Motor Impairment Keyboard

MIKey Final Report

Jackson Hagood

Andrew Imwalle

Brittany Jenkins

Connie Liu

Abhishek More

Max Smith

Department of Computer Science

Texas A&M University

12/5/2022

Table of Contents

1 Executive summary

Those who suffer from Parkinson's are using computers more and more as younger generations grow older. Traditional keyboard offerings pose a challenge for these users as the keys are often small and difficult to press. Furthermore, keyboards for desktop use do not implement predictive features like auto-complete, something which could cut down on the number of keystrokes that those with motor function impairments must press (and therefore improving their typing experience).

MIKey sought to solve this problem by developing a keyboard for those with motor function impairments. This keyboard aims to make the typing experience for those with Parkinson's and other motor function impairments better by designing a keyboard for them. This solution was designed to be affordable and accessible with the core function to make typing easier and quicker. It was also determined that the best design should be compatible with any platform (Windows, Mac, etc.) and easy to adapt to a variety of use cases.

With these objectives in mind, MIKey began the design process. It was determined that the keyboard would be a custom, over-sized "QWERTY" layout. The keycaps would be physically larger than those of a normal keyboard and they would be oriented in a grid to be more easily located. The keyboard would also have three displays that showed auto-complete suggestions depending on what the user had typed. These suggestions would be determined by a Raspberry Pi which received hardware interrupts from a Raspberry Pi Pico that scans the keyboard to determine keystrokes. When the user finds a suggestion that they wish to use, three macros below the spacebar would be used to select the appropriate one, and then be output to the receiving computer. Such a design would require software, hardware, and modeling teams for successful implementation.

The keyboard was designed using an agile development methodology. Five main sprints were outlined at the beginning with a rough schedule of tasks to be completed in each. Each of the six team members were placed into at least one technical team (hardware, software, or modeling) as well as a non-technical role (manager, scribe, etc.). Despite initial assignments, several members floated to other roles as the requirements of the project changed. Development was mostly steady with tasks typically being completed on schedule. Time was intentionally left at the end for debugging and the final prototype was successfully constructed before the deadline.

The final prototype successfully implements the main design goals outlined at the beginning of the semester. The keyboard is oversized and uses auto-complete to assist the user's typing experience. Several usability tests were conducted to gauge how well certain features assisted the user. It was found that the auto-complete algorithm could save around 30 % of the total keystrokes needed to type sample passages, with the suggestions correctly predicting upwards of 90 % of the typed words. The keyboard was confirmed to work in all the ways a normal keyboard does and did not have a noticeably higher latency (one test suggested just 10 ms more latency than a standard keyboard). While user testing with actual participants that have motor function impairments could not be conducted due to IRB issues, the team is satisfied with the ability of their design to cut down on how many keystrokes must be taken. It follows that reducing the number keystrokes reduces the chance for error and increases typing speed.

2 Project background

The world is in a digital age where usage and navigation of computers has become increasingly important for the population. In 2018, the American Community Survey (ACS) found that 92 percent of households have a computer, and this number has been trending upwards [4]. Unfortunately, not everyone is able to efficiently use computers, as they are not designed for those with physical disabilities.

2.1 Needs statement

Those with tremors or motor function impairments (such as Parkinson's) experience trouble when interacting with traditional computer peripherals (namely a keyboard). Therefore, as use of technology becomes more prevalent, there exists a need for a simple assistive technology that provides better computer hardware accessibility to individuals with limited motor function. Existing assistive technologies are often too restrictive for those with mild to moderate motor function impairments (such as eye-trackers) and tend to sacrifice efficiency for the user. As a result, there is a need for a more traditional form of interaction that is easier for those with mild impairments to utilize. There also exists a need for a more affordable solution to this problem, as assistive devices can cost upwards of thousands of dollars, making them unattainable for a large section of the population.

2.2 Goal and objectives

The goal of our solution was to make the typing experience easier for those who have motor function impairments. The solution should be cost-effective, accessible, and should also improve the experience of interacting with their computer (namely typing or otherwise inputting text). Priority must be placed on ensuring the solution is easy to learn, as a large percentage of those with motor function impairments are typically older in age and likely to be less proficient with computers. Therefore, avoiding the need to install device drivers or other software is preferable. The solution should also be generic enough to be customizable depending on the specific needs of each user (like variable actuation force or form factor). The design should not compromise any of the abilities present in a normal keyboard. Ideally, a solution would add to a normal keyboard's capabilities. The aim of these capabilities should be to reduce the user's opportunity to make mistakes while typing and improve their typing speed. Another important objective is to keep the solution as cost effective as possible in order to keep the product in reach of the general user. A final objective is to follow a "helping the disabled helps all" philosophy. Many devices for those with disabilities end up helping others in unintended ways. A solution to this problem could end up having uses for a wide variety of circumstances, even for those without motor function impairments.

2.3 Design constraints and feasibility

Several constraints presented in the needs statement involve the price, functionality, and compatibility of the product. In terms of price, we aim to deliver a final product that does not exceed $200, we anticipated development of a prototype exceeding this amount. Additionally, users will ideally be able to set up and take advantage of core functionalities of the product on their

own. We also want the user to be able to use this product with any computer in a variety of settings, so it must be reasonably portable and compatible across Linux, Windows, and Mac. To do this, our solution will be based on existing technologies with the form factor of a standard USB-A, QWERTY American-English keyboard. Lastly, our development time is constrained at 10 weeks. We have determined that a majority of these constraints are feasible given our technical expertise, but our biggest challenges throughout the development of this project included the final cost of the product as well as limitations on development time.

2.4 Literature and technical survey

When analyzing the market for existing relevant technologies, three main categories are present. First, alternative forms of computer interaction are common for those with more severe motor function impairments (such as an eye tracker or a mouse stick). While these are not completely in line with the intentions of the proposed design, they are useful for considering accessibility features. More in-line with the project's goals are more traditional keyboards. These include oversized keyboards, or sunken key keyboards. The last type of relevant technology is existing auto-complete and auto-correct software which could potentially be leveraged to better a potential solution.

Those with sever motor function impairments often use devices that fall under alternative forms of computer interaction. One of the most well-known of these is an eye-tracking device. This device allows users that cannot interact directly with their hands to select on-screen items using their eyes. This device is often used by those with ALS, muscular dystrophy, or spinal cord injuries [5]. Another device under this category is a mouth stick. This device is placed into the user's mouth where they use it to blow onto on-screen items. Again, this device is intended for those who do not have use of their hands. Other devices in this category follow the theme of finding alternative avenues for users to interact with their screen, most of which make use of an on-screen keyboard. While these items do not fulfill all the objectives specified in Goals and Objectives, they do provide some insight into how devices are developed for those with disabilities. All these devices, for example, feature a relatively high level of customizability. The mouth stick, for example, can come in a plethora of shapes and sizes depending on the user's needs, and some eye trackers have modes for stationary and mobile heads. It can be surmised that when developing devices for those with motor disabilities, it is important to keep the solution customizable and expandable as all users have a specific set of circumstances.

Many with more mild motor impairments use more traditional keyboard peripherals. One of these devices is what is known as an oversized keyboard. The largest manufacturer of this device is BigKeys with nearly two decades in the market. These keyboards feature large keycaps and a wide form factor that makes it easier for the user to avoid selecting the wrong key. The users of these keyboards include children and those with motor function impairments. By grouping these categories together, however, many with motor function impairments may feel put-off by the child-oriented design. Furthermore, the keyboard is "bare-bones" (no auto-completion) but costs over $100 [1]. With a comparable, standard-sized keyboard costing as little as $10, this device feels lacking. Another type of more traditional keyboard is a sunken key keyboard. In this design, the keys are sunken into a casing with guards between them. This makes it so users who often press

neighboring keys on accident can no longer press the wrong key. This fits in well for those with Parkinson's disease. The largest issue with this keyboard, however, is its availability. Only one model is easily found online, and it costs greater than $200 [7]. Both keyboards seem like substantial solutions to the Needs Statement, but their price and availability place them out of reach for many with motor-function impairments.

The last type of relevant technology is auto-complete software. For years, auto complete has been quite universally used on smartphones to allow users to type faster. Auto-complete solves two major problems with typing on smartphones. First, the difficulty of pressing the correct key on a small keyboard is alleviated by having to type less when suggestions are chosen instead of typing the entirety of the word. The second problem is improving the speed at which users type on their smartphones, with most typing far slower on their phone than a full-sized physical keyboard. While auto-complete was not designed for those with motor function impairments, its benefits could be leveraged for these users. In the same way the average smartphone user has difficulty selecting the correct key on their phone's keyboard, many with motor function impairments have trouble typing the intended key on their physical keyboard. Furthermore, the speed improvements provided by auto-complete on a smartphone could potentially be applied to a physical keyboard experience. Therefore, it stands to reason that auto-complete could greatly assist those with motor function impairments to type on a physical keyboard.

