Predicative Text Application for Adaptive Human Interface Devices

by

Bryant Evangelista Johnson & John Monroe Gould

An Honors Capstone
Submitted in partial fulfillment of the requirements
for the Honors Diploma
to

The University Honors Program

of

The University of Alabama in Huntsville

April 19, 2016

Honors Capstone Director: Dr. Buren Wells
Professor

_____
Student (signature) Date

_____
Student (signature) Date

_____
Director (signature) Date

_____
Department Chair (signature) Date

_____
Honors College Dean (signature) Date

Honors College
Frank Franz Hall
+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax) honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted as a bound part of the thesis.
In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____	_____
Student Name (printed)	Student Name (printed)
_____	_____
Student Signature	Student Signature
_____	_____
Date	Date

Table of Contents

Abstract

While there exists an abundance of alternative keyboard devices to cater to people with various disabilities, there exits an appalling lack of software support to augment use of the modified keyboards. This project aims to fill the gap left in most alternative interface solutions, by offering a predictive text program for existing devices available as a web extension. To demonstrate its effectiveness, the application will be tested on the deliverables provided by the Adaptive Human Interface Device senior design project. The alternative keyboard from the senior design project will showcase the effectiveness of the application, as well as display the versatility between different hardware targets.

Introduction

Our team is researching and building Adaptive Technology human interface devices to help people living with disabilities interact with computers. Adaptive technology (AT) is defined as a device or component which is specifically designed for persons with disabilities. Human Interface Devices (H.I.D) is a method by which a human interacts with an electronic information system either by input or output. Computers have dropped in price, grown in capability, and leapt into consumer markets in the past few years. There has a been a fairly recent explosion in the prevalence of computers in modern society. Despite this massive increase in importance and distribution, Adaptive Technologies which help persons living with disability interface with a computer have been left behind. Solutions available are expensive and difficult to obtain. There is a distinct need for human interface devices which meet many AT requirements and are available to wider audiences for more reasonable cost.

This project was student lead and noncommercial in nature. We wish to make clear that our design decision to present the project as open source incurs additional work for us to have a

clean operating environment that others can approach and utilize in separate projects, not a sidestepping of features we do not intend to implement. Indeed, after the scope of the project is complete, the open source design ensures the group, or anyone else with good intentions, can pick up where the original project ended and contribute additional research and improvements to the design. The primary hardware component, the sliding keyboard, will require no finger movement to type characters, instead utilizing shoulder and elbow ranges. We are interfacing with local disability service group United Cerebral Palsy (UCP) of Huntsville.

We established our marketing and engineering requirements based upon UNICEF's World Report on Disability. UNICEF states, "Manufacturing or assembling [AT] products locally, using local materials, can reduce cost and ensure that devices are suitable for the context." Priority of the design was for cheap and readily available materials and techniques—allowing for the product to be easy to build and modify. Consumers should be able to, with relative ease, assemble the device themselves. This required a design that supported a wide user base being able to obtain the necessary materials for construction, as well as providing a design that supported a relatively easy method of assembling the device. Methods required to complete physical implementations should require no further implements than those found at a typical makerspace (such as low-level machine equipment). Additionally, we wanted to build a product with equivalent or better functionality without increasing complexity of user interaction. The design should be durable and adhere to the standards of current AT marketable solutions. The Marketing Requirements are as follows:

- Open Source Design
    - Require only items available to common makerspaces
    - Easy to iterate and customize

    - Low cost requirements for parts and machinery
- Easy to Operate
    - Must provide as good or better functionality than existing solutions
- Reliable
    - Must accept improper input and have a robust mechanical design.

The Engineering Requirements were based upon current mechanical keyboard design standards. We needed the hardware to react to changes under 6 milliseconds while in use. Each switch will need to last for at least 10 Million cycles while supporting at least 30 words per minute operation at 90% accuracy. The device should be operational from -10 Degrees Celsius up to 70 Degrees Celsius. These requirements are typically found on the product speciation section of modern mechanical keyboards by various brands on the marketplace. Research was conducted by browsing and comparing specifications of listed items in retail stores. The aggregate of the standards was compiled, and the results were applied to the design specifications of the keyboards produced for the project.

The primary addition from the honors group consisted of adding a predictive text interface to the proposed keyboard designs. As one of the primary goals is to aide in the streamlining of interface with computers, designing efficient hardware can only go so far. To further improve the efficiency of the designs, the honors team approached the problem from the software angle. The result was a program designed to help users by learning their vocabulary habits and attempting to improve delivery speed by guessing user input. The program begins in a default state with a large library of words. As the user begins to type a word, the program takes the current input and attempts to match it to the most likely result. The initial guesses are simply by proximity to the input, followed by alphabetical order. However, the program tracks the usage

of each individual word, and assigns a weight to each word based on the frequency of use. This allows the program to, on subsequent runs, attempt to match to a much more likely guess, as the frequency provides a much more powerful weight than pure alphabetical proximity. With this system in place, the user should ideally be able to rely on the program to shorten the average time taken for key, repeated phrases to be input via the keyboard. Another solid point is that the program is compatible with all of the designs, as it is purely concerned with the attempted keyboard input from the user (as such, the hardware which merely provides the means of conveying the data does not significantly affect the program).

Background of Existing Implementations

There are numerous shortcomings in the field of AT. The main issue is that there are high development costs to a low volume market. Existing products include a Large Switch Keyboard which is aimed towards people with motor disabilities and lack of precision. It is less costly but does not necessarily solve the problem: just lessens the possibility of error by expanding the zone of valid input. Such a “solution” does little more than mitigate the symptoms of the disability, rather than directly address the issue. Though to be fair, it is a decent fix for very minor precision issues. Shown below is an image of one type of enlarged button keyboard.

