Gesture Recognition Glove: A Tilt and Flex Sensor-Based Approach

---

By: Mazen Belal
Made On: October 15, 2024

Abstract:

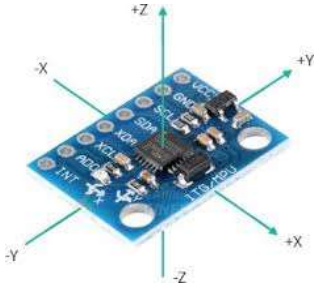
The Gesture Recognition Glove is an embedded system designed to recognize hand gestures and map them to predefined words or phrases. Utilizing an MPU6050 for tilt detection and flex sensors for finger bending, this glove aims to allow deaf and dumb people to communicate freely with anyone not familiar with sign language.

Table of Contents

Hardware Components

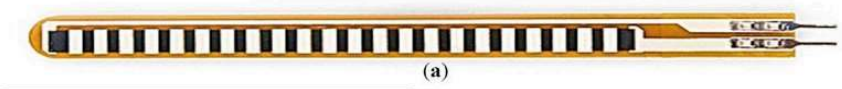
List of Components

1. **MPU6050 Accelerometer and Gyroscope Sensor**
    - **Role**: Measures tilt and motion data for tilt-based word mapping.
    - **Specifications**:
        - Operating Voltage: 3.3V-5V
        - Gyroscope Range: ±250, ±500, ±1000, ±2000°/sec
        - Accelerometer Range: ±2g, ±4g, ±8g, ±16g
        - Communication: I2C protocol
    - **Reason for Use**: Provides real-time motion detection with high accuracy.



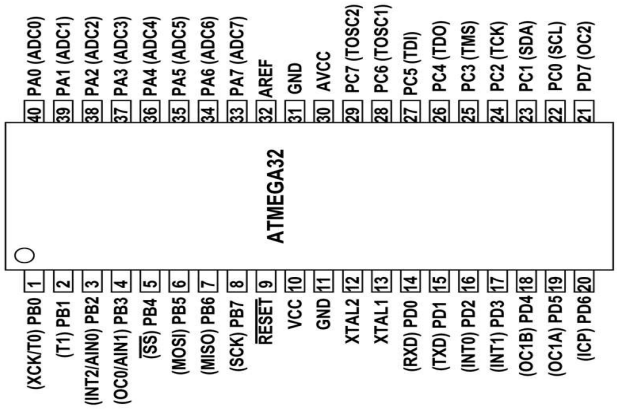
---

2. **Flex Sensors (5 units)**
    - **Role**: Detects bending of fingers to generate binary codes for word mapping.
    - **Specifications**:
        - Bend Resistance: 10KΩ (flat) to 40KΩ (fully bent)
        - Length: ~2.2 inches
        - Operating Voltage: 3.3V-5V
    - **Reason for Use**: Captures the bending of each finger accurately.


(a)

---

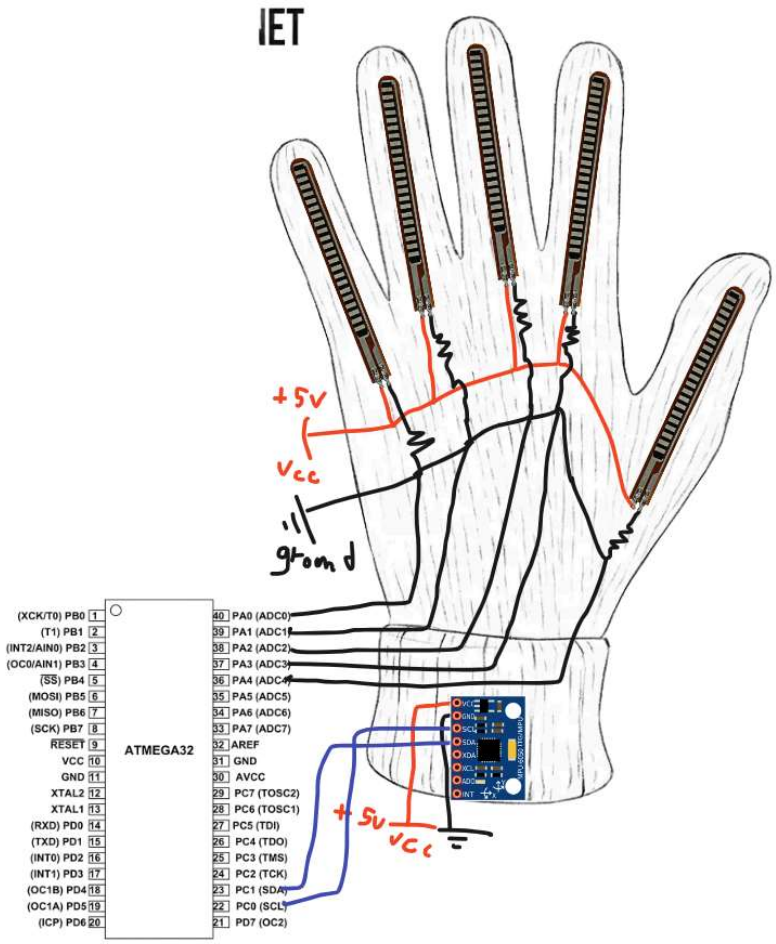
3. **ATmega32 Microcontroller**
    - **Role**: Core processor for managing sensors, processing data, and driving the display.
    - **Specifications**:
        - Operating Voltage: 5V
        - Flash Memory: 32KB
        - I/O Pins: 32
        - Communication: I2C, UART, SPI
    - **Reason for Use**: Offers enough processing power and GPIOs for sensor interfacing.