Looking at these three categories of devices, three main lessons can be learned. First, devices designed for those with disabilities should be customizable to fit the needs of each specific user. It is therefore best to strive for a design that can be deconstructed or configured without destroying it. Second, existing keyboards for those with motor function impairments are too expensive for a lot of people. A useful design should therefore be relatively affordable for the average person. Finally, auto-complete has significant potential to improve the typing experience of those with motor-function impairments. The design proposed in this document will be less expensive, more customizable, and more advanced than existing alternative keyboards.

2.5 Evaluation of alternative solutions

One solution to the problem outlined in the Needs Statement is simply to use a large touch-screen display. This device obviously already exists and may be a good fit for some users. The large screen can enable the user to avoid typing the wrong key, and on-screen auto-complete is a feature in many such devices. This device has two problems, however. First is the price, with many touch-screen computers or displays costing thousands of dollars. The second problem is the lack of tactile response. Many with motor function impairments may benefit from tactile feedback that their keystroke has registered.

Another alternative solution to our problem is more theoretical and does not exist at the moment. This solution is essentially the same as the touchscreen solution proposed earlier, but fixes the missing feedback issue by placing multiple haptic motors underneath the screen. In this way, the keys that show up on the screen can be wholly configurable to the user's preferences while also providing the user with a better typing experience with haptic feedback. However, one drawback to this design is the cost, as well as the limitation of our team's technical experience.

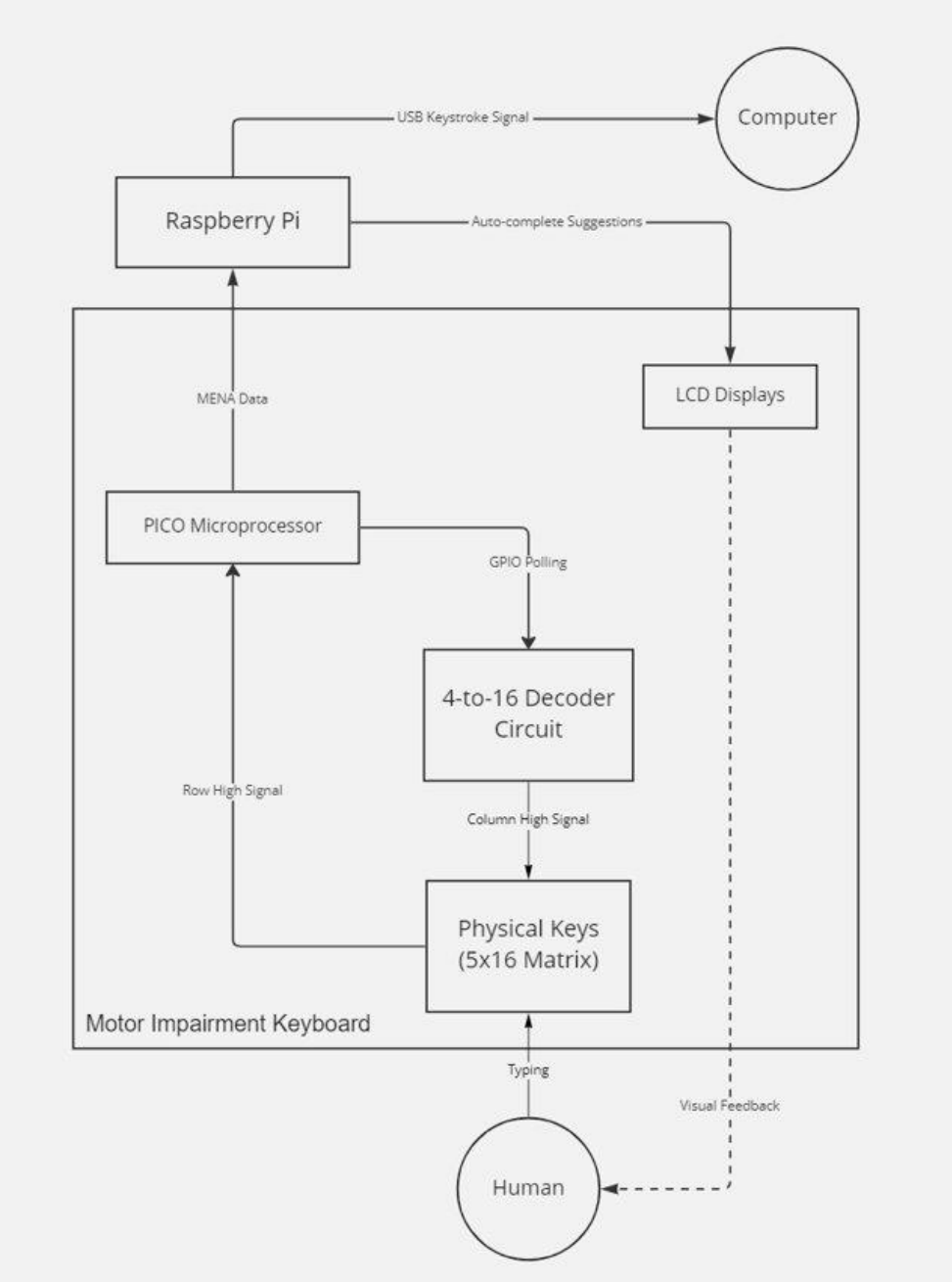
An additional solution is using an eye-tracking device alongside an on-screen keyboard, but also providing some auto-complete or text suggestion software. This would increase the user's typing efficiency compared to simply using an onscreen keyboard. The downside of this solution is the high barrier to entry. Eye trackers recommended for use with the Windows Eye Control software are sold at a minimum of $250 [2], which exceeds the budget of our audience. Additionally, the user's typing speed will still be slower compared to some of the other solutions.

An alternative option that was considered was incorporating speech to text as a feature on a physical keyboard. While this could be a good stretch goal for our project, it would not be a more efficient solution for users with some function in their hands. As described above, we are aiming to help improved computer interaction for people with mild to moderate motor function impairments. Speech to text is not a feasible solution for our target users, and we are looking to solve an issue where we see a gap in the market.

Another means of reducing the amount of typing that a motor-impaired user must do to complete a given task is to allow for customizable keyboard macros to be set. To the user, this means that a single button press can be set up to complete a complex task, such as opening a browser and navigating to a specific website. If used for multiple frequently-executed tasks, this feature has the capacity to greatly reduce the amount of typing and mouse navigation required for a user to carry out regular activities on a computer. The downside to this solution is that it would require an additional software component that users may have difficulties both installing and navigating.

3 Final design

3.1 System description



Our product comes in the form of a traditional keyboard, with some modifications that make typing easier for motor-impaired users. The user-facing hardware (the keycaps, key switches, and the external frame) is described further in section 2.3.1.

The Motor Impairment Keyboard (MIK) is surrounded by an input and output. To start off the input stage, the user will type their desired text on a physical key switch in the 5x16 Matrix. The GPIO polling from the Pico Microprocessor will utilize the 4-to-16 decoder circuit logic to

individually power each column of the key matrix. Once the column of a pressed switch has been powered, the corresponding row of the switch will be connected and detected as high input back to the Pico Microprocessor. At this point, the Pico will encode the specific keypress into its corresponding MENA representation and generate an interrupt to the Raspberry Pi. Once the Pi has received the MENA data from the Pico through its serial port, it will send the proper scan code to the computer via USB and generate the proper auto-completion suggestions to be displayed on the LCD displays. The Pi is also responsible for generating the user's desired auto-complete suggestion to be shown on the computer screen. Further details on the system components can be found in section 3.2.

3.2 Complete module-wise specifications

3.2.1 Software

Programming for the MIKey can be split into two halves: The Raspberry Pi's driver program (primarily C++) and the PICO's key scanning (Python). The two devices communicate to one another using a hardware interruption that denotes the beginning of a MENA value being sent. Otherwise, the two components are mostly isolated. MENA stands for MENA Encoding Not ASCII and is the basis for storing keystrokes in a two-byte integer (short in C++). The first four bits of MENA are used to denote whether special shortcut keys (ALT, CTRL, SHIFT, and WIN) are being pressed. Binary 1 would mean that such a key is pressed. The final byte is then dedicated to storing which "character" key is being pressed. Below are the MENA standard definitions.

