Cortex-A72 Smart Home Hub

Ian M Tavener

Ensign College

IT 316 Computer Architecture

Shane Stailey

December 15, 2024

Abstract

This paper presents the architectural design of a smart home monitoring hub utilizing the low-powered Cortex-A72 CPU. The design implements FreeRTOS, while integrating WiFi 802.11ax, Bluetooth, and Zigbee modules. Using the ARMv8-A instruction set, the system is optimized for efficient communication and energy management, with a low-power ARM based CPU and eMMC storage. Digital logic and microarchitecture are tailored for smart home applications, handling control signals such as door locks and thermostat adjustments with precision. The software layer includes OpenHAB, which provides centralized control, automation, and third-party integration via MQTT and REST APIs. It supports secure communication and firmware updates, enabling users to interact with their smart home via a smartphone app or voice control through Alexa or Google Assistant. This setup allows seamless integration and management of devices, addressing key challenges like connectivity and security. Two case examples are presented: a doorbell press and creating a thermostat schedule from a smartphone. These scenarios highlight the system's ability to manage and automate different smart home tasks through the four abstraction layers of the architecture.

Table of Contents

Introduction

In the age of connected homes, the demand for a smart, efficient, and secure monitoring hub is increasingly critical. By utilizing the Internet-of-Things, we are able to create a network of smart devices, to be managed by a central computer within our home. This computer is known as a smart home monitoring hub. The hub enables monitoring, automation, and optimization of these devices from one central location.

Some of the common smart home devices which may be connected to a hub include lights, locks, thermostats, and cameras. Using sensors and schedules for input, we are able to fully automate the operation of our home.

Our goal, in this paper, is to outline the design of a new smart home monitoring hub, and describe it's applications. Our hub will meet the requirements of:

- An energy efficient design.
- Integration with necessary networking protocols, including WiFi, Zigbee, and Bluetooth.
- Secure communication with both IoT devices and the cloud.

By defining the architecture through all abstraction layers, and analyzing workflow and system cohesion, we will prove how this design will provide the framework necessary to support your smart home.

Abstraction Layers

Hardware Layer

Components:

CPU

It is crucial that we select a CPU which provides a balance of performance and energy efficiency. For this reason, we want to choose a low-powered ARM (Advanced RISC Machine)-based CPU. The Cortex-A72 would fit well with this design. It uses 20% less power and provides 90% greater performance than it's predecessor, the A57. This is a 4-core CPU, with clock speeds of up to 2.2 GHz. For caches, it has 80KiB L1, and up to 4MiB for L2, with no L3 cache.

This choice is optimal for an always-on system, where the hub won't be powered off or ever put in a reduced function state in order to save power. The architecture diagram in **Appendix A** demonstrates the internal design and some of the key features of the Cortex-A72.

Wireless Modules

There are 2 main considerations for wireless communication with this device. It will need to communicate with sensors and other nearby devices, and it will need to connect to the cloud. We plan to implement the usage of 3 different wireless modules, each with distinct purposes, and all providing Advanced Encryption Standard (AES) encryption.

WiFi 802.11ax will allow for connections across 2.4GHz and 5GHz bands. It will provide high throughput for devices such as smart home security cameras, as well as compatibility with WiFi smart plugs that allow smart home control and monitoring over any wall socket in the home.

Bluetooth will be used for device pairing for remote smart home control from a smart phone or tablet, without the need for internet connectivity, as well as compatibility with Bluetooth smart home devices.

Zigbee will be the standard for most network devices. Zigbee is an open standard, which can use radio bands separate from WiFi, thus avoiding network congestion. It can be used to create a mesh network, which means devices will retain connectivity as long as they can communicate with any other device, and will still be reliable despite being outside of WiFi range. Most importantly, it is an energy efficient protocol which can be used in battery powered devices like door locks.

Storage

For storage, we are electing to go with a standard technology used for mobile devices, embedded MultiMediaCard (eMMC). eMMC provides NAND flash memory and a storage controller. Flash memory has the benefit of being persistent, which means it is retained through power loss. With an integrated controller, the smart home hub CPU will not have to handle the storage management for the device (Kirvan, 2024).

