****

**Technical university of Sofia**

Faculty of German Engineering Education and Industrial Management

Mechatronics and Information technology

**THESIS**

Development of a blockchain system for processing sensory information

Student: Sladun Arsenov

Faculty number: 901219011

Supervisor: Dr. Aleksandar Marinchev

Sofia Date: 10.09.2023

Empty Page

**Declaration**

The undersigned, Sladun Kamenov Arsenov, faculty number 901219011, declare that the thesis presented by me on the development of “blockchain system for the processing of sensory information” was developed independently.In the work, I have complied with the copyright of other sources or resources that I have used and cited in the text. I have not used any other copyrighted materials other than those listed in the "References" section. I declare, that the thesis does not represent the achievement of the previous sessions of the thesis.

Declarator: Date: 10.09.2023

Sofia

List of abbreviations

API Application Programming Interface

ARP Address Resolution Protocol

Build tools Software automating code compilation and packaging

cmd Command

DHCP Dynamic Host Configuration Protocol

DLL Dynamic Link Library

ESP-IDF Espressif IoT Development Framework

FTP File Transfer Protocol

GCC GNU Compiler Collection

Git No abbreviation (Git is the name of the version control system)

Hash Value, derived from data for security

HTTP Hypertext Transfer Protocol

IANA Internet Assigned Numbers Authority

IP Internet Protocol

IPv4 Internet Protocol version 4

IoT Internet of Things

MAC Media Access Control

MinGW Minimalist GNU for Windows

mutex Mutual Exclusion

Node A device in a network for data processing

OSI Open Systems Interconnection

P2P Peer-to-Peer

PoS Proof of Stake

PoW Proof of Work

TCP Transmission Control Protocol

Toolchain A set of development tools for creating software

UML Unified Modeling Language

UDP User Datagram Protocol

**Table of Contents**

[​ 1. Introduction 6](#__RefHeading___Toc3315_1122408748)

[​ 2. Theory: 7](#__RefHeading___Toc3317_1122408748)

[2.1 What is a blockchain? 7](#__RefHeading___Toc3319_1122408748)

[2.1.1 Key Components of a blockchain: 7](#__RefHeading___Toc3321_1122408748)

[2.1.2 How a Blockchain Works: 7](#__RefHeading___Toc3323_1122408748)

[2.1.3 Use Cases of Blockchain: 8](#__RefHeading___Toc3325_1122408748)

[2.2 What is ESP32-C6 ? 8](#__RefHeading___Toc3327_1122408748)

[2.3 Networking 10](#__RefHeading___Toc3329_1122408748)

[2.3.1 Introduction 10](#__RefHeading___Toc3331_1122408748)

[​ 3. Software Architecture 11](#__RefHeading___Toc3335_1122408748)

[3.1 Introduction 11](#__RefHeading___Toc3337_1122408748)

[3.2 Architectural Goals and Constraints 12](#__RefHeading___Toc3339_1122408748)

[3.3 Prerequisites 13](#__RefHeading___Toc3341_1122408748)

[3.4 Folder structure 15](#__RefHeading___Toc3343_1122408748)

[3.5 Short introduction to the C Programming language syntax 17](#__RefHeading___Toc3345_1122408748)

[3.6 Structures overview – computer side 19](#__RefHeading___Toc3347_1122408748)

[3.7 Code overview – computer side 22](#__RefHeading___Toc3349_1122408748)

[3.7.2 Start 23](#__RefHeading___Toc3351_1122408748)

[3.7.4 Parallelism/Concurrency 30](#__RefHeading___Toc3353_1122408748)

[3.8 Code overview – ESP32-C6 side 35](#__RefHeading___Toc3355_1122408748)

[​ 4. Experimental studies. 40](#__RefHeading___Toc3357_1122408748)

[4.1 Achieving the goals 42](#__RefHeading___Toc3359_1122408748)

[​ 5. Conclusion 43](#__RefHeading___Toc3361_1122408748)

[​ 6. Sources: 44](#__RefHeading___Toc3363_1122408748)

[​ 7. Source code 45](#__RefHeading___Toc3377_1122408748)

# 1. Introduction

In an era driven by data, the effective collection, secure storage, and responsible sharing of sensory information is of great importance. The project’s aim is to transform the way sensory data is handled by leveraging the power of blockchain technology. There are four main objectives that the project seeks to achieve:

1) Development of decentralized blockchain system that runs on multiple computers, collecting sensory information.

2) Connecting and sending data from the sensors to the computers.

3) Implementing a robust security measures to protect sensory data from unauthorized access and tampering.

4) Designing an intuitive way for the collected information to be used.

Achieving all of these points will be better explained in the next chapters.

There are also multiple ways, in which the blockchain project can be useful to numerous industries. Some of the examples are:

1) Environmental Monitoring: By providing real-time data on air quality, temperature, and pollution, the environmental challenges can be better understood and addressed.

2) Healthcare: Continuous monitoring of vital signs and health metrics can lead to more proactive and personalized healthcare.

3) Industry: Factories and industrial facilities can benefit from real-time sensor data to optimize operations, improve safety, and reduce downtime.

Blockchain technology provides a couple of solid qualities that can improve the foundation of the project’s needs:

1) Security: The stored data is tamper-proof, ensuring integrity.

2) Decentralization: A distributed network of nodes (computers), reducing central points of failure.

3) Privacy: Users have control over the data, sharing it securely when they choose.

4) Interoperability: Different sensory data metrics from various sources and devices can be integrated seamlessly.

# 2. Theory:

## 2.1 What is a blockchain?

A blockchain is a revolutionary distributed and decentralized digital ledger technology that serves as a transparent and secure method for recording and verifying transactions across a network of computers. The name "blockchain" is derived from its fundamental structure, consisting of a series of interconnected blocks, each containing a collection of transactions. These transactions can represent various forms of data, such as cryptocurrency transfers, digital contracts, or any digital record. The size and capacity of each block may vary depending on the specific blockchain's design and use case.

### 2.1.1 Key Components of a blockchain:

- Blocks: The building blocks of a blockchain, these contain groups of transactions. Each transaction within a block represents a digital record of an event or action. In blockchain networks, such as those used for cryptocurrencies, these transactions often involve the transfer of digital assets or tokens.

- Chain: Blocks in a blockchain are not isolated entities; rather, they are intricately linked together in a sequential order. Each block contains a unique reference known as a cryptographic hash, which securely points to the previous block in the chain. This linkage ensures the integrity, immutability, and chronological order of all transactions within the blockchain.

- Decentralization: A hallmark characteristic of blockchain is its decentralized nature. Unlike traditional systems that rely on a central authority or intermediary, blockchain networks operate on a vast network of distributed nodes. Each node independently maintains a complete copy of the entire blockchain ledger. This decentralization enhances security, reduces the risk of censorship, and eliminates single points of failure.

- Consensus Mechanism: To add new transactions to the blockchain, nodes must collectively agree on their validity. This consensus is achieved through a defined mechanism, which can vary from one blockchain network to another. Common consensus mechanisms include Proof of Work (PoW) and Proof of Stake (PoS). These mechanisms ensure that only valid transactions are added to the blockchain.

- Cryptographic Security: Blockchain relies on advanced cryptographic techniques to secure data. Every transaction is cryptographically signed, guaranteeing its authenticity and integrity. Furthermore, blocks are interconnected through cryptographic hashes, making it exceedingly challenging for malicious actors to alter or manipulate data.

### 2.1.2 How a Blockchain Works:

The operational workflow of a blockchain involves several key steps:

- Transaction Creation: Users initiate transactions by creating digital records. For example, in a cryptocurrency blockchain, this could involve transferring tokens from one user to another.

- Transaction Verification: Transactions are broadcast to the network of nodes. Each node independently verifies the validity of transactions, checking factors like digital signatures and available funds.

- Block Creation: Valid transactions are grouped together into a block. Miners (in PoW blockchains) or validators (in PoS blockchains) compete to solve complex mathematical puzzles. The first to solve the puzzle gets the right to create the next block.

- Consensus: The newly created block is broadcast to the network. Other nodes validate the block's transactions and reach consensus that the block is valid. If consensus is reached, the block is added to the blockchain.

- Linking Blocks: The new block contains a reference (hash) to the previous block, creating a link. This link is crucial for maintaining the chronological order of transactions.

- Repeat: The process continues with new transactions being added to new blocks, which are sequentially linked to the previous ones. This forms a growing and immutable blockchain.

### 2.1.3 Use Cases of Blockchain:

Blockchain technology has transcended its origins in cryptocurrency and has found applications across a diverse spectrum of industries and sectors, including but not limited to:

- Cryptocurrency: The most well-known use case, with cryptocurrencies like Bitcoin and Ethereum operating on blockchain technology.

- Smart Contracts: Self-executing contracts with predefined conditions, automating various types of agreements.

- Supply Chain Management: Tracking and verifying the movement of goods and products throughout the supply chain.

- Digital Identity: Creating secure and verifiable digital identities for individuals.

- Voting Systems: Developing secure and transparent voting systems.

- Healthcare: Storing and sharing electronic health records securely.

- Asset Tokenization: Representing real-world assets (e.g., real estate, stocks) as digital tokens for efficient trading.

The essence of blockchain lies in its capacity to deliver secure, transparent, and tamper-resistant record-keeping. As blockchain technology continues to evolve and mature, it holds the potential to revolutionize an array of industries, redefining how we interact with digital data, transactions, and trust in a rapidly advancing digital age.

## 2.2 What is ESP32-C6 ?

ESP32-C6 is a system on a chip, created by “Espressif Systems”that integrates the following features:

* Wi-Fi 6 (2.4 GHz band)
* Bluetooth Low Energy
* 802.15.4 Thread/Zigbee
* High performance 32-bit RISC-V single-core processor
* Multiple peripherals
* Built-in security hardware

Powered by 40 nm technology, ESP32-C6 provides a robust, highly integrated platform, which helps meet the continuous demands for efficient power usage, compact design, security, high performance, and reliability.

Espressif provides basic hardware and software resources to help application developers realize their ideas using the ESP32-C6 series hardware. The software development framework by Espressif is intended for development of Internet-of-Things (IoT) applications with Wi-Fi, Bluetooth, power management and several other system features.

## 2.3 Networking

### 2.3.1 Introduction

Computer Networking is the practice of connecting computers together to enable communication and data exchange between them. In general, Computer Network is a collection of two or more computers. It helps users to communicate more easily. In this article, we are going to discuss the basics which everyone must know before going deep into Computer Networking.

2.3.2 How Does a Computer Network Work?

Basics building blocks of a Computer network are Nodes and Links. A Network Node can be illustrated as Equipment for Data Communication like a Modem, Router, etc., or Equipment of a Data Terminal like connecting two computers or more. Link in Computer Networks can be defined as wires or cables or free space of wireless networks.

The working of Computer Networks can be simply defined as rules or protocols which help in sending and receiving data via the links which allow Computer networks to communicate. Each device has an IP Address, that helps in identifying a device.

2.3.3 Basic Terminologies of Computer Networks

* Network: A network is a collection of computers and devices that are connected together to enable communication and data exchange.
* Nodes: Nodes are devices that are connected to a network. These can include computers, Servers, Printers, Routers, Switches, and other devices.
* Protocol: A protocol is a set of rules and standards that govern how data is transmitted over a network. Examples of protocols include TCP/IP, HTTP, and UDP.
* Topology: Network topology refers to the physical and logical arrangement of nodes on a network. The common network topologies include bus, star, ring, mesh, and tree.
* Service Provider Networks: These types of Networks give permission to take Network Capacity and Functionality on lease from the Provider. Service Provider Networks include Wireless Communications, Data Carriers, etc.
* IP Address: An IP address is a unique numerical identifier that is assigned to every device on a network. IP addresses are used to identify devices and enable communication between them.
* Firewall: A firewall is a security device that is used to monitor and control incoming and outgoing network traffic. Firewalls are used to protect networks from unauthorized access and other security threats.

**2.3.4 Types of Computer Network Architecture**

Computer Network falls under these broad Categories:

* **Client-Server Architecture:** Client-Server Architecture is a type of Computer Network Architecture in which Nodes can be Servers or Clients. Here, the server node can manage the Client Node Behavior.
* **Peer-to-Peer Architecture:** In P2P (Peer-to-Peer) Architecture there is not any concept of a Central Server. Each device is free for working as either client or server.

**2.3.5 Network Devices**

An interconnection of multiple devices, also known as hosts, that are connected using multiple paths for the purpose of sending/receiving data or media. Computer networks can also include multiple devices/mediums which help in the communication between two different devices; these are known as Network devices and include things such as routers, switches, hubs, and bridges.