4. **16x2 Character LCD**
    - **Role**: Displays detected words and binary codes.
    - **Specifications**:
        - Character Size: 5x8 pixels varies
        - Operating Voltage: 4.7V-5.3V
        - Communication: Parallel
    - **Reason for Use**: Easy-to-read display for real-time output.

5. **Resistors (Various Values)**
    - **Role**: Used for voltage division and pull-up/down configurations.
    - **Specifications**: Various resistances (1KΩ)
    - **Reason for Use**: Ensure proper voltage levels and circuit stability.

Circuit Design and Assembly:



Software Design

Libraries and Functions Used

Below is a detailed list of the libraries used in the project, the functions utilized from each library, and their respective explanations.

---

1. MPU6050 Interface (self made)

Handles communication with the MPU6050 sensor to retrieve accelerometer and gyroscope data.

- **Functions**:
    1. `MPU6050_voidInit()`
        - Initializes the MPU6050 sensor.
        - Configures power management, sample rate, and sensitivity settings for accelerometer and gyroscope.
    2. `MPU6050_voidReadAccel(int *Acc_x, int *Acc_y, int *Acc_z)`
        - Reads raw accelerometer data for X, Y, and Z axes.
        - Outputs the data as integer values.

---

2. CLCD Interface

Controls the 16x2 character LCD for displaying text and numbers.

- **Functions:**
    1. `CLCD_voidInit()`
        - Initializes the LCD in 4-bit or 8-bit mode.
        - Prepares the LCD for displaying characters.
    2. `CLCD_voidSendString(char *str)`
        - Displays a string of characters on the LCD.

- Useful for outputting tilt states and binary code interpretations.

3. `CLCD_voidSetPosition(u8 line, u8 position)`
   - Sets the cursor to a specific line and position on the LCD.
   - Enables targeted updates to specific sections of the screen.
4. `CLCD_voidClearDisplay()`
   - Clears all content on the LCD screen.
   - Used before updating the display to ensure a clean output.

---

3. ADC Interface

Performs analog-to-digital conversions for the flex sensor signals.

- **Functions:**
  1. `ADC_voidInit()`
     - Configures the ADC for operation.
     - Sets the reference voltage, prescaler, and resolution (e.g., 10-bit ADC).
  2. `ADC_u16GetChannelReading(u8 channel)`
     - Reads the analog value from the specified ADC channel.
     - Converts the signal into a digital value (0-1023).

---

4. DIO Interface

Manages digital I/O pin configuration and control.

- **Functions:**
    1. `DIO_voidSetPinDir(u8 port, u8 pin, u8 direction)`
        - Sets the direction of a specific pin as input or output.
        - Used to configure the LCD control pins, I2C pins, and flex sensor pins.
    2. `DIO_voidSetPortDir(u8 port, u8 direction)`
        - Configures all pins on a specified port as input or output.
    3. `DIO_voidSetPinVal(u8 port, u8 pin, u8 value)`
        - Sets a specific pin to HIGH or LOW.
        - Used for controlling LEDs or resetting sensors.

---

5. Timer Interface

Provides periodic interrupts for tasks like debouncing and regular sensor updates.

- **Functions:**
    1. `TIMER0_voidInit()`
        - Initializes Timer0 with the desired prescaler and mode (e.g., normal mode).
    2. `TIMER0_e_SetCallBackOVF(void (*callback)(void))`
        - Sets a callback function to be executed on Timer0 overflow.
        - Used for debouncing tilt detection.
    3. `TIMER0_voidSetPreLoad(u8 preload_value)`
        - Sets the Timer0 preload value to adjust the overflow timing.