The eMMC will be used to store the operating system for the embedded system, as well as custom scripts for device automation.

Power Supply

This hub would use approximately 6-11W to power all the existing modules. To meet these demands, we would suggest a 12V, 2A USB-C power adapter. This will provide 24W of power, which would place it only under 50% load during regular operation, and provide extra headroom for future module expansion.

Numbering Systems:

Each device in our smart home WiFi network will be identified with a 32-bit IPv4 address. This is a series of 32 1's or 0's, split into 4 octets, ranging from 0-256. Each device in the Zigbee network will be given a unique ID in a Personal Area Network (PAN), which is a 16-bit number. Bluetooth devices will be identified by their Media Access Control (MAC) address, which is a 48-bit number, represented as 12 hexadecimal digits.

The multitude of devices on our smart home network will be in constant communication with the hub. The data will relate to readings from the sensors within the home, and controls signals being sent to locks, thermostats, or switches. This data is passed via wireless signal as binary values, which will then be interpreted into human readable form by the hub.

Digital Logic:

The control signals mentioned above, in essence, are a series of on or off signals. A door is either locked or unlocked, and the state is determined with digital logic. There may be a schedule to consider, events which override the schedule, or manual activation by the user. All these factors will be used in if/else statements that will be processed through logic gates to determine whether an on or off signal should be generated. For example if the schedule says 'unlock' OR an event says 'unlock' then the door will unlock, but NOT if the user has manually locked the door. This same logic can be applied for activating motion controlled lights, adjusting a thermostat, or recording with a security camera.

This logic will be handled in the CPU by the Arithemtic Logic Unit (ALU), with the end result being a 1 or 0 sent to the appropriate device to signal whether it should be on or off.

Microarchitecture:

ARM is based on the RISC CPU architecture. According to Sheldon (2022), "Arm processors can execute many more millions of instructions per second than Intel processors. By stripping out unneeded instructions and optimizing pathways, an Arm processor can deliver outstanding performance while using much less energy than a CISC-based processor." With the simplified instruction set, ARM is able to perform better than a complex instruction set, using the same hardware.

The main benefit of a RISC processor is the simplified instruction set. This highly optimized set allows for faster execution and more efficient use of CPU clock cycles. The CPU requires fewer transistors to do the same amount of work through this simplified instruction set and is especially efficient with repetitive tasks like those that would be used for process data from monitors and sensors.

Operating System Layer

Type:

For our OS, we've chosen to go with FreeRTOS, which is an open source real-time operating system (RTOS) for embedded systems. This is our preference because it provides a much faster response to external events than a traditional operating system. As Costagli (2024) explains, "An RTOS differs in that it typically provides a hard real time response, providing a fast, highly deterministic reaction to external events." This means swift and predictable responses to sensors and schedules in our smart home network.

Some of the key elements of an RTOS include:

- **Priority Based Scheduling:** Being able to separate tasks based on their criticality, providing priority to those that are more important.

- **Improved Efficiency:** The OS will operate based on events instead of polling for information from a process that hasn't occurred.
- **Idle Processing:** Background processes are separated into an idle task. This ensures that CPU and other system resources remain available for main processing.

Instruction Set Architecture:

The Cortex-A72 that we have chosen in this build utilizes the ARMv8-A instruction set. Though not originally supported by FreeRTOS, it can used by configuring FreeRTOS into AArch32 mode. This will be compatible with the ARMv8-A instruction set architecture. This option provides the benefits of a lightweight RTOS, but also allows us to switch to a more feature rich OS, without changing hardware. This can be useful depending on the specific network configuration, and our priorities with our smart home network.

Security:

Large amounts of personally sensitive information will be managed by our smart home hub, including camera feeds and home/away schedules. For these reasons, security is of utmost importance in our design. For these reasons we will make use of secure boot and OS level encryption.

The Cortex-A72 stores a Root of Trust (ROT) within the ROM, which prevents unsigned code from being inserted and booted. This will protect our hub from unauthorized or malicious code during the boot process.