MENA Structure

Bit(s)	0	1	2	3	4-7	8-15
Key	ALT	CTRL	SHIFT	WIN	(unused)	Character Key

MENA Character Key Table

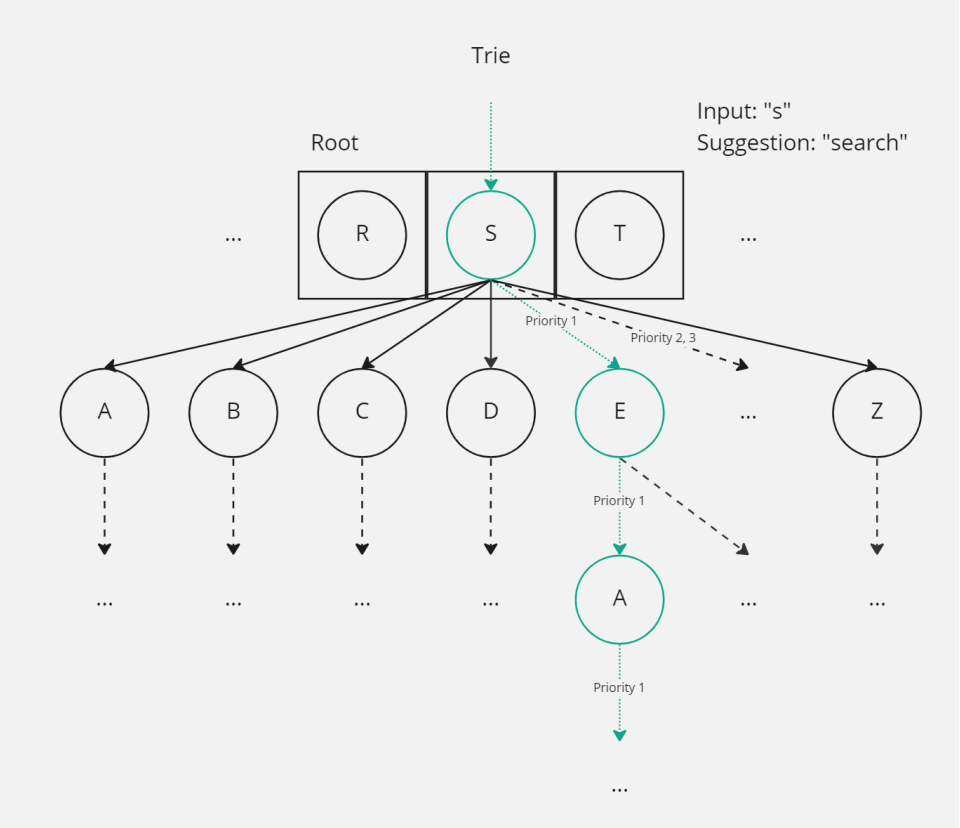
Integer Value	Character Key
0-9	0-9
10-35	a-z
36	`
37	-
38	=
39	[
40	]
41	\
42	;
43	‘
44	<

45	>
46	/
47	Space Bar
48	Tab
49	Enter
50	Backspace
51	Delete
52	Esc
53	Caps Lock
54-65	F1-F12
66	UP
67	DOWN
68	LEFT
69	RIGHT
70	Macro 1
71	Macro 2
72	Macro 3

The Raspberry Pi serves three main purposes: determining auto-complete suggestions, outputting suggestions to LCDs, and sending keystrokes to the receiving computer. These comprise most of the heavy compute power that is needed (meaning the PICO can focus on key scanning and issuing interruptions alone). The Raspberry Pi runs a bash script on startup that does two things: starts a pre-compiled C++ driver program and runs another bash script to handle input from the PICO. The driver program oversees the Raspberry Pi's main functionality. The program utilizes C++ principles in object-oriented programming and leverages several classes. The program runs an infinite while loop that runs no commands. Instead, the driver program configures GPIO 21 for hardware interrupts and runs a function when these interrupts are received. This function then takes in the sent MENA using standard in, calculates the new suggestions, and sends the received keystroke.

The auto-complete suggestions work by using a specialized data structure: a Trie. Our implementation of this data structure was purpose built for our project. On program startup, the trie is constructed by reading from a dictionary file. This dictionary file is comprised of the 10 thousand most popular English words (in priority order). The auto-complete algorithm would work if this dictionary file were changed for another. For example, if the user were a programmer and wanted C++ keywords higher in priority, they could use another dictionary file without issue. The constructor opens the dictionary file to build the trie. The trie is denoted by 26 root nodes. Nodes in the trie are representations of a character that exists in at least one word in the dictionary. These 26 root nodes are for the beginning letters of every word in the dictionary. Each node also has

space for 26 pointers to other letters in words. For example, the word “the” is found by going to the root node for “t” and then going to that node’s “h” child and finally that next node’s “e” child. Each node keeps track of these 26 child pointers in alphabetic order. Another component of each node is a boolean to keep track of whether or not a potential word can end here. For example, the previously discussed “e” node (for “the”) would have this boolean set to true. The final part of the node objects is an array of the four highest priority children. These are what allow the auto-complete algorithm to find the most relevant words. Outside of the constructor, a get-candidates function is the other main major component, in charge of traversing the created Trie using a partial word to find suggestions of the highest priority. Below is an example of how the Trie is traversed.



The driver program leverages this auto-complete algorithm by maintaining a string representing the partial word the user has typed and running the get-candidates function every time it is changed. It then stores the three resulting suggestions. These suggestions are output to the LCD’s over GPIO (using Wiring Pi, a C++ library). This is done using a class written for interacting with the three LCD’s using the serial interface on the Raspberry Pi.

After the auto-complete suggestions are determined, the driver program sends the keystroke to the computer. If the keystroke is a macro, then the selected auto-complete suggestion is sent instead. To send the keystrokes, Key Mime Pi is used. This allows shell commands to send keystrokes to whatever device the Raspberry Pi is plugged into. To do this within the driver program, the C++ program forks and executes shell commands. For each keystroke two commands are sent: one to send the keystroke, and one to send the signal for it being released. Logic has been implemented to convert MENA values into Key Mime Pi commands.

The Raspberry Pi Pico is responsible for detecting users' key presses, translating both individual key presses and combinations of key presses into the associated MENA value, generating an interrupt through GPIO to notify the Pi that there was a key press, and transmitting the MENA value to the Pi. The CircuitPython `code.py` contains all of this functionality.

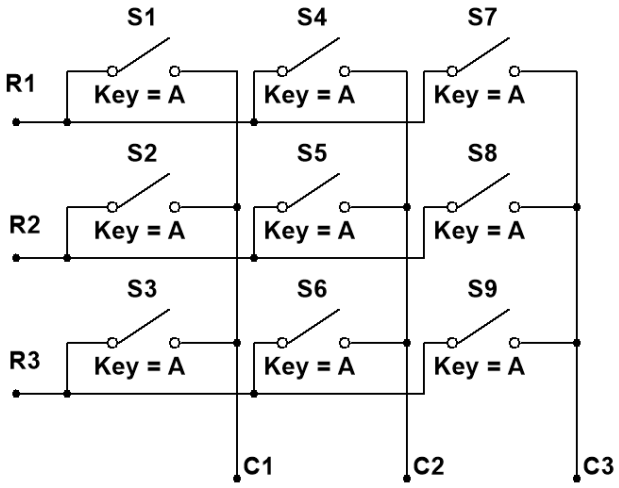
In order to detect key presses, the Pico iterates through the entire key matrix, outputting a high signal to each individual column and checking to see whether any of the rows are powered. If a row is read as powered, then the key located at that row and the column receiving power from the Pico is being pressed.

The Pico contains a mapping from locations in the key matrix to the associated MENA value, allowing for easy retrieval once a key press is detected. Using bit masking, a MENA encoding is generated for the current keyboard state. If there is a non-null encoding, an interrupt is generated by the Pico and the MENA data is sent over the serial connection to the Pi.

3.2.2 Hardware

A custom layout for the keys has been created by the team to ensure that only the necessary keys would be present in the keyboard. This layout is a slightly modified QWERTY layout to ensure that the user is familiar with the positioning of the keys. The keyboard and individual keys will be oversized to assist with accuracy. The keys will be sunken into the keyboard housing, similar to a sunken keyboard, also to assist with accuracy and speed. The keycaps will be a slim design as to not make the keyboard too tall, which is an aesthetic as well as functional decision. The three macros placed underneath the spacebar is to allow the user to select from the auto complete options.

The LCD display will be built into the keyboard and will receive GPIO signal input from the microprocessor. It will display up to three suggestions that that the user can select. As the user continues to type, the LCD will update. The font will be large and high contrast to ensure that the user can read the suggestions clearly.



In order to retrieve key presses from the keyboard, MIK will use a key scanning matrix, as shown above. This circuitry will be driven by a secondary microprocessor that will generate interrupts to send to the primary processing unit, along with a MENA code. In order read key presses, the secondary microprocessor will continuously poll the hardware by individually powering each row of the matrix and checking each column to see if the switch at that specific row/column coordinate has been closed. If a high signal has been detected at a specific row, the row and column are used to index a mapping from the key scanning matrix location to a MENA encoding. As mentioned above, an interrupt is then generated and the MENA code is sent to the primary processing unit to be handled. In order to reduce the number of GPIO pins used, a decoder will likely be used for the powering of the rows. This will reduce the number of pins from $R + C + N$ to $lg(R) + C + N$, where R and C are the number of rows and columns respectively, and N is the number of pins used for other purposes.

