DD-UART Protocol Frame Format

1. Frame Structure

`[START][LENGTH][COMMAND][DATA][CHECKSUM][END]`

2. Field Details

- **START**:
  - Description: Start byte, indicates the beginning of the message.
  - Value: `$` (ASCII)
  - Size: 1 byte
- **LENGTH**:
  - Description: Total length of the message, including all fields (START, LENGTH, COMMAND, DATA, CHECKSUM, END).
  - Value: Unsigned integer
  - Size: 2 bytes
- **COMMAND**:
  - Description: Action identifier (e.g., `READ_SENSOR`).
  - Value: Command code
  - Size: 2 bytes
- **DATA**:
  - Description: Command parameters or payload.
  - Value: Optional field; size varies depending on the message.
  - Size: 0–93 bytes
- **CHECKSUM**:
  - Description: Simple checksum for validation (e.g., XOR of all bytes except START and END).
  - Value: Unsigned integer
  - Size: 1 byte
- **END**:
  - Description: End byte, indicates the end of the message.
  - Value: `\n` (ASCII)
  - Size: 1 byte

Maximum Frame Size

- Minimum: `1 + 2 + 2 + 0 + 1 + 1 = 7 bytes`
- Maximum: `1 + 2 + 2 + 93 + 1 + 1 = 100 bytes`

Field Name Size (Bytes) Description

START 1 Start byte, `$` (ASCII), marks the beginning of the frame.

LENGTH 2 Total length of the frame, including all fields.

COMMAND 2 Action identifier (e.g., `READ_SENSOR`).

DATA 0–93 Optional payload or parameters for the command.

CHECKSUM 1 XOR or modular checksum for validation.

END 1 End byte, `\n` (ASCII), marks the end of the frame.

the table with the fields and their corresponding length in bytes:

[START]	[LENGTH]	[COMMAND]	[DATA (0–93 bytes)]	[CHECKSUM]	[END]
1	2	2	0–93	1	1

3. command list and their optional parameters:

Command	Description	Optional Parameter
`00`	Reset perimetral (sensors IMU, VL53L7)	None
`01`	Read IMU - Gyro value	None
`02`	Read IMU - Accelerometer value	None
`03`	Read Distance Sensor	`id` (e.g., `01`)
`04`	Read continuous distance sensor	`id` and frequency (in seconds)
`05`	Set MOTOR_FORWARD	None
`06`	Set MOTOR_REVERSE	None
`07`	Set MOTOR_STOP	None
`08`	Set MOTOR_SPEED	Speed of the motor in RPM

STM32 Implementation

Checksum and Length Validation

- **Steps**:
    1. Calculate the expected checksum and length from the received message.
    2. Compare the calculated values with the received ones.
- **Code Snippet**:

```
#define START_BYTE '$'
#define END_BYTE '\n'

// Updated Validation Function
int validate_message(uint8_t *message, uint16_t length) {
    // Check start and end bytes
    if (message[0] != START_BYTE || message[length - 1] != END_BYTE) {
        return 0; // Invalid start or end byte
    }

    // Validate length
    uint16_t received_length = (message[1] << 8) | message[2];
    if (received_length != length) {
        return 0; // Length mismatch
    }

    // Calculate checksum
    uint8_t checksum = 0;
    for (uint16_t i = 1; i < length - 2; i++) {
        checksum ^= message[i];
    }

    // Validate checksum
    if (checksum != message[length - 2]) {
        return 0; // Invalid checksum
    }

    return 1; // Message is valid
}

// Updated Command Processing
void process_message(uint8_t *message, uint16_t length) {
    if (!validate_message(message, length)) {
        printf("Invalid message!\n");
        return;
    }

    uint16_t command = (message[3] << 8) | message[4]; // Command field
    switch (command) {
        case 0x0001: // Read IMU - Gyro Value
            int16_t gyro_x = 9, gyro_y = -60, gyro_z = -66;
            send_response(0x0001, gyro_x, gyro_y, gyro_z);
            break;
        case 0x0002: // Read IMU - Accelerometer Value
            int16_t accel_x = 16519, accel_y = -1042, accel_z = 399;
            send_response(0x0002, accel_x, accel_y, accel_z);
```

```
            break;
        case 0x0003: // Read Distance Sensor
            // Example implementation for sensor ID
            uint8_t sensor_id = message[5];
            printf("Reading distance sensor ID: %d\n", sensor_id);
            break;
        default:
            printf("Unknown command\n");
    }
}

// Updated Send Response Function
void send_response(uint16_t command, int16_t x, int16_t y, int16_t z) {
    uint8_t message[100];
    uint16_t length = snprintf((char *)message + 5, sizeof(message) - 7, "%04X,%d,%d,%d",
command, x, y, z) + 7;

    message[0] = START_BYTE;
    message[1] = (length >> 8) & 0xFF; // High byte of length
    message[2] = length & 0xFF;       // Low byte of length
    message[3] = (command >> 8) & 0xFF;
    message[4] = command & 0xFF;

    uint8_t checksum = 0;
    for (uint16_t i = 1; i < length - 2; i++) {
        checksum ^= message[i];
    }

    message[length - 2] = checksum;
    message[length - 1] = END_BYTE;

    HAL_UART_Transmit(&huart1, message, length, HAL_MAX_DELAY);
}
```