**2.3.6 Protocol**

A protocol is a set of rules or algorithms which define the way how two entities can communicate across the network and there exists a different protocol defined at each layer of the OSI model. A few such protocols are TCP, IP, UDP, ARP, DHCP, FTP, and so on.

**2.3.7 Unique Identifiers of Network**

- IP Address (Internet Protocol address): Also known as the Logical Address, the IP Address is the network address of the system across the network. To identify each device in the world-wide-web, the Internet Assigned Numbers Authority (IANA) assigns an IPV4 (Version 4) address as a unique identifier to each device on the Internet. The length of an IPv4 address is 32 bits, hence, there are 232 IP addresses available. The length of an IPv6 address is 128 bits.  
Typing “ipconfig” in the command prompt and pressing ‘Enter’, gives the IP address of the device.

- MAC Address (Media Access Control address): Also known as physical address, the MAC Address is the unique identifier of each host and is associated with its NIC (Network Interface Card). A MAC address is assigned to the NIC at the time of manufacturing. The length of the MAC address is: 12-nibble/ 6 bytes/ 48 bits. Typing “ipconfig/all” in the command prompt and pressing ‘Enter’, gives the MAC address.

- Port: A port can be referred to as a logical channel through which data can be sent/received to an application. Any host may have multiple applications running, and each of these applications is identified using the port number on which they are running. A port number is a 16-bit integer, hence, there are 216 ports available.

# 3. Software Architecture

## 3.1 Introduction

This section provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions, which have been made on the system.

## 3.2 Architectural Goals and Constraints

There are some key requirements and systems constraints that have significant bearing on the architecture. They are:

1) When the program starts, the node should send a message, asking to join the blockchain network.

2) If it doesn’t get the blockchain, meaning that it is the first user, it should generate it.

3) If it is not the first node, it should receive the blockchain.

4) After joining the system, three parallel actions should start

5) It should start listening for other nodes that want to join, and send them the blockchain.

5) Parallel to that, block generation should be happening, calculating a mathematical problem for hash generation.

6) Parallel to that, data transactions should be happening, filling them in the currently generating block’s transaction pool.

7) There should be an ESP32-C6 program, that collects data and sends it to the nodes.

9) After successful block generation, the process should start repeating.

Security is also an important factor. Key requirements are:

1) Decentralization

2) Cryptographic hash codes, for every block

3) Tamper proof blocks

4) Tamper proof chain between the blocks

5) Secure sharing of output data

## 3.3 Prerequisites

A couple of questions should be asked, and a couple of decisions should be made, before the development of the project has begun:

**1) Why was this particular project chosen?**

Security of data is of great importance in these days. By collecting the data inside a blockchain, the security is massively increased. Nobody can tamper with the data, and the data can be shared whenever and however the user wants to. That’s why blockchain systems are used for handling the sensory information.

The microcontroller, ESP32-C6 is a compact but powerful microcontroller, which has access to WiFi and has sufficient computing power. But the project’s idea is not limited to just this microcontroller. Many other types of data from various sources can be collected and securely stored.

The project’s idea can be used in various fields, and can easily be modified and adapted for the needed purposes.

**2) What operating system should the project be written for?**

Currently, the computer part of the project can only be executed on Windows, because Windows is the most common operating system right now, but it can effortlessly be adapted to other operating systems.

**3) What programming language should be used and why?**

The C programming language is used in the development of both the computer part and the microcontroller part. It is selected because of it’s features, that make it better than the rest of the languages:

- C is simple and effective. The basic syntax style C is simple and easy to learn. This makes the language easily comprehensible and enables a programmer to modify the application in a timely manner.

- Speed: C is one of the, if not, the fastest programming language. Statically typed programming languages are faster than dynamic ones. C is a statically typed programming language, which gives it an edge over other dynamic languages. Also, unlike Java and Python, which are interpreter-based, C is a compiler-based program. This makes the compilation and execution of codes faster.

Another factor that makes C fast is the availability of only the essential and required features. Newer programming languages come with numerous features, which increase functionality but reduce efficiency and speed. Since C offers limited but essential features, the headache of processing these features reduces, resulting in increased speed.

- Portability: Another feature of the C language is portability. To put it simply, C programs are machine-independent which means that you can run the fraction of a code created in C on various machines with none or some machine-specific changes. Hence, it provides the functionality of using a single code on multiple systems depending on the requirement.

- Extensibility: You can easily (and quickly) extend a C program. This means that if a code is already written, you can add new features to it with a few alterations. Basically, it allows adding new features, functionalities, and operations to an existing C program.

**4) What tools are necessary for the development?**

The necessary tools for developing the project are not a lot. They are:

- Compiler system: The compiler system, selected for the project is MinGW (Minimalist GNU for Windows). It is a native Windows port of the GNU Compiler Collection (GCC). It contains the C compiler for windows, plus all the other necessary tools for the successful compilation, linking and execution of the project.

- Version Control System: The version control system is Git (Global Information Tracker). Git is used for source code management in the process of software development. It is a distributed system, that is stored globally. Every developer then gets their local repository with full commit history. It is also useful for tracking changes in the development, making it possible to revert to a previous version, if a problem occurred. Also, in case of a hardware error, like burnt hard disk, the project can be stored in a code hosting platform, with the help of Git, like GitHub, without the possibility of it being lost.

- Build system: Makefile is a set of commands (similar to terminal commands) with variable names and targets to create object file and to remove them. In a single Makefile multiple targets can be compiled to create executable file. The projec can be compiled any number of times by using Makefile. The most common use of Makefiles is to manage the dependencies of the source files of the programs during the compilation and linking (build) phase, that is, to compile only the files that need to be compiled by looking at the dependencies on each other and the last modified dates of the source files while the programs are being compiled.

## 3.4 Folder structure

There are two main folders for the project. The computer part and the ESP32-C6 part. The computer part can be found in blockchain/ and the ESP32-C6 – in esp32-c6/ folder. There is also a .gitignore file, which says which files should not be tracked from the Git version control system.

The folder structure of the project is the following:

blockchain/

|

|- src/

| |-include/

| | |-blockchain.h

| | |-connect.h

| | |-fileIO.h

| | |-util.h

| |-blockchain.c

| |-connect.c

| |-fileIO.c

| |-main.c

| - Makefile

esp32-c6/

|

| - main/

| | - CmakeLists.txt

| | - udp\_client.c

| | - Kconfig.projbuild

| - CmakeLists.txt

.gitignore

The blockahin/ folder is the main folder for the computer part of the project. It contains the Makefile, the sources folder and the sources folder contains the header files folder. The executable “run.exe” can be found here after the Makefile has be run.

The src/ folder contains the source code “.c” files, where the C program is written. They contain the instructions and logic that dictate how a program operates. It also contains the “main.c” file, which contains the “main()” function. The main function serves as the entry point of the program’s execution.

The include/ folder contains the header files. The header files’ main use is to allow functions and data usage, defined in one “.c” file in another “.c” file. These header files contain function prototypes and declarations that allow other “.c” files to use the functions, without needing to see the implementation details. This separation of interface and implementation helps with code organization and encapsulation.

The esp32-c6/ folder contains one “.c” file called “udp\_client.c”. The file contains the “main()” function, which describes the entry point of the microcontroller’s program execution.

CMakeLists.txt files are configuration files used with Cmake, a popular build system and build configuration tool. Cmake is designed to automate the build process for software projects written in C, C++ and other programming languages. It is the substitution of the Makefile in the computer’s part.

The esp32-c6/ project has a custom set of instructions for compilation and execution, that need to be followed.

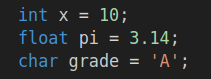
The Kconfig.projbuild file contains the desired settings for the esp32-c6 project.

## **3.5 Short introduction to the C Programming language syntax**

C is a powerful and widely used programming language known for its simplicity and efficiency. Before the code explanations from the project has started, a couple of core concepts should be explained. C code is a set of instructions that are given to the computer to perform various tasks.

**3.5.1 Statements, Variables and Data Types**

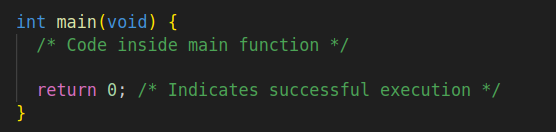
C programs are made up of statements. Each statement typically ends with a semicolon “;”. Variables are like containers that store data. They are described by a “type”, “name” and “value”. Int’s contain whole numbers, float’s contain decimal numbers and char’s contain characters.



uint32\_t and other unsigned numbers cannot be negative.

**3.5.2 Functions and Comments**

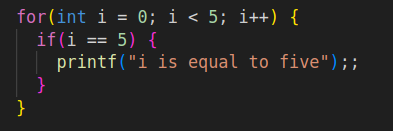
Functions are blocks of code that perform specific tasks. The “main()” function is a special function where program execution begins. Comments are ignored by the computer and are there to help the developers understand the code. They provide explanations or notes.



**3.5.3**

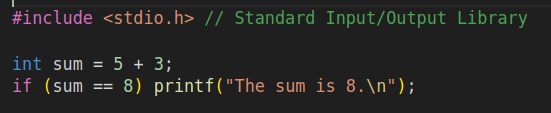
**Control Structures and Output**

Control structures control the flow of the program. Common ones include “if” statements for conditional execution and loops like “for” and “while” for repetition. The printf(“text”); function is used to display information on the screen.



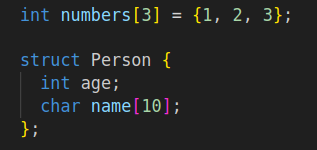
**3.5.4 Libraries and Operators**

Libraries (header files) can be included from outside the file using the #include <libname.h> statement. Operators are symbols used for calculations and comparisons. For example, “+” for addition and “==” for equality.

****

**3.5.5 Arrays and Structures**

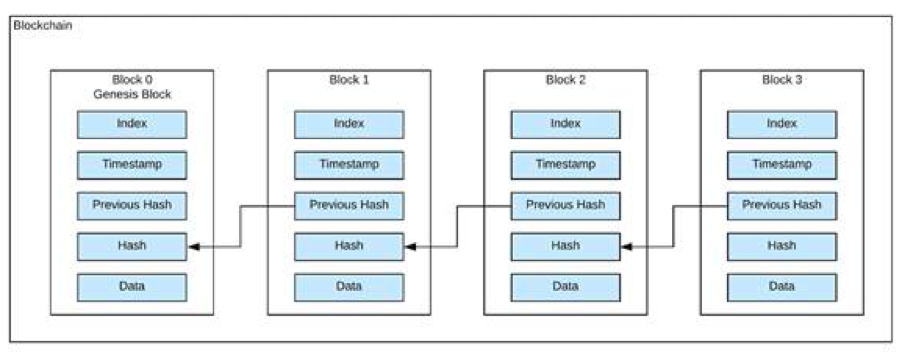
Arrays are collections of data of the sane type, stored in contiguous memory locations. Structure is a composite data type, that allows grouping of multiple and different variables into a single unit

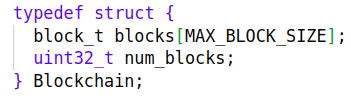
****

## **3.6 Structures overview – computer side**

**3.6.1 Blockchain structure**

As already explained, the Blockchain contains blocks, which are connected by chains. The blockchain structure, inside the project is described in the following way.

Each block contains an index, timestamp, current hash, previous has and data. The blockchain is the reason why everything is secure and transactions can be verified.

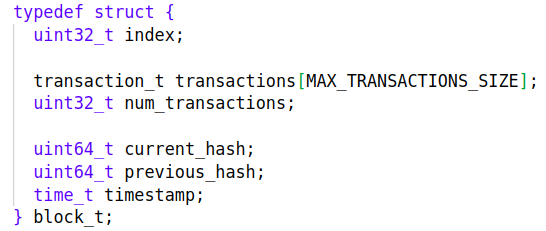


- block\_t blocks[MAX\_BLOCK\_SIZE] – Statically allocated array of blocks with the size “MAX\_BLOCK\_SIZE”, which is a preprocessor directive, used for the creation of symbolic constants. Type is block\_t, which is another structure.



- uint32\_t num\_blocks – The current number of blocks in the blockchain. Type unsigned integer 32 bits of space.

**3.6.2 Block structure**



The block structure, called block\_t contains all the necessary variables for describing the block.

- uint32\_t index – The index of the block, which block it is in the blockchain system. Type unsigned integer 32 bits of space.

- transaction\_t transactions – Statically alllocated array of transactions (in the case of the project data) of the type struct transaction\_t, which have been made during the block generation. The size of the array is:



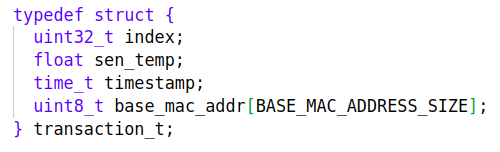
- uint32\_t num\_transaction – the number of transactions in the current block. Type unsigned integer 32 bits of space.

