Modbus communication C/C++ code for TCP socket without libmodbus-dev library

Implementing Modbus TCP communication over a socket in C or C++ involves opening a TCP socket, constructing Modbus protocols, and sending/receiving data through the socket. Let's create a basic example for reading and writing Modbus registers using a direct socket connection without using the libmodbus-dev library.

**Basic Requirements**

* A TCP/IP connection.
* Knowledge of the Modbus TCP protocol for framing and addressing.

**Socket Setup**

First, we'll set up the TCP socket for communication.

**Socket Initialization**

#include <iostream>

#include <cstring>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

// Function prototypes

int initializeSocket(const char\* ip, int port);

int main() {

const char\* ip = "192.168.1.100"; // Change to your target IP

int port = 502; // Modbus TCP default port

int sockfd = initializeSocket(ip, port);

if (sockfd == -1) {

return -1;

}

// Additional code for Modbus request and response will go here

// Close the socket

close(sockfd);

return 0;

}

int initializeSocket(const char\* ip, int port) {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

std::cerr << "Failed to create socket: " << strerror(errno) << std::endl;

return -1;

}

struct sockaddr\_in serverAddr;

memset(&serverAddr, 0, sizeof(serverAddr));

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(port);

if (inet\_pton(AF\_INET, ip, &serverAddr.sin\_addr) <= 0) {

std::cerr << "Invalid address: " << ip << std::endl;

close(sockfd);

return -1;

}

if (connect(sockfd, (struct sockaddr\*)&serverAddr, sizeof(serverAddr)) < 0) {

std::cerr << "Connection failed: " << strerror(errno) << std::endl;

close(sockfd);

return -1;

}

return sockfd;

}

C++

Copy code

**Constructing Modbus TCP Requests and Responses**

Here's how to construct and send a Modbus TCP request.

**Function to Write a Single Register**

bool writeRegister(int sockfd, uint8\_t slaveID, uint16\_t registerAddress, uint16\_t value) {

uint8\_t frame[12];

static uint16\_t transactionID = 0; // Unique transaction ID

transactionID++;

// Construct Modbus TCP ADU

frame[0] = (transactionID >> 8) & 0xFF; // Transaction ID High Byte

frame[1] = transactionID & 0xFF; // Transaction ID Low Byte

frame[2] = 0x00; // Protocol ID High Byte

frame[3] = 0x00; // Protocol ID Low Byte

frame[4] = 0x00; // Length High Byte

frame[5] = 0x06; // Length Low Byte (6 bytes to follow)

frame[6] = slaveID; // Unit ID (slave address)

frame[7] = 0x06; // Function Code (Write Single Register)

frame[8] = (registerAddress >> 8) & 0xFF; // Register Address High Byte

frame[9] = registerAddress & 0xFF; // Register Address Low Byte

frame[10] = (value >> 8) & 0xFF; // Value High Byte

frame[11] = value & 0xFF; // Value Low Byte

// Send frame to the socket

if (send(sockfd, frame, 12, 0) == -1) {

std::cerr << "Failed to send frame: " << strerror(errno) << std::endl;

return false;

}

// Receive response

uint8\_t response[12];

if (recv(sockfd, response, 12, 0) != 12) {

std::cerr << "Failed to receive response: " << strerror(errno) << std::endl;

return false;

}

// Verify response (could add additional checks)

if (response[7] != 0x06) {

std::cerr << "Invalid response function code: " << (int)response[7] << std::endl;

return false;

}

return true;

}

C++

Copy code

**Function to Read Holding Registers**

bool readRegisters(int sockfd, uint8\_t slaveID,

C++

Copy code

Copy

Good response

Bad response

continue

Here's the continuation of the readRegisters function along with the integrated example:

bool readRegisters(int sockfd, uint8\_t slaveID, uint16\_t startAddress, uint16\_t numRegisters, uint16\_t \*buffer) {

uint8\_t frame[12];

static uint16\_t transactionID = 0; // Unique transaction ID

transactionID++;

