

can you generate a crc16 function in c for me

Absolutely! Here's a simple and common implementation of **CRC-16 (Modbus)** in C. It's widely used, efficient, and easy to understand.

```
c                                     Copy code

#include <stdint.h>

// CRC-16 (Modbus) polynomial: 0x8005 (reversed 0xA001)
uint16_t crc16(const uint8_t *data, uint16_t length)
{
    uint16_t crc = 0xFFFF; // Initial value
    for (uint16_t i = 0; i < length; i++)
    {
        crc ^= data[i];
        for (uint8_t j = 0; j < 8; j++)
        {
            if (crc & 0x0001)
                crc = (crc >> 1) ^ 0xA001;
            else
                crc >>= 1;
        }
    }
    return crc;
}
```

```
from shared_queues import plotting_queue
import matplotlib.pyplot as plt
from typing import Dict

def plotting_thread():
    # Enable interactive mode
    plt.ion()
    fig, ax = plt.subplots()

    # Lists to store data
    timestamps = []
    a_x = []
    a_y = []
    a_z = []

    # Create line objects for each axis
    line_x, = ax.plot([], [], label='a_x')
    line_y, = ax.plot([], [], label='a_y')
    line_z, = ax.plot([], [], label='a_z')

    ax.set_xlabel('Timestamp (ms)')
    ax.set_ylabel('Acceleration (mg)')
    ax.set_title('Real-Time Accelerometer Data')
    ax.legend()

    while True:
        try:
            # Get the next accelerometer message from the queue
            accel_data: Dict = plotting_queue.get() # Blocks if queue is empty

            # Update lists with new data
            timestamps.append(accel_data["timestamp_ms"])
            a_x.append(accel_data["a_x_mg"])
            a_y.append(accel_data["a_y_mg"])
            a_z.append(accel_data["a_z_mg"])

            # Update line data
            line_x.set_data(timestamps, a_x)
            line_y.set_data(timestamps, a_y)
            line_z.set_data(timestamps, a_z)

        except KeyboardInterrupt:
            break
```

 Copy code

```
line_z, = ax.plot([], [], label='a_z')  
Copy code  
  
ax.set_xlabel('Timestamp (ms)')  
ax.set_ylabel('Acceleration (mg)')  
ax.set_title('Real-Time Accelerometer Data')  
ax.legend()  
  
while True:  
    try:  
        # Get the next accelerometer message from the queue  
        accel_data: Dict = plotting_queue.get() # Blocks if queue is empty  
  
        # Update lists with new data  
        timestamps.append(accel_data["timestamp_ms"])  
        a_x.append(accel_data["a_x_mg"])  
        a_y.append(accel_data["a_y_mg"])  
        a_z.append(accel_data["a_z_mg"])  
  
        # Update line data  
        line_x.set_data(timestamps, a_x)  
        line_y.set_data(timestamps, a_y)  
        line_z.set_data(timestamps, a_z)  
  
        # Adjust axes limits dynamically  
        ax.relim()  
        ax.autoscale_view()  
  
        # Draw updated figure  
        fig.canvas.draw()  
        fig.canvas.flush_events()  
  
        # Mark the queue item as done  
        plotting_queue.task_done()  
    except Exception as e:  
        print(f"Error in plotting thread: {e}")
```



can you give me a function to take an integer square root

Absolutely! Computing an **integer square root** means finding the largest integer r such that $r \cdot r \leq n$. There are several ways to do this efficiently in C without floating-point math. Here's a robust and simple implementation using **binary search**, which works for 32-bit integers:

C

 Copy code

```
#include <stdint.h>

// Integer square root of a 32-bit unsigned number
uint32_t isqrt(uint32_t n) {
    uint32_t low = 0;
    uint32_t high = n;
    uint32_t mid;
    uint32_t ans = 0;

    while (low <= high) {
        mid = low + ((high - low) / 2);
        uint64_t mid_squared = (uint64_t)mid * mid; // prevent overflow

        if (mid_squared == n) {
            return mid;
        } else if (mid_squared < n) {
            ans = mid; // store the last valid mid
            low = mid + 1; // search higher
        } else {
            high = mid - 1; // search lower
        }
    }

    return ans;
}
```



can you give me a python unpack function for
AccelDataMessage bytes received over TCP with this
format:

```
typedef struct __attribute__((packed)){  
    uint8_t header_byte_1;  
    uint8_t header_byte_2;  
    uint32_t sample_number;  
} TcpHeader;  
  
typedef struct __attribute__((packed)){  
    TcpHeader header;  
    uint64_t timestamp_ms;  
    int16_t a_x_mag;  
    int16_t a_y_mag;  
    int16_t a_z_mag;  
    uint32_t a_magnitude_counts;  
    uint16_t crc;  
} AccelDataMessage;
```