- uint64\_t current\_hash – The hash of the current block. Cryptographic hashes are based on the transactions of the block. They make altering any block in the chain extremely difficult as it would require changing the data in the block and generating new hash, based on the mathematical algorithm. Type unsigned integer 64 bits of space.

- uint64\_t previous\_hash – The hash of the previous block. This field is the pillar of the so called “chain”. It lets the block be arranged consecutively. Altering one block also means that the next block should also be altered, which once again increases security. Type unsigned integer 64 bits of space.

- time\_t timestamp – The timestamp of the block generation. Type time\_t, signed integer 64 bits of space.

**3.6.3 Transactions structure**



- uint32\_t index – The index of the current transaction inside the block array. Type unsigned integer 32 bits of space.

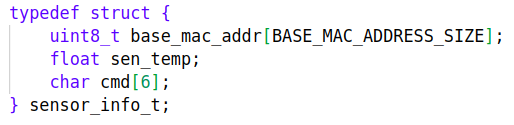
- float sen\_temp – The main data inside the transaction. Storing securely the sensor temperature is the main goal of the project. Type is floating point number, 32 bits of space.

- time\_t timestamp – The timestamp when the transaction was completed. Type time\_t, signed integer 64 bits of space.

- uint8\_t base\_mac\_addr[BASE\_MAC\_ADDRESS\_SIZE] – An array, describing the base mac address of the sensor. In the case of the project it is used as an unique identifier. Each device has an unique base mac address. Type of the array is unsigned integer, 6 bits of space. Size is:



**3.6.4 Sensor information structure**



The Sensor information structure is the structure received from the sensor, with the sensor data. It is a helper structure, which gives the transaction structure the necessary information to complete it, like the base mac address and the sensor temperature.

- uint8\_t base\_mac\_addr[BASE\_MAC\_ADDRESS\_SIZE] – An array, describing the base mac address of the sensor. Type of the array is unsigned integer, 6 bits of space. Size is:

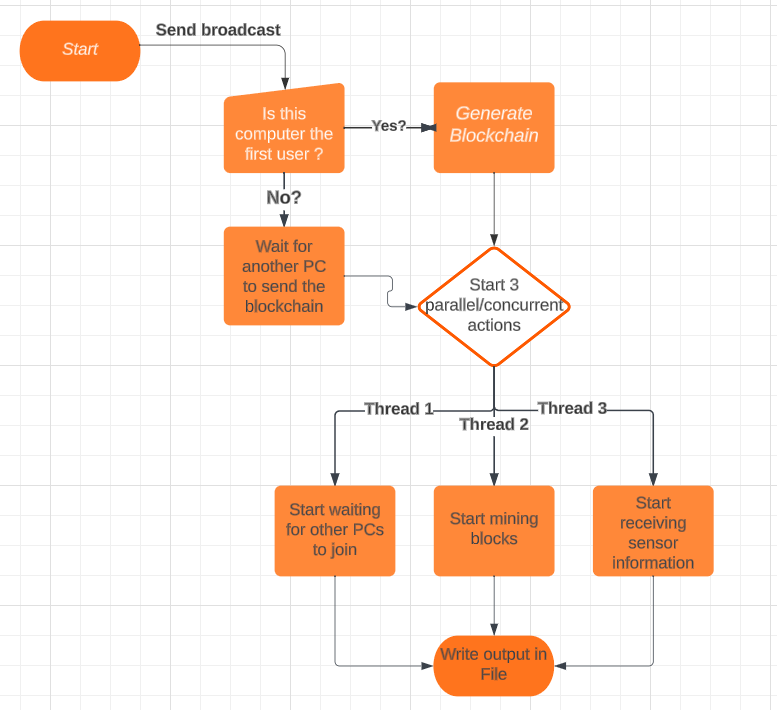


- float sen\_temp – Temperature, sent by the sensor. Type is floating point number, 32 bits of space.

- char cmd[6] – A char array with size the size ‘6’. It describes a command, sent by the sensor. It is used for the verification, that a legitimate device has sent a legitimate data. If the command is not what is expected, the transaction won’t be accepted.

## 3.7 Code overview – computer side

The following UML Diagram represents the overview and sequence of the processes, executed by the program.

****

**3.7.1 Makefile**

For the successful compilation and libraries linkage, the following Makefile has been created.

exec = run.exe

sources = $(wildcard src/\*.c)

objects = $(sources:.c=.o)

flags = -g -Wall -lm -fPIC -lwsock32 -lWs2\_32 -lkernel32

$(exec): $(objects)

gcc $(objects) $(flags) -o $(exec)

%.o: %.c include/%.h

gcc -c $(flags) $< -o $@

clean:

-rm \*.out

-rm \*.exe

-rm \*.o

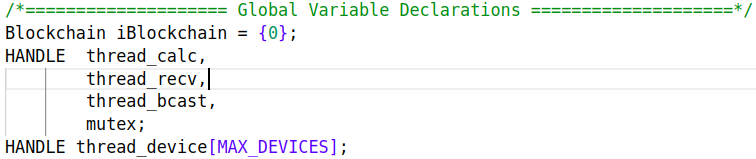
-rm \*.a

-rm src/\*.o

### 3.7.2 Start

Before even the “main()” function has started it’s execution. All the global variables are allocated. Global variables are declared and define outside of any function or code block. The scope of the variable extends throughout the entire program. They are visible and accessible from any part of the code, inside the “main.c” file.

The global variables are:

****

The blockchain is created.

The three main threads are also created globally, and the threads that collect information from the devices are also created.

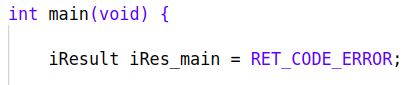
**- Threads:** In the C programming language, threads refer to the concept of multithreading, which allows a program to execute multiple threads of execution concurrently within the same process. Each thread represents an independent sequence of instructions that can run concurrently with the other threads, allowing parallelism and improved performance in certain situations. Threads share the same memory space of the process, which means they can easily share data and resources.

The usage of threads introduces challenges related to synchronization, data sharing and potential race conditions. That’s why there is also a mutex introduced to the program.

**- Mutex:** In C, a mutex (short for “mutual exclusion”) is a synchronization primitive used to protect shared resources in a multithreaded program. A mutex ensures that only one thread can access a shared resources at a time, preventing data races and ensuring proper synchronization between threads.

By locking a mutex before entering the critical section and unlocking it after leaving, it can be ensured that only one thread can execute that section of code. Other threads attempting to access the same section will have to wait until the mutex is released.

Now that everything has been allocated, the “main()” function begins it’s execution.

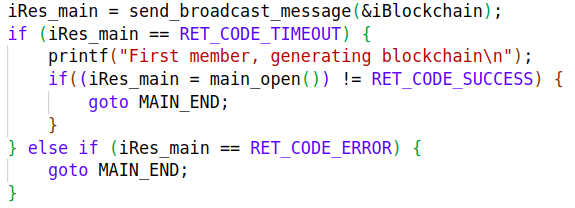


The main function returns type int, and takes nothing as an argument. When a function doesn’t accept a function argument, void should be written. The first thing done in the function is the initialization of the “return” variable, called iRes\_main. It is of type iResult which is:

typedef uint32\_t iResult;

The return value is useful for checking the successful execution of every single function. If a function is successful, it will return “RET\_CODE\_SUCCESS” and if a function fails, it will return “RET\_CODE\_ERROR”. By checking the return code of the functions, everything can be tracked.

**3.7.3 Send broadcast**

****

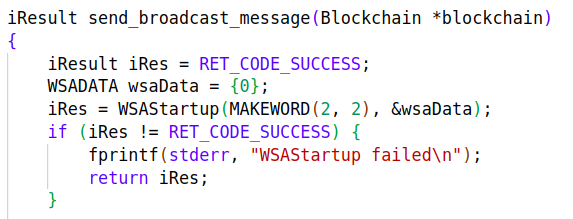
The first function, called from the main, is “send\_broadcast\_message”. It accepts the address of the blockchain as an argument. Why the address? Because it a change of the value of the blockchain is expected. When the address of the variable is passed (a pointer), the function can access and modify the actual data stored at that memory location. If the variable itself is passed, a copy of that variable is created inside the function, and any changes done will not persist outside the scope of the function.

The goal of the function is to send a broadcast message, to everyone connected to the network. To achieve this goal, a socket must be created, and data should be sent through the socket. Sockets allow communication and data exchange between computers and devices over a network. Sockets are operational system dependent, so the sockets in Linux and in Windows differ.

The libraries in Windows OS, that support socket creation are :

#include <winsock2.h>

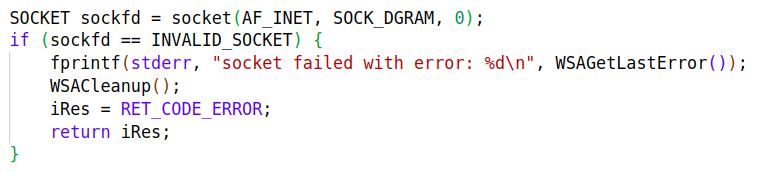
#include <windows.h>

**3.7.3.1 WSAStartup()**

To create a socket inside Windows OS, the WSAStartup() function must be the first Windows Sockets function called by an application or DLL. It allows an application or DLL to specify the version of Windows Sockets required and retrieve details of the specific Windows Sockets implementation. The application or DLL can only issue further Windows Sockets functions after successfully calling WSAStartup().

After WSAStartup() has been succesfully executed, a socket can be created.

**3.7.3.2 Socket()**

****

The socket creation function accepts the parameters:

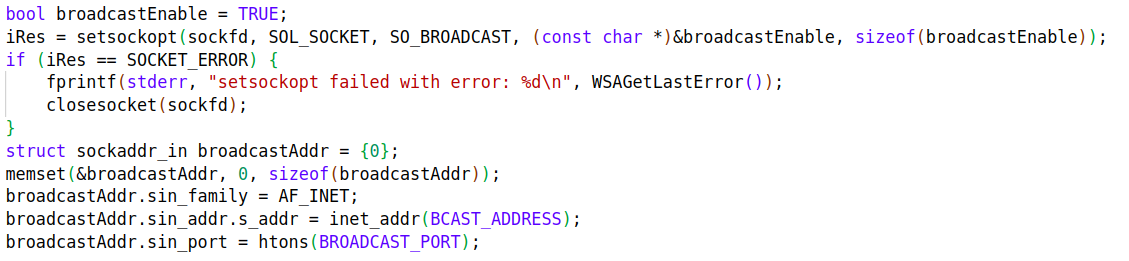
- address\_family – In the project’s case, it’s “AF\_INET”, meaning the Internet Protocol version 4 (IPv4) address family.

- type – In this case, it is “SOCK\_DGRAM”, which is connectionless buffers of fixed (small amount) of data. This socket type uses the User Datagram Protocol (UDP).

- protocol – When 0 is called, there is no specified protocol.

**3.7.3.3 Setsocketopt()**

After that, the broadcast address and options are set.

****

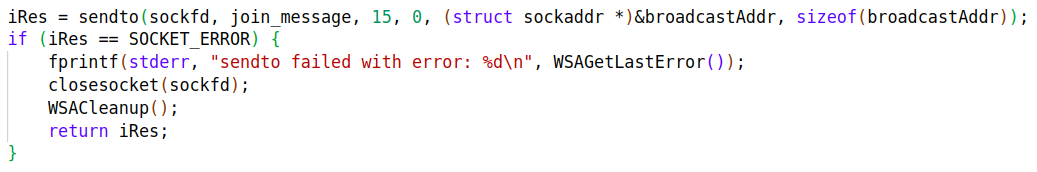
- setsocketopt() - Controls the socket’s behavior.

- After that the broadcastAddr’s address and port are also set. To ease the use of the program, “255.255.255.255” is set as an address, but the normal IPv4 address of the router could also be set. Here it should be noted, that the port should be the same at both sides of the socket communication, otherwise no data would be caught after being sent.

The port for the program is set as “12345”.

**3.7.3.4 Sendto()**

The sendto function is used to write outgoing data on a socket.

****

Parameters:

- Socket - A descriptor identifying a (possibly connected) socket.

- buffer - A pointer to a buffer containing the data to be transmitted. In this case it is a join\_message, specific for the implementation. It is “I want to join”

- buffer length - The length, in bytes, of the data pointed to by the *buffer* parameter.