---

6. I2C Interface(self made)

Enables communication between the microcontroller and the MPU6050 using the I2C protocol.

- **Functions:**
    1. `I2C_voidInit()`
        - Configures the I2C communication settings (e.g., clock rate, master/slave mode).
    2. `I2C_u8Start(u8 address)`
        - Sends a START condition followed by the device address.
        - Initiates communication with the MPU6050.
    3. `I2C_u8Write(u8 data)`
        - Sends a byte of data over the I2C bus.
        - Used for writing to MPU6050 registers.
    4. `I2C_u8ReadAck()`
        - Reads a byte of data and sends an ACK to the sender.
        - Used for reading multiple bytes from the MPU6050.
    5. `I2C_voidStop()`
        - Sends a STOP condition to terminate communication.

---

7. Standard Utility Library (<stdlib.h> and <stdbool.h>)

Provides utility functions and data types for effective program implementation.

- **Functions/Features:**
    1. `malloc()`
        - Dynamically allocates memory (if used).
    2. `Boolean Type (bool)`
        - Used for handling flags like `devmode`.

---

8. Delay Library (<util/delay.h>)

Introduces time delays for stabilization and visual clarity.

- **Functions:**
    1. `_delay_ms(double delay)`
        - Delays the program execution by the specified number of milliseconds.
        - Used for ADC stabilization and LCD updates.

Explanation of Code Structure:

- [ ] Included Libraries

```
#include <util/delay.h>
#include "MPU6050_interface.h"
#include "I2C_interface.h"
#include "CLCD_interface.h"
#include "DIO_interface.h"
#include "GI_interface.h"
#include "TIMER0_interface.h"
#include "ADC_interface.h"
#include <string.h>
#include <stdbool.h>
```

Purpose: These libraries provide essential functions and definitions required for the project.

Explanation:

- `<util/delay.h>`: Provides functions like `_delay_ms()` for precise delays.
- `MPU6050_interface.h`: Contains functions to interact with the MPU6050 accelerometer.
- `I2C_interface.h`: Defines functions for I2C communication, required for the MPU6050.
- `CLCD_interface.h`: Contains functions for the character LCD display.
- `DIO_interface.h`: Digital Input/Output library to control pins.
- `GI_interface.h`: Enables global interrupts for handling the timer.
- `TIMER0_interface.h`: Provides timer functions, including overflow interrupt setup.
- `ADC_interface.h`: Used for reading values from analog-to-digital converters (flex sensors).
- `<string.h>`: For string operations, like `strcmp` and `strcpy`.
- `<stdbool.h>`: For using `true` and `false` in boolean logic.

☐ Global Variables

```
//Tilt Detection Variables
char currentTilt[16] = "Starting";
char last_displayed_tilt[16] = "Starting";
int stableCounter = 0;
const int stableThreshold = 10;
volatile u8 Timer0_Flag = 0;
int Acc_x, Acc_y, Acc_z;
u8 Acc_x_mapped, Acc_y_mapped, Acc_z_mapped;
```

Purpose: These variables handle tilt detection.

Explanation:

- `currentTilt`: Stores the current tilt direction (e.g., "Tilted Left").
- `last_displayed_tilt`: Stores the last tilt direction displayed on the LCD.
- `stableCounter`: Ensures the tilt is stable before acting on it.
- `stableThreshold`: Defines how many consistent readings are required for stability.
- `Timer0_Flag`: Set to `1` when the timer interrupt occurs, signaling that tilt detection should run.
- `Acc_x`, `Acc_y`, `Acc_z`: Raw accelerometer data.
- `Acc_x_mapped`, `Acc_y_mapped`, `Acc_z_mapped`: Accelerometer values mapped to a 0–255 range.

```
//Flex Sensor Variables
#define NUM_SENSORS 5
#define AVG_SAMPLES 3
#define DEBOUNCE_THRESHOLD 2
u16 sensor_values[NUM_SENSORS];
u8 sensor_channels[NUM_SENSORS] = {PIN0, PIN1, PIN2, PIN3, PIN4};
u8 binary_states[NUM_SENSORS] = {0};
u8 binary_code = 0;
u8 last_binary_code = 0xFF;

const u8 lower_threshold[NUM_SENSORS] = {30, 27, 25, 30, 30};
const u8 upper_threshold[NUM_SENSORS] = {40, 40, 40, 40, 40};
```

Purpose: Configures and tracks the state of the flex sensors.