3.3 Approach for design validation

Our approach for validating our design included testing the functionality of each key to ensure that all worked as expected. Once this was completed, we began typing example paragraphs twice, with one run ignoring the auto-complete feature and the second run utilizing the auto-complete feature whenever possible. This provided us with insight as to how useful the auto-complete feature was when saving the user from unnecessary keystrokes. Finally, we conducted several latency tests within the team by utilizing reaction test software with both a normal laptop keyboard and our own prototype. This was done in order to determine if there was a significant difference in the latency of a keystroke between an in-built keyboard and our own design. The results of these design validation tests can be found in section 5.

4 Implementation notes

4.1 Software

Starting with the Raspberry Pi's software design, the auto-complete algorithm is implemented with two classes: a node and a trie. This trie data structure provides a traversable

representation of a collection of English words. This collection is formatted as a dictionary text file where each word is on its own line. Crucially, these words should be ordered by frequency where the most typed words are at the beginning and the least typed words are at the end. For our implementation, a dictionary found from a Google study is used compiling the 10 thousand most typed English words in frequency order (https://github.com/first20hours/google-10000-english). For the trie to represent this dictionary, a node class is necessary.

Each node in the trie structure is a representation of a character that exists in at least one word from the dictionary (directly stored as a character within the node). Each node can have parents and children depending on its position in a word. For example, node 'h' in the word "the" has a parent 't' that points to it. The 'h' node also points to a child 'e' node. If a node has no parents, then it denotes the beginning of a word. To keep track of children, each node has an array of 26 node pointers which is enough space to point to all the potential children (one spot for each letter in the alphabet). This array is in alphabetic order and is therefore referenced by index. For example, if a node's 'a' child was to be obtained, the array would be indexed at position 0. If the node's 'z' child was desired, the array should be indexed at position 25. Each element in this array of children has the capacity to be a nullptr, indicating that there is no child for that letter.

Outside of having children, each node has the capacity of being a terminal node (meaning it denotes the end of a word). This is stored by a boolean variable that, when set to true, marks the node as a potential end to a word. A node can be both terminal and point to other nodes (for example, the 'e' node in "the" can also point to an 'r' node in "there"). The final component of the node class is a priority array. This array stores the 4 highest priority children. The values of this priority array correspond to indices in the children array. The priority array itself is ordered from highest to lowest. Value in the priority array can also have two special values. If the value is 26, then the priority array at that location indicates the priority of the node being terminal. If the value is 27, then there is no child for that priority. The reasoning for having just 4 elements in the array is to have enough space for at most three suggestions (the keyboard suggests 3 words at a time). An extra spot is needed in case the node being terminal is in the top three (and therefore another suggestion might be needed if the node had already been typed by the user). One final note about the priority array is what happens if a child appears twice in it. This is possible when re-traversing the same child should be done for a node if the two highest priority words continue with the same character ("the" and "there" in the 'h' node, for example). This is accounted for through integer division. Each value of the priority array is offset by 28 multiplied by the number of times it appears earlier in the array. For example, if the 'a' child of a node appears twice before the current index, the value of the priority array is 0 added with 28 multiplied by 2 (56). This means that integer division with 28 will give the number of times this child occurs previously in the array and modulus will give the alphabetic index.

All data members in the node class are kept private and are interacted with using 9 member functions. First, a parameterized constructor is used to make the node using a character. The constructor simply sets the node's character and initializes the children and priority arrays to initial values. Four getter functions have also been implemented for each of the four member variables (character, children, terminal, and priority).

When a node is added to a trie, two main functions in the node class should be used. First of all, an add-node function adds a child to a node's array of children (using that child's character to get the correct index). This function does not do anything about priority, that is the job of the add-priority function. This function accepts a character and attempts to add it to the highest priority in the priority array. To do this, the function iterates through the priority array, counting the number of times this character has already appeared in the array. Once an empty spot in the array has been found (value 27) the value is changed to the alphabetic position of the character (0 for 'a') added with the number of times the character already appears in the array multiplied by 28. This is so if that child already exists, the trie structure can know that simply by looking at the value (mod for character, integer division for previous occurrences). Like before, a value of 27 indicates the priority of the node being terminal. The last function on the node class it to set the terminal boolean to true (it is otherwise false by default).

Trie Node Class Interface

```
class Node {
    private:
        char character;
        bool isTerminal;
        Node* children[26];
        unsigned char priority[4];

    public:
        Node(const char c);

        void addNode(Node* n)
        void setTerminal();
        void addPriority(unsigned char n);

        char getChar() const;
        Node* getNode(char c) const;
        Node* getNode(unsigned char i) const;
        unsigned char getPriority(unsigned char i) const;
};
```

The node class alone is not enough to accomplish the auto-complete traversal. The trie data structure is in charge of managing these nodes and is the interface the driver program works with most directly. The trie begins the reference to all nodes using its only member variable, 26 root nodes. One root node is used for each beginning letter in the English alphabet. From these root nodes, all characters in all words in the provided dictionary file can be referenced.

The trie implements the rule of three through a copy constructor, copy assignment operator, and destructor, but the majority of the trie's functionality is in two other functions. The first of these functions is a constructor that accepts a file name. This constructor opens the provided dictionary file to create the trie from scratch. This function iterates through each word in the dictionary and then iterates through each letter (starting at the root array). For each letter, the function checks to see if the node exists. If it does not, one is created and added. Then, the function sees if there is an empty spot in the priority array, and if there is it is added. In the final node of each word, the terminal boolean is set.

With the node class definition and the trie constructor it is now possible to implement the auto-complete algorithm. The get-candidates function is what implements this. This function accepts a partial word as a parameter, indicating what the user has typed. The function then starts by traversing the trie to the last typed character. If the function fails to do so before then (meaning no such word can exist) it terminates. Next, a loop iterates four times to compute the three auto-complete suggestions. An extra iteration is needed in case the node being terminal is one of the priority values (which would not help the user). The loop begins with the last known node's priority array and then iterates through children using their priority arrays. Once a full candidate is found (ended with a terminal node) it is added to an array of strings that are returned.

Trie Class Interface

```
class Trie {
    private:
        Node* root[26];

    public:
        Trie(const Trie& other);
        Trie& operator=(const Trie& other);
        ~Trie();

        Trie(const std::string fileName);

        int getCandidates(std::string partial, std::string candidates[3]);
};
```

To send suggestions to the LCD, we created an LCD and a LCDList class, located in LCD.h. The LCDs we used were 16x02 LCDs, meaning that they support 16 characters horizontally and two characters vertically. The Pi was responsible for sending the computed suggestions to the LCDs.

An I2C connection was used to send data to the LCDs. The alternate connection is SPI; however, for our use case, I2C was the best option. I2C connections only require two wires, serial clock (SCL) and serial data (SDA), to transmit data bi-directionally. Additionally, the address system and shared bus of I2C allows for multiple devices to be connected via the same wires. Compared to SPI, which uses four lines, I2C is simpler in complexity and requires less GPIO pins. The only downside of I2C is the speed of data transfer, but this is negligible since we are only transmitting three strings at a time.

We used the WiringPi library to interface with the LCDs via an I2C connection. The LCD operates at a low level, reading in hexadecimal instructions from the I2C channel. The WiringPi library makes it easier for the developer by allowing the use of plain strings and built-in commands like 'clear' and 'toggle_backlight'. The LCD class is used to initialize a single display, based on the I2C address. The default address is 0x27 for all LCDs, but it can be adjusted by bridging terminals on the back. Because we needed to send output to three separate LCDs, we modified two of them to use different addresses.

The LCDList class, on construction, takes in N addresses and initializes N LCDs. The purpose of this class is to manage all LCDs so that the driver program does not need to keep track

of the different displays. The code supports any number of addresses, so that we if decide to change the number of suggestions and macros, all we need to do is add or remove addresses from the constructor. This class contains a suggest() method, which takes in a string array of suggestions and passes individual suggestions to each existing LCD in the list. Once the driver calls this function, the LCDs are correctly updated in order of the suggestions. After a user has moved onto the next word, either by pressing space or a macro, the LCDList will clear all screens so new suggestions can be made. Error checking is in place, such that suggesting too many or too little suggestions is handled properly.

LCD Class Interface

```
class LCD {
    public:
        int addr;
        int fd;

        LCD(int addr);
        void clear(void);
        void lcdLoc(int line);
        void typeChar(char val);
        void suggest(const char *s);
        void lcd_byte(int bits, int mode);
        void lcd_toggle_enable(int bits);
        void lcd_init();
};

class LCDList {
    public:
        std::vector<LCD*> LCDs;

        LCDList();
        LCDList(std::vector<int> addrs);
        void clear();
};
```

All these components are combined within a driver program which implements main. The driver program has global variables to keep track of the current auto-complete suggestions and the user's partial word. This main function sets up some components of Wiring Pi and then simply runs an infinite loop (without any commands). While this loop runs, hardware interrupts from the PICO (GPIO 21) issue a function to implement the main functionality. Hardware interrupts mark the beginning of a MENA value which is sent over USB. MENA's specifications are for a two-byte integer. This is used to represent a keystroke as a combination of special keys (CTRL, ALT, SHIFT, and WIN) and "normal" keys (alphabetic, numeric, etc.). This standard was outlined in the Design section of this document.

