NAIT

Edmonton, Alberta

Accessible Universal Remote

As a submission to

Mr. Kelly Shepherd, Instructor

English and Communications Department

&

Mr. Kevin Moore, Instructor

Computer Engineering Department

Submitted by

Brandon Carpenter, Student

&

Cole Yaremko, Student

CMPE2965

Computer Engineering Technology

April 26, 2024

11762 - 106 Street
Edmonton, AB T5G 2R1
April 26, 2024

Mr. Kelly Shepherd, Mr. Kevin Moore
Instructors
NAIT
11762 – 106 Street NW
Edmonton, AB T5G 2R1

Dear Mr. Shepherd and Mr. Moore,

As per the requirements of CMPE2965, we are submitting the report titled Accessible Universal Remote for evaluation.

This document elaborates on the comprehensive planning and execution of the universal remote project. It also discusses both the achievements and challenges faced during the project's development. The report delves into the electronic components used, programming methodologies adopted, and web development aspects integrated into the project.

I want to express my gratitude to my instructors throughout my time at NAIT for their patience and dedication to educating their students. Their genuine interest in our success fosters a strong desire to learn and succeed.

Sincerely,

Brandon Carpenter and Cole Yaremko, CNT Students

Table of Contents

List of Figures

Statement of Contribution

This document was written by Brandon Carpenter and Cole Yaremko. The table below describes the contribution to the paper and views the effort each put in to be approximal equal.

Section	Author	Reviewer
Title Page, Table of Contents, Cover Letter, List of Figures and Tables, and general formatting	Cole Yaremko	Brandon Carpenter
Abstract	Cole Yaremko	Brandon Carpenter
Intro	Cole Yaremko	Brandon Carpenter
Background	Brandon Carpenter	Cole Yaremko
Micro Controllers Design	Brandon Carpenter	Cole Yaremko
Mechanical Design	Brandon Carpenter	Cole Yaremko
Embedded Design	Brandon Carpenter	Cole Yaremko
Web Design	Cole Yaremko	Brandon Carpenter
Implementation	Both	Cole Yaremko
Project Results	Cole Yaremko	Brandon Carpenter
Conclusion	Brandon Carpenter	Cole Yaremko
References	Both	Cole Yaremko

Abstract

The Accessible Universal Remote was an exploration into embedded systems development as well as internet of things technology. An infrared remote control was developed to adjust and alter common remote-using devices such as televisions and satellite boxes. The main remote control is run by an STM32 micro controller running C and C++ code to drive an infrared transmitter and receiver. The STM32 communicates over serial with an ESP8266 to allow for internet access. The internet access is required to allow users to access a React web page where they will be able to program the remote and which macros it will be able to send. The React page also allows a user to set schedules to automate when the remote will send macros for specific channels. The React page and ESP8266 communicate with an SQL database that will be used to store the user's macros, schedules, and each remote they own.

The results of the project were measured by programming the remote and attempting to use it to interact with infrared devices. The tests confirmed that the remote was able to effectively send and receive infrared signals. The remote was then tested with the web interface by making changes on both ends of the system. The tests confirmed that the remote can be altered from the web interface and vice versa. These results were confirmed by checking the database directly. The final test was confirmed by setting up a schedule and observing the remote send infrared signals on its

own, at the given time intervals. This confirmed that all elements of the project were working as intended.

1.0 Introduction

Remote controls are a common tool used by many people in their daily lives to operate various devices. A problem is that individuals who are impaired struggle to operate the modern universal remote. In addition to providing accessible hardware to those who may be impaired, an intuitive programming interface can ensure accessibility for both users and caregivers.

The purpose of this report is to outline the findings and decisions that went into the development of the Accessible Universal Remote. This report will initially go over the background of the project and a brief description of some of the higher-level concepts. Following the background, the report will go over the design and implementation for each of the major components of the project. These components include the hardware design for the remote, the software running on the remote, and the web interface. To conclude this report the project will be reviewed wholistically, and the design specifications will be measured and tested. The success or failure of this project will be determined by the ability to meet the specifications described further in this introduction.

The main objective of the Accessible Universal Remote is to offer a remote that is easier to use for individuals who may be impaired. The solution is to offer a simple user interface in the form of large easy to use

buttons on the remote itself while allowing the user or caregiver to easily program the remote via web interface. The remote needs to be able to communicate with multiple devices using infrared. It also needs to be able to be programmed through an infrared receiver on the remote. The web interface allows for programming of macros as well as setting a schedule to automatically perform macros at user-specified times.

The background of the project will be provided in section 2, including the inspiration for and inception of the project. The third section will be dedicated to the overall design of the project including the hardware, firmware and web interface. Implementation of the design will be covered in the fourth section. The final body section will be used to outline the testing and overall outcome of the project.

2.0 Background

The main inspiration for this project comes from an unfinished Hackathon project from January 2023. A hackathon is an event where programmers, designers, and other creative minds come together to collaborate on software projects over a short period. After a long discussion with a family member, Brandon Carpenter was driven to develop an accessible TV remote since the current state of TV remotes limits individuals with impairments to turn on a television, let alone change from HDMI 1 to HDMI 2. This appeared to be a quite viable solution.

Although the initial prototype from the Hackathon was not complete, after tinkering around, Brandon was able to get a simple prototype working at sending and receiving infrared remote data and storing that in memory. This remote was limited in its ability without an internet connection or scheduling features, as well as limited space due to the micro controller being used.

In the fall of 2023, during one of the embedded sessions, one of the instructors, Carlos Estay-Oyarzo, inspired Cole Yaremko and Brandon to experiment with bare metal STM32 programming. Bare metal programming refers to writing code directly for hardware without the need for an operating system or additional software layers. This approach grants developers' greater control and efficiency over the hardware. Due to Carlos's heavy influence, a lot of the STM32 direction was determined by this push.