// Construct Modbus TCP ADU

frame[0] = (transactionID >> 8) & 0xFF; // Transaction ID High Byte

frame[1] = transactionID & 0xFF; // Transaction ID Low Byte

frame[2] = 0x00; // Protocol ID High Byte

frame[3] = 0x00; // Protocol ID Low Byte

frame[4] = 0x00; // Length High Byte

frame[5] = 0x06; // Length Low Byte (6 bytes to follow)

frame[6] = slaveID; // Unit ID (slave address)

frame[7] = 0x03; // Function Code (Read Holding Registers)

frame[8] = (startAddress >> 8) & 0xFF; // Start Address High Byte

frame[9] = startAddress & 0xFF; // Start Address Low Byte

frame[10] = (numRegisters >> 8) & 0xFF; // Number of Registers High Byte

frame[11] = numRegisters & 0xFF; // Number of Registers Low Byte

// Send frame to the socket

if (send(sockfd, frame, 12, 0) == -1) {

std::cerr << "Failed to send frame: " << strerror(errno) << std::endl;

return false;

}

// Calculate expected response length: Transaction ID + Protocol ID + Length + Unit ID + Function Code + Byte Count + Data

int responseLength = 9 + 2 \* numRegisters; // Fixed header is 9 bytes + data

uint8\_t response[responseLength];

// Receive response

if (recv(sockfd, response, responseLength, 0) != responseLength) {

std::cerr << "Failed to receive response: " << strerror(errno) << std::endl;

return false;

}

// Verify response (could add additional checks)

if (response[7] != 0x03) {

std::cerr << "Invalid response function code: " << (int)response[7] << std::endl;

return false;

}

// Extract data

for (int i = 0; i < numRegisters; ++i) {

buffer[i] = (response[9 + 2 \* i] << 8) | response[10 + 2 \* i];

}

return true;

}

int main() {

const char\* ip = "192.168.1.100"; // Change to your target IP

int port = 502; // Modbus TCP default port

int sockfd = initializeSocket(ip, port);

if (sockfd == -1) {

return -1;

}

uint8\_t slaveID = 1;

uint16\_t registerAddress = 0;

uint16\_t value = 1234;

// Write a single holding register

if (!writeRegister(sockfd, slaveID, registerAddress, value)) {

std::cerr << "Failed to write register" << std::endl;

close(sockfd);

return -1;

}

// Read holding registers

const int numRegisters = 5;

uint16\_t buffer[numRegisters] = {0};

if (!readRegisters(sockfd, slaveID, registerAddress, numRegisters, buffer)) {

std::cerr << "Failed to read registers" << std::endl;

close(sockfd);

return -1;

}

// Print the read registers

std::cout << "Read registers:" << std::endl;

for (int i = 0; i < numRegisters; ++i) {

std::cout << "Register " << i << ": " << buffer[i] << std::endl;

}

// Close the socket

close(sockfd);

return 0;

}

C++

Copy code

**Summary**

This example provides a basic implementation of Modbus TCP communication over a socket in C++. It includes:

1. Establishing a TCP connection to the Modbus server.
2. Writing a single holding register using Modbus function code 0x06.
3. Reading multiple holding registers using Modbus function code 0x03.
4. Sending and receiving the appropriate Modbus Application

Copy

Good response

Bad response

continue

Modbus Application Data Unit (ADU) frames over the TCP socket, verifying responses, and extracting register values.

Let's break down the key components included in the example:

1. **Socket Initialization**:
   * Opens a TCP socket and connects to the specified IP and port.
2. **Modbus Frame Construction**:
   * Constructs the Modbus TCP ADU frames for both writing a single register (writeRegister) and reading multiple registers (readRegisters).
3. **Frame Transmission**:
   * Sends the Modbus frame to the server using the socket.
4. **Response Handling**:
   * Receives the response from the server.
   * Parses and validates the response to ensure it adheres to the expected Modbus format.
5. **Data Extraction**:
   * Extracts data from the valid response frames and stores it in the provided buffer.