The most significant differences between competing products compared to our product is the low price and iterative design. We strive to create and publish an open source product that will allow people to develop custom, low cost AT wherever a makerspace is available. While predictive text applications appear quite common in mobile devices, they are surprisingly sparse in the field of AT. Despite the wide-spread prevalence of such applications (facilitated by the enormous rise in popularity of mobile devices in recent years), there are little to no open solutions to port such software to AT devices. This came as quite a surprise considering such functionality is now essentially a given in modern cellular devices. The honors team aims to bridge the gap to AT, allowing our proposed keyboards, as well as any input device, to utilize the program to improve typing proficiency.

Analysis of Design Aspects

Various products were considered in the design of this project. We have reviewed the possibility of creating four different systems to be part of our suite of low cost AT devices that we will deliver as the outcome of our project. With our trade-off analysis we investigate the options available for our project and the design decisions which lead to our chosen systems.

Sliding Keyboard

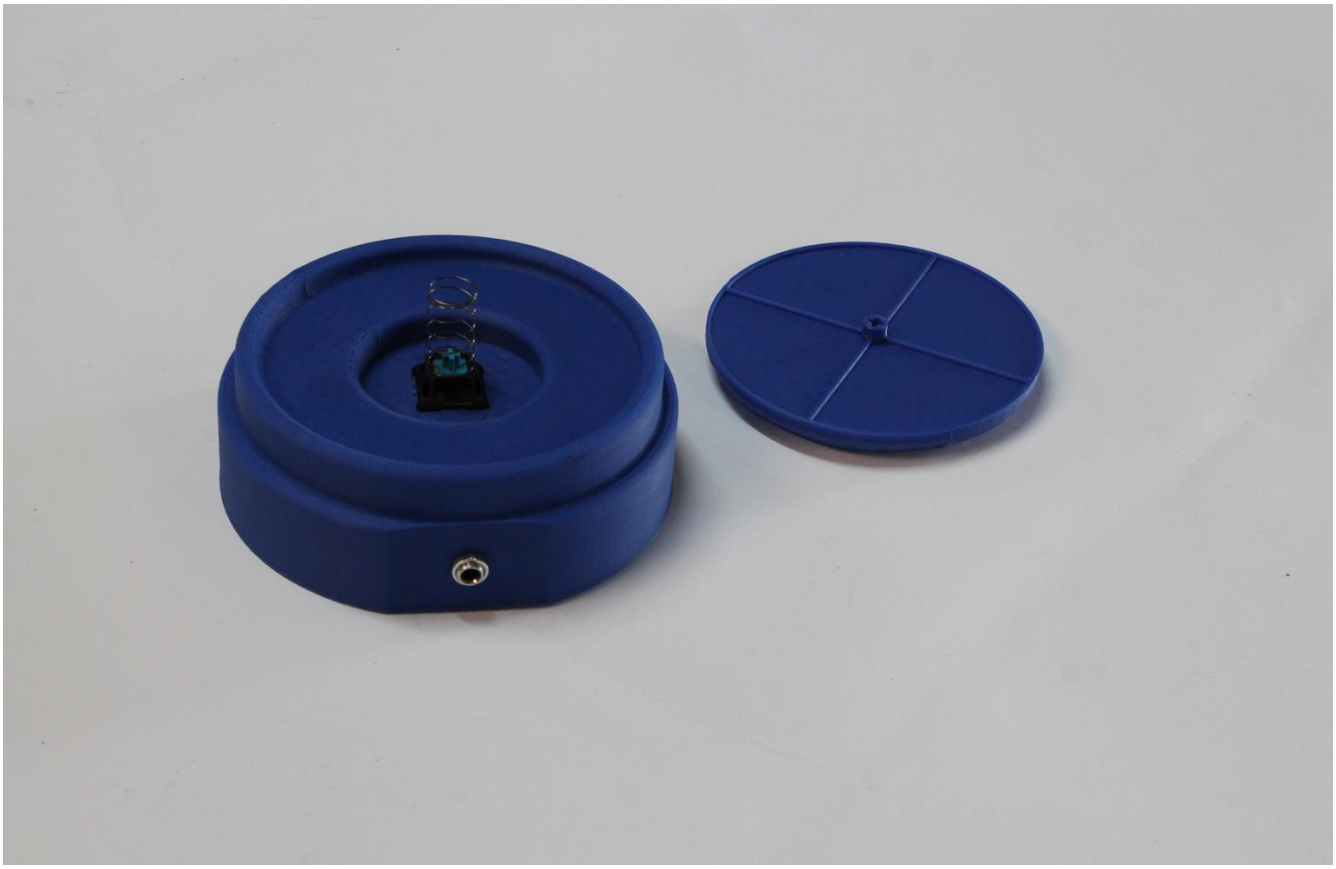
Our first product is a keyboard to limit the required dexterity of typing. Targeted at users with conditions commonly associated with amyotrophic lateral sclerosis or cerebral palsy, this keyboard removes keys from the user's path and instead places two horizontal "pucks" down as control surfaces. Pucks can be a custom design from a handle to a joystick, anything that the user can operate to slide the base of the puck around an inset hexagon that it resides on. At each corner of the hexagon are mechanical switches, totaling twelve between the two pucks. If each

puck is moved to contact a switch, then depending on the key map a range of either 36 or 72 characters can be represented.