Cryptographic extensions are available for storing data in our hub, which use the ISA's hardware assisted encryption. Network traffic will make use of TLS, ensuring all data transmitted over WiFi, Zigbee, and Bluetooth is done so securely.

Drivers:

In our system, drivers are essential for communication with the various devices within our smart home network. These will typically be written in C or Assembly, thereby bypassing the upper abstraction layers and providing more efficient code execution. Efficiency is critical with our real time system design, which is fully supported through low-level languages. An example would be a driver for our WiFi module would allow computer commands to be translated into a signal which could then be communicated to our smart home camera to pan, tilt, or zoom the camera.

Software Layer

Application & Automation:

In order to interact with our smart home network, we will need software that is able to interact with many different platforms, including Message Querying Telemetry Transport (MQTT), HTTP, and Zigbee. For this purpose, we are choosing OpenHAB. This is an open-source application, which can run on FreeRTOS.

MQTT is a lightweight protocol that allows for bidirectional communication with devices through a publish-subscribe model. MQTT supports encryption through TLS, ensuring secure communication sessions.

OpenHAB provides a dashboard for monitoring and control of our devices, as well as a rules engine to create automated triggers and processes. For example, you could set up rules, or a schedule for turning on a particular light. OpenHAB is written in Javascript, and allows scripting through Python or JavaScript. These are all feature-rich, high-level languages, which allow for complex logic to be easily scripted.

Devices can be managed centrally, within OpenHAB. It supports network based firmware updates for devices in our smart home.

APIs for Third Party Integration:

OpenHAB comes complete with Representational State Transfer (REST) API. This allows for any third party device or service to connect to OpenHAB. This includes physical devices within our network, such as cameras, locks, and sensors, as well as services such as weather and security monitoring.

Through this API, the home user will also be able to interact with their smart home network through AWS Lambda or Google Cloud Functions.

User Layer

User Interaction:

The most popular method of interaction with this system will most likely be a smartphone app. This could be done through the OpenHAB Android or iOS app. This will allow for remote monitoring, when away from home, but also ease of use while at home. Users can set schedules, receive notifications, and create automation scripts from anywhere. This app also allows for interaction with Alexa or Google Assistant to give users voice control over their smart home devices.

For a lightweight build, this smart home hub will not typically have a keyboard and monitor left attached it. However, this option is possible for advanced troubleshooting and maintenance needs.

Security:

The user is typically the weakest point in the system's security. Multi-factor Authentication is a critical security feature at this layer of the model. This would involve the user signing into the smart home hub through factors such as: something you know, something you have, something you are, etc. For example, this could be a password and a fingerprint. If we rely only on one factor, like a password, this makes unauthorized access much easier for an attacker.

Analysis & Cohesion

Example Workflows

Adjusting the thermostat schedule

Diagram included as **Appendix B**.

1. Interacting with smartphone app

The user wants to to create a thermostat schedule to match their work schedule. They don't want to waste energy heating an empty home. To accomplish this, they interact with the OpenHAB app on their smartphone. The schedule they create includes specific times and specific temperature targets.

Layer: User

2. Communication from smartphone to FreeRTOS hub

The OpenHAB app converts the user input into a high-level language script, in either Python or JavaScript. It then communicates with OpenHAB on our hub by sending a request to the FreeRTOS hub, containing the thermostat schedule information. This data is encapsulated in a Bluetooth message. The FreeRTOS application checks the contents of the Bluetooth packet and verifies the schedule details for completeness, before processing the new information.

Layer: Software

3. Processing on FreeRTOS hub

The FreeRTOS operating system provides the drivers and API necessary to communicate with the smartphone. These drivers are primarily written in C. The OS send the automation logic to the CPU. The OS uses the MQTT library to create a message to be sent to the smart home thermostat

device in our home. FreeRTOS is also responsible for session management and authentication with the IoT devices.