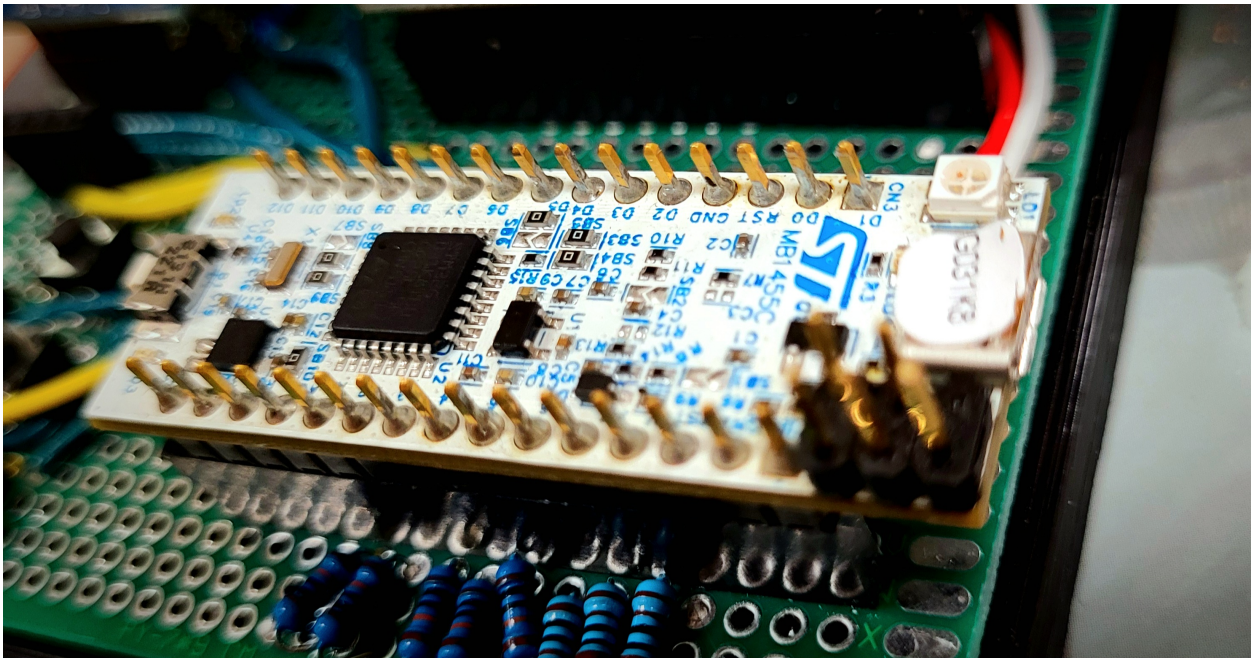
3.0 Design

3.1 Mechanical Design

3.1.1 Micro controllers

When developing the original Hackathon prototypes, one of the hiccups encountered was that the original micro had limited SRAM and limited flash memory, not enough to allow for the program to function. Already possessing skills in bare metal STM32, it was an easy choice to design around it due to the larger flash and SRAM on the device. The STM32G032K8TU itself contains "64 Kbytes of Flash memory and 8 Kbytes of SRAM" (UM2591, 2019, pg 2). One of the testing boards is shown below (figure 1).

Figure 1: Nucleo32G031



Source: Brandon Carpenter

The one limitation of the STM32 is that this board does not support WIFI or Bluetooth, hence the second choice of the ESP8266. This was chosen since it has built-in internet connectivity, also because it has support for MicroPython. This allows the project to encapsulate both bare metal C and MicroPython. According to Kevin Moore, the industry is starting to use micro python in rapid development micro projects.

After some testing and deliberations with MicroPython, it was discovered that the ESP8288 does not have a proper SSL certificate that can be supported with the web server of the project. In the final weeks of development, due to compatibility issues, the internet-connected microcontroller was swapped from an ESP8266 to an ESP32. While the ESP32 and ESP8266 share many similarities, this change required a slight adjustment in the code. Fortunately, transitioning the code designed for the ESP8266 to the ESP32 was a straightforward process, requiring minimal effort.

3.1.2 Mechanical

For the Easy remote, it was apparent that there would need to be at least eight buttons: one for power, two for volume, and five for custom macros. Also, there needed to be a display or some sort of way to view what was taking place. The selection of an SSD1306 OLED display was chosen as it is well-supported and documented.

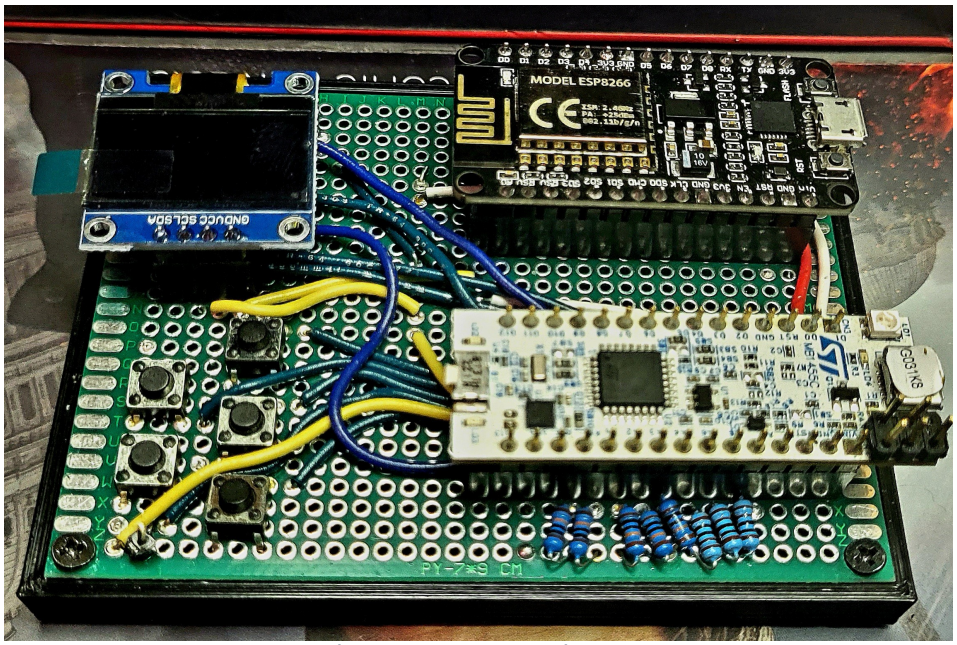
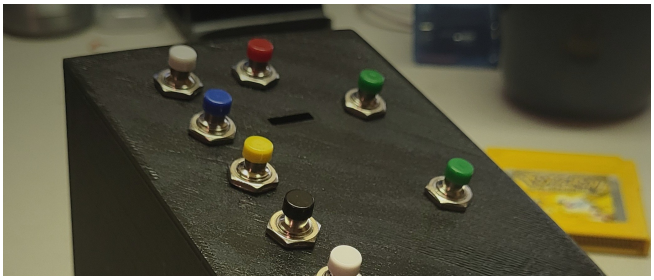


Figure 2: Proto-Boad v 1.0

The first iteration of testing and developing for the remote was this (Figure 3) proto-board. The board contained five buttons, an SSD1306 display, direct serial connection to the ESP8266, and under the screen an IR reviver and IR led for transmitting.