The core functionality does two main things. First of all, it decodes the MENA value sent by the PICO over USB into the components of the keystroke. Then, using this information, the auto-complete suggestions are updated. If something other than a character key is pressed, the suggestions are cleared. Otherwise, when an alphabetic character is pressed, the Trie's get-

candidates function is run with the new partial word. With the new suggestions, the LCDs are updated using the LCDList class interface.

With the new auto-complete suggestions, keystrokes are sent to the receiving computer. If a macro was pressed, then the selected auto-complete suggestion is sent instead, one character at a time. Keystrokes are sent to the receiving computer using Key Mime Pi (https://mtlynch.io/key-mime-pi/). Key Mime Pi allows the Raspberry Pi to act like a standard keyboard to whatever device it is plugged into. Key Mime Pi does this through shell commands that have their own format. Logic in the driver program is present to convert the MENA values into this format. With this completed. Shell commands are executed within the driver program using the system commands to make a new process. Two commands are sent for each keystroke: one to replicate the key being pressed down, and another for the key being released.

Two bash programs, `serial.sh` and `run_keyboard.sh`, are stored on the Raspberry Pi. The script `serial.sh` is responsible for reading in MENA data sent from the Pico to the Raspberry Pi's serial port (`/dev/ttyAMC0`). The MENA values are output to standard out. The script `run_keyboard.sh` runs both `serial.sh` and `driver`, piping the output of `serial.sh` into the driver to be processed. This method of handling input from the Pico has been the most effective of those attempted. Initially, UART was used to communicate data between two Pico units, but there was poor support for UART when attempting to communicate with the Raspberry Pi. The second method implemented was to set up the Pico as a USB Human Interface Device to send each byte to the Pi as a key press. This was not very efficient and incurred latency.

The code on the Raspberry Pi Pico (written in CircuitPython) is responsible for reading key presses from physical the keyboard, converting key presses (whether individual keys or combinations of keys) into a MENA value, and sending that value to the Raspberry Pi to be processed. This functionality is all in the file `code.py`. CircuitPython searches for and immediately runs this file after the Pico starts up.

In order to detect key presses, the Raspberry Pi Pico continuously powers each column of the key matrix individually and scans each row. When a key is pressed, its corresponding row and column become connected; thus, its row will be detected by the Pico when its column is powered. In order to power the proper column, the Pico encodes the column to be powered as a 4-bit signal to send to the hardware. The key scanning code utilizes a doubly-nested loop structure, essentially checking each location of the matrix to see whether or not the corresponding key is pressed.

The Pico keeps a mapping from locations in the key matrix to the associated MENA value. When a key press is detected, the corresponding MENA value is retrieved from the mapping. Bit masking is used to handle simultaneous key presses, such as those that occur as combinations of modifier keys (Ctrl, Shift, Alt, Windows) and regular keys. The MENA encoding was designed in such a way as to enable modifier keys to be easily added or removed from a MENA value, allowing for simple key press encoding, decoding, and processing. When non-modifier keys are pressed, the last 8 bits of MENA value to be sent are first cleared, resulting in potential overwriting of values when multiple non-modifier keys are held down simultaneously. The resulting functionality would be that the key that is scanned latest in the nested loop cycle is the key press that is registered (along with any modifier keys, since these are not overwritten within a loop). This is the intended

functionality, but would not be difficult to alter if a more elaborate handing of key presses is desired.

Once the entire 5 x 16 matrix is scanned, an interrupt is generated through a GPIO pin on the Pico if any keys were pressed, and the MENA value is sent to the Pi through the serial connection. Due to a minor oversight in the design of the MENA encoding that initially resulted in incorrect handling of modifier keys, the Pico uses Bit 4 to keep track of whether or not there has been a non-modifier key pressed during the current scanning cycle. This bit is zeroed out before any data is sent to the Pi and is not used to transmit any information outside of the Pico.

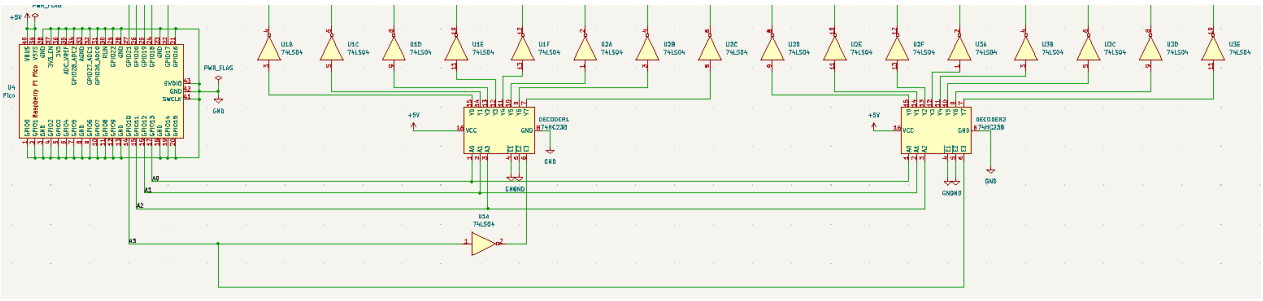
Currently, the Pico handles all of the timing of key presses, including the timing involved when the same key is continuously held down. The handling of held-down keys is likely something that would be altered in the future, as it introduces some issues for non-typing keyboard applications.

Lastly, there is another file on the Pico named `boot.py` that specifies the boot configuration for the Pico. This was required in order to disable mass storage mode, which causes the Pico to be read as a USB storage device and results in a prompt on the Raspberry Pi. In order to access the Pico in mass storage mode, GPIO pin 2 has to be grounded prior to powering the Pico. This must be done in order to make modifications to the code on the microcontroller.

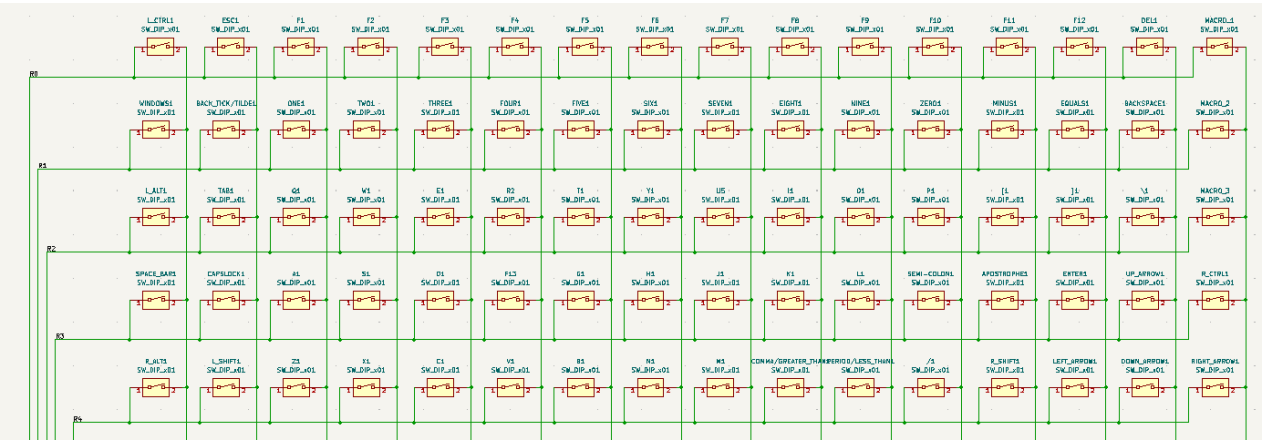
4.2 Hardware

The initial steps to begin designing our hardware was to select the correct chips for our implementation. We opted to find as many chips as we could in lab to save time when ordering the PCB since placing that order was a bigger time constraint. This meant we had to make some design decisions that we would not have implemented in a market product. The decoder that we were able to find in the lab were 74HC138's. They are 3-to-8 decoders that produce one active low output while the other 7 remain high. In a final product we would switch to 74HC238's which do the inversion described below automatically, producing one active high output. We would also consider switching to a standalone 4-to-16 decoder and eliminate the need for additional inverters in our design entirely. The inverters we found in lab were 74SL05's which are standard 6-gate input and output integrated circuits. Finally, we decided the Pico would be soldered onto the PCB to save space and present a more complete final design. All Datasheets are in our Appendices.