Absolutely! Since your C structs are **packed**, we can use Python's `struct` module to unpack the bytes directly. Here's a clean function for your `AccelDataMessage`:

python

 Copy code

```
import struct  
from typing import Dict  
  
# Define the struct format  
# '<' = little-endian (assume little-endian, adjust if needed)  
# B = uint8, H = uint16, I = uint32, Q = uint64, h = int16  
ACCEL_DATA_MESSAGE_FORMAT = '<BBI Q h h h I H'  
ACCEL_DATA_MESSAGE_SIZE = struct.calcsize(ACCEL_DATA_MESSAGE_FORMAT)  
  
def unpack_accel_data_message(data: bytes) -> Dict:
```

 Copy code

```
def unpack_accel_data_message(data: bytes) -> Dict:
    """
    Unpack a byte array representing an AccelDataMessage into a dict.

    Args:
        data (bytes): The byte array received over TCP.

    Returns:
        dict: Dictionary with keys corresponding to fields in AccelDataMessage.
    """
    if len(data) < ACCEL_DATA_MESSAGE_SIZE:
        raise ValueError(f"Data too short. Expected {ACCEL_DATA_MESSAGE_SIZE} bytes, got {len(data)}")

    unpacked = struct.unpack(ACCEL_DATA_MESSAGE_FORMAT, data[:ACCEL_DATA_MESSAGE_SIZE])

    return {
        "header_byte_1": unpacked[0],
        "header_byte_2": unpacked[1],
        "sample_number": unpacked[2],
        "timestamp_ms": unpacked[3],
        "a_x_mag": unpacked[4],
        "a_y_mag": unpacked[5],
        "a_z_mag": unpacked[6],
        "a_magnitude_counts": unpacked[7],
        "crc": unpacked[8]
    }

# Example usage
# raw_bytes = tcp_socket.recv(28) # example size
# message = unpack_accel_data_message(raw_bytes)
# print(message)
```

 Notes:

1. The format string `<BBI Q h h h I`  corresponds exactly to your C struct:
 - `BBI` → `header_byte_1 + header_byte_2 + sample_number`

Can you build me a source file for a tcp socket source file based on this header?

```
#ifndef TCP_INTERFACE_H
#define TCP_INTERFACE_H

#include <stdint.h>

typedef struct {
    int server_fd;
    int socket_fd;
} TcpConfig;

int tcp_server_init(TcpConfig* server, uint16_t port);

int tcp_send(TcpConfig* server, const uint8_t* data,
int data_len);

int tcp_receive(TcpConfig* server, uint8_t* buffer,
int buffer_size);

void tcp_close(TcpConfig* server);

#endif // TCP_INTERFACE_H
```



Absolutely! I can write a robust TCP socket source file (`tcp_interface.c`) that implements your header. I'll include proper error handling, `bind()`, `listen()`, `accept()`, and simple send/receive functions. Here's a clean implementation:

```
int tcp_server_init(TcpConfig* server, uint16_t port)
{
    if (!server) return -1;

    // Create server socket
    server->server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server->server_fd < 0) {
        perror("socket");
        return -1;
    }

    // Allow reuse of address and port
    int opt = 1;
    if (setsockopt(server->server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)) < 0)
        perror("setsockopt");
    close(server->server_fd);
    return -1;
}

// Bind socket to port
struct sockaddr_in address;
memset(&address, 0, sizeof(address));
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(port);    ↓
```

```
if (bind(server->server_fd, (struct sockaddr*)&address, sizeof(address)) < 0) {
    perror("bind");
    close(server->server_fd);
    return -1;
}

// Listen for incoming connections
if (listen(server->server_fd, 1) < 0) {
    perror("listen");
    close(server->server_fd);
    return -1;
}

printf("Server listening on port %d...\n", port);

// Accept a single client connection (blocking)
socklen_t addrlen = sizeof(address);
server->socket_fd = accept(server->server_fd, (struct sockaddr*)&address, &addrlen);
if (server->socket_fd < 0) {
    perror("accept");
    close(server->server_fd);
    return -1;
}

printf("Client connected.\n");
return 0;
}

int tcp_send(TcpConfig* server, const uint8_t* data, int data_len)
{
    if (!server || server->socket_fd < 0 || !data || data_len <= 0)
        return -1;

    int total_sent = 0;
    while (total_sent < data_len) {
        int sent = send(server->socket_fd, data + total_sent, data_len - total_sent);
        if (sent < 0) {
            perror("send");
            close(server->socket_fd);
            return -1;
        }
        total_sent += sent;
    }
}
```

```
int total_sent = 0; Copy code
while (total_sent < data_len) {
    int sent = send(server->socket_fd, data + total_sent, data_len);
    if (sent < 0) {
        perror("send");
        return -1;
    }
    total_sent += sent;
}
return total_sent;
}

int tcp_receive(TcpConfig* server, uint8_t* buffer, int buffer_size)
{
    if (!server || server->socket_fd < 0 || !buffer || buffer_size < 0)
        return -1;

    int received = recv(server->socket_fd, buffer, buffer_size, 0);
    if (received < 0) {
        perror("recv");
    }
    return received;
}

void tcp_close(TcpConfig* server)
{
    if (!server) return;

    if (server->socket_fd >= 0) {
        close(server->socket_fd);
        server->socket_fd = -1;
    }
    if (server->server_fd >= 0) {
        close(server->server_fd);
        server->server_fd = -1;
    }
}
```