The first enclosure apparatus was a 3D printed square box with enough height to account for the proto-board underneath. The buttons contained in

the box were attached to a multi wire DuPont connector, that would slip over pins A0-A7 of the stm32 and connect to their respective buttons. One of the difficulties with this setup is although the buttons function properly, the when the ssd1306 was connected with wires that were closely together from the I2C breakout to the top of the shell, the I2C lanes cause noise in the parallel lines, requiring them to be separated for communication to occur.

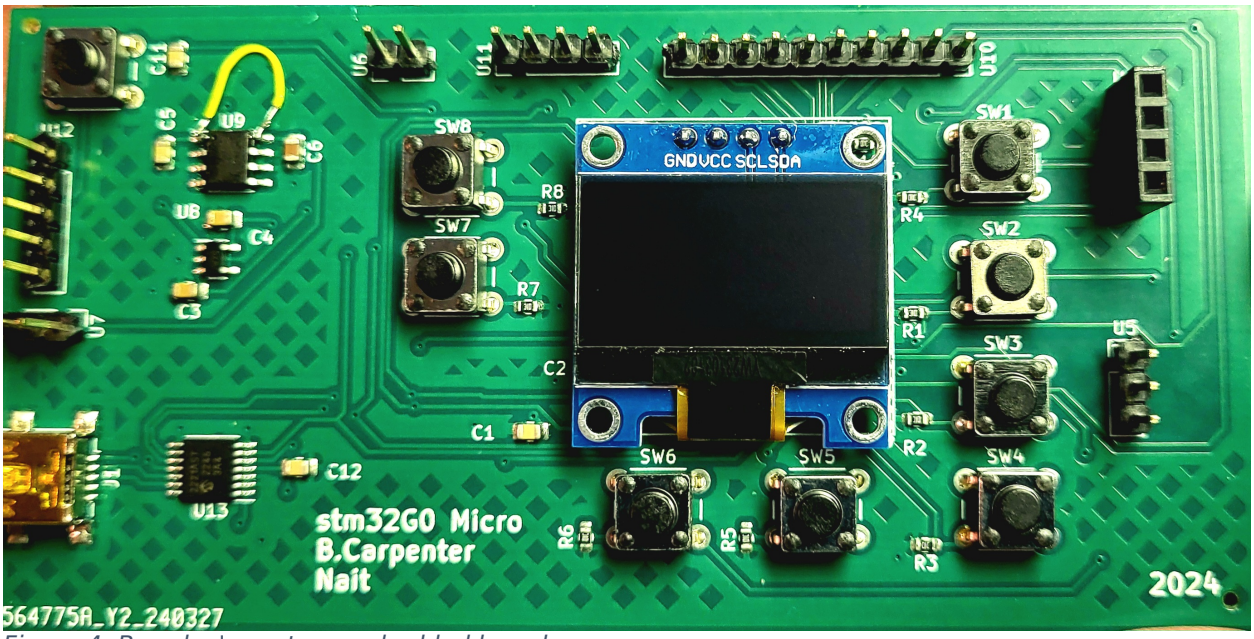


Figure 4: Brandon's custom embedded board

When the demand for a new micro arose, The ESP32, it made the current case and proto-board incompatible with the project. A new case was proto-board setup was then used, a bread-boarded ESP32 and a newly designed stm32 embed breakout board created by Brandon Carpenter (Figure 4). After connecting it to serial and developing uploading the stored macros from the database, the new proto-board setup was functioning better and faster than the original.

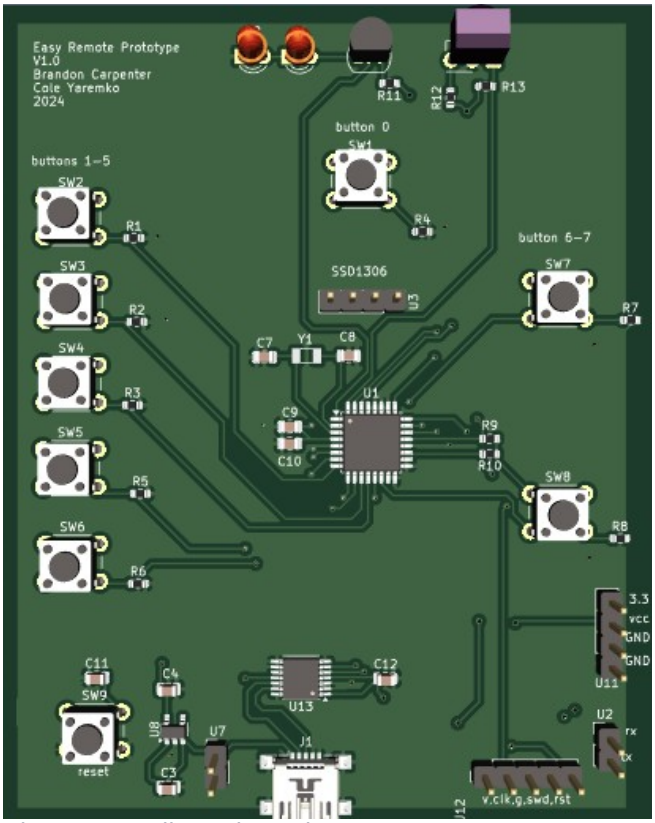


Figure 5: Dedicated IR micro

The next step in design was to create a self-contained board for the stm32 with the IR reviver, transmitter and interface properly organized in one board (see Figure 5).