Layer: Operating System

4. Sending the command to the thermostat

The hub uses its ARM Cortex-A72 CPU to manage and execute the automation logic for the thermostat schedule. It then sends an MQTT message to the thermostat to update its settings, based on the user-defined schedule. MQTT provides secure communication with minimal latency. Using a TCP connection, it ensures the data reaches its destination. The CPU interfaces with the Zigbee module to then send the command to the thermostat. This data is passed in binary form through the network.

Layer: Hardware

Receiving a notification from a doorbell

Diagram included as **Appendix C**.

1. Pressing the doorbell of the smart home network

A visitor presses the doorbell. The doorbell generates a signal via MQTT, which is transmitted via Zigbee to the smart home hub. The signal is picked up by the hub's transceiver module that is connected to the Cortex-A72 CPU. The CPU then sends the signal up through the OS to reach the OpenHAB application.

Layer: Hardware

2. Processing the signal

Once the signal is captured, it is passed from the hardware up through the FreeRTOS operating system. The operating system is able to make sense of this signal using drivers for the Zigbee module. The operating system prepares the signal for processing using task scheduling, interrupt handling, and resource management. It then communicates this signal up to the next layer by using it's APIs.

Layer: Operating System

3. Preparing the notification

OpenHAB receives the MQTT message that was originally generated from the doorbell. This signal has been forward by the operating system. The OpenHAB application verifies accuracy of the message and determines which processes should occur next with it's notification settings, which have been written in Java. The application identifies the user's smart phone device and sends the notification via Wi-Fi.

Layer: Software

4. Receiving the notification

The user's phone alerts the user in the configured manner (vibrate, ring, silent, etc). The user can then interact with their phone and see the specific details of the doorbell alert which has been generated, including date, time, and image from the doorbell's camera. The user can then utilize options in the app to interact with their smart home network, by unlocking a door or activating a microphone.

Layer: User

System Cohesion

This design implements an ARM-based Cortex-A72 CPU. This CPU utilizes the ARMv8-A instruction set. Though this ISA is not listed in the supported instruction sets for FreeRTOS, porting options exist to gain functionality. We thought this was a more modular approach than using a lighter weight ARMv8-M architecture. With this approach, we retain the option of upgrading the RTOS to a more feature rich OS such as Linux, if our needs require.

Our design provides all the necessary elements at each layer of the abstraction model. A full featured app is available for users on the smartphone, with multi-factor authentication. We will make use of the OpenHAB application to manage schedules and build automation scripts. We use the FreeRTOS real time operating system to minimize latency and increase efficiency with processing input from sensors. And all of this is built on top of an ARM-based CPU, with an emphasis on low-power operation.

One of the main problems that would be encountered with this system is a loss of internet connectivity. This is mitigated by allowing users to connect directly to the hub from their smart phone using a Bluetooth connection. With schedules and automation rules retained within the hub storage, our smart home network will retain all functionality despite losing internet connectivity. The only functions that will be lost are being able to remotely connect from outside the home, and downloading updates from connected services such as weather or security platforms.

Another common problem will be a power outage. Many critical devices in this network, such as door locks, will be battery operated and are unaffected by a power outage. For other devices, such as cameras, which require more power, we recommend connection through an uninterruptible power

source (UPS). The hub, itself, could also be connected to a UPS. Due to it's low power consumption, it can get several hours of extended use through a power outage.

Automation rules can be built in the OpenHAB application logic, based on user input. Users will not need to have any coding experience to be able to build these rules, which OpenHAB will then convert into high-level language scripts.

An important consideration with any system is adherence to the principles of CIA (Confidentiality, Integrity, and Availability). In our design, confidentiality is ensured through the encryption protocols used for data transmission through TLS and AES, as well as the OS based encryption on the system. Integrity is ensured with secure boot and the FreeRTOS Root of Trust model, which prevents unsigned code from being executed. Scripts and automation rules could also be backed up to the cloud to ensure a swift restoration process in the event of critical system failure. Lastly, availability is maintained through the backup power options mentioned above.