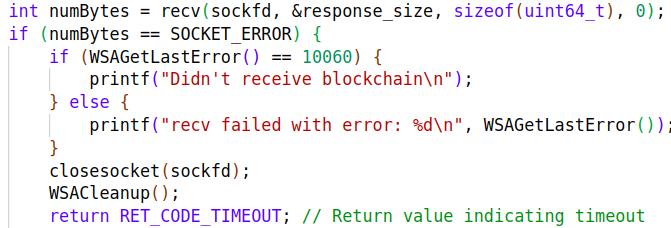
- flags - A set of flags that specify the way in which the call is made. In this case, it is unspecified.

- to - An optional pointer to a sockaddr structure that contains the address of the target socket. That is the previously set broadcastAddr.

- sockaddr length - The size, in bytes, of the address pointed to by the *to* parameter.

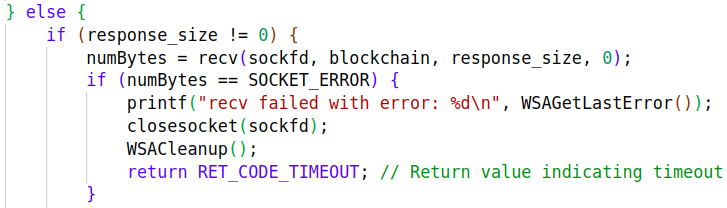
**3.7.3.5 recvfrom()**

After the successful data transfer, the application sets new socket option - timeout. The timeout says how long the program should wait to receive data, before it times out. The recvfrom() function expects to receive the size of the blockchain from the same socket descriptor.

****

If the function fails and times-out, the program decides that no other nodes participate in the blockchain system, because no node sent the blockchain to it.

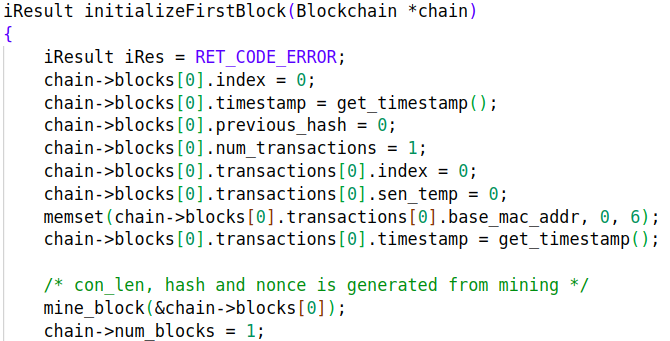
**- second recvfrom() -** If the program, however, receives data through the socket, that means, that another node has already created the blockchain. It waits to receive the whole blockchain.



That is the way the synchronization works between computers. In two words, as it is shown in the UML Diagram, it asks the network for existing participants. If it times-out, that means it is the first participant and starts the whole process. Otherwise it receives the blockchain and joins the process.

**3.7.3.6 InitializeFirstBlock()**

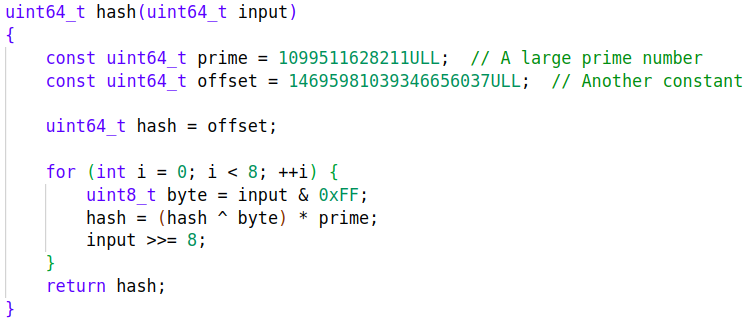
If the current computer is the first node though, it has to be the starter of the process. It has to initialize the first block, called also the genesis block.



The genesis block is the very first block in a blockchain. It serves as the foundation upon which all other blocks are added. InitializeFirstBlock(Blockchain \*blockchain) is a plain function, that sets the values of all fields, besides the timestamp and hash, in the genesis block to zero. It also creates an empty transaction. The hash is created based on the timestamp, because the transactions are empty.

**3.7.3.7 Hash()**

The hash generation algorithm is simple but secure algorithm, that creates a uint64\_t number from given input. It is impossible to receive the same answer with two different values. The algorithm is also a one way cryptographic function as the original data cannot be retrieved via decryption. Hashes are the main source of security for the blockchain.

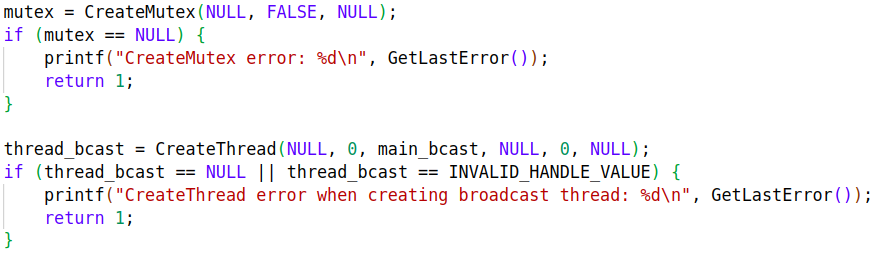


The hash is generated at the end of every block generation, based on the transactions, received during the generation. By a sequence of calculations, the hash is generated. It is close to impossible for a human to be able to restore the initial input. For a machine it wont be impossible, but even the strongest algorithms can be decrypted with a super-computer, given the time.

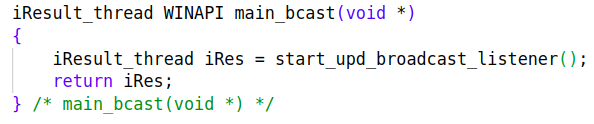
### 3.7.4 Parallelism/Concurrency

Here is the important part of the computer’s side of the project. Three actions should be happening at the same time, throughout the whole program.

**3.7.4.1 Thread\_bcast and mutex**

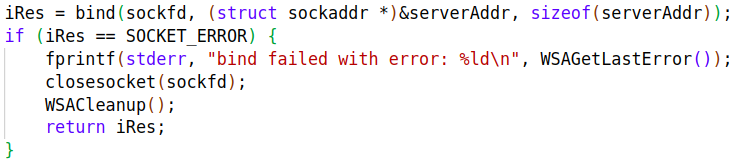


- thread\_bcast – The first thread execution, that is started, is the broadcasting thread. The function that it executes is the main\_bcast() function.

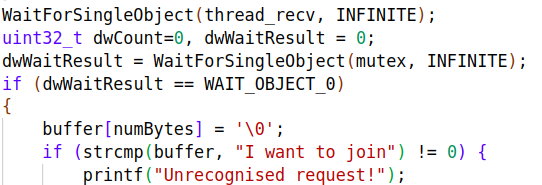


The main\_bcast function takes as an argument void\*, and returns iResult\_thread, which is unsigned long. Inside only one function is executed, the start\_upd\_broadcast\_listener(). The reason why the thread doesn’t call the listener function directly, is to make the program easily extensible.

The start\_upd\_broadcast\_listener() has similarities to the previously called send\_broadcast\_message. This is not a coincidence, because these two functions are the two sides of the socket communication. Both functions open sockets on the same port number. The difference is, that the listener function binds the socket. This step is not necessary, because the OS will automatically do it, but it is clearer for the developer.



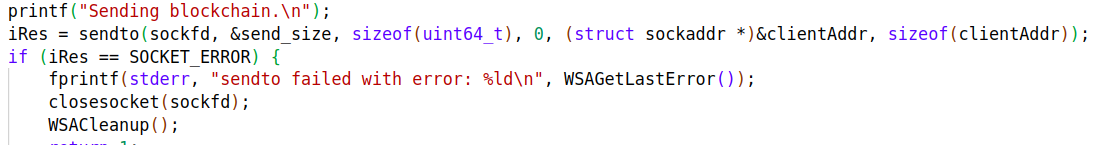
The interesting part starts here. After the function calls recvfrom() and actually receives data, it will lock the mutex.



Why would it lock the mutex? Because the sender function will try to send the whole blockchain. The blockchain is an important structure, that is accessed from everywhere all the time. That’s why it will wait for all the other threads to finish their tasks, like putting transactions and generating the blocks. Then it will lock the mutex, which is also used in the transaction loading part, to be the only thread with access to the blockchain. This ensures, that the whole blockchain will be sent to the requesting node, without missing transactions. If this data is not locked at that moment, there is a possibility, that the split second the blockchain is sent, another transaction is added. This transaction will not be recorded in the other node’s blockchain and the whole security will crumble.

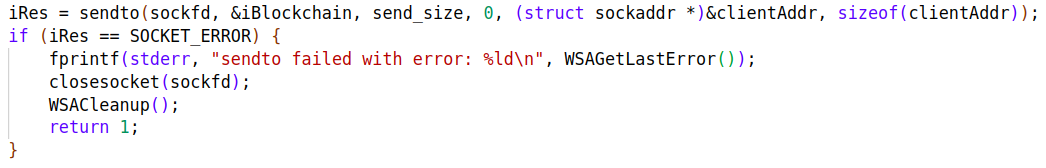
After successfully locking the mutex, the program will check if the correct request was sent to it. If it is not the recognizable command, it will just skip sending anything back.

If it is, however, the correct “I want to join” option, the sending process starts.

****

First, it sends the size of the blockchain. This is done, because the receiver should now how much bytes should be expect.

If the sendto function is successful, the whole blockchain is sent.

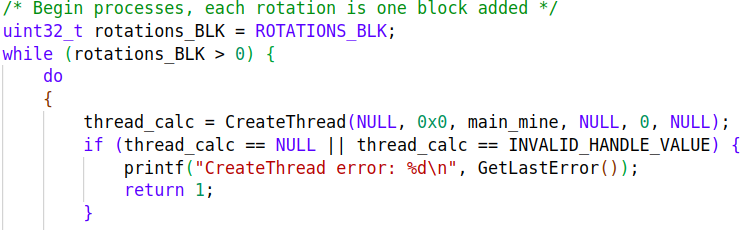


Here it is important, that after successful data transfer, the

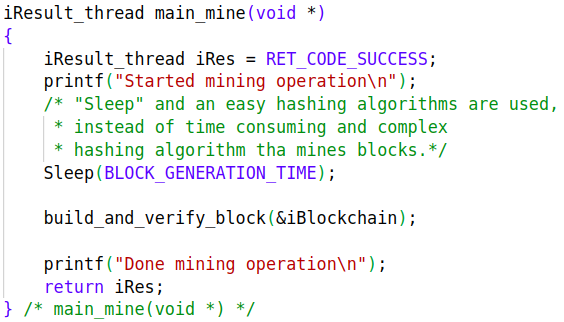
ReleaseMutex(mutex) function should be called, to unlock the mutex, and allow other threads to have access to their desired resources.

**3.7.4.2 thread\_calc**

After the broadcasting thread is started, an loop is started, which rotates until the desired block amount in the blockchain is achieved.

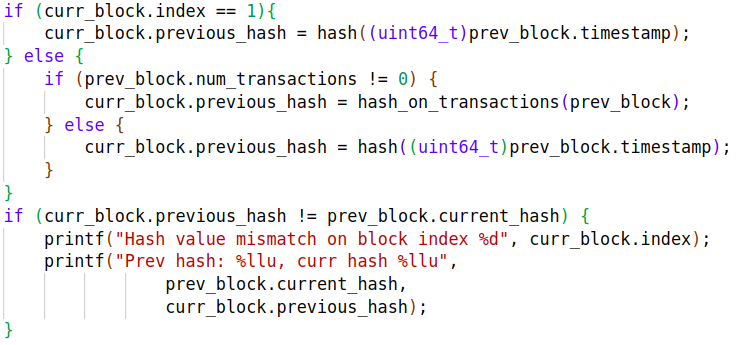


After the iterator has started, the thread\_calc thread is started, with the function main\_mine()



The mining function is the function that generates the block’s parameters, aside from the transactions and the current\_hash field. Here it is a little bit different from the popular blockchain systems, because there is no complex mathematical problem to be resolved, prior to the block generation. Here the block is created, and to simulate more real-life examples, a sleep is incurred. This gives the receiving function to fill transactions for a given amount of time.

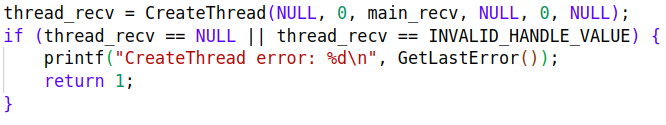
The build\_and\_verify\_block(&iBlockchain) function is still useful. It once again increases the security of the blockchain.