Explanation:

- `NUM_SENSORS`: Number of flex sensors.
- `AVG_SAMPLES`: Number of samples for averaging sensor readings.
- `DEBOUNCE_THRESHOLD`: Minimum stable readings required to confirm a sensor state.
- `sensor_values[]`: Stores ADC readings for each flex sensor.
- `sensor_channels[]`: Assigns ADC pins to each sensor.
- `binary_states[]`: Tracks if each sensor is bent (1) or not bent (0).
- `binary_code`: Combines all `binary_states` into a 5-bit code.
- `last_binary_code`: Stores the previously displayed binary code.
- `lower_threshold` and `upper_threshold`: Define bending thresholds for each sensor.

- [ ] Initialization

```
I2C_voidInit();
CLCD_voidInit();
MPU6050_voidInit();
TIMER0_voidInit();
ADC_voidInit();
GI_voidEnable();
```

Purpose: Initializes all hardware peripherals.

Explanation:

- `I2C_voidInit`: Prepares I2C communication for the MPU6050.
- `CLCD_voidInit`: Configures the LCD.
- `MPU6050_voidInit`: Sets up the accelerometer.
- `TIMER0_voidInit`: Starts Timer0 for periodic tilt checks.
- `ADC_voidInit`: Configures the ADC for flex sensor readings.
- `GI_voidEnable`: Enables global interrupts.

- [ ] Tilt Detection Function

```
void DetectTilt(void)
{
    if (Acc_x_mapped < 40)
        strcpy(currentTilt, "Tilted Left");
    else if (Acc_x_mapped > 215)
        strcpy(currentTilt, "Tilted Right");
    else if (Acc_y_mapped < 15)
        strcpy(currentTilt, "Tilted Backward");
    else if (Acc_y_mapped > 200)
        strcpy(currentTilt, "Tilted Forward");
    else if (Acc_x_mapped > 130 && Acc_x_mapped < 200 && Acc_y_mapped > 50 && Acc_y_mapped
        strcpy(currentTilt, "Upright");

    if (strcmp(currentTilt, last_displayed_tilt) == 0)
        stableCounter++;
    else
        stableCounter = 0;

    if (stableCounter >= stableThreshold)
    {
        stableCounter = 0;
    }
```

Purpose: Detects the tilt of the glove based on accelerometer data and assigns a tilt status (e.g., "Tilted Left", "Upright") to the `currentTilt` variable. It ensures stability in tilt detection using a counter to avoid rapid fluctuations.

Explanation:

1. **Tilt Conditions**:
    - Compares the mapped accelerometer values (`Acc_x_mapped` and `Acc_y_mapped`) against thresholds.
    - Assigns tilt directions (e.g., "Tilted Left") based on these values.
    - `strcpy` is used to update `currentTilt`.
2. **Stability Counter**:
    - Compares `currentTilt` with `last_displayed_tilt`.
    - Increments `stableCounter` if the tilt remains the same, ensuring stability in detection.
    - Resets `stableCounter` if the tilt changes.
3. **Stable Threshold**:
    - Once `stableCounter` exceeds `stableThreshold`, it confirms the tilt and resets the counter.

☐ Timer Overflow ISR

```
void Timer0_ISR(void)
{
    Timer0_Flag = 1;
}
```

Purpose: Sets a flag (`Timer0_Flag`) when the timer overflows, providing a timed mechanism for tasks such as tilt detection.

Explanation:

- Used to debounce the tilt detection by introducing a periodic check.
- The main loop checks the flag before reading accelerometer data.

- [ ] Get Averaged ADC Reading

```
u16 getAveragedReading(u8 channel)
{
    u32 sum = 0;
    for (u8 i = 0; i < AVG_SAMPLES; i++)
    {
        sum += ADC_u16GetChannelReading(channel);
    }
    return (u16)(sum / AVG_SAMPLES);
}
```

Purpose: Reads and averages multiple ADC readings for a given channel, reducing noise.

Explanation:

1. **Loop**:
   - Reads `AVG_SAMPLES` ADC values from the specified `channel`.
   - Accumulates the readings in `sum`.
2. **Averaging**:
   - Returns the average by dividing `sum` by `AVG_SAMPLES`.
3. **Noise Reduction**:
   - Helps minimize random variations in sensor readings.

- [ ] Get Binary State

```
int getBinaryState(u16 sensor_value, u8 sensor_index)
{
  if (sensor_value < lower_threshold[sensor_index] || sensor_value > upper_threshold[sensor_index])
    return 1;
  else
    return 0;
}
```

Purpose: Determines the binary state (0 or 1) of a flex sensor based on its value and predefined thresholds.