PC Implementation

Python Example with Validation

```
import serial

START_BYTE = b'$'
END_BYTE = b'\n'

def calculate_checksum(message):
    """
    Calculate XOR checksum for the message.
    :param message: Bytes for checksum calculation.
```

```
    :return: Checksum byte.
    """
    checksum = 0
    for byte in message:
        checksum ^= byte
    return checksum

def create_message(command, data=None):
    """
    Create a DD-UART protocol-compliant message.
    :param command: 2-byte command ID (e.g., 0x0001).
    :param data: Optional data as a bytes object.
    :return: Complete message as bytes.
    """
    command_bytes = command.to_bytes(2, 'big')
    data = data if data else b''
    length = 7 + len(data)  # Start, length (2 bytes), command (2 bytes),
checksum, end
    length_bytes = length.to_bytes(2, 'big')
    message = START_BYTE + length_bytes + command_bytes + data
    checksum = calculate_checksum(message[1:])
    return message + bytes([checksum]) + END_BYTE

def validate_message(message):
    """
    Validate a received message.
    :param message: Full message as bytes.
    :return: True if valid, False otherwise.
    """
    # Validate start and end bytes
    if message[0:1] != START_BYTE or message[-1:] != END_BYTE:
        return False

    # Validate length
    length = int.from_bytes(message[1:3], 'big')
    if len(message) != length:
        return False

    # Validate checksum
    checksum = calculate_checksum(message[1:-2])
    if checksum != message[-2]:
        return False

    return True

def send_command(ser, command, data=None):
    """
    Send a command to the STM32.
    :param ser: Open serial connection.
    :param command: 2-byte command ID.
    :param data: Optional data as bytes.
    """
    message = create_message(command, data)
    ser.write(message)
    print(f"Sent: {message}")

def read_response(ser):
```

```
    """
    Read and validate a response from the STM32.
    :param ser: Open serial connection.
    :return: Decoded response if valid, None otherwise.
    """
    response = ser.read_until(END_BYTE)
    if validate_message(response):
        length = int.from_bytes(response[1:3], 'big')
        command = int.from_bytes(response[3:5], 'big')
        data = response[5:-2]
        return {
            'length': length,
            'command': command,
            'data': data
        }
    else:
        print("Invalid response received.")
        return None

# Example Usage
if __name__ == "__main__":
    ser = serial.Serial('COM3', 115200, timeout=1)

    # Send a command to read gyro
    send_command(ser, 0x0001)

    # Read and process the response
    response = read_response(ser)
    if response:
        print(f"Response: {response}")

    ser.close()
```

Benefits

1. **Robustness**: Length and checksum ensure messages are not corrupted during transmission.
2. **Scalability**: Easy to extend with new commands.
3. **Debugging**: Clear structure makes debugging straightforward.