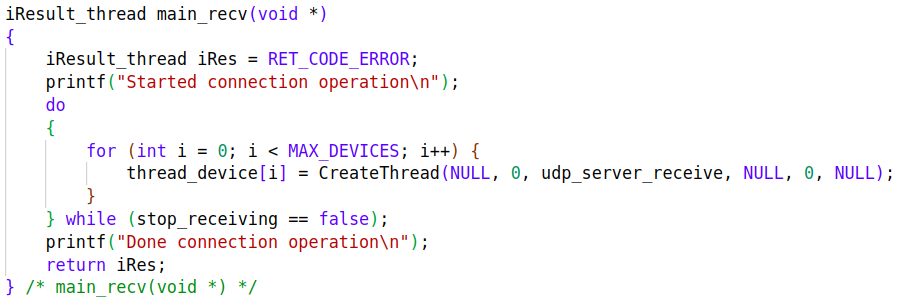
Inside the function, the block that is currently being generated will once again re-generate the hash of the previous block. That way it verifies, that no changes have been made to either the transactions, or the hash, making it tamper-proof.

**3.7.4.3 thread\_recv**

The final of the three main threads. Inside this thread happens the receiving of the of transactions or the sensor data. The thread’s function is the main\_recv() function.



The special parts of this function, is that inside the thread\_recv, another three threads are opened.

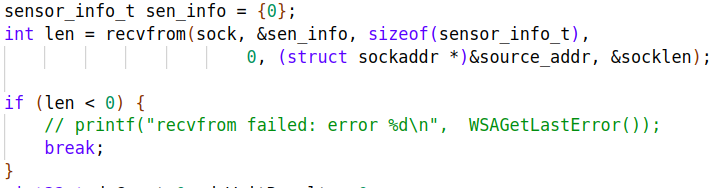
****

The most important reason for that is concurrency. Multiple threads allow the computer to handle multiple device connections simultaneously. That way, the chances of missing a transaction are minimal.

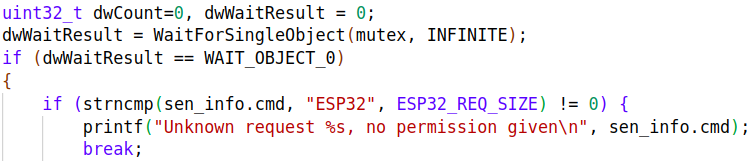
There are also better resources utilization and scalability. Adding a couple of more threads allows for more device connections.

**- second udp\_server\_receive()**

The procedure inside this function is similar to the broadcast listener function, mentioned earlier. It opens and binds a socket to the desired port.

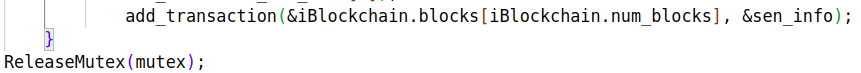


Then it expect to receive data from the socket, which fills the sen\_info structure. It checks if a valid data was received. If data is invalid, repeats the process, until valid data is received.



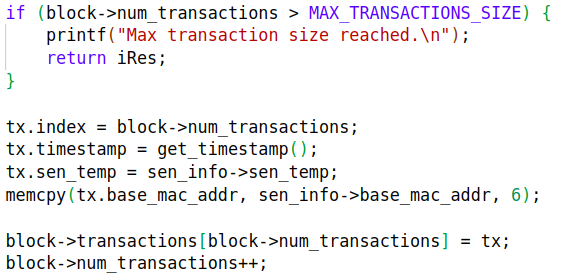
After valid data is received, the mutex will be locked. The same mutex as the one that sends the blockchain is used. This allows it to be locked at only one of the two places at a time, ensuring data completeness.

First the sen\_info.cmd parameter is compared with the expected command. If they do not match, no permission is given to track the transaction.



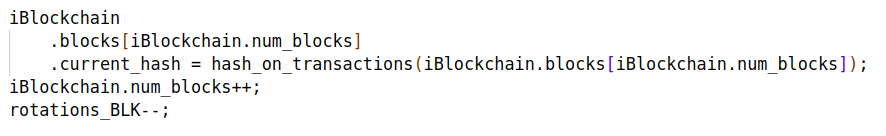
Otherwise, if they match, the transaction is recorded inside the block and the mutex is released.

Inside the add\_transaction() function, two things happen.



If the transaction pool of the block is filled, the transaction will not be recorded. Otherwise, the sen\_info parameters are recorded inside the transaction and everything is complete.

Finally, after the block has been generated, and the transactions have been completed, the block’s hash is generated.



The hash generation is based on the sum of data of all the transactions inside the block. That way all the transactions participate in the hash generation, and modifying even a single one of them will be caught.

## **3.8 Code overview – ESP32-C6 side**

**3.8.1 Introduction and prerequisites**

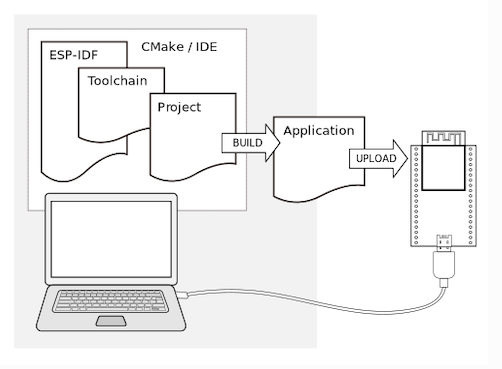
The following software is required for the successful usage of ESP32-C6:

- **Toolchain** to compile code for ESP32-C6

- **Build tools** - CMake and Ninja to build a full Application for ESP32-C6

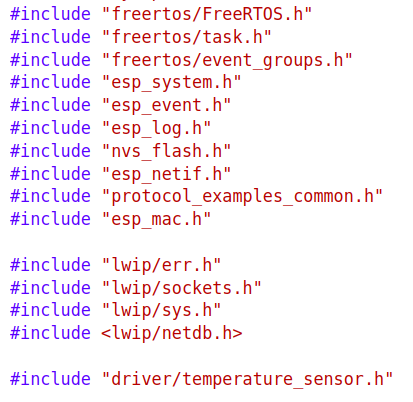
- **ESP-IDF** that essentially contains API (software libraries and source code) for ESP32-C6 and scripts to operate the Toolchain

Python, Git, cross-compilers, CMake and Ninja build tools are the software, which were used for this project.

(image from ESP32-C6 documentation)

The CmakeLists.txt files were taken from the pre-exisitng project examples, installed together with the Espressif IDE. Apart from that, the project “udp\_client.c” was used as a starting point, as it matches the desired program’s goals the most.

There are multiple esp libraries needed for the file.



None of these have been developed for the sake of the project, or by the creator of the project. They have been created by Espressif studio.

Before the program is executed and flashed to the device, the ESP-IDF 5.1 PowerShell should be opened.

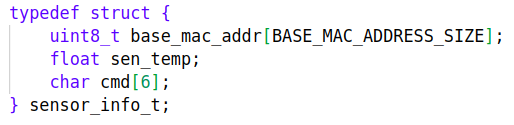
There are three commands that need to be executed for the program to work:

- idf.py set-target esp32c6 – This command sets the target device as esp32-c6.

- idf.py menuconfig – This command opens a menu, from which the user should select the desired port and ip address configurations, to which the data will be sent. Afterwards a config file is created, from which the program reads.

- idf.py -p (PORT) flash monitor – This command is actually a couple of commands. The first part is “idf.py -p (PORT) flash”, where “PORT” is the port of the computer, to which the device is connected. For exemplary purposes, this command was previously executed with “COM4” as the port. Then the “flash” command not only flashes the device, but also builds the project. “Monitor” will show the output of the program on the PowerShell.

**3.8.2 Structures overview – ESP32-C6 part**



The sensor information structure contains all the necessary data, needed to fulfill the transactions.

- uint8\_t base\_mac\_addr[BASE\_MAC\_ADDRESS\_SIZE] – An array, describing the base mac address of the sensor. Type of the array is unsigned integer, 6 bits of space. Size is:



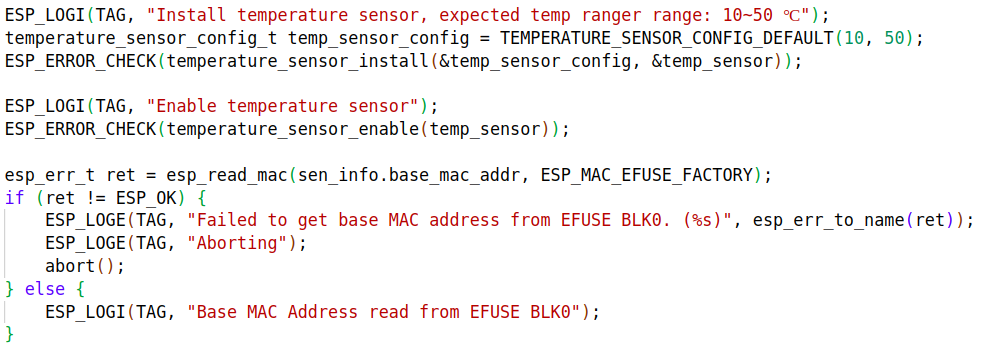
- float sen\_temp – Temperature, sent by the sensor. Type is floating point number, 32 bits of space.

- char cmd[6] – A char array with size the size ‘6’. It describes the command, sent by the device. It is used for the verification, that a legitimate device has sent a legitimate data. The command sent is “ESP32-C6”

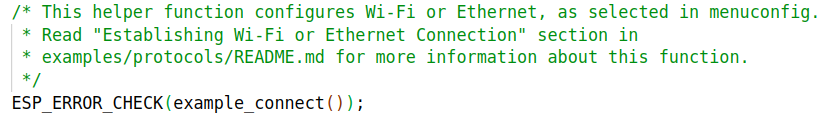
**3.8.3 Code overview – ESP32-C6 part**

The program execution starts the same way it starts on the computer side, from the main function. In this case, it is called “app\_main()”.

It takes no arguments and returns no arguments.

First, the necessary temperature sensor settings are enabled and the base mac address is retrieved.

Afterwards a connection with Wi-Fi is established.

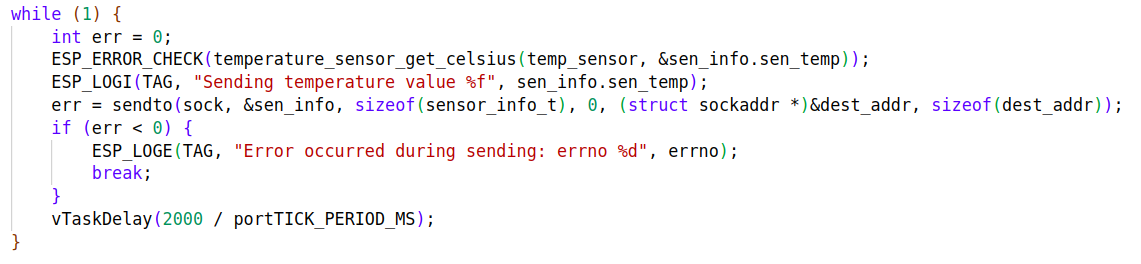


Finally, a freeRTOS task is created :

xTaskCreate(udp\_client\_task, "udp\_client", 4096, NULL, 5, NULL);

The task executes the function, specified as an argument. In this case, it is “udp\_client\_task()”. By the already familiar procedure, a socket is generated. The parameters for the address and port are taken from the config file, generated by the “idf.py menuconfig” command.

Then an endless loop starts.

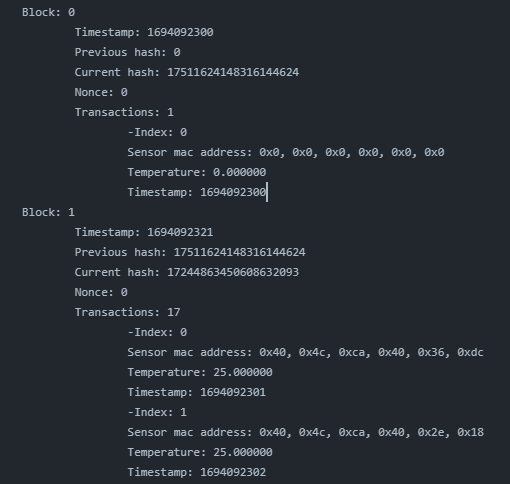


Every time the loop starts, the temperature is measured, which keeps the data actual and valid. It is then placed inside the sen\_info structure, which already has set “base\_mac\_address” and “cmd” parameters. Then the structure is sent through the socket to everyone in the network. Finally a delay is set, making the data transfer happen every two seconds.

# 4. Experimental studies.

The experiment consist in simulating a real environment, which tests the validity of the project - running multiple devices, with multiple nodes/computers. Two ESP32-C6 devices started sending data to the network. Two computers started program execution, one after the other.

The first computer successfully collected data from both devices. Afterwards, the second node joined the system. It received successfully the whole blockchain, with zero invalid information. Both of them then started receiving information from both sensors.