**Completing the Example**

To ensure the code is self-contained and functional, you can follow the entire setup as described:

Here is the complete example code:

#include <iostream>

#include <cstring>

#include <unistd.h>

#include <arpa/inet.h>

#include <sys/socket.h>

// Function prototypes

int initializeSocket(const char\* ip, int port);

bool writeRegister(int sockfd, uint8\_t slaveID, uint16\_t registerAddress, uint16\_t value);

bool readRegisters(int sockfd, uint8\_t slaveID, uint16\_t startAddress, uint16\_t numRegisters, uint16\_t \*buffer);

int main() {

const char\* ip = "192.168.1.100"; // Change to your target IP

int port = 502; // Modbus TCP default port

int sockfd = initializeSocket(ip, port);

if (sockfd == -1) {

return -1;

}

uint8\_t slaveID = 1;

uint16\_t registerAddress = 0;

uint16\_t value = 1234;

// Write a single holding register

if (!writeRegister(sockfd, slaveID, registerAddress, value)) {

std::cerr << "Failed to write register" << std::endl;

close(sockfd);

return -1;

}

// Read holding registers

const int numRegisters = 5;

uint16\_t buffer[numRegisters] = {0};

if (!readRegisters(sockfd, slaveID, registerAddress, numRegisters, buffer)) {

std::cerr << "Failed to read registers" << std::endl;

close(sockfd);

return -1;

}

// Print the read registers

std::cout << "Read registers:" << std::endl;

for (int i = 0; i < numRegisters; ++i) {

std::cout << "Register " << i << ": " << buffer[i] << std::endl;

}

// Close the socket

close(sockfd);

return 0;

}

int initializeSocket(const char\* ip, int port) {

int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

if (sockfd == -1) {

std::cerr << "Failed to create socket: " << strerror(errno) << std::endl;

return -1;

}

struct sockaddr\_in serverAddr;

memset(&serverAddr, 0, sizeof(serverAddr));

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_port = htons(port);

if (inet\_pton(AF\_INET, ip, &serverAddr.sin\_addr) <= 0) {

std::cerr << "Invalid address: " << ip << std::endl;

close(sockfd);

return -1;

}

if (connect(sockfd, (struct sockaddr\*)&serverAddr, sizeof(serverAddr)) < 0) {

std::cerr << "Connection failed: " << strerror(errno) << std::endl;

close(sockfd);

return -1;

}

return sockfd;

}

bool writeRegister(int sockfd, uint8\_t slaveID, uint16\_t registerAddress, uint16\_t value) {

uint8\_t frame[12];

static uint16\_t transactionID = 0; // Unique transaction ID

transactionID++;

// Construct Modbus TCP ADU

frame[0] = (transactionID >> 8) & 0xFF; // Transaction ID High Byte

frame[1] = transactionID & 0xFF; // Transaction ID Low Byte

frame[2] = 0x00; // Protocol ID High Byte

frame[3] = 0x00; // Protocol ID Low Byte

frame[4] = 0x00; // Length High Byte

frame[5] = 0x06; // Length Low Byte (6 bytes to

C++

Copy code

Copy

Good response

Bad response

continue

Certainly! Here's the continuation and completion of the writeRegister function and the complete program:

frame[5] = 0x06; // Length Low Byte (6 bytes to follow)

frame[6] = slaveID; // Unit ID (slave address)

frame[7] = 0x06; // Function Code (Write Single Register)

frame[8] = (registerAddress >> 8) & 0xFF; // Register Address High Byte

frame[9] = registerAddress & 0xFF; // Register Address Low Byte

frame[10] = (value >> 8) & 0xFF; // Value High Byte

frame[11] = value & 0xFF; // Value Low Byte

// Send frame to the socket

if (send(sockfd, frame, 12, 0) == -1) {

std::cerr << "Failed to send frame: " << strerror(errno) << std::endl;

return false;

}

// Receive response

uint8\_t response[12];

if (recv(sockfd, response, 12, 0) != 12) {

std::cerr << "Failed to receive response: " << strerror(errno) << std::endl;

return false;

}

// Verify response (could add additional checks)

if (response[7] != 0x06) {

std::cerr << "Invalid response function code: " << (int)response[7] << std::endl;

return false;

}

return true;

}

bool readRegisters(int sockfd, uint8\_t slaveID, uint16\_t startAddress, uint16\_t numRegisters, uint16\_t \*buffer) {

uint8\_t frame[12];

static uint16\_t transactionID = 0; // Unique transaction ID

transactionID++;