Conclusion

Our design demonstrates a balance between performance, efficiency, and security. The power of the Cortex-A72 running FreeRTOS ensures low-latency processing for real-time communication with connected devices, while operating on a minimal amount of power. By implementing OpenHAB management, we've created a versatile platform for managing of a wide variety of devices, operating on WiFi, Bluetooth, or Zigbee.

The system's architecture, when analyzed across all four abstraction layers, demonstrates its cohesive nature. Users can easily create schedules for devices, such as thermostats, using their smartphone. These inputs are then converted into scripts and reliably communicated to the hub, for efficient processing and management of devices. The interaction highlights how the design integrates the user, software, operating system, and hardware layers of the architecture.

The hub address key needs of a smart home network. It provides centralized device management, energy-efficient operation, and secure communication. Its security measures protect sensitive user data in transit and at rest, mitigated risks of data compromise.

There are a couple limitations worth stressing, which are the dependence on internet connectivity and susceptibility to power failures. Though the device could maintain adherence to the CIA triad during extended losses of internet connectivty, it is particularly vulnerable during extended power outages. Battery backup would support critical function during power loss, but extended outages would eliminate hub functionality.

In conclusion, this hub provides a robust solution to meet the demands of a modern home. Though limitations exist, this design's integration of technology ensures reliable, secure, and efficient operation of the smart home network.

References

Andersen, G., MoldStud Research Team. (2024, January 17). *Exploring the Role of Embedded Software Engineering in Smart Homes*. MoldStud. https://moldstud.com/articles/p-exploring-the-role-of-embedded-software-engineering-in-smart-homes

Arm. (n.d.). *CPU Cortex-A72*. https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a72

Baskar, N. (2023, June 29). *Smart Homes & Beyond: Unleashing the Power of Embedded Systems*. Skill-Lync. https://skill-lync.com/blogs/smart-homes-beyond-unleashing-the-power-of-embedded-systems

Costagli, E. (2024, February 23). *What is a Real-Time Operating System (RTOS)?*. High Integrity Systems. https://www.highintegritysystems.com/rtos/what-is-an-rtos/

Crudu, A., MoldStud Research Team. (2024, February 1). *Systems Engineering for Smart Home Technologies: Designing Connected Living Spaces*. MoldStud. https://moldstud.com/articles/p-systems-engineering-for-smart-home-technologies-designing-connected-living-spaces

Duran, J. (2018, October 18). *We analyze openHAB, the home automation platform at our fingertips*. Panel. https://www.panel.es/en/plataforma-openhab-domotica-a-nuestro-alcance/

Dusun IoT. (2024, January 18). *Smart Home Hub, All You Need To Know in 2024*. https://www.dusuniot.com/blog/smart-home-hub-all-you-need-to-know-in-2023/

EMQX Team. (2024, August 26). *MQTT Protocol Explained: Ultimate Guide for IoT Beginners*. EMQX. https://www.emqx.com/en/blog/the-easiest-guide-to-getting-started-with-mqtt

FreeRTOS. (2024, November). *Using FreeRTOS on ARM Cortex-A9 Embedded Processors*. https://www.freertos.org/Using-FreeRTOS-on-Cortex-A-Embedded-Processors

Frumusanu, A. (2015, April 23). *ARM Reveals Cortex-A72 Architecture Details*. AnandTech. https://www.anandtech.com/show/9184/arm-reveals-cortex-a72-architecture-details

Hassan, S., Eassa, A. (2022). *A Proposed Architecture for Smart Home Systems Based on IoT, Context-awareness and Cloud Computing*. International Journal of Advanced Computer Science and Applications (IJACSA), 13(6), 89-96. https://thesai.org/Downloads/Volume13No6/Paper_12-A_Proposed_Architecture_for_Smart_Home_Systems.pdf

Kirvan, P. (2024, July). *What is an embedded MultiMediaCard (eMMC)?*. TechTarget. https://www.techtarget.com/searchstorage/definition/eMMC-embedded-MultiMediaCard

Sen, E. A. (2024, April 30). *Revolutionizing Architecture: The Impact of Smart Homes on Future Designs*. Illustrarch. https://illustrarch.com/articles/architectural-technology/25274-impact-of-smart-homes-on-future-designs.html