The whole code execution will be described tersely. Shown above is the output file, created from the successful completion of the program on a computer.

First of all, a node tried joining the blockchain network. It didn’t join, so it created the genesis block itself. It created a hash for it, based on the timestamp, as no valid transactions exist.

Afterwards the generation of the second block started. It verified the hash of the previous block, and then inherited it, becoming the second block of the blockchain.

While the block was being generated, multiple transactions from ESP32-C6 devices arrived. It correctly recorded the data from both devices. Proof of that is, that the unique identifiers – sensor base mac address, differ.

After the block generation has been completed, a hash was generated, based on the temperatures received from the sensors, as it is the important data.

Then a new node joined and it received the whole blockchain. Then both nodes continued receiving data from the sensors.

Finally, both nodes produced a .txt file, which contained the whole blockchain. It could be easily shared and the data could be easily analyzed with the desired purposes in mind. It was verified, that the blockchains matched.

## 4.1 Achieving the goals

After everything was verified and worked as expected, the experimental studies were put to an end with a successful result. The goals, created at the beginning of the project were all ticked out.

1) Development of decentralized blockchain system that runs on multiple computers, collecting sensory information – Verified from the experimental studies.

2) Connecting and sending data from the sensors to the computers – Verified from the experimental studies

3) Implementing a robust security measures to protect sensory data from unauthorized access and tampering – Verified from the experimental studies

4) Designing an intuitive way for the collected information to be used – Verified from the experimental studies

- Security: The stored data is tamper-proof, ensuring integrity

Security is the main goal of the project. Multiple methods were created, and afterwards tested, to ensure security. These methods include the hash generation of every block, based on the data collected, making changing both the data and the hash close to impossible. Then the blocks are linked to each other, preventing the removal of blocks, which keeps everything sequential.

Network security is another method, used to increase the security. Unauthorized devices and nodes were not allowed to participate in the blockchain system.

- Decentralization: A distributed network of nodes (computers), reducing central points of failure.

Decentralization was successfully incorporated inside the program execution. Each nodes has the same blockchain and even one computer somehow meets hardware issues, the other nodes will still be keeping the valid blockchain copy locally.

- Privacy: Users have control over the data, sharing it securely when they choose.

Privacy and control was achieved by writing all the output (the blockchain structure) inside a file, which can then be securely handled however the user wants. No information is sent anywhere to the outside world.

- Interoperability: Different sensory data metrics from various sources and devices can be integrated seamlessly.

The sensor temperature is not the only data that can be sent. Additional metrics can easily be included in the process by modifying a couple of values inside both the computers and ESP32-C6 code and by integrating additional peripheral sensors and sending their output data over.

# 5. Conclusion

In conclusion, the blockchain-based sensory information collection project has been incredibly successful in achieving the predefined goals. All the points – Efficient Data Collection, Data Integrity and Security, Transparency and Trust, Scalability and Interoperability and Efficiency have been achieved one way or another. The journey has been one of innovation.

Here it should be noted, that while the blockchain technology provides strong security features, it is not immune to all attacks. Like everything in the software engineering world, multiple functionalities can be improved in many ways. One example is the hashing algorithm for the block hash generation. It’s breakability can be reduced by increasing the difficulty of the mathematical principles, standing behind the logic of the algorithm. Other algorithms and functions could most likely also be improved, but for the experimental studies, the low recourse and processing power options were chosen.

Security is an ongoing concern around the world, and the whole blockchain community continually works to improve and adapt security measures to address new vulnerabilities and threats as they arise.

# 6. Sources:

The source code is available on: https://github.com/Talaxika/blockchain

[1] What is a blockchain, <https://www.ibm.com/topics/blockchain>, 10.09.2023

[2] Networking, <https://www.geeksforgeeks.org/basics-computer-networking/>, 10.09.2023

[3] Espressif/ESP32-C6 programming guides and documentation, https://docs.espressif.com/projects/esp-idf/en/latest/esp32c6/get-started/index.html, 10.09.2023

[4] Windows sockets, <https://learn.microsoft.com/en-us/windows/win32/api/winsock/>, 03.09.2023

[5] Windows threads, <https://learn.microsoft.com/en-us/windows/win32/procthread/processes-and-threads>, 03.09.2023

[5] Hashing algorithms, <https://www.okta.com/identity-101/hashing-algorithms/>, 10.09.2023

# 7. **Source** code

The whole code will be shown, file by file, starting from the header files.

**blockchain.h**

#ifndef BLOCKCHAIN\_H

#define BLOCKCHAIN\_H

#include "util.h"

#include "connect.h"

#define MAX\_BLOCK\_SIZE (8U)

#define MAX\_TRANSACTIONS\_SIZE (16U)

#define BLOCK\_GENERATION\_TIME (5000U)

typedef struct {

uint32\_t index;

float sen\_temp;

time\_t timestamp;

uint8\_t base\_mac\_addr[BASE\_MAC\_ADDRESS\_SIZE];

} transaction\_t;

typedef struct {

uint32\_t index;

transaction\_t transactions[MAX\_TRANSACTIONS\_SIZE];

uint32\_t num\_transactions;

uint64\_t current\_hash;

uint64\_t previous\_hash;

time\_t timestamp;

} block\_t;

typedef struct {

block\_t blocks[MAX\_BLOCK\_SIZE];

uint32\_t num\_blocks;

} Blockchain;

time\_t get\_timestamp();

iResult initializeFirstBlock(Blockchain \*chain);

iResult build\_and\_verify\_block(Blockchain \*chain);

iResult mine\_block(block\_t \*block);

uint64\_t hash\_on\_transactions(block\_t block);

iResult add\_transaction(block\_t \*block, sensor\_info\_t \*sen\_info);

void print\_blockchain(Blockchain chain);

/\* Function is defined here, and not in connect.h,

\* because access to the Blockchain structure is needed\*/

iResult send\_broadcast\_message(Blockchain \*blockchain);

#endif /\* BLOCKCHAIN\_H \*/

**connect.h**

#ifndef CONNECT\_H

#define CONNECT\_H

#include "util.h"

#define DEFAULT\_PORT 12233

#define BROADCAST\_PORT 12345

#define MAX\_TIMEOUT\_THREAD (5U)

#define BCAST\_ADDRESS "255.255.255.255"

#define BROADCAST\_MESSAGE "Hello"

#define ESP32\_REQ\_SIZE (6u)

iResult connect\_open(void);

iResult connect\_close(void);

#endif /\* CONNECT\_H \*/

**fileIO.h**

#ifndef FILEIO\_H

#define FILEIO\_H

#include "util.h"

#include "blockchain.h"

iResult write\_to\_file(Blockchain blockchain);

#endif /\* FILEIO\_H \*/

**util.h**

#ifndef UTIL\_H

#define UTIL\_H

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <time.h>

#include <stdbool.h>

#include <stdint.h>

#include <winsock2.h>

#include <windows.h>

#include <ws2tcpip.h>

#define RET\_CODE\_ERROR (1U)

#define RET\_CODE\_SUCCESS (0U)

#define RET\_CODE\_TIMEOUT (200U)

#define BASE\_MAC\_ADDRESS\_SIZE (6U)

typedef uint32\_t iResult;

typedef long unsigned int iResult\_thread;

typedef struct {

uint8\_t base\_mac\_addr[BASE\_MAC\_ADDRESS\_SIZE];

float sen\_temp;

char cmd[6];

} sensor\_info\_t;

#endif /\* UTIL\_H \*/

**blockchain.c**

#include "include/connect.h"

#include "include/blockchain.h"

time\_t get\_timestamp()

{

time\_t t;

time(&t);

return t;

}

uint64\_t hash(uint64\_t input)

{

const uint64\_t prime = 1099511628211ULL; // A large prime number

const uint64\_t offset = 14695981039346656037ULL; // Another constant

uint64\_t hash = offset;

for (int i = 0; i < 8; ++i) {

uint8\_t byte = input & 0xFF;

hash = (hash ^ byte) \* prime;

input >>= 8;

}

return hash;

}

uint64\_t hash\_on\_transactions(block\_t block)

{

uint64\_t sum = 0;

for (uint64\_t i = 0; i < block.num\_transactions; i++)

{

sum+=(uint64\_t)block.transactions[i].sen\_temp;

}

return hash(sum);

}

iResult build\_and\_verify\_block(Blockchain \*blockchain)

{

block\_t curr\_block = blockchain->blocks[blockchain->num\_blocks];

block\_t prev\_block = blockchain->blocks[blockchain->num\_blocks - 1];

curr\_block.index = blockchain->num\_blocks;

curr\_block.timestamp = get\_timestamp();

if (curr\_block.index == 1){

curr\_block.previous\_hash = hash((uint64\_t)prev\_block.timestamp);

} else {

if (prev\_block.num\_transactions != 0) {

curr\_block.previous\_hash = hash\_on\_transactions(prev\_block);

} else {

curr\_block.previous\_hash = hash((uint64\_t)prev\_block.timestamp);

}

}

if (curr\_block.previous\_hash != prev\_block.current\_hash) {

printf("Hash value mismatch on block index %d", curr\_block.index);

printf("Prev hash: %llu, curr hash %llu",

prev\_block.current\_hash,

curr\_block.previous\_hash);

}

blockchain->blocks[blockchain->num\_blocks] = curr\_block;

return RET\_CODE\_SUCCESS;

}

iResult mine\_block(block\_t \*block)

{

iResult iRes = RET\_CODE\_ERROR;

block->current\_hash = hash((uint64\_t)block->timestamp);

iRes = RET\_CODE\_ERROR;

return iRes;

}

iResult add\_transaction(block\_t \*block, sensor\_info\_t \*sen\_info)

{

iResult iRes = RET\_CODE\_ERROR;

transaction\_t tx = {0};

if (block->num\_transactions > MAX\_TRANSACTIONS\_SIZE) {

printf("Max transaction size reached.\n");

return iRes;

}

tx.index = block->num\_transactions;

tx.timestamp = get\_timestamp();

tx.sen\_temp = sen\_info->sen\_temp;

memcpy(tx.base\_mac\_addr, sen\_info->base\_mac\_addr, 6);

block->transactions[block->num\_transactions] = tx;

block->num\_transactions++;

iRes = RET\_CODE\_SUCCESS;

return iRes;

}

iResult initializeFirstBlock(Blockchain \*chain)

{

iResult iRes = RET\_CODE\_ERROR;

chain->blocks[0].index = 0;

chain->blocks[0].timestamp = get\_timestamp();

chain->blocks[0].previous\_hash = 0;

chain->blocks[0].num\_transactions = 1;

chain->blocks[0].transactions[0].index = 0;

chain->blocks[0].transactions[0].sen\_temp = 0;

memset(chain->blocks[0].transactions[0].base\_mac\_addr, 0, 6);

chain->blocks[0].transactions[0].timestamp = get\_timestamp();

mine\_block(&chain->blocks[0]);

chain->num\_blocks = 1;

iRes = RET\_CODE\_SUCCESS;

return iRes;

}

void print\_block(block\_t block) {

printf("Block %d:\n", block.index);

printf(" Timestamp: %llu\n", block.timestamp);

printf(" Previous hash: %llu", block.previous\_hash);

printf(" Current hash: %llu", block.current\_hash);

printf(" Nonce: %llu\n", block.nonce);

printf(" Transactions: %d\n", block.num\_transactions);

for (int i = 0; i < block.num\_transactions; i++)

{

printf(" -Index: %d\n", block.transactions[i].index);

printf(" Temperature: %f\n", block.transactions[i].sen\_temp);

printf(" Timestamp: %llu\n", block.transactions[i].timestamp);

printf(" Sensor mac address: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",

block.transactions[i].base\_mac\_addr[0],

block.transactions[i].base\_mac\_addr[1],

block.transactions[i].base\_mac\_addr[2],

block.transactions[i].base\_mac\_addr[3],

block.transactions[i].base\_mac\_addr[4],

block.transactions[i].base\_mac\_addr[5]);

}

}

void print\_blockchain(Blockchain chain) {

int i = 0;

for (i = 0; i < chain.num\_blocks; i++) {

print\_block(chain.blocks[i]);

}

}

**connect.c**

#undef UNICODE

#define WIN32\_LEAN\_AND\_MEAN

#include "include/blockchain.h"

#include "include/connect.h"

static const char \*join\_message = "I want to join";

iResult connect\_open(void)

{

iResult iRes = RET\_CODE\_ERROR;

WSADATA wsaData = {0};

// Initialize Winsock

iRes = WSAStartup(MAKEWORD(2,2), &wsaData);

if (iRes != 0) {

printf("WSAStartup failed with error: %d\n", iRes);

return iRes;

}

return iRes;

}

iResult connect\_close(void)