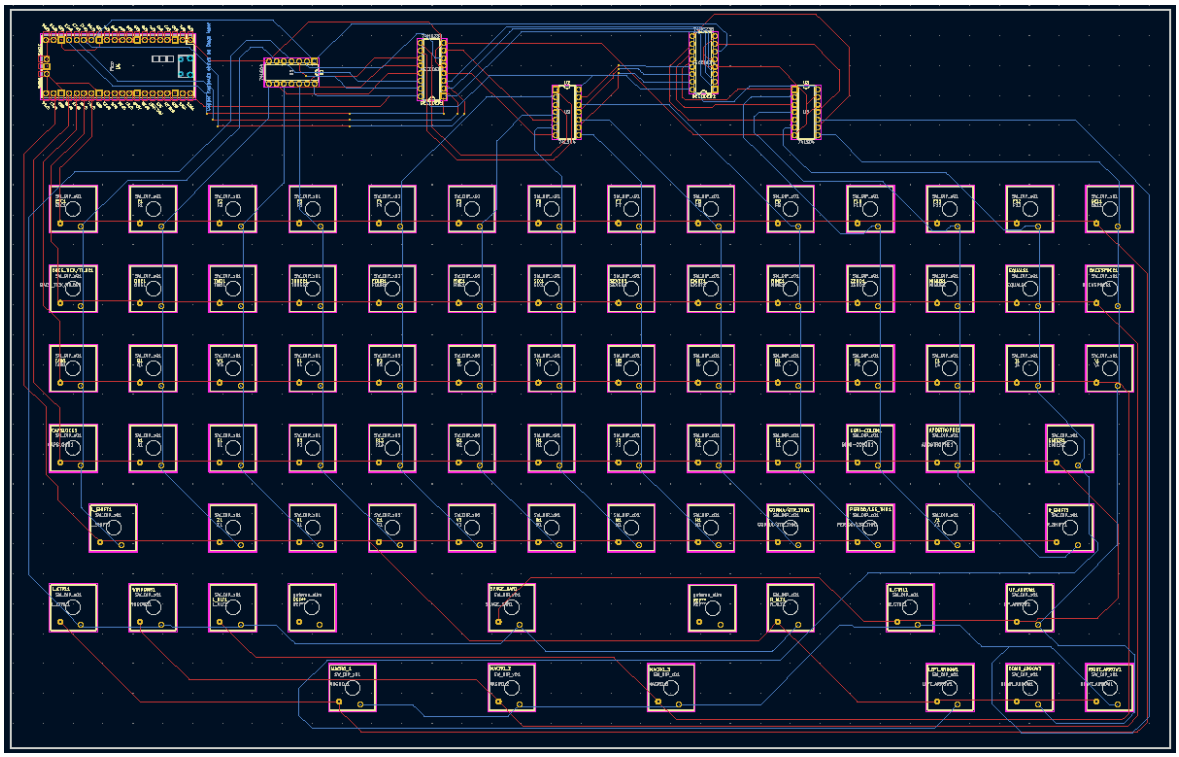
In the lab, we were able to find 3-to-8 decoders, but no 4-to-16 decoders. Rather than use 6 GPIO pins, that would have no ability to determine which 8 columns were being polled, we opted to use knowledge learned in our electrical engineering coursework to combine 2 of the decoders found in lab into 1 4-to-16 decoder. In addition to implementing the 3 existing input pins, we were able to use the enable bit on each decoder to function as a fourth input. To ensure each set of 8 pins were referred to, the first decoder inverted the enable signal before receiving it creating 16 unique signals that were 4-bits each.



The picture above shows the 4 GPIO signals at the schematic level decoded into 16 signals. One signal is used for each column during polling to power a specific row depending on the key pressed in the Matrix. After the signals were decoded, it was necessary to invert each signal because the Decoders we were using output high power until true, this would have caused unessential power consumption and cause issues with the software that had already been created. The inverters ensured all signals were low except the one being polled, which in turn will power one row in the matrix. The software then determines which key has been pressed.



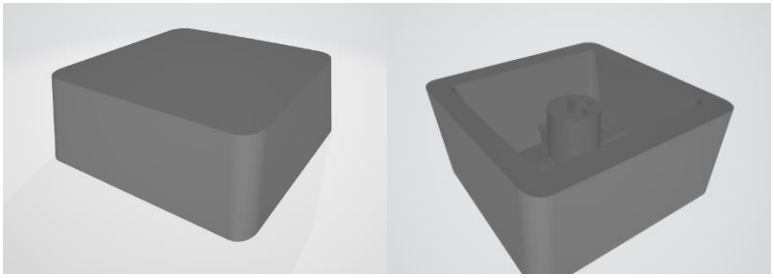
Above is the previously mentioned key scanning matrix. It receives inputs from the decoders as the Pico polls each column, once a key is pressed the switch will complete the circuit and power a specific row. The Pico can then determine which key was pressed based on the combination of the row that is sending a high signal and the current column being polled. Once this Matrix was determined, we mapped specific keys to each location and were able to move them to their proper place on the keyboard in the PCB routing phase. This is shown below.



In the routing phase, we were able to ensure our board met all specifications to be printed at the chosen manufacturer, PCBWay. Our board was a 2-layer through-hole plated PCB with the following specifications. It is important to note that our PCB design grounds all unused pins on the Pico, this caused a booting issue with the microprocessor, so we had to solder only the pins we needed onto the Pico for it to operate properly.

Board type :	Single pieces	Panel Way :	
Different Design in Panel:	1	X-out Allowance in Panel :	
Size :	388 x 247 mm	Quantity :	5
Layer :	2 Layers	Material :	FR-4: TG150
Thickness :	1.6 mm	Min Track/Spacing :	6/6mil
Min Hole Size :	0.3mm ↑	Solder Mask :	Green
Silkscreen :	White	Edge connector :	No
Surface Finish :	HASL with lead	"HASL" to "ENIG"	No
Via Process :	Tenting vias	Finished Copper :	1 oz Cu
Remove pcb identification number :		Additional Options :	UL Marking:None,
PO No. :		Manufacturing :	If there is no extra cost we would p refer our board to be Through-hole plated.

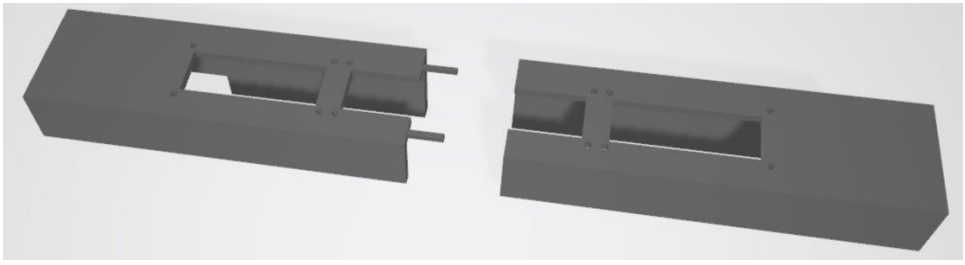
4.3 Modeling

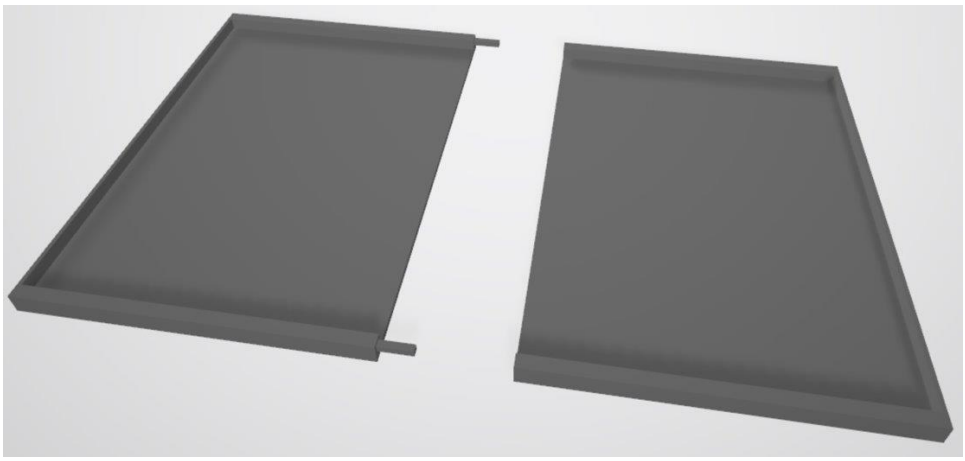


The initial design of the keycap was similar to a traditional mechanical keyboard keycap with a few changes. The stem of the keycap is a traditional interface with a mechanical keyboard switch. For this part, there was some tolerance given so the parts could be successfully 3D printed. The minor changes were to enlarge the surface area to 0.9 inches in order to allow the key to be easier to press, and to inset the top of the key into the outer rim to ensure that when a key is being pressed, the user would not accidentally press another key.

