. New Concepts and Syntax

- \*\*millis()\*\*: This function helps track the elapsed time without pausing the program. Unlike `delay()`, `millis()` lets other code run simultaneously.

- \*\*Bitwise Operations\*\*: Use bitwise operators like `<<` (bit shift) to assign LEDs based on the binary count.

- \*\*Tracking Switch State Changes\*\*: We’ll use a variable to store the previous state of the switches, so the 15-second timer only resets when the state actually changes.

### 2. Code Implementation

Here’s a code template that combines both phases of functionality:

```cpp

byte switchState = 0;

byte previousSwitchState = 0; // Store the previous switch state

unsigned long lastSwitchChangeTime = 0;

bool isCounting = false; // Flag to switch between modes

byte counter = 0;

void setup() {

// Set up pins for DIP switch and LEDs

pinMode(2, INPUT);

pinMode(3, INPUT);

pinMode(4, INPUT);

pinMode(10, OUTPUT);

pinMode(11, OUTPUT);

pinMode(12, OUTPUT);

}

void loop() {

// Read the DIP switch state

switchState = (digitalRead(4) << 2) | (digitalRead(3) << 1) | digitalRead(2);

// Check if switch state has changed

if (switchState != previousSwitchState) {

previousSwitchState = switchState; // Update previous state

lastSwitchChangeTime = millis(); // Reset the timer

isCounting = false; // Stay in DIP switch mode

displaySwitchState(switchState); // Display current switch state

}

// Check the elapsed time to determine mode

if (millis() - lastSwitchChangeTime > 15000) { // After 15 seconds

isCounting = true;

}

if (isCounting) {

displayBinaryCounter();

} else {

displaySwitchState(switchState); // Display switch state within 15 seconds

}

}

// Function to display DIP switch state

void displaySwitchState(byte state) {

digitalWrite(10, state & 0b001 ? HIGH : LOW); // LED on pin 10 (1)

digitalWrite(11, state & 0b010 ? HIGH : LOW); // LED on pin 11 (2)

digitalWrite(12, state & 0b100 ? HIGH : LOW); // LED on pin 12 (4)

}

// Function to display binary counter from 0 to 7

void displayBinaryCounter() {

digitalWrite(10, counter & 0b001 ? HIGH : LOW); // LED on pin 10

digitalWrite(11, counter & 0b010 ? HIGH : LOW); // LED on pin 11

digitalWrite(12, counter & 0b100 ? HIGH : LOW); // LED on pin 12

delay(1000); // Hold each counter state for 1 second

counter = (counter + 1) % 8; // Loop back to 0 after reaching 7

}

```

### Explanation of the Code

- \*\*Timer-Based Switching\*\*: The code checks `millis()` to measure elapsed time since the last switch change. Once it’s been 15 seconds, the program switches modes.

- \*\*State Display vs. Counting\*\*:

- `displaySwitchState()` shows the DIP switch position with the LEDs for the first 15 seconds.

- `displayBinaryCounter()` takes over afterward, cycling through binary numbers from 0 to 7 and using a 1-second delay to hold each state.

This setup gives you control over both functions, blending interactive input with a timed binary counter display. Try it out, and let me know if you want to tweak anything further!

counter = (counter + 1) % 8

Sure! The command `counter = (counter + 1) % 8` is a neat way of incrementing the `counter` variable while looping back to zero after reaching a specified maximum—in this case, `7`.

Here’s a breakdown of each part:

1. \*\*Incrementing the Counter\*\*:

- `(counter + 1)` simply adds `1` to the current value of `counter`, moving it to the next count in the sequence.

- So, if `counter` starts at `0`, `(counter + 1)` will make it `1`, and so on.

2. \*\*Modulo Operation (`% 8`)\*\*:

- The `%` symbol is the \*\*modulo operator\*\*, which finds the remainder after division. Here, it ensures that `counter` stays within the range of `0` to `7`.

- When `counter + 1` reaches `8`, the expression becomes `8 % 8`, which equals `0`, looping `counter` back to zero.

3. \*\*Result\*\*:

- This line makes `counter` count up by `1` each loop cycle, resetting to `0` when it reaches `8`. So `counter` cycles through `0, 1, 2, ..., 7` repeatedly.

Using this approach is efficient for creating a repeating cycle and is especially handy in cases like binary counting, where each value represents a specific combination of LED states or binary outputs.