{

iResult iRes = RET\_CODE\_ERROR;

/\* Terminates Windows Sockets operations for all threads \*/

iRes = WSACleanup();

if (iRes == SOCKET\_ERROR) {

printf("WSACleanup failed with error: %d\n", WSAGetLastError());

}

return iRes;

}

iResult send\_broadcast\_message(Blockchain \*blockchain)

{

iResult iRes = RET\_CODE\_SUCCESS;

WSADATA wsaData = {0};

iRes = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iRes != RET\_CODE\_SUCCESS) {

fprintf(stderr, "WSAStartup failed\n");

return iRes;

}

SOCKET sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sockfd == INVALID\_SOCKET) {

fprintf(stderr, "socket failed with error: %d\n", WSAGetLastError());

WSACleanup();

iRes = RET\_CODE\_ERROR;

return iRes;

}

bool broadcastEnable = TRUE;

iRes = setsockopt(sockfd, SOL\_SOCKET, SO\_BROADCAST, (const char \*)&broadcastEnable, sizeof(broadcastEnable));

if (iRes == SOCKET\_ERROR) {

fprintf(stderr, "setsockopt failed with error: %d\n", WSAGetLastError());

closesocket(sockfd);

}

struct sockaddr\_in broadcastAddr = {0};

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

broadcastAddr.sin\_family = AF\_INET;

broadcastAddr.sin\_addr.s\_addr = inet\_addr(BCAST\_ADDRESS);

broadcastAddr.sin\_port = htons(BROADCAST\_PORT);

iRes = sendto(sockfd, join\_message, 15, 0, (struct sockaddr \*)&broadcastAddr, sizeof(broadcastAddr));

if (iRes == SOCKET\_ERROR) {

fprintf(stderr, "sendto failed with error: %d\n", WSAGetLastError());

closesocket(sockfd);

WSACleanup();

return iRes;

}

struct sockaddr\_in responseAddr = {0};

// Set a timeout for receiving responses

uint32\_t timeout = 4000;

iRes = setsockopt(sockfd, SOL\_SOCKET, SO\_RCVTIMEO, (const char \*)&timeout, sizeof(timeout));

if (iRes < 0) {

perror("setsockopt timeout");

WSACleanup();

return iRes;

}

char responseBuffer[sizeof(Blockchain)];

uint64\_t response\_size = 0;

int numBytes = recv(sockfd, &response\_size, sizeof(uint64\_t), 0);

if (numBytes == SOCKET\_ERROR) {

if (WSAGetLastError() == 10060) {

printf("Didn't receive blockchain\n");

} else {

printf("recv failed with error: %d\n", WSAGetLastError());

}

closesocket(sockfd);

WSACleanup();

return RET\_CODE\_TIMEOUT; // Return value indicating timeout

} else {

if (response\_size != 0) {

numBytes = recv(sockfd, blockchain, response\_size, 0);

if (numBytes == SOCKET\_ERROR) {

printf("recv failed with error: %d\n", WSAGetLastError());

closesocket(sockfd);

WSACleanup();

return RET\_CODE\_TIMEOUT; // Return value indicating timeout

}

print\_blockchain(\*blockchain);

printf("Received response from %s:%d: %s\n", inet\_ntoa(responseAddr.sin\_addr), ntohs(responseAddr.sin\_port), responseBuffer);

}

}

closesocket(sockfd);

WSACleanup();

return iRes;

}

**fileIO.c**

#include "include/fileIO.h"

iResult write\_to\_file(Blockchain blockchain)

{

// Open the file for writing

HANDLE hFile = CreateFile(

(LPCSTR)"Blockchain.txt", // File name

GENERIC\_WRITE, // Access mode (write)

0, // No sharing

NULL, // Security attributes

CREATE\_ALWAYS, // Create if not exists, overwrite if exists

FILE\_ATTRIBUTE\_NORMAL, // File attributes

NULL // Template file

);

if (hFile == INVALID\_HANDLE\_VALUE) {

perror("Error opening file");

return 1;

}

// Convert the structure fields to bytes and write to the file

DWORD bytesWritten = 0;

char buffer[256] = {0};

for (uint32\_t num\_blk = 0; num\_blk < blockchain.num\_blocks; num\_blk++)

{

block\_t block = blockchain.blocks[num\_blk];

snprintf(buffer, sizeof(buffer), "Block: %d\r\n", block.index);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\tTimestamp: %llu\r\n", block.timestamp);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\tPrevious hash: %llu\r\n", block.previous\_hash);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\tCurrent hash: %llu\r\n", block.current\_hash);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\tNonce: %llu\r\n", block.nonce);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\tTransactions: %d\r\n", block.num\_transactions);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

for (uint32\_t num\_trx = 0; num\_trx < block.num\_transactions; num\_trx++)

{

snprintf(buffer, sizeof(buffer), "\t\t-Index: %d\r\n", block.transactions[num\_trx].index);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\t\tSensor mac address: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\r\n",

block.transactions[num\_trx].base\_mac\_addr[0],

block.transactions[num\_trx].base\_mac\_addr[1],

block.transactions[num\_trx].base\_mac\_addr[2],

block.transactions[num\_trx].base\_mac\_addr[3],

block.transactions[num\_trx].base\_mac\_addr[4],

block.transactions[num\_trx].base\_mac\_addr[5]);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\t\tTemperature: %f\r\n", block.transactions[num\_trx].sen\_temp);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

snprintf(buffer, sizeof(buffer), "\t\tTimestamp: %llu\r\n", block.transactions[num\_trx].timestamp);

WriteFile(hFile, buffer, strlen(buffer), &bytesWritten, NULL);

}

}

// Close the file

CloseHandle(hFile);

printf("Structure written to file successfully\n");

return RET\_CODE\_SUCCESS;

}

**main.c**

#include "include/blockchain.h"

#include "include/connect.h"

#include "include/fileIO.h"

#define ROTATIONS\_BLK (2)

#define MAX\_DEVICES (3u)

/\*==================== Global Variable Declarations ====================\*/

Blockchain iBlockchain = {0};

HANDLE thread\_calc,

thread\_recv,

thread\_bcast,

mutex;

HANDLE thread\_device[MAX\_DEVICES];

block\_t block = {0};

volatile int stop\_listening = 0;

volatile bool stop\_receiving = false;

/\*======================================================================\*/

/\*==================== Global Function Declarations ====================\*/

/\* Main initializing function.

\* - Opens the sockets for connection.

\* - Initializes the blockchain

\*

\*\*/

iResult main\_open(void);

/\* Main receiving function.

\* - This is the receive function that is ran from thread\_recv.

\* - It's purpose is to receive, while the thread\_calc (main\_mine)

\* function is generating the block. The goal is to achieve parallelism.

\* - Everything about receiveing from a socket is collected here, to ease

\* running it through a thread.

\*\*/

iResult\_thread WINAPI main\_recv(void \*);

/\* Supportive receiving function.

\* Main function, called from main\_recv(). It is defined inside the main file,

\* because the mutexes and threads are needed to ease its use and synchronization.

\* The purpose is signal the function that the program is ready to be finished.

\*\*/

iResult\_thread udp\_server\_receive(void \*);

/\* Main mining function.

\* - This is the mine function that is ran from thread\_mine.

\* - It's purpose is to mine, while the thread\_recv (main\_recv)

\* function is receiving. The goal is to achieve parallelism.

\* - Everything about mining a block and filling it's header structure,

\* except the transactions, which are generated from main\_recv,

\* is generated from this function.

\* - After the main\_mine and main\_recv functions are completed

\* successfully, the process of block generation is successful.

\*\*/

iResult\_thread WINAPI main\_mine(void \*);

/\* Main broadcasting function.

\* - This is the broadcasting function that is ran from thread\_bcast.

\* - If this is the first computer, that joined the blockchain, it will

\* generate the blockchain, otherwise it will wait for the blockchain to

\* be sent to it. Then it starts waiting other PC's, and will be the sender.

\* The goal is to achieve parallelism. While the block generation and

\* transactions are being done, this will be the synchronisation between PC's.

\*\*/

iResult\_thread WINAPI main\_bcast(void \*);

/\* Supportive broadcasting function.

\* Main function, called from main\_bcast(). It is defined inside the main file,

\* because the "stop\_listening" variable is needed. The purpose is signal the function

\* that the program is ready to be finished.

\*\*/

iResult start\_upd\_broadcast\_listener(void);

/\* Main de-initializing function.

\* - Closes the conenctions.

\* - Prints the blockchain

\* - Writes the output in a file

\*\*/

iResult main\_close(void);

/\*======================================================================\*/

int main(void) {

iResult iRes\_main = RET\_CODE\_ERROR;

iRes\_main = send\_broadcast\_message(&iBlockchain);

if (iRes\_main == RET\_CODE\_TIMEOUT) {

printf("First member, generating blockchain\n");

if((iRes\_main = main\_open()) != RET\_CODE\_SUCCESS) {

goto MAIN\_END;

}

} else if (iRes\_main == RET\_CODE\_ERROR) {

goto MAIN\_END;

}

mutex = CreateMutex(NULL, FALSE, NULL);

if (mutex == NULL) {

printf("CreateMutex error: %d\n", GetLastError());

return 1;

}

thread\_bcast = CreateThread(NULL, 0, main\_bcast, NULL, 0, NULL);

if (thread\_bcast == NULL || thread\_bcast == INVALID\_HANDLE\_VALUE) {

printf("CreateThread error when creating broadcast thread: %d\n", GetLastError());

return 1;

}

/\* Begin processes, each rotation is one block added \*/

uint32\_t rotations\_BLK = ROTATIONS\_BLK;

while (rotations\_BLK > 0) {

do

{

thread\_calc = CreateThread(NULL, 0x0, main\_mine, NULL, 0, NULL);

if (thread\_calc == NULL || thread\_calc == INVALID\_HANDLE\_VALUE) {

printf("CreateThread error: %d\n", GetLastError());

return 1;

}

stop\_receiving = false;

thread\_recv = CreateThread(NULL, 0, main\_recv, NULL, 0, NULL);

if (thread\_recv == NULL || thread\_recv == INVALID\_HANDLE\_VALUE) {

printf("CreateThread error: %d\n", GetLastError());

return 1;

}

WaitForSingleObject(thread\_calc, INFINITE);

WaitForMultipleObjects(MAX\_DEVICES, thread\_device, TRUE, INFINITE);

stop\_receiving = true;

} while (WaitForSingleObject(thread\_recv, INFINITE));

iBlockchain

.blocks[iBlockchain.num\_blocks]

.current\_hash = hash\_on\_transactions(iBlockchain.blocks[iBlockchain.num\_blocks]);

iBlockchain.num\_blocks++;

rotations\_BLK--;

}

MAIN\_END:

/\*\*\*\*\*\*Clean up, print, etc...\*\*\*\*\*\*\*/

iRes\_main = main\_close();

return iRes\_main;

} /\* main(void) \*/

/\*==================== Global Function Definitions ====================\*/

iResult main\_open(void)

{

iResult iResult = RET\_CODE\_SUCCESS;

/\* Initiate the genesis block \*/

if((iResult = initializeFirstBlock(&iBlockchain)) != RET\_CODE\_SUCCESS) {

printf("%s(): Unsuccessful Blockchain initialization", \_\_func\_\_);

goto END;

}

if((iResult = connect\_open()) != RET\_CODE\_SUCCESS) {

printf("%s(): Unsuccessful connection initialization", \_\_func\_\_);

goto END;

}

END:

return iResult;

} /\* main\_open(void) \*/

iResult\_thread main\_recv(void \*)

{

iResult\_thread iRes = RET\_CODE\_ERROR;

printf("Started connection operation\n");

do

{

for (int i = 0; i < MAX\_DEVICES; i++) {

thread\_device[i] = CreateThread(NULL, 0, udp\_server\_receive, NULL, 0, NULL);

}

} while (stop\_receiving == false);

printf("Done connection operation\n");

return iRes;

} /\* main\_recv(void \*) \*/

iResult\_thread main\_mine(void \*)

{

iResult\_thread iRes = RET\_CODE\_SUCCESS;

printf("Started mining operation\n");

/\* "Sleep" and an easy hashing algorithms are used,

\* instead of time consuming and complex

\* hashing algorithm tha mines blocks.\*/

Sleep(BLOCK\_GENERATION\_TIME);

build\_and\_verify\_block(&iBlockchain);

printf("Done mining operation\n");

return iRes;

} /\* main\_mine(void \*) \*/

iResult\_thread WINAPI main\_bcast(void \*)

{

iResult\_thread iRes = start\_upd\_broadcast\_listener();

return iRes;

} /\* main\_bcast(void \*) \*/

iResult main\_close(void)