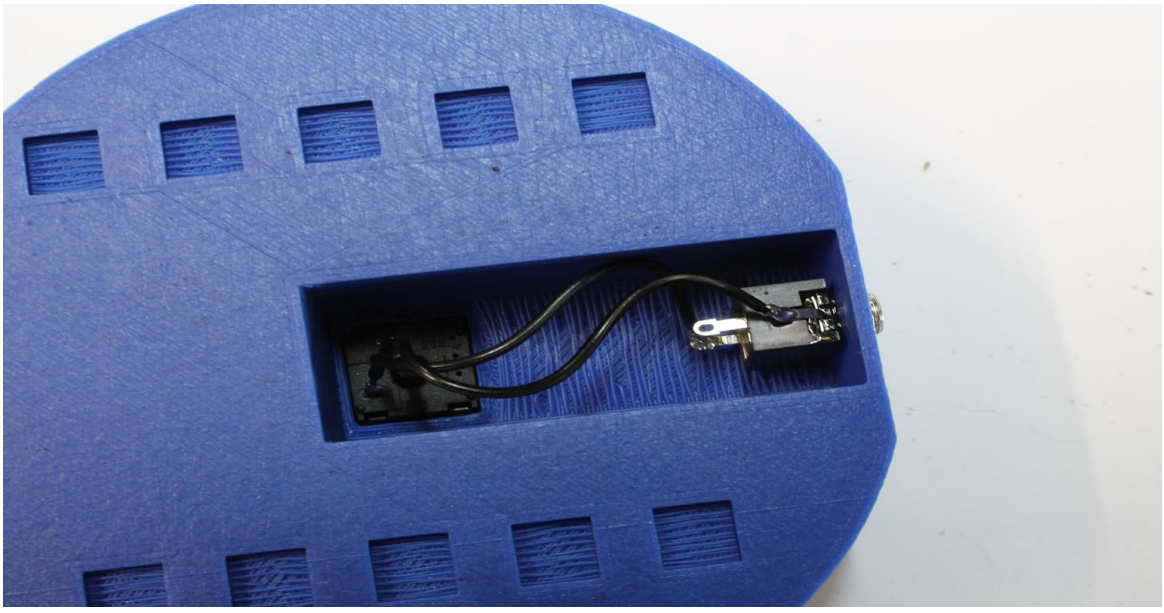
AT Button

While our customer expressed general interest in the above solutions, we also learned a lot about the current state of AT in use at UCP and received enough direction to build out our case study plan for more AT. One of the more common systems for actuation in use at UCP are SPST buttons -- roughly four-inch diameter pads -- that terminate to a 3.5mm mono audio pole. These buttons are then plugged into a variety of host items in order to provide a greater ease of access to the item. At UCP a conflict has occurred where the market price for these buttons is high -- in the $60 to $80 range -- and their use level is low -- an example is sub $20 toy adapting. We suspect this is caused by the quality of the buttons, if the button is associated with a device to request food or water, then it *must* work each and every time. Toy adapting is not quite as mission critical though, and a solution which provides “button to 3.5mm jack” at a lower price would be ideal. Thus we intend to investigate using mechanical keyboard switches with thermoplastic or acrylic cases to create more affordable actuation devices and test them as part of our case study.

Shown below is an image of the AT button. The simplicity of design makes for an easy print job: reducing complexity also lessens the number of moving parts, which should help keep button lifetime high. The image displays the center base, which can withstand significant pressure (upwards of 200 lbs. resting force before suffering from stress fractures). Embedded in the center is a keyboard switch, which interfaces with the button pad located to the right of the switch. When assembled, the pad is placed onto the switch, giving the user access to a very large area to trigger the switch.

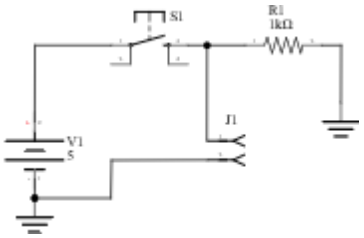


The following image displays the bottom of the AT switch. This image displays the channel that contains the wires connecting the keyboard switch to the input port on the side of the button.

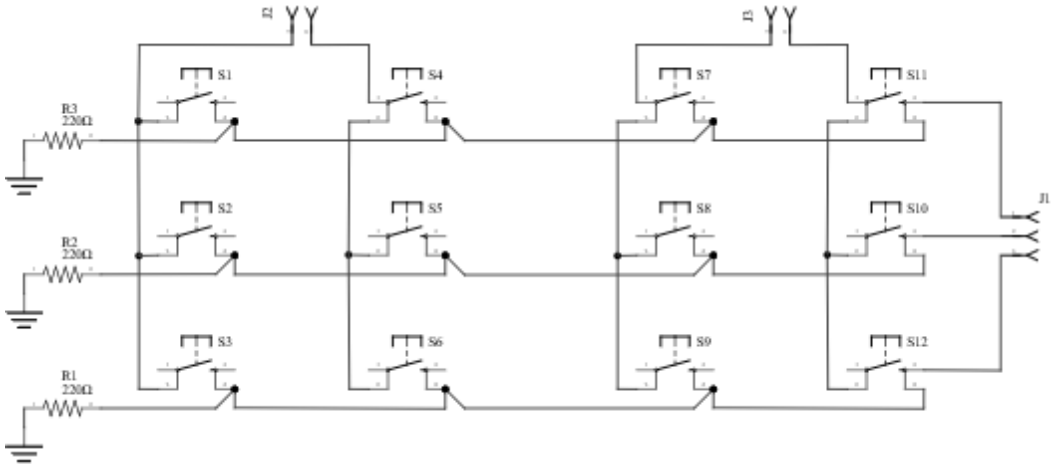


Switch Circuits

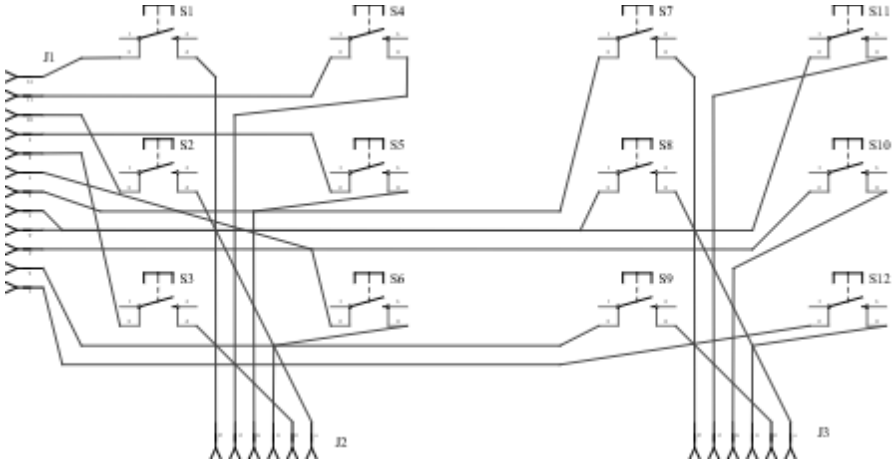
For switches, we will use port inputs on either the MSP430 or atMega and arrange switches with pulldown resistors.