3.2 Software Design

3.2.1 Embedded Firmware

Easy Remote requires two distinct firmware, one for handling user input and Infrared and another to deal with communicating with the web service. On the STM32 the firmware is built from the ground up using bare metal C to handle all user input, display the current status, and send and receive IR signals using the IRMP library. The ESP32 uses MicroPython and

connects to the database and gets all current macros and schedules stored in the database and sends them over serial to the STM32.

IRMP is a library that can handle sending and receiving up to 46 different protocols. When a micro receives an IR signal it is stored in a struct (IRMP 2020) (see Figure 6). This struct is then used to send and receive all IR data.

Figure 6: IRMP_DATA struct

IRMP_DATA	
Uint8_t	Protocol
Uint16_t	Address
Uint16_t	Command
Uint8_t	Flags

In the Easy Remote each button stores a macro made of multiple IRMP_DATA structs. To store multiple, another struct, IRMP_MACRO is used (see Figure 7). This stores a macro number or identifier, a count of how many IRMP_DATA structs are stored, and an array of IRMP_DATA structs.

Figure 7: IRMP_MACRO struct

IRMP_MACRO	
char	macro_num
char	data_count
IRMP_DATA	signals[4]

Figure 8: Packet structure

Description	Type	Data	End
Clear a macro to be updated and then store the count and identifier in the struct.	“U” Update	Identifier, count	\r
Store a IRMP_DATA to the struct being updated.	“I” IRMP	Proc, Address, Command	‘\r’
Update the current struct.	“W” Write	NA	‘\r’
Send the corresponding macro	“S” Send	macro_num	‘\r’

Once one of the micros would like to communicate it uses a packet system between them to send, store, clear, and activate macros. (see figure 8) The packet system begins with and Identifier, either S,I,U,or W. After the identifier the data part of the packet is sent until a ‘/r’ is received to trigger the end of the packet.

On boot or reset the ESP32 pulls all macro information for the remote from the web service, where it is then sent over serial to the STM32 and stored in flash memory. After pulling the macros the ESP32 then pulls the schedule data storing it in a class that every 5 minutes checks if there is a

macro scheduled to send it will send “S” + macro_num +’\r’ to tell the stm32 to send the corresponding macro.

The STM32 can send macros by receiving the send command but also by pressing the corresponding button on the proto-board. Once a button is pressed it will activate that macro. If a button is held for five seconds after a macro is finished sending, it will begin programming mode.

In programming mode, the STM32 waits to receive either four IRMP_DATA signals from a remote, or until the button held to program it is pressed again. After programming, the micro stores the new IRMP_MACRO into flash memory and send it over UART to the ESP32 to update the database.

3.2.2 Web Interface Design

To give users more control over the remote, the team decided to design a web interface to be used alongside the remote itself. The web interface was developed to allow users a secondary way to program the remote as well as create schedules to automatically change channels based on a set time. This interface consists of two main parts, the main React app and the server-side program.

3.2.2.1 React App Design

The React app is the main client-side program that will be directly used by the user. The interface was developed using React which allows for the usage of HTML elements “directly in JavaScript code” (Hutsulyak, 2022).

React uses its own syntax known as JSX which incorporates this combination of HTML and JavaScript. React had to be used in conjunction with cascading style sheets (CSS) to style the web interface. The React app was deployed to GitHub pages which allows users to host a single website for free. To deploy the app a step-by-step tutorial outlining how to create the app and get it running on GitHub pages was used (gitname, n.d.). The web page was designed to exist on a single page that changes components based on the state of the user's inputs. Axios, "a promise-based HTTP Client for node.js" (Axios, n.d.) was used to retrieve information from the database and update the interface dynamically.

Upon loading onto the web page, the user is faced with a simple login / sign up screen. Through an Axios post call, new users will be entered into the database and be automatically logged in. For existing users their entered username and password will be checked as well using Axios to see if their credentials match what's in the database. Once the user is logged in, this signifies a state change in the application, which will automatically update the UI to the main app without needing a reload.

Once the user is in the main application, they can register a remote, which links an individual device to that specific user. Once a device has been registered, users can navigate to the button mapping component. This component will initially make an Axios post call to get all the remotes that belong to the user, each macro that belongs to the current remote, as well as every infrared protocol stored in the database. Using this data the

application will populate HTML select elements. These elements will serve as drop down menus so users can pick which remote they are currently programming, and then alter the macros belonging to that remote.

The final section of the main React page is the scheduling component. This component is split into two smaller components. One allows the user to create new schedules, this will make an Axios call to the database to pull all the macros for a given remote. These macros will be used to populate more drop-down selects, and each one will be associated with a time input. The user can specify between one and ten channels that will be used by the remote to send macros throughout the day. Once the user has set the schedule, they may click the “Create Schedule" button to store their schedule in the database as well as set the schedule as their current set schedule. The other component associated with scheduling allows users to look at and edit their existing schedules. On loading into this component, it has similar controls to the create schedule component, however an additional drop-down select is needed to change between the user’s schedules. Depending on which remote is selected, an Axios call is made which will load all the stored data for the schedule. This component has three buttons at the bottom of the screen, one to update the schedule if the user has made any changes. One to delete the schedule outright, and another to set the schedule as the user’s currently selected schedule. Each of these buttons are linked to an Axios call that will be used to change the database.

Since React doesn't support PHP sessions for retaining user data across different components, a module called "UserProfile" was created. This module exclusively handles storing user data without relying on global variables and isn't involved in rendering HTML to the webpage. Its primary purpose is to persist user data beyond the initial login, as it's essential for making all the Axios calls on the webpage.

The development of a React application with its seamless integration of HTML and JavaScript through JSX allowed for the creation of a user-friendly interface. Deployment on GitHub pages provided a cost-effective solution for hosting the application. The application’s dynamic nature, facilitated by Axios for database interactions, ensures real-time updates and responsiveness to user inputs. With the front-end architecture in place, a database was required to support both the application as well as the remote itself.