{

iResult iResult = RET\_CODE\_SUCCESS;

stop\_listening = 1;

WaitForSingleObject(thread\_bcast, MAX\_TIMEOUT\_THREAD);

CloseHandle(thread\_calc);

CloseHandle(thread\_recv);

CloseHandle(thread\_bcast);

CloseHandle(mutex);

print\_blockchain(iBlockchain);

write\_to\_file(iBlockchain);

/\* Shutdown connections, close existing sockets \*/

iResult = connect\_close();

return iResult;

} /\* main\_close(void) \*/

iResult start\_upd\_broadcast\_listener(void)

{

iResult iRes = RET\_CODE\_SUCCESS;

WSADATA wsaData = {0};

iRes = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (iRes != RET\_CODE\_SUCCESS) {

fprintf(stderr, "WSAStartup failed\n");

return iRes;

}

SOCKET sockfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sockfd == INVALID\_SOCKET) {

fprintf(stderr, "socket failed with error: %ld\n", WSAGetLastError());

WSACleanup();

iRes = RET\_CODE\_ERROR;

return iRes;

}

struct sockaddr\_in serverAddr;

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

serverAddr.sin\_family = AF\_INET;

serverAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

serverAddr.sin\_port = htons(BROADCAST\_PORT);

iRes = bind(sockfd, (struct sockaddr \*)&serverAddr, sizeof(serverAddr));

if (iRes == SOCKET\_ERROR) {

fprintf(stderr, "bind failed with error: %ld\n", WSAGetLastError());

closesocket(sockfd);

WSACleanup();

return iRes;

}

char buffer[1024] = {0};

struct sockaddr\_in clientAddr;

int clientAddrLen = sizeof(clientAddr);

while (!stop\_listening) {

int numBytes = recvfrom(sockfd, buffer, sizeof(buffer), 0, (struct sockaddr \*)&clientAddr, &clientAddrLen);

if (numBytes == SOCKET\_ERROR) {

fprintf(stderr, "recvfrom failed with error: %ld\n", WSAGetLastError());

closesocket(sockfd);

WSACleanup();

iRes = RET\_CODE\_ERROR;

return iRes;

}

WaitForSingleObject(thread\_recv, INFINITE);

uint32\_t dwCount=0, dwWaitResult = 0;

dwWaitResult = WaitForSingleObject(mutex, INFINITE);

if (dwWaitResult == WAIT\_OBJECT\_0)

{

buffer[numBytes] = '\0';

if (strcmp(buffer, "I want to join") != 0) {

printf("Unrecognised request!");

} else {

printf("Received message from %s:%d: %s\n", inet\_ntoa(clientAddr.sin\_addr), ntohs(clientAddr.sin\_port), buffer);

uint64\_t send\_size = (uint64\_t) sizeof(iBlockchain);

printf("Sending blockchain.\n");

iRes = sendto(sockfd, &send\_size, sizeof(uint64\_t), 0, (struct sockaddr \*)&clientAddr, sizeof(clientAddr));

if (iRes == SOCKET\_ERROR) {

fprintf(stderr, "sendto failed with error: %ld\n", WSAGetLastError());

closesocket(sockfd);

WSACleanup();

return 1;

} else {

iRes = sendto(sockfd, &iBlockchain, send\_size, 0, (struct sockaddr \*)&clientAddr, sizeof(clientAddr));

if (iRes == SOCKET\_ERROR) {

fprintf(stderr, "sendto failed with error: %ld\n", WSAGetLastError());

closesocket(sockfd);

WSACleanup();

return 1;

}

}

}

ReleaseMutex(mutex);

}

}

closesocket(sockfd);

WSACleanup();

return 0;

} /\* start\_upd\_broadcast\_listener(void) \*/

iResult\_thread udp\_server\_receive(void \*)

{

iResult\_thread iRes = RET\_CODE\_SUCCESS;

char rx\_buffer[20] = {0};

int addr\_family = AF\_INET;

int ip\_protocol = 0;

struct sockaddr\_in6 dest\_addr;

struct sockaddr\_in \*dest\_addr\_ip4 = (struct sockaddr\_in \*)&dest\_addr;

dest\_addr\_ip4->sin\_addr.s\_addr = htonl(INADDR\_ANY);

dest\_addr\_ip4->sin\_family = AF\_INET;

dest\_addr\_ip4->sin\_port = htons(DEFAULT\_PORT);

ip\_protocol = IPPROTO\_IP;

SOCKET sock = socket(addr\_family, SOCK\_DGRAM, ip\_protocol);

int err = 0;

if (sock == SOCKET\_ERROR) {

// printf("Unable to create socket: error %d", WSAGetLastError());

}

#ifdef PRINT\_DEBUG

printf("Socket created\n");

#endif

// Set timeout

uint32\_t timeout = 3000;

err = bind(sock, (struct sockaddr \*)&dest\_addr, sizeof(dest\_addr));

if (err == SOCKET\_ERROR) {

// printf("Socket unable to bind: error %d", WSAGetLastError());

}

#ifdef PRINT\_DEBUG

printf("Socket bound, port %d", DEFAULT\_PORT);

#endif

err = setsockopt(sock, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&timeout, sizeof(timeout));

if (err == SOCKET\_ERROR) {

#ifdef PRINT\_DEBUG

printf("setsocketopt error: %d", WSAGetLastError());

#endif

}

struct sockaddr\_storage source\_addr;

socklen\_t socklen = sizeof(source\_addr);

do {

sensor\_info\_t sen\_info = {0};

int len = recvfrom(sock, &sen\_info, sizeof(sensor\_info\_t),

0, (struct sockaddr \*)&source\_addr, &socklen);

if (len < 0) {

// printf("recvfrom failed: error %d\n", WSAGetLastError());

break;

}

uint32\_t dwCount=0, dwWaitResult = 0;

dwWaitResult = WaitForSingleObject(mutex, INFINITE);

if (dwWaitResult == WAIT\_OBJECT\_0)

{

if (strncmp(sen\_info.cmd, "ESP32", ESP32\_REQ\_SIZE) != 0) {

printf("Unknown request %s, no permission given\n", sen\_info.cmd);

break;

} else {

printf("Transaction number: %d\n", iBlockchain.blocks[iBlockchain.num\_blocks].num\_transactions);

printf("Sensor temperature: %f Sensor mac address: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",

sen\_info.sen\_temp,

sen\_info.base\_mac\_addr[0],

sen\_info.base\_mac\_addr[1],

sen\_info.base\_mac\_addr[2],

sen\_info.base\_mac\_addr[3],

sen\_info.base\_mac\_addr[4],

sen\_info.base\_mac\_addr[5]);

add\_transaction(&iBlockchain.blocks[iBlockchain.num\_blocks], &sen\_info);

}

ReleaseMutex(mutex);

}

} while (stop\_receiving == false);

if (sock != -1) {

#ifdef PRINT\_DEBUG

printf("Shutting down socket and restarting...");

#endif

closesocket(sock);

return 0;

}

return 1;

} /\* udp\_server\_receive(void \*) \*/

/\*=====================================================================\*/

**Makefile**

exec = run.exe

sources = $(wildcard src/\*.c)

objects = $(sources:.c=.o)

flags = -g -Wall -lm -fPIC -lwsock32 -lWs2\_32 -lkernel32

$(exec): $(objects)

gcc $(objects) $(flags) -o $(exec)

%.o: %.c include/%.h

gcc -c $(flags) $< -o $@

clean:

-rm \*.out

-rm \*.exe

-rm \*.o

-rm \*.a

-rm src/\*.o

**udp\_client.c**

#include <string.h>

#include <sys/param.h>

#include "freertos/FreeRTOS.h"

#include "freertos/task.h"

#include "freertos/event\_groups.h"

#include "esp\_system.h"

#include "esp\_event.h"

#include "esp\_log.h"

#include "nvs\_flash.h"

#include "esp\_netif.h"

#include "protocol\_examples\_common.h"

#include "esp\_mac.h"

#include "lwip/err.h"

#include "lwip/sockets.h"

#include "lwip/sys.h"

#include <lwip/netdb.h>

#include "driver/temperature\_sensor.h"

#ifdef CONFIG\_EXAMPLE\_SOCKET\_IP\_INPUT\_STDIN

#include "addr\_from\_stdin.h"

#endif

#if defined(CONFIG\_EXAMPLE\_IPV4)

#define HOST\_IP\_ADDR CONFIG\_EXAMPLE\_IPV4\_ADDR

#elif defined(CONFIG\_EXAMPLE\_IPV6)

#define HOST\_IP\_ADDR CONFIG\_EXAMPLE\_IPV6\_ADDR

#else

#define HOST\_IP\_ADDR "255.255.255.255"

#endif

#define PORT CONFIG\_EXAMPLE\_PORT

static const char \*TAG = "UDP Client:";

static const char \*payload = "ESP32";

typedef struct {

uint8\_t base\_mac\_addr[6];

float sen\_temp;

char cmd[6];

} sensor\_info\_t;

static sensor\_info\_t sen\_info = {0};

temperature\_sensor\_handle\_t temp\_sensor = NULL;

static void udp\_client\_task(void \*pvParameters)

{

int addr\_family = 0;

int ip\_protocol = 0;

strncpy(sen\_info.cmd,"ESP32", 6);

while (1) {

#if defined(CONFIG\_EXAMPLE\_IPV4)

struct sockaddr\_in dest\_addr;

dest\_addr.sin\_addr.s\_addr = inet\_addr(HOST\_IP\_ADDR);

dest\_addr.sin\_family = AF\_INET;

dest\_addr.sin\_port = htons(PORT);

addr\_family = AF\_INET;

ip\_protocol = IPPROTO\_IP;

#endif

int sock = socket(addr\_family, SOCK\_DGRAM, ip\_protocol);

if (sock < 0) {

ESP\_LOGE(TAG, "Unable to create socket: errno %d", errno);

break;

}

ESP\_LOGI(TAG, "Socket created, sending to %s:%d", HOST\_IP\_ADDR, PORT);

while (1) {

int err = 0;

ESP\_ERROR\_CHECK(temperature\_sensor\_get\_celsius(temp\_sensor, &sen\_info.sen\_temp));

ESP\_LOGI(TAG, "Sending temperature value %f", sen\_info.sen\_temp);

err = sendto(sock, &sen\_info, sizeof(sensor\_info\_t), 0, (struct sockaddr \*)&dest\_addr, sizeof(dest\_addr));

if (err < 0) {

ESP\_LOGE(TAG, "Error occurred during sending: errno %d", errno);

break;

}

vTaskDelay(2000 / portTICK\_PERIOD\_MS);

}

if (sock != -1) {

ESP\_LOGE(TAG, "Shutting down socket and restarting...");

shutdown(sock, 0);

close(sock);

}

}

vTaskDelete(NULL);

}

void app\_main(void)

{

ESP\_LOGI(TAG, "Install temperature sensor, expected temp ranger range: 10~50 ℃");

temperature\_sensor\_config\_t temp\_sensor\_config = TEMPERATURE\_SENSOR\_CONFIG\_DEFAULT(10, 50);

ESP\_ERROR\_CHECK(temperature\_sensor\_install(&temp\_sensor\_config, &temp\_sensor));

ESP\_LOGI(TAG, "Enable temperature sensor");

ESP\_ERROR\_CHECK(temperature\_sensor\_enable(temp\_sensor));

esp\_err\_t ret = esp\_read\_mac(sen\_info.base\_mac\_addr, ESP\_MAC\_EFUSE\_FACTORY);

if (ret != ESP\_OK) {

ESP\_LOGE(TAG, "Failed to get base MAC address from EFUSE BLK0. (%s)", esp\_err\_to\_name(ret));

ESP\_LOGE(TAG, "Aborting");

abort();

} else {

ESP\_LOGI(TAG, "Base MAC Address read from EFUSE BLK0");

}

ESP\_LOGI(TAG, "Sensor mac address: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",

sen\_info.base\_mac\_addr[0],

sen\_info.base\_mac\_addr[1],

sen\_info.base\_mac\_addr[2],

sen\_info.base\_mac\_addr[3],

sen\_info.base\_mac\_addr[4],

sen\_info.base\_mac\_addr[5]);

ESP\_ERROR\_CHECK(nvs\_flash\_init());

ESP\_ERROR\_CHECK(esp\_netif\_init());

ESP\_ERROR\_CHECK(esp\_event\_loop\_create\_default());

/\* This helper function configures Wi-Fi or Ethernet, as selected in menuconfig.

\* Read "Establishing Wi-Fi or Ethernet Connection" section in

\* examples/protocols/README.md for more information about this function.

\*/

ESP\_ERROR\_CHECK(example\_connect());

xTaskCreate(udp\_client\_task, "udp\_client", 4096, NULL, 5, NULL);

}