The plan is to implement a scanning grid for the switches as shown below:

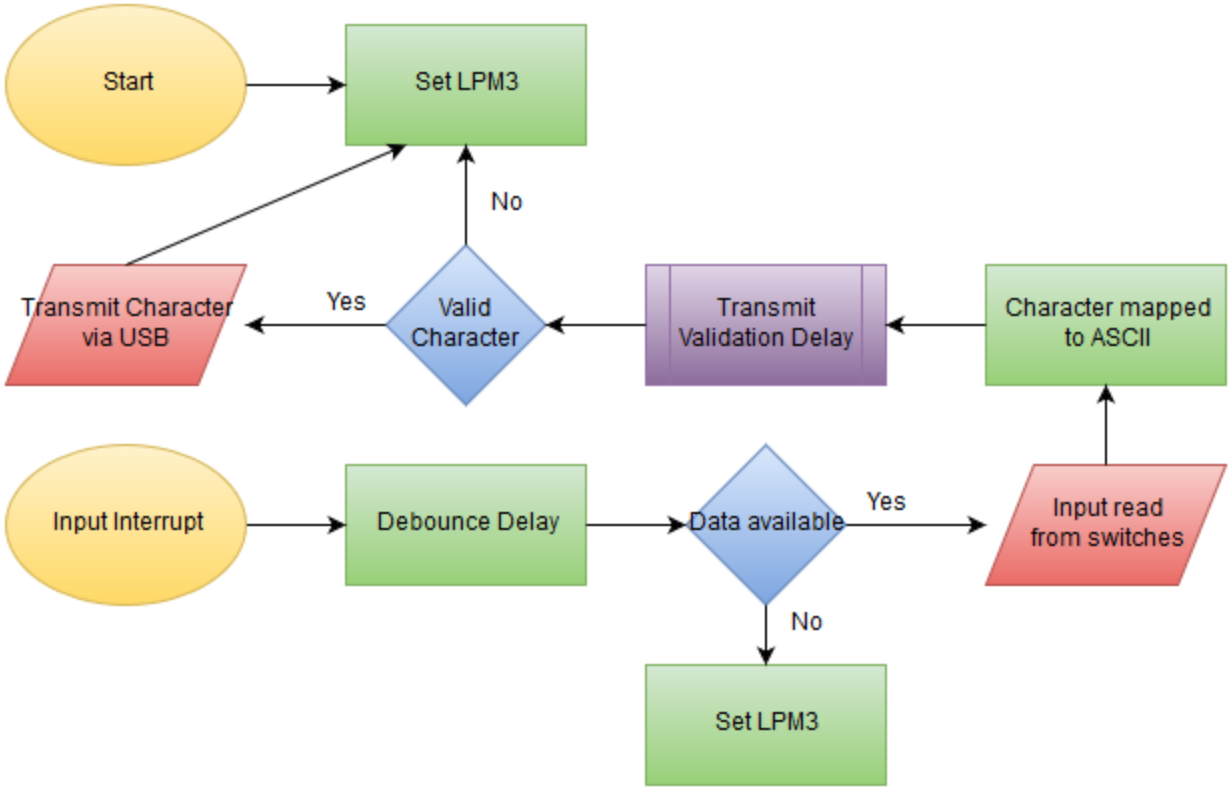


To test the firmware and hardware setups, a set of twelve isolated switches which can be wired into breadboards were constructed. The tradeoff gained is the ease of configuration in exchange for a large amount of tidiness.



Hardware Flow Chart

Below is a flowchart detailing the flow of how the hardware reacts to a user giving input to the keyboard system:

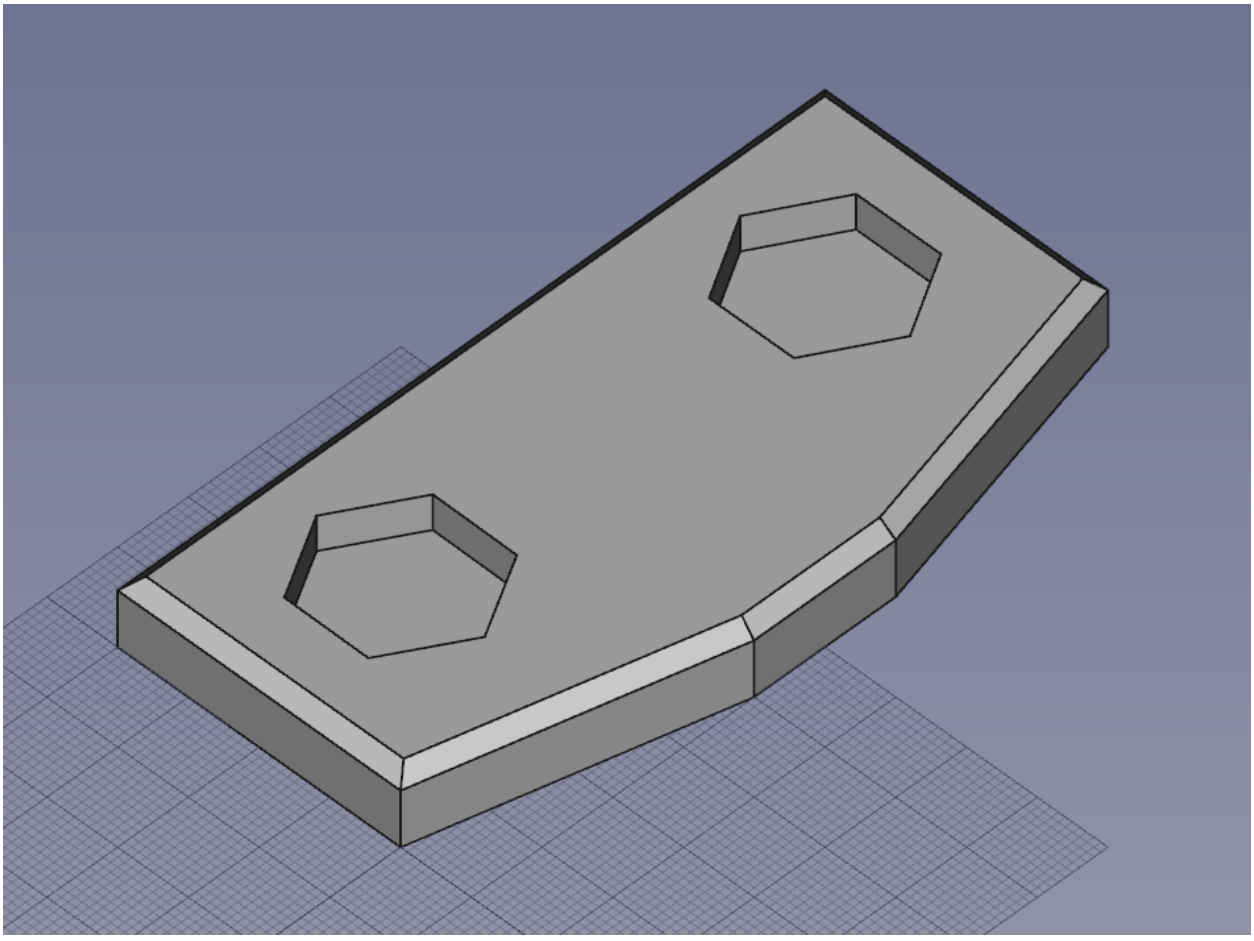


Prototype and Developmental Stages

As most of the developmental lifetime was spent working on the sliding keyboard, this section will be heavily focused on such. The sliding keyboard was the main focus of attention

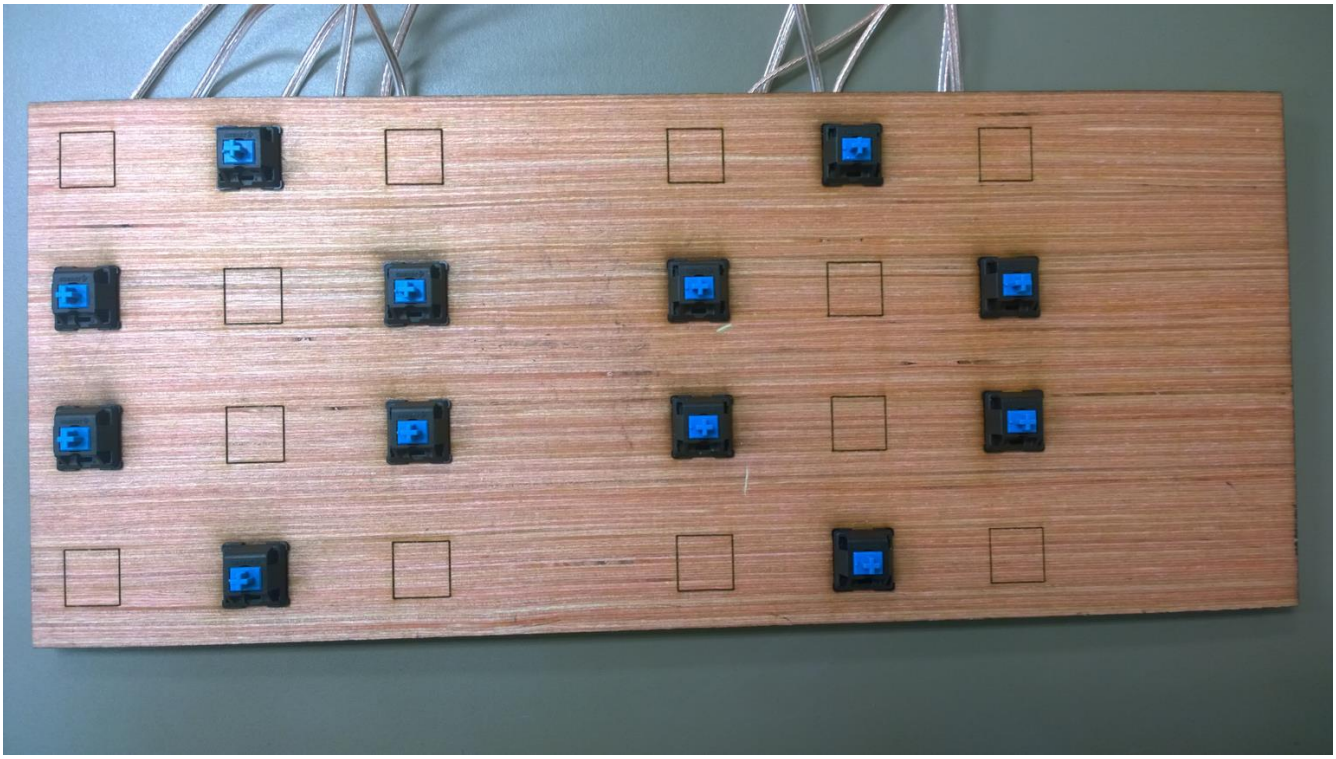
when considering reviews from UCP of Huntsville; the sliding keyboard also served as the primary test bed for developing the predictive text application. As it is central to the conclusion of the project, it is the focus of the prototyping section.