3.2.2.2 Web Back-End Design

Given the substantial amount of data required for this project, the team recognized the necessity of implementing a database to facilitate efficient data management and retrieval. Consequently, an SQL database was chosen and hosted on NAIT's Thor server, utilizing PHP MyAdmin for administration. Leveraging these tools provided by NAIT not only streamlined development but also offered a cost-effective solution for database management. In the end a total of nine tables were implemented to store the data as seen in the figure below.

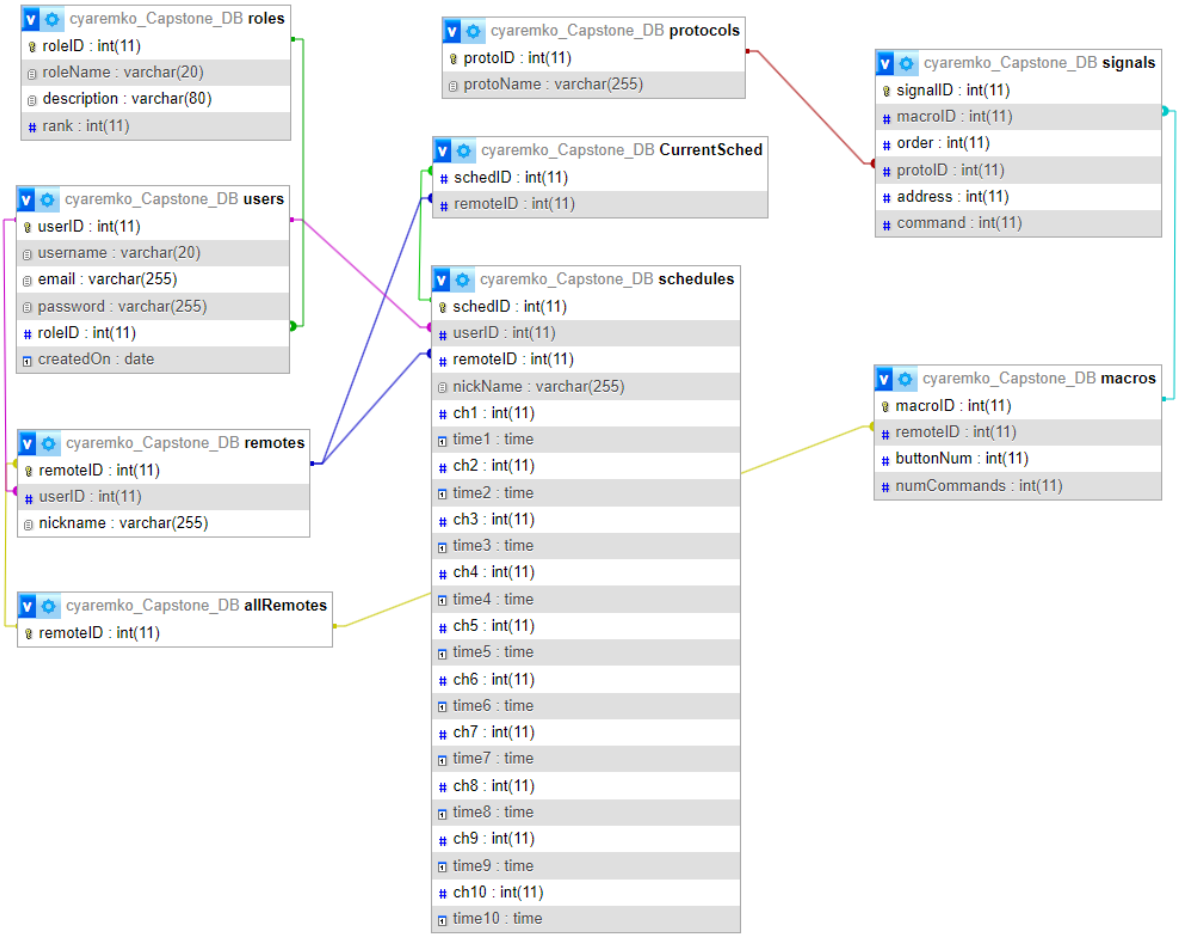


Figure 9: Database Schema

The initial database setup began with the creation of two key tables: “roles” and “users” as seen in the top left of figure 9. The “roles” table was designed to distinguish between different user types, allowing for tailored access permissions across various sections of the web page based on the assigned role. Meanwhile, the “users” table is used to store essential user information necessary for logging onto the webpage. Upon user registration, a new entry was generated in the “users” table, with the password being hashed for enhanced security measures. As explained by Hoffman (n.d.), password hashing is a crucial security measure that converts passwords into

a seemingly random string of characters using encryption algorithms, thereby safeguarding user credentials, especially in the event of a security breach.

The next tables added were “allRemotes” and “remotes.” The “allRemotes” table serves to store the ID of each hypothetical remote. This table was needed to allow users to have multiple remotes registered under the same account. The “remotes” table serves as a link between the physical remote and the user who owns it. Entries into this table are created automatically as users register their remotes through the webpage.

Following the creation of tables for remotes, tables for storing data used in the IRMP library were needed. The “protocols” table simply stores each of the 46 infrared protocols given by the library and assigns them to an ID number. The “signals” table stores data that represents a singular button press on a separate third party remote. The “macros” table represents a macro of up to four separate signals that can be sent by a press of a button on the universal remote. These tables can be updated either from the webpage or from the user’s universal remote directly.

The final two tables added relate to the scheduling portion of the project. The “schedules” table relates to a singular schedule for a given remote, the user may have multiple schedules for each remote if they choose. The schedule has slots for up to ten channels and ten times, each channel refers to a macro or button on the physical remote. The

“CurrentSched” table stores which schedule the user has selected currently and references the schedule that will be pulled by the remote on startup. These nine tables allowed the team to store all the necessary data needed to implement the web side of the project. To link the database with the React application and remote a web service was required.

The web service utilized in this project was written in PHP. Each individual component of the React application as well as the remote received their own PHP script to hold functions pertaining to their operation. Once connection was established to the database using PHPs built in mysqli object, the React app would send JSON to the web service using Axios. Likewise, the remote would send JSON using the requests module. A function from the script is selected based on what value is supplied to the “action” key in the JSON object. Each function makes a call to a stored procedure in the database that uses SQL to retrieve or alter the data necessary for whatever operation is being executed. Using this PHP web service along with the SQL database allowed for all the data storage, retrieval, and alteration that was necessary for the project.