When the first set of keycaps were printed, there were a few issues. The first of which was the fact that the stem which interfaces with the switch was not consistent and uniform in the sizing. This proved to be a non-issue as the switches still fit inside the keycap consistently. Another issue was due to the shape of the top of the keycap. The keycap is printed upside down to ensure that the stem of the part would be printed with the best orientation to ensure a successful print. This means that there needs to be support on the inset part of the key. This proved to be difficult to remove from the model. When printing 80 keycaps for a prototype, this would not be efficient. The benefit that they inset model provided would also be covered by the fact that the keys will sit inside a rigid frame. This ensures that the user does not have any issues pressing the correct key and removes the need for an inset keycap. Due to these issues, the top of the keycap was changed so that it would be flat while remaining oversized.

Two other versions of the keycaps were created, one that was simply double wide on the top surface, and the spacebar keycap. For the double wide keycap, the first model was simply adapted, as the stem needed to remain consistent while having a larger area to press the key. This proved to be rigid enough that there was no need to stabilize the keycap. The spacebar on the other hand was unstable from one switch. The design of this part used the same stem from the previous part. Two stems were spaced six keys apart including spacing, and a middle key was placed directly bisecting the space between the two. Using three switches on the board, the keycap was stable and was able to be pressed consistently.



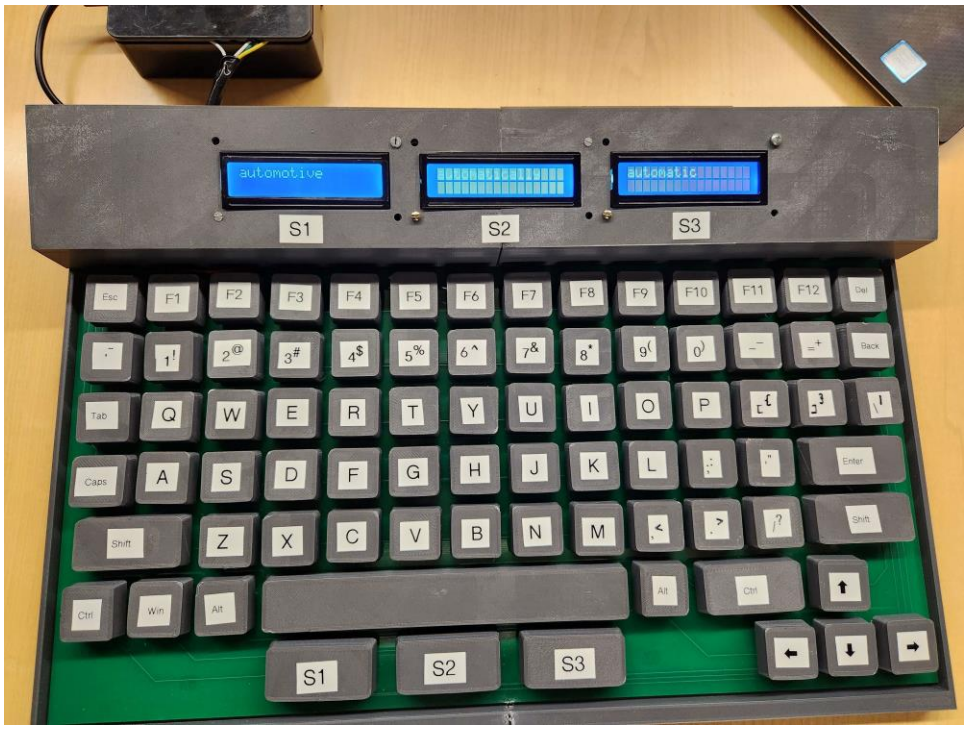


The next part that was modeled and printed is the housing for the keyboard. This was initially done by creating an inset into a rectangle that was the correct dimensions to allow our printed circuit board to nestle into the empty space. This would serve as the base of the keyboard. The next section designed was the housing to cover the circuitry at the top of the keyboard. The dimensions of the circuitry were taken to ensure that there would be no overlap with the keycaps. Next, a hollow shell was created that would house the wiring along with the circuitry that the user would not need. The top of this shell would be angled toward the user to facilitate the user's ability to see the LCDs. Finally, holes at the top of the shell were created to be large enough to see the screens of the LCDs. Additionally, holes were created to secure the LCDs to the top of the shell. This shell would be placed on top of the other housing section.

These two parts combined would create the shell of the keyboard. Unfortunately, the 3D printers that would be used to fabricate these models do not have a large enough build volume to print these parts without breaking them up. To do this, both the bottom and the shell were divided in half. To join these split parts together in a rigid fashion, a peg and hole were created on the halves to interlock these sections together. This forms the housing for the keyboard.

5 Experimental results

Below is a picture of the final prototype. As can be seen, the keyboard part of the device is mostly standard QWERTY, but with a grid design. Above the keyboard are the three LCD's which display the three auto-complete suggestions (which correspond to three macros below the spacebar).



The first test was used to measure how many keystrokes the auto-complete functionality could save. This was done for two example passages. The first passage represents a quite standard collection of words of varying length. The other passage represents the worst case-scenario, with a few obscure words, lots of punctuation, and verbose vocabulary. Below are the two passages.

Paragraph 1: "In a hole in the ground there lived a hobbit. Not a nasty, dirty, wet hole, filled with the ends of worms and an oozy smell, nor yet a dry, bare, sandy hole with nothing in it to sit down on or to eat: it was a hobbit-hole, and that means comfort."

Paragraph 2: "Paragraphs are the building blocks of papers. Many students define paragraphs in terms of length: a paragraph is a group of at least five sentences, a paragraph is half a page long, etc. In reality, though, the unity and coherence of ideas among sentences is what constitutes a paragraph."

For the test, each passage was typed twice: once without using any auto-complete suggestions (representing a normal keyboard's usage) and again using as many auto-complete suggestions as possible. For each, the number of total keystrokes and number of used suggestions was measured. The table below summarizes the results.

Category	Total Words	Without Auto-complete	With Auto-complete
Passage 1: Total keystrokes	49	247	196
Passage 1: Predicted Words		-	28
Passage 2: Total keystrokes	44	288	200
Passage 2: Predicted Words		-	43

As can be seen in the table, the auto-complete suggestions saved a significant number of keystrokes for the user. While it is unlikely the user will every use the auto-complete suggestions to 100 % accuracy, this does demonstrate how the feature can save users on the number of times they must press keys. Furthermore, it was found while testing that, due to the algorithm's static nature (suggestions do not change over time given the same input), the user can come to memorize what keys must be pressed for certain words (for example "the" can be typed with "t" and suggestion 1). Of particular note is that the auto-complete algorithm was able to predict most standard English words. Furthermore, words that would be expected to be more common (such as "the") were suggested first.

Another test was aimed at finding the perceived latency for the MIKey device against a standard keyboard. To do this, a reaction measuring test was used. In this test, once a color flashed on the screen, the user was to press any key on the keyboard. The program would then state the time between the flash and the key press being received. While this will also measure items like the screen's latency, differences between the two devices can show any shortcomings. This test was performed on five team members. Each team member was to do this test 10 times on a standard keyboard and 10 times on the MIKey device. Different participants used the two devices in different orders to eliminate variables. Below are the results of the study.

(average ± std deviation)	Normal Keyboard	MIKey Device
Participant 1	304.6 ± 23.7 ms	314.5 ± 35.8 ms
Participant 2	294.3 ± 21.1 ms	352.2 ± 30.6 ms
Participant 3	274.1 ± 17.95 ms	265.9 ± 17.4 ms
Participant 4	252.2 ± 20.1 ms	262.1 ± 12.1 ms
Participant 5	275.2 ± 26.6 ms	312.6 ± 44.8 ms
Total	276.5 ms	288.8 ms

Looking at the table, users were around 10 ms quicker with the normal keyboard. However, when one looks at the standard deviations, it can be seen that these far exceed this difference. Additionally, one of the five users was actually faster with the MIKey device. This indicates that the MIKey device may be slightly slower, but other factors, such as variance in reaction time and fatigue play a far more important role. Therefore, it can be concluded that the MIKey device does not present a noticeably higher latency than the normal keyboard (this was also found qualitatively through user reports).

6 User's Manuals

6.1 Startup

In order to use the Motor Impairment Keyboard, first plug the keyboard into the USB-A port of a computer device. Once the MIKey has started up (may take approximately one minute), the three LCD screens will each be illuminated, and the keyboard will be responsive to key presses. There is no need to install any software or drivers to the computer device for operation.

6.2 Operation

The MIKey can be used in the same manner as a regular keyboard, enabling users to type text into programs on their devices. It supports typical special characters (‘!’, ‘@’, ‘#’, ‘$’, etc.), function keys (F1, F2, F3, etc.), as well as key press combinations that include the Ctrl, Shift, Alt, and Windows keys.