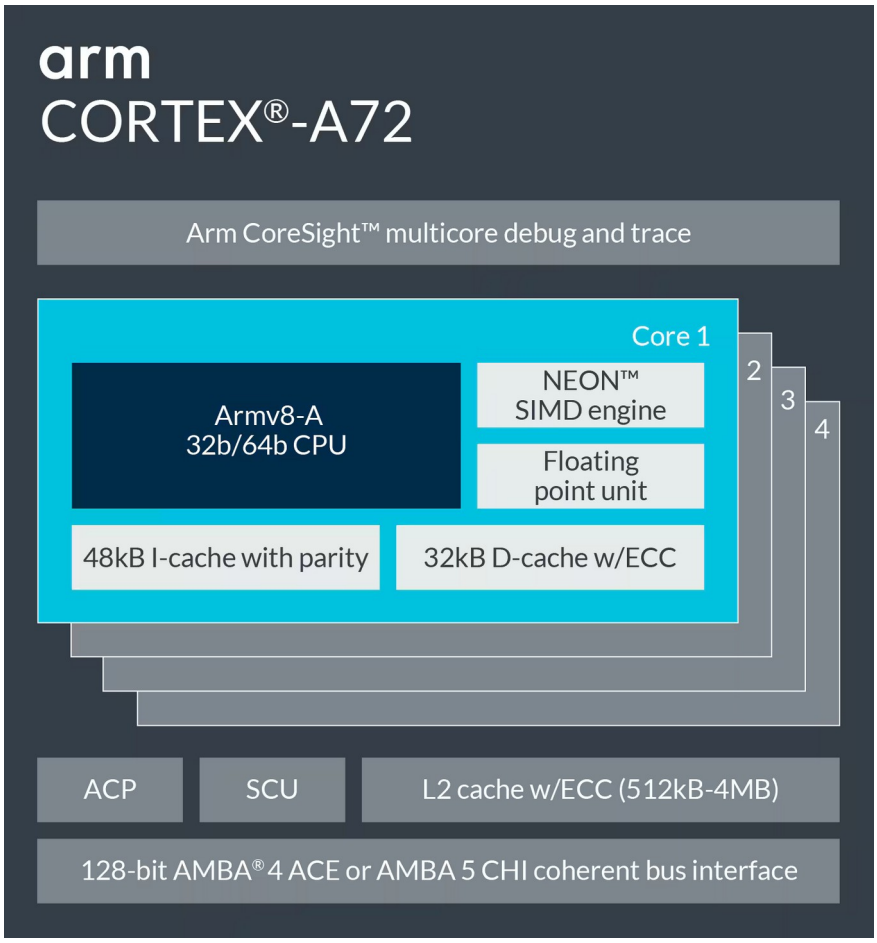
Sheldon, R. (2022, May). *Arm processor.* TechTarget. https://www.techtarget.com/whatis/definition/ARM-processor

Appendices

Appendix A - Cortex-A72 architecture

Title: Image of Cortex-A72 architecture**.**

Description: This diagram highlights key components and features of the processor. Each core contains a CPU, SIMD Engine for advanced vector operations, FPU for floating point arithmetic, instruction and data caches.