4.0 Implementation

4.1 Embedded Implementation

4.1.1 Bare Metal STM32

To get the STM32 functioning, the following libraries were required:

- GPIO

- Clock
- UART
- I2C
- Timer
- PWM
- SSD1306(OLED)

The development of the bare metal libraries took much of the development time for the embedded side of the project.

The bare metal library that took the longest was I2C. Since the project's only peripheral was the SSD1306 the only way to test I2C was to have both a functioning I2C library and a working SSD1306 library. To

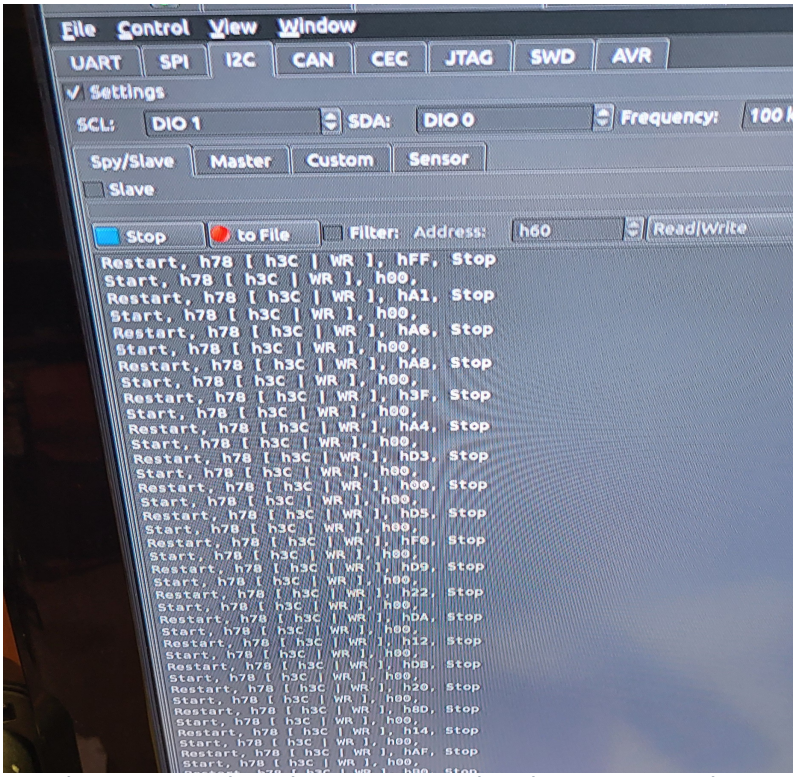
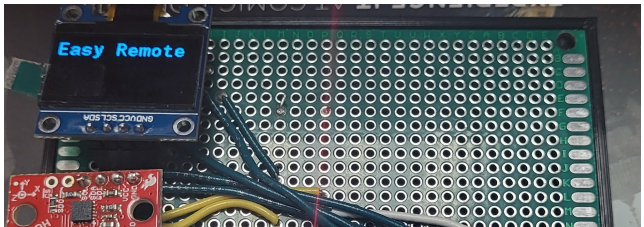


Figure 10: Analog Discovery 2 scoping the I2C connection

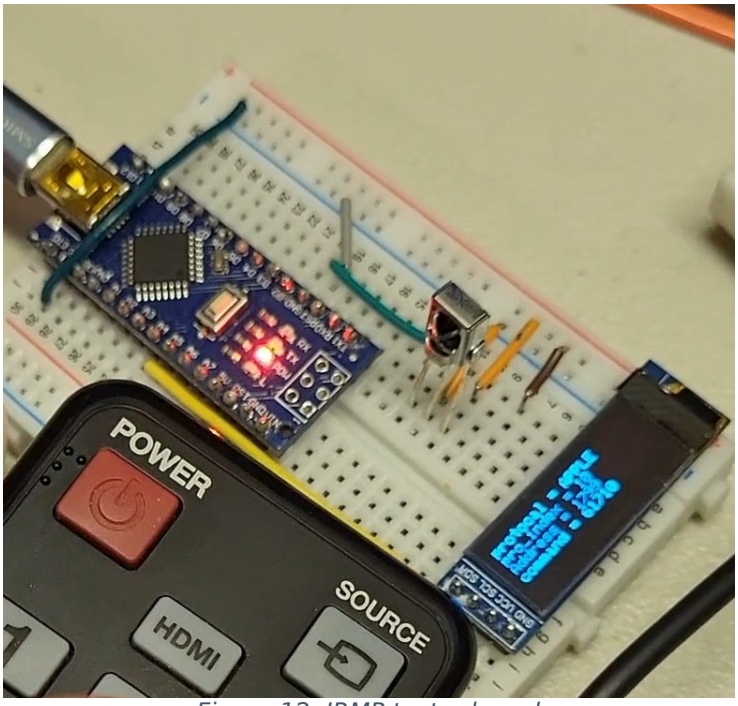
implement the SSD1306(OLED) there was already a pre-written library by Afiskon (STM32 SSD1306, 2024) designed for the ST-hall libraries. Once the library was adapted to use the bare metal



I2C implementation instead of the hall it was inhibited by the I2C library not functioning properly. After extensive testing with the Analog Discovery 2, (Figure 11) the library started to send acknowledge bits, and with further trial and error the screen started to function properly (Figure 10)

After all the STM32 bare metal libraries were written, the next step in development was to implement the IRMP (Infrared multiple protocols) library into the STM32. This took a long time as there were around 12000 lines of code that needed to be examined until there was a working understanding of what it was doing.

IRMP works in two modes, sending and receiving. In receiving mode, the library requires that there is an ongoing ISR (interrupt service routine) running 15,000 times a second. Each time that the ISR is triggered it checks the state of the pin connected with the receiver and then proceeds to compare it against 46 other models to determine what protocol might be in use. After an IRMP_DATA struct has been identified, it will return that to the user to store or use.



Sending mode works in a similar way. It also runs an ISR at 15,000 times a second, but instead of reading a pin, it compares it’s time frame to that of the protocol and IRMP_DATA struct to determine whether to set the sending pin high or low.