Explanation:

1. **Threshold Comparison**:
   - Compares `sensor_value` against `lower_threshold` and `upper_threshold` for the specified sensor.
   - Returns `1` if the value is outside the range; otherwise, returns `0`.
2. **Custom Thresholds**:
   - Each sensor has its own threshold values (`lower_threshold` and `upper_threshold`).

☐ Update Debounced States

```
void updateDebouncedStates(u8 *binary_states)
{
    static u8 stable_count[NUM_SENSORS] = {0};
    static u8 last_binary_states[NUM_SENSORS] = {0};

    for (u8 i = 0; i < NUM_SENSORS; i++)
    {
        u8 current_state = getBinaryState(sensor_values[i], i);
        if (current_state == last_binary_states[i])
        {
            stable_count[i]++;
        }
        else
        {
            stable_count[i] = 0;
        }

        if (stable_count[i] >= DEBOUNCE_THRESHOLD)
        {
            binary_states[i] = current_state;
            stable_count[i] = 0;
        }

        last_binary_states[i] = current_state;
    }
}
```

Purpose: Stabilizes binary states of the flex sensors by debouncing rapid state changes.

Explanation:

1. **Stable Count**:
    - Maintains a counter (`stable_count`) for each sensor.
    - Increments the counter if the current state matches the last state.
2. **Debounce Threshold**:
    - Updates the binary state only if `stable_count` exceeds `DEBOUNCE_THRESHOLD`.
    - Prevents rapid fluctuations from being registered as state changes.
3. **Last State Update**:
    - Updates `last_binary_states` for the next iteration.

These are the main codes that have complex structures that need explanation

Conclusion:

☐ **Limitations and Challenges**

Throughout the course of this project, we encountered several challenges. While we successfully addressed many of them, a few limitations remain:

1. **Finding the Flex Sensors**
   Flex sensors were difficult to source due to their rarity in the local market. This led to delays as we had to search extensively across various suppliers before obtaining the necessary components.
2. **Selecting and Preparing the Glove**
   Finding a suitable glove that was both flexible and compatible with the sensors was another challenge. We tested multiple materials to ensure seamless motion tracking. Ultimately, we chose a fabric material glove and found that sewing the sensors in place while gluing the terminal parts provided the most stable attachment for accurate motion detection.
3. **Sensor Sensitivity and Calibration**
   Achieving consistent and meaningful readings from the sensors required rigorous calibration. Each motion and state needed a specific voltage range, which was challenging due to the varying accuracy and response times of the sensors. We invested significant effort in testing across different motion ranges and environmental conditions to enhance sensitivity and minimize errors.
4. **MPU6050 Sensitivity**
   The MPU6050 provided excellent motion detection in two axes but was notably less sensitive in the third axis. This imbalance posed challenges in achieving accurate multi-directional motion detection and required fine-tuning of thresholds for tilt recognition.
5. **Speaker Integration**
   Integrating a speaker module for audio output presented significant challenges. While we explored modules and researched ways to produce specific words or play pre-recorded audio, we lacked access to expertise or resources to implement this feature effectively during the project timeline.

Potential Improvements

To further enhance the functionality and usability of this project, several improvements can be made:

1. **Speaker Integration**
   Replacing the LCD with a speaker for audio output would significantly improve the glove's user experience. The system could use synthesized or pre-recorded audio to convey information, making it more accessible and intuitive.
2. **Gesture-Based Control**
   Expanding the glove's functionality to include gesture-based control opens the door to numerous applications. With minor modifications to the code, the glove could be adapted to control robots, smart home devices, vehicles, and other systems through intuitive hand gestures.
3. **Enhanced Sensor Technology**
   Using advanced flex sensors with improved sensitivity and accuracy would address some limitations, allowing for faster and more precise gesture recognition.
4. **Improved Glove Design**
   A custom-designed glove that integrates sensors directly into its fabric could enhance durability, comfort, and ease of use, reducing the need for manual attachment.
5. **Dynamic Calibration Phase**

   Implementing a calibration phase during the glove's first boot would allow it to adapt to individual users. Instead of relying on constant threshold values, the glove could prompt the user to perform specific motions, such as fully opening and closing their hand, tilting in different directions, and relaxing their hand. These actions would establish personalized threshold values for each sensor and motion, significantly improving accuracy and usability.

Conclusion

The smart glove project demonstrates the potential of integrating motion sensing and flex sensor technologies to create a versatile tool for communication and control. Despite the challenges faced, including sourcing components, sensor sensitivity, and the integration of advanced features like audio output, the project showcases a strong proof of concept.

By expanding on the current design and addressing its limitations, the glove could evolve into a highly functional product with applications ranging from assisting individuals with disabilities to controlling devices in various domains. This project serves as a stepping stone toward more sophisticated wearable technology innovations.