In addition to regular keyboard functionality, MIKey supports auto-complete in order to reduce the number of keystrokes required by a user to complete a given task. Three auto-complete suggestions for the word currently being typed are displayed on the three LCDs (labeled S1, S2, and S3). In order to use the auto-complete functionality, users can select one of the three suggested words by pressing any of the S1, S2, or S3 keys to complete the word on the corresponding display. The labeled displays and suggestion keys are shown in the image below:

2 displays. Upon selection of an auto-complete suggestion, the remainder of the selected word is typed into the computer, followed with a space. Selection of an auto-complete suggestion also clears each of the three suggestion displays.

6.3 Safety

Since the MIKey is an electronic device, users must use caution when operating to not spill liquids on or near the keyboard.

7 Course debriefing

The team followed an Agile development process to organize our tasks and workflows into a total of 5 sprints. All team members were also divided into both technical and non-technical

roles. At the start of the project, we attempted to match each individual team member with the technical skillset that best fit their background and knowledge. However, once we began working on our tasks, many team members ended up swapping roles. The final changes in our team structure are shown in the table below:

Technical Role	**Members Involved**
Hardware Team	Andrew, Brittany, Connie
Software / Embedded	Jackson, Max, Abhishek
3D Printing / Modeling	Andrew, Connie

As you can see in the above table, team members Max and Abhishek swapped places with Brittany and Connie from their original placements on the hardware and software teams. Additionally, Connie replaced Jackson on the 3D printing and modeling team. Overall, these changes did not have a negative impact on the team's productivity or dynamics, since the roles that were set at the beginning of the project were done so as a tentative measure. The flexibility and adaptability that all members of the team exercised in order to accommodate these changes demonstrates a level of synergy that not all teams would be able to endure together.

If we were given the opportunity as a team to re-do this project, the only thing we likely would do differently in terms of teamwork would be to create more structure for the division of tasks and labor. Since the level of expertise amongst the team wasn't very even, a lot of the technical development fell into the hands of only a couple members while less knowledgeable members had difficulty picking up tasks. Other than this point, all other team interactions went rather smoothly.

Although the end product likely won't raise any safety concerns for our users, there were several safety concerns for the team to consider during the development of our prototypes. These include, but are not limited to, the various soldering needed for both the PCB and wiring, power delivery to our hardware components, and safety hazards when 3D printing the housing components. The best way to approach these safety hazards was to stay alert while following all outlined standard safety protocols and trainings.

Because the volume of production of our device will likely be fairly small, the environmental impact of our product will also be reasonably manageable. If we were to do the project again, one thing we can consider to decrease our environmental impact is to look towards using recycled/recyclable materials for the housing and the keycaps of the keyboard.

8 Budgets

Final Itemized Parts List

Product Name	Cost (dollars)	Purchase Site
Raspberry Pi Pico	20.82	Amazon
LCD Displays	23.79	Amazon
Keyboard Switches	43.29	Amazon

PCB Material	43.16	PCBWay
Total	131.06	

Our group only had to pay for four of the components of the keyboard. We were able to secure the other components free of charge. One of our members had an existing Raspberry Pi 4, and we were able to print the keycaps and housing for free from the Fischer Engineering Design Center.

Below is the detailed budget of all costs expected to be incurred during the project (e.g., parts, fabrication services) created in the proposal. Our actual final cost was $128.88 lower than the predicted cost. This occurred due to the lower cost of the PCB. At the time of the proposal, we did not have a schematic file ready, so we were not able to get an accurate estimate of the PCB cost. After creating the schematic file, we submitted it to multiple vendors such as jlcpcb and PCBWay. PCBWay ended up having the lowest cost and the fastest delivery time. Therefore, we chose PCBWay over jlcpcb and this is why the cost was less expensive than our proposal. The omittance of certain parts, as well as natural price fluctuations, allowed us to have a save enough money to stay in budget for the project.

The biggest difference between the final overall costs and the proposal occurred for a variety of reasons. We realized that the Raspberry Pi 0 was not needed and could be replaced by the cheaper Raspberry Pi Pico. We used an existing Raspberry Pi 4 to handle intense computations and the Pico to only feed keystrokes to the Pi. Next, after receiving the keyboard switches, we realized that we did not need O-rings, since the low-profile switches we received already reduced the actuation distance. Additionally, we realized that we could print the keycaps and the housing for free at the FEDC.

Proposed Itemized Parts List

Product Name	Cost	Purchase Site
Raspberry Pi 0	79.99	Amazon
LCD Displays	15.99	Amazon
Keyboard Switches	47.98	Amazon
O-rings	6.99	Amazon
PLA Filament	18.99	Amazon
PCB Material	~90	pcbway.com
Total	259.94	

We built this product for demo purposes and easy development, given the one semester time frame of the project. If we were to manufacture this product at scale, then we would use two Raspberry Pi Picos, instead of the stronger Pi, to carry out all computations. This would save space and money. Below is the manufacturing unit cost that would arise after bulk ordering parts for 1000 keyboards.

Manufactured Unit Cost

Product Name	Cost (dollars)
Raspberry Pi Pico (2)	7
LCD Displays	4.5
Keyboard Switches	32
PCB Material	7.41
Housing Material	9.99
Total	60.90

9 References

"BigKeys keyboards," *BigKeys Company*, 2017. [Online]. Available: http://www.bigkeys.com/. [Accessed: 14-Sep-2022].

"Get started with eye control in Windows," *Microsoft Support*, 2022. [Online]. Available: https://support.microsoft.com/en-us/windows/get-started-with-eye-control-in-windows-1a170a20-1083-2452-8f42-17a7d4fe89a9. [Accessed: 17-Sep-2022].

Kinesis, "Why get a programmable keyboard," *Kinesis*, 04-Oct-2019. [Online]. Available: https://kinesis-ergo.com/programmable-keyboards/. [Accessed: 18-Sep-2022].

M. Martin, "Computer and Internet use in the United States: 2018," *census.gov*, 2021. [Online]. Available: https://www.census.gov/content/dam/Census/library/publications/2021/acs/acs-49.pdf. [Accessed: 19-Sep-2022].

"Motor Disabilities Assistive Technologies," *Motor Disabilities*, 12-Oct-2012. [Online]. Available: https://webaim.org/articles/motor/assistive. [Accessed: 17-Sep-2022].

S. J. Davies, "Our new keyguard alternative," *LimeTech*, 29-Jan-2022. [Online]. Available: https://limetech.uk/keyguard-alternative/. [Accessed: 16-Sep-2022].

T. Rudnicki and T. ATI, *Eye Control Empowers People with Disabilities*, 2022. [Online]. Available: https://www.abilities.com/community/assistive-eye-control.html. [Accessed: 19-Sep-2022].

10 Appendices

IRB Approval was a longer process than we first expected. We were told we had to submit a determination then apply for an exemption, and the determination was not reviewed until the week before Thanksgiving break. This did not allow us to have time to resubmit to obtain an exemption. To reference our application, our IRB number is **IRB2022-1301**.

IRB Approval:

Submission

Demographic & Questionnaire:

- Q1: Do you suffer from a motor function impairment or physical disability that affects typing?
- Q2: What is the severity of your motor impairment?
- Q3: How often do you type using a keyboard?
- Q4: Do you currently use any assistive keyboard technology?
- Q5: What is your age range?
    - 18-25
    - 26-35
    - 35-50
    - 51-65
    - 66-75
    - 76+

Questionnaire

Q1: The Motor Impairment Keyboard was easier to use overall than the normal keyboard.				
Strongly Agree	Slightly Agree	Neutral	Slightly Disagree	Strongly Disagree
Q2: I felt I typed faster on the Motor Impairment Keyboard than the normal keyboard.				
Strongly Agree	Slightly Agree	Neutral	Slightly Disagree	Strongly Disagree
Q3: I felt I typed more accurately on the Motor Impairment Keyboard than the normal keyboard.				
Strongly Agree	Slightly Agree	Neutral	Slightly Disagree	Strongly Disagree
Q4: The auto-complete screen on the Motor Impairment Keyboard helped me type faster.				
Strongly Agree	Slightly Agree	Neutral	Slightly Disagree	Strongly Disagree
Q5: The recessed keys helped my fingers stay on the correct keys.				
Strongly Agree	Slightly Agree	Neutral	Slightly Disagree	Strongly Disagree
Q6: If this keyboard was on the market, I would consider purchasing it for personal use.				
Strongly Agree	Slightly Agree	Neutral	Slightly Disagree	Strongly Disagree

Datasheets:

Inverter 74LS05

Decoder 74HC138

Raspberry Pi Pico

Raspberry Pi 4

LCD

Project files:

https://github.com/MIKeyTAMU/MotorImpairmentKeyboard/

MIKey Demonstration

Resources:

https://mtlynch.io/key-mime-pi/

http://wiringpi.com/

https://github.com/first20hours/google-10000-english