To test the functionality of the prototype, a second board was developed using the Arduino language and an Elegoo nano as well as the IRMP library to determine if the bare metal implementation was functioning properly (Figure 12).

4.1.2 MicroPython and Packets

During the initial stages of developing for the ESP8266, one of the primary tasks involved ensuring its capability to effectively send and receive packets. In the C programming language, variables are essentially ordered structures of binary data, allowing them to be copied and moved around seamlessly.

In contrast, MicroPython is inherently type-safe in all operations. However, one challenge arises from Python's stricter type safety compared to C—variables' types can change without triggering any alerts. This posed a hurdle when exchanging packets between C and Python. In MicroPython, each received packet needed to be converted into a byte array. Additionally, any ASCII values within the packet had to be converted separately, as values in the lower range (< 13) could cause errors during ASCII conversion.

Moreover, sending packets from MicroPython required even more conversion and encoding compared to C. These additional steps were necessary to perform tasks that are relatively trivial in C programming.

After Packets, the next step was to communicate with the database. To communicate with the database the micro needed to send get and post calls to a PHP page that would use stored procedures to update or get any macro information. While trying to connect to the PHP page it became apparent that the ESP8266 does not contain a compatible SSL certificate in the latest version of micro python to connect to Thor. Due to this inability, the ESP8266 was replaced with the ESP32. The ESP32, after some compatibility fixes to the code, was able to pull and update from the database.

During the development of the connection between the ESP32 and the PHP web service, a challenge arose where the data from the ESP's POST request wasn't appearing in the PHP $_POST superglobal. To circumvent this issue, a workaround was implemented, as depicted in Figure 13 below.

```
$data = json_decode(json_decode(file_get_contents('php://input'), true), true);
```

Figure 13: $_POST work around.

Initially, a single json_decode function call was used to decode the incoming data. However, upon reviewing the data in the PHP error log, it was observed that the data remained encoded as a JSON string. As a result, a second json_decode call was added. Following this adjustment, the ESP successfully communicated with the database without encountering any further issues.

4.2 Web Implementation

To implement the React web application a single page design was utilized. This single page, shown below, would have different components appear depending on the state of the application.

```
return (
  <div>
    {isLoggedIn ? (<Main />) : (<LoginSignup onSuccess={handleSuccess} />)}
  </div>
);
```

Figure 14: Main React App

The code shown in figure 14 shows that depending on the state of “isLoggedIn” either the Main component or the LoginSignup component will be visible. Each of these components can also contain other components in a nesting structure, which was utilized to create the main application. To pull data from the database, Axios post calls were used. An example Axios post is shown below.

```
// Function to make an ajax call to get all the users remotes from the db
useEffect(() => {
    axios.post('https://thor.cnt.sast.ca/~cyaremko/CapStone/buttonLogic.php', {
        action: 'remote',
        userID: UserProfile.getUserId()
    })
    // success function for using remote data
    .then(function (response) {
        console.log('Successfully pulled remote data: ');
        console.log(response);

        if (response.data && response.data.error) {
            // No error returned
        } else {
            setRemotes(response.data);
            UpdateData(response.data[0].remoteID);
        }
    })
    // error function - couldn't reach web service
    .catch(function (error) {
        console.log(error);
    });
}, []);
```

Figure 15: Axios Example

Axios posts consist of three parts, post, then, and catch. The post contains the URL pertaining to this call as well as the data that will be sent over to the web service. The data is automatically converted to JSON before being sent for ease of use. The then function dictates what will happen if the website

successfully gets a response back from the webservice. In this case, printing the response to the console, checking if the JSON returned had the property “error” if it does not, states are being updated with the response data. The catch portion will only happen if there was an error interacting with the web service. Multiple of these Axios post requests were needed for each component, depending on what data needed to be retrieved or updated. The PHP webservice would then check what the “action” is set to and call the associated function. One such function is shown in figure 16.

```php
// function to get all protocols from the DB
function GetProtocols()
{
    global $mysql_connection; //comes from util

    // put the data from the query into an array
    $protos = array();
    if ($results = mysqlQuery("call GetProtocols()"))
    {
        //we have valid data coming back
        while ($row = $results->fetch_assoc())
        {
            $newrow["protoID"] = $row["protoID"];
            $newrow["protoName"] = $row['protoName'];
            array_push($protos,$newrow);
        }
    }
    else
    {
        return $response['error'] = "Something went wrong, please try again later.";
    }

    // return the protocols
    return json_encode($protos);
}
```

Figure 16: PHP function

These functions typically make a call to a stored procedure within the database, and then return the data from this stored procedure, to be used by

```
BEGIN
    DECLARE schedule_count INT;

    -- Check if a schedule exists for the given remoteID
    SELECT COUNT(*) INTO schedule_count
    FROM CurrentSched
    WHERE remoteID = rID;

    IF schedule_count = 0 THEN
        -- If no schedule exists, insert a new row
        INSERT INTO CurrentSched (remoteID, schedID)
        VALUES (rID, sID);
    ELSE
        -- If a schedule exists, update the existing row
        UPDATE CurrentSched
        SET schedID = sID
        WHERE remoteID = rID;
    END IF;
END
```

the web interface. These stored procedures are created using SQL through the PHP MyAdmin interface. Figure 17 shows an example of a stored procedure used for the scheduling component.

Figure 17: SQL stored procedure

This procedure first selects the count of the schedules for the given remote into a stored variable. The variable is then checked to see if the current set schedule needs to be altered or created entirely. Using each of these different methods a single cohesive web interface was implemented.