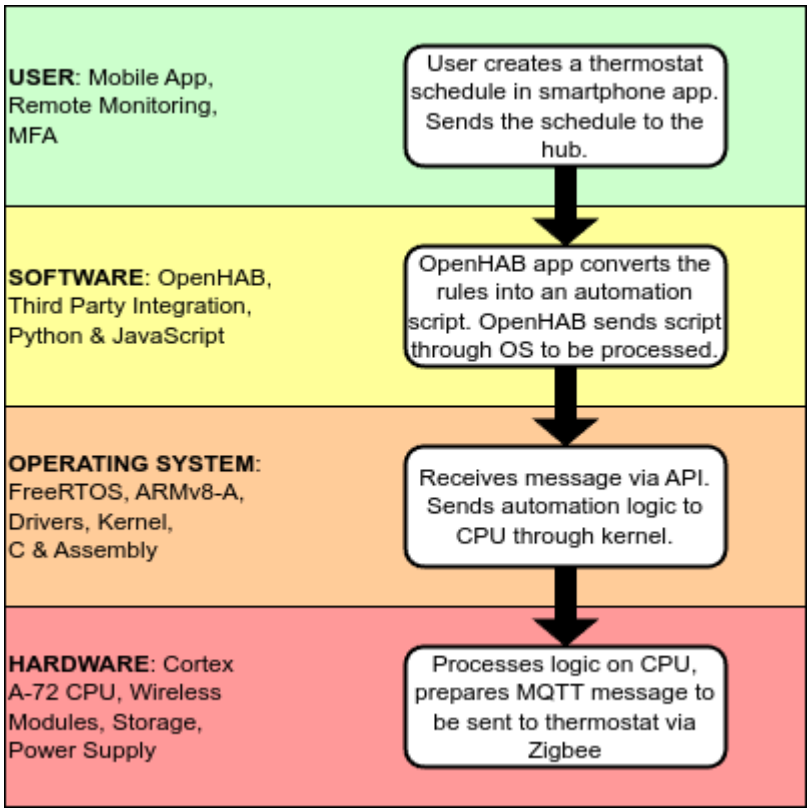


Source: Arm. (n.d.). *CPU Cortex-A72*. *Architecture Diagram*

Appendix B - Thermostat abstraction layer model

Title: Abstraction layer diagram for adjusting thermostat process.

Description: This diagram shows the steps taken in each of the 4 abstraction layers, starting with the schedule being created in the smartphone, to the signal being pushed to the thermostat.

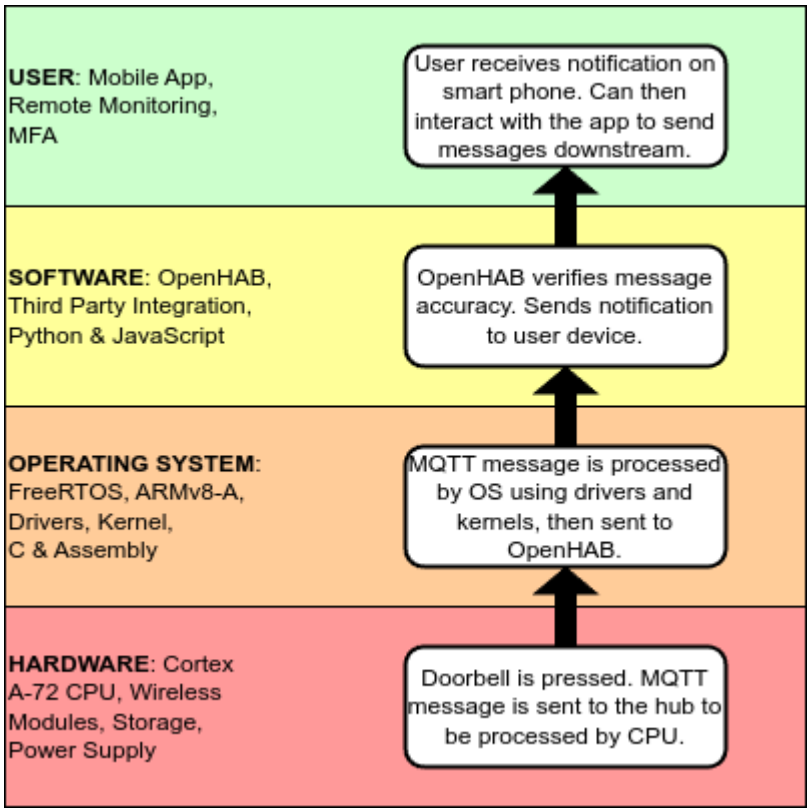


Source: Created by author.

Appendix C - Doorbell abstraction layer model

Title: Abstraction layer diagram for a doorbell press.

Description: This diagram shows the steps taken in each of the 4 abstraction layers, starting with the smart home doorbell being pressed, to the notification being delivered to the user's smartphone.



Source: Created by author.