// Construct Modbus TCP ADU

frame[0] = (transactionID >> 8) & 0xFF; // Transaction ID High Byte

frame[1] = transactionID & 0xFF; // Transaction ID Low Byte

frame[2] = 0x00; // Protocol ID High Byte

frame[3] = 0x00; // Protocol ID Low Byte

frame[4] = 0x00; // Length High Byte

frame[5] = 0x06; // Length Low Byte (6 bytes to follow)

frame[6] = slaveID; // Unit ID (slave address)

frame[7] = 0x03; // Function Code (Read Holding Registers)

frame[8] = (startAddress >> 8) & 0xFF; // Start Address High Byte

frame[9] = startAddress & 0xFF; // Start Address Low Byte

frame[10] = (numRegisters >> 8) & 0xFF; // Number of Registers High Byte

frame[11] = numRegisters & 0xFF; // Number of Registers Low Byte

// Send frame to the socket

if (send(sockfd, frame, 12, 0) == -1) {

std::cerr << "Failed to send frame: " << strerror(errno) << std::endl;

return false;

}

// Calculate expected response length: Transaction ID + Protocol ID + Length + Unit ID + Function Code + Byte Count + Data

int responseLength = 9 + 2 \* numRegisters; // Fixed header is 9 bytes + data

uint8\_t response[responseLength];

// Receive response

if (recv(sockfd, response, responseLength, 0) != responseLength) {

std::cerr << "Failed to receive response: " << strerror(errno) << std::endl;

return false;

}

// Verify response (could add additional checks)

if (response[7] != 0x03) {

std::cerr << "Invalid response function code: " << (int)response[7] << std::endl;

return false;

}

// Extract data

for (int i = 0; i < numRegisters; ++i) {

buffer[i] = (response[9 + 2 \* i] << 8) | response[10 + 2 \* i];

}

return true;

}

int main() {

const char\* ip = "192.168.1.100"; // Change to your target IP

int port = 502; // Modbus TCP default port

int sockfd = initializeSocket(ip, port);

if (sockfd

C++

Copy code

Copy

Good response

Bad response

continue

Certainly! Let's continue and complete the main function and finalize the program implementation:

if (sockfd == -1) {

return -1;

}

uint8\_t slaveID = 1;

uint16\_t registerAddress = 0;

uint16\_t value = 1234;

// Write a single holding register

if (!writeRegister(sockfd, slaveID, registerAddress, value)) {

std::cerr << "Failed to write register" << std::endl;

close(sockfd);

return -1;

}

// Read holding registers

const int numRegisters = 5;

uint16\_t buffer[numRegisters] = {0};

if (!readRegisters(sockfd, slaveID, registerAddress, numRegisters, buffer)) {

std::cerr << "Failed to read registers" << std::endl;

close(sockfd);

return -1;

}

// Print the read registers

std::cout << "Read registers:" << std::endl;

for (int i = 0; i < numRegisters; ++i) {

std::cout << "Register " << i << ": " << buffer[i] << std::endl;

}

// Close the socket

close(sockfd);

return 0;

}

C++

Copy code

**Explanation of the Complete Code:**

1. **Main Function:**
   * This function serves as the entry point for the program.
   * It initializes the socket using the initializeSocket function.
   * It assigns a slave ID, and register address, and the value to be written.
   * It writes a value to a single holding register using the writeRegister function.
   * It reads multiple holding registers starting from the same address using the readRegisters function.
   * It prints the values of the read registers.
   * Finally, it closes the socket.
2. **Initialize Socket Function (initializeSocket):**
   * It creates a socket and connects it to the specified IP address and port.
   * It returns the socket file descriptor if successful, or -1 if it fails.
3. **Write Register Function (writeRegister):**
   * It constructs a Modbus frame for writing a single holding register.
   * It sends the frame to the Modbus server using the socket.
   * It receives the response from the server and verifies it.
   * It returns true if successful, or false if it fails.
4. **Read Registers Function (readRegisters):**
   * It constructs a Modbus frame for reading multiple holding registers.
   * It sends the frame to the Modbus server using the socket.
   * It calculates the expected response length based on the number of registers to read.
   * It receives the response from the server and verifies it.
   * It extracts the register values from the response and stores them in the buffer.
   * It returns true if successful, or false if it fails.

**Final Remarks:**

* **Error Handling:** The code includes basic error handling (e.g., checking return values of socket operations). You might want to enhance this based on specific requirements.
* **Expandability:** The basic structure allows for easy expansion to support more Modbus functions or custom protocols.

Is there anything specific you want to adjust or expand upon in this implementation?