5.0 Project Results

Overall, the project was a success and managed to meet all the

planned features. The remote was able to effectively send infrared signals to interact with two brands of televisions. This was tested by programming the remote and then lining up the infrared transmitter of the remote with the infrared receiver of the television and observing that the remote was able to make changes on the television. The remote was able to receive infrared signals to reprogram it. This was tested by setting the remote into programming mode, holding up a third-party remote to the infrared receiver on the universal remote and pressing several buttons to be repeated back. The LCD on the universal remote updated with the correct information and was able to mimic the signals back. The remote was able to store macros to the cloud effectively. This was tested by allowing the remote to attempt to sync its currently held macros with the database. Once the sync finished, both the webpage and database itself were refreshed and the data was observed to match that which was on the physical remote. The remote was also able to then retrieve macros and signals from the database. To test this the web interface was used to make changes to the existing macros and signals for the remote. The remote was then reset so it would grab the latest set of macros from the database. Once the reset occurred, multiple buttons were tested to see if the changes on the website were reflected in the physical remote. On inspection of the LCD screen, it was confirmed that the changes went through to the remote. The feature to create schedules that would activate the infrared macros at different times was a success. This was tested by setting a schedule from the web page that would trigger a different

macro every five minutes. Upon resetting the remote so it would pull the latest schedule it was observed that the correct macro was sent every five minutes as expected. Overall, through testing the team was able to demonstrate all the remote’s planned features.

6.0 Conclusion

The modern remote ecosystem is not currently accessible to those with limited motor function or without technological expertise from gaining full access to their televisions or IR devices. The goal of the Easy Remote Project was to explore the ability to create a remote that could be used in conjunction with a care provider, to create a personalized remote that is accessible to everyone.

Key Achievements:

1. **Bare metal STM32 development**: The team's choice to utilize bare metal C contributed to an overall efficiency of the project and deeper knowledge of the processes of processing and sending IR data.
2. **Dual micro communication**: By using a custom packet system the team was able to design and leverage faster speeds in sending and receiving data.
3. **Web Interface**: Through the use of React the team was able to create a user-friendly web interface that can adapt to multiple devices.
4. **Database Management**: SQL databases allowed the team to create stored procedures that allowed the group to create easily accessible updates and access the data stored.

To Verify the functionality of the device **several Key areas were tested**:

1. Ability to **send and receive IR macros:** *verified*
2. Ability to **update and store macros in a database*:*** *verified*
3. Ability to **activate macros on a schedule*:*** *verified*

After seeing successful verification tests with the ability to program macros or IR signals containing an individual's favorite channel numbers, as well as being able to turn on multiple devices with one button press; the easy remote removes the complexity of having confusion on what a remote does or how to turn on an individual's devices. The schedule test proves that the

remote can now be set on a timer so that those who cannot even use a remote without the ability to function it.

Currently the device is just a prototype that is not user friendly, but the functionality proves that such a device can be manufactured that makes remotes access-able to those who cannot operate one.

A special thanks to Kevin Moore, Kelly Shepherd, Carlos Estay-Oyarzo, and Steven Dytiuk; for their mentorship, guidance, advice, and encouragement on this development. Another thanks to our classmates who encouraged, joked and spurred us on throughout the process. Without these men and women, the project would not be what it is.

We look forward to future development of this project and exploration of technology.

Sincerely,

Brandon Carpenter, and Cole Yaremko

References

Afiskon. (2024). *STM32 SSD130*. GitHub. https://github.com/afiskon/stm32-ssd1306

Axios (n.d.) *Getting Started*. Axios. https://axios-http.com/docs/intro

Gitname (n.d.) *react-gh-pages*. GitHub. https://github.com/gitname/react-gh-pages

Hoffman, B. (n.d.). *How do passwords work?* Delinea. https://delinea.com/blog/how-do-passwords-work#:~:text=Password%20hashing%20turns%20your%20password

Hutsulyak, O. (2023). *10 Key Reasons Why You Should Use React for Web Development*. TechMagic. https://www.techmagic.co/blog/why-we-use-react-js-in-the-development/

IRMP. (2022). *IRMP*. GitHub. https://github.com/IRMP-org/IRMP

STMicroelectronics (2019). UM2591: User manual STM32G0 Nucleo-32 board (MB1455). https://www.st.com/content/ccc/resource/technical/document/user_manual/group1/b5/f7/d3/3c/11/82/44/a8/DM00622380/files/DM00622380.pdf/jcr:content/translations/en.DM00622380.pdf

STMicroelectronics (2020). RM0444: Reference manual STM32G0x1 advanced Arm®-based 32-bit MCUs. https://www.st.com/content/ccc/resource/technical/document/reference_manual/group0/2f/21/cb/33/78/80/42/64/DM00371828/files/DM00371828.pdf/jcr:content/translations/en.DM00371828.pdf

Verification Test

Project: Accessible Universal Television Remote

Team: Brandon Carpenter, Cole Yaremko

Date: Feb. 5, 2024

1. Can effectively send IR signals to interact with a TV or other IR device through a button press using at least two IR protocols, from one main micro-controller.

Test	Press the on button and observe if a TV and second IR device turn on from this one press. Press subsequent buttons and observe if they, can control the two devices. Record the distance and angles the devices can communicate at.
Results	*To be determined*
Conclusion	*To be determined*

2. Can receive IR signals to reprogram the remote, through an infrared sensor interpreted by the micro-controller.

Test	Set the remote into Programming mode. Place another pre-existing remote in front of the IR receiver. Press a combination of buttons on the remote to be mimicked for the device to send back to the interface. Check if the display updates properly. Press the newly programmed button on the easy-remote and observe if it sends the same macro when pressed. (this may be limited by the number of key presses it can mimic) Record the distance and angle at which this programming can occur at.
Results	*To be determined*
Conclusion	*To be determined*

3. Can store macros of IR remote signal sequences (example: the number sequence 4,7,5) in the cloud to be recalled later.

Test	After a system sync has occurred, refresh the web page to see if the data has updated in the database and subsequently displayed properly on the web page.
Results	*To be determined*
Conclusion	*To be determined*

4. The internet-connected microcontroller can retrieve IR macros from a database and transfer them to the main IR-transmitting microcontroller via serial communication.

Test	On the web interface, swap the button layout, and after a system sync check if the button and display respond accordingly on the remote.
Results	*To be determined*
Conclusion	*To be determined*

5. The user can set a schedule from a website that can communicate with the internet connected micro for when the device is to send certain IR signals. When the user creates a new schedule, it will be stored in the database to be swapped as needed.

Test	Set up a schedule to change the channel every minute and sync the systems. Wait and observe if the appropriate IR signal is sent when the real time clock on the IR sending micro reaches the appointed times.
Results	*To be determined*
Conclusion	*To be determined*