Perhaps the most intriguing aspect of the project was the wealth of redesign and improvement that occurred throughout the scope of the timeline. The original specification for the sliding keyboard was based on a hexagonal design, with the contact points for the switches located in the corners. The hexagon would be arranged so that one corner was at the top and bottom, while there would be two to each side. An early model image is given below.

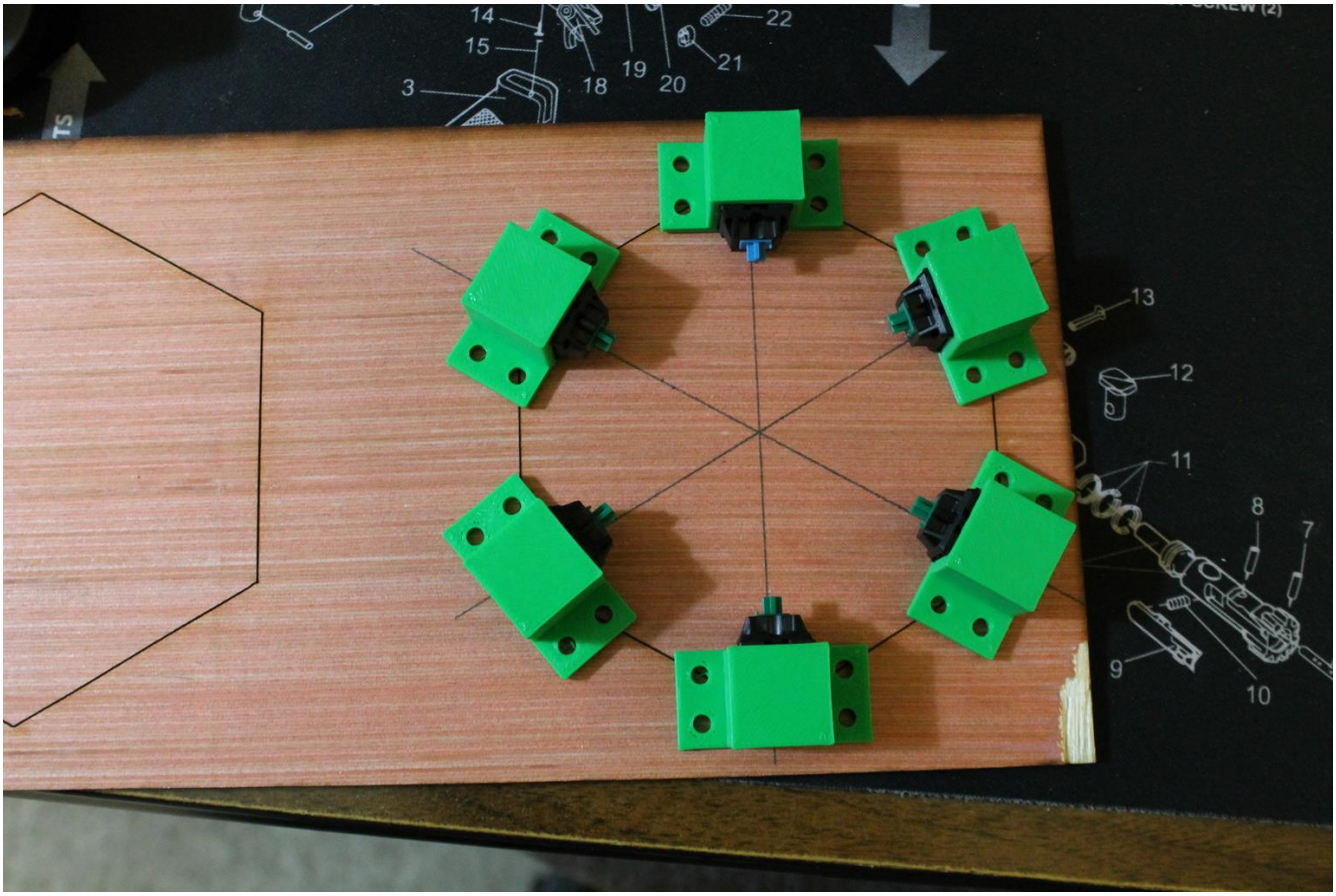


Using the model as a basis for going forward, a simple prototype was constructed. This

prototype phase was built using a laser-cut plywood frame to house the switches, with all of the contact points being somewhat messily funneled into the MSP430. This bare-bones version served as a good visual aide for demonstrating our intent to UCP of Huntsville, as well as allowing for preliminary evaluation of potential ergonomic downfalls. The main goal was to act as a blueprint, and provide physical re-assurance that the project was (at the time) on schedule. The result is displayed here:

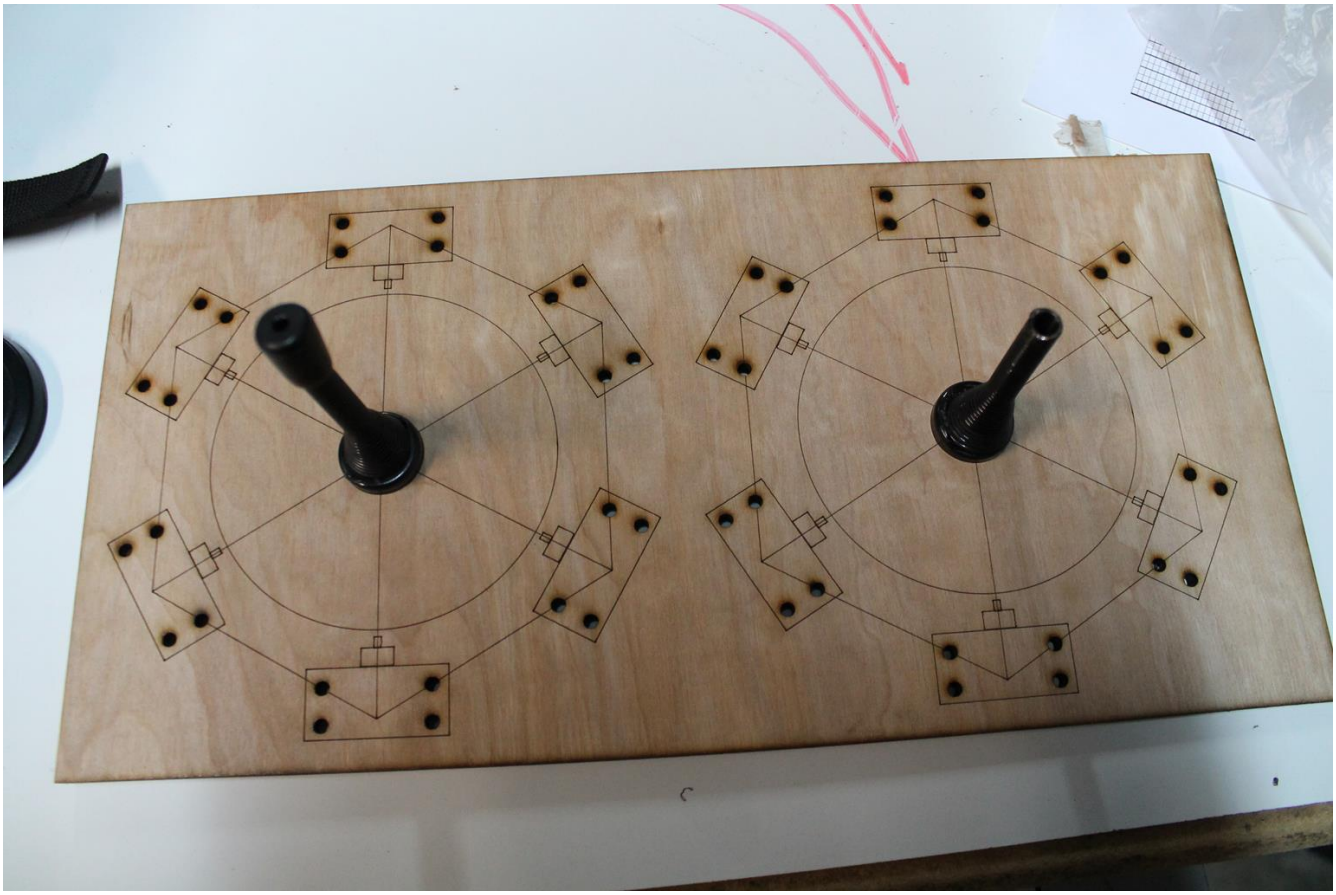


After the switch layout was determined using the previous model, the version was then improved. The switches were adjusted from a standard keyboard slot (embedded into the floor of the device, facing upwards), and were instead placed in 3D printed braces the would allow the switches to be placed upright. In this design, the switches would face inwards towards the sliding paddle, rather than face upward and require a more complicated contact scheme. Shown below is the intermediate phase when the braces had been finished and the switches were slotted and in the process of being measured out on the keyboard:

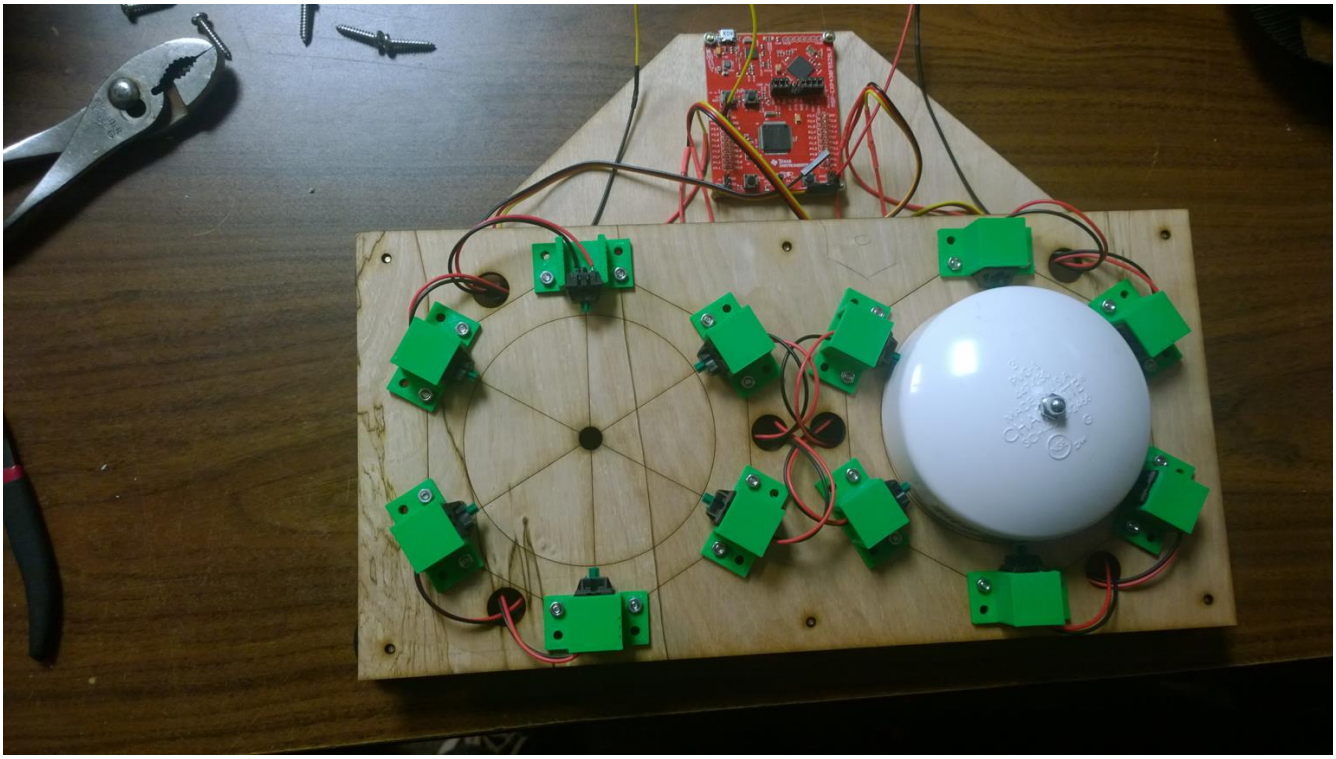


Once the switch placement issue was resolved, the need for the sliding button was approached. As one of the primary tenets of the project was to reduce cost, a remarkably affordable solution was devised. The button itself was prototyped using a standard PVC pipe cap. The shape and size were ideal, the cost is appropriately inexpensive, and availability is thankfully high due to the abundance of home improvement and hardware stores. The issue of movement was a tad more difficult to resolve. The main problem was finding a solution for the range of motion that did not conflict with our design goals. Springs were considered fairly early on in when trying to resolve the issue: regrettably that would necessitate much higher precession when aligning and installing the springs, which conflicts with the simplicity of design aspect. Elastic materials were dismissed, as the lifetime is rather poor, and they would not respond to stress in an even manner, causing the paddle to develop odd movement habits. Thankfully, a

rather clever solution was provided in the form of a door stopper. When placed upright in the center of the paddle, the stopper provides an excellent source of returning tension so the paddle can come back to rest once a key is pressed. The design means it is resistant to compression (ideal since it becomes the backbone of the paddle), and the resistance of the paddle can be adjusted by purchasing different door stoppers (again high availability and low cost) or by simple tampering with a set of pliers. It provided a perfect solution: simple, cheap, and very effective. The iterations of these additions are displayed below. First, the door stopper core:

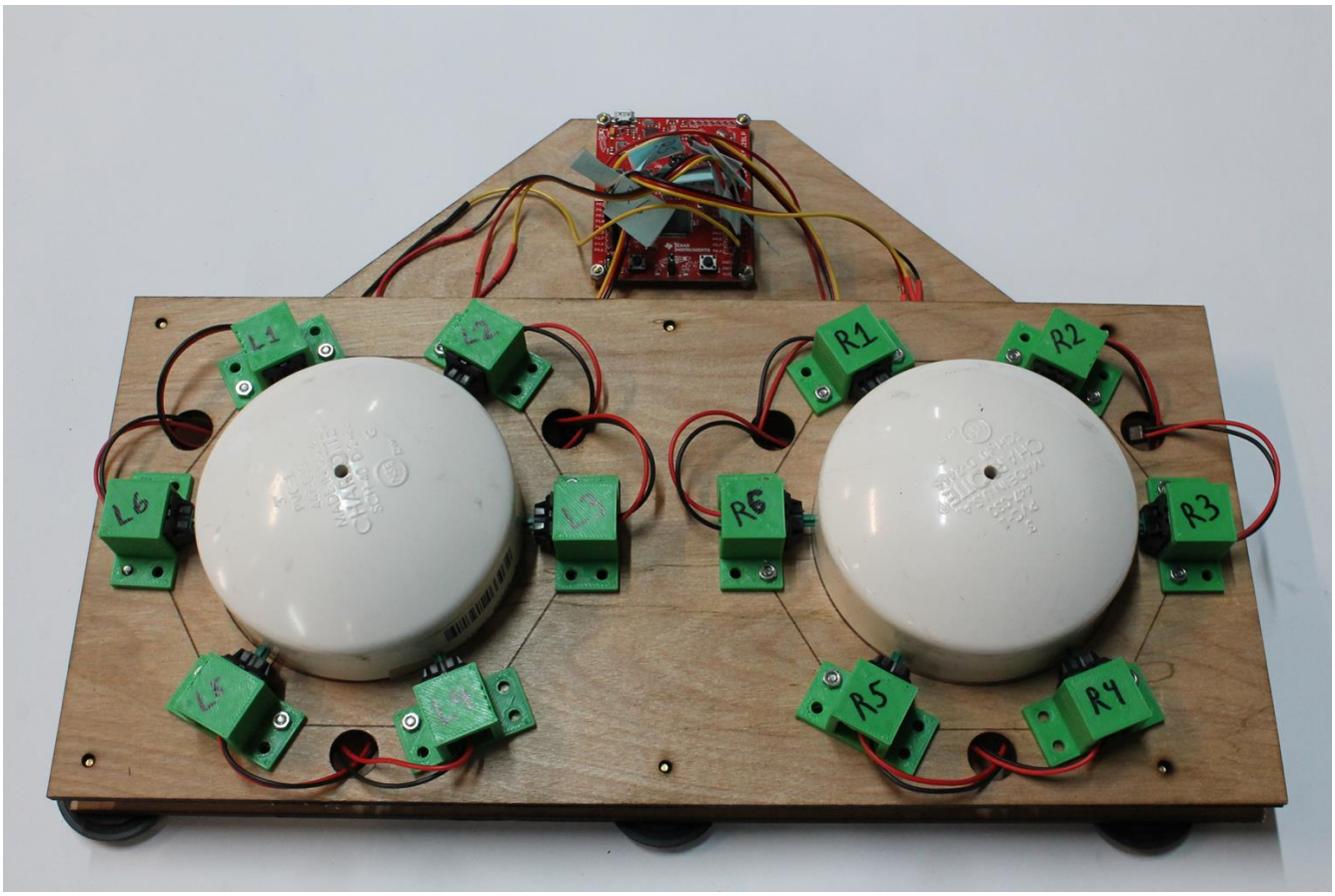


Then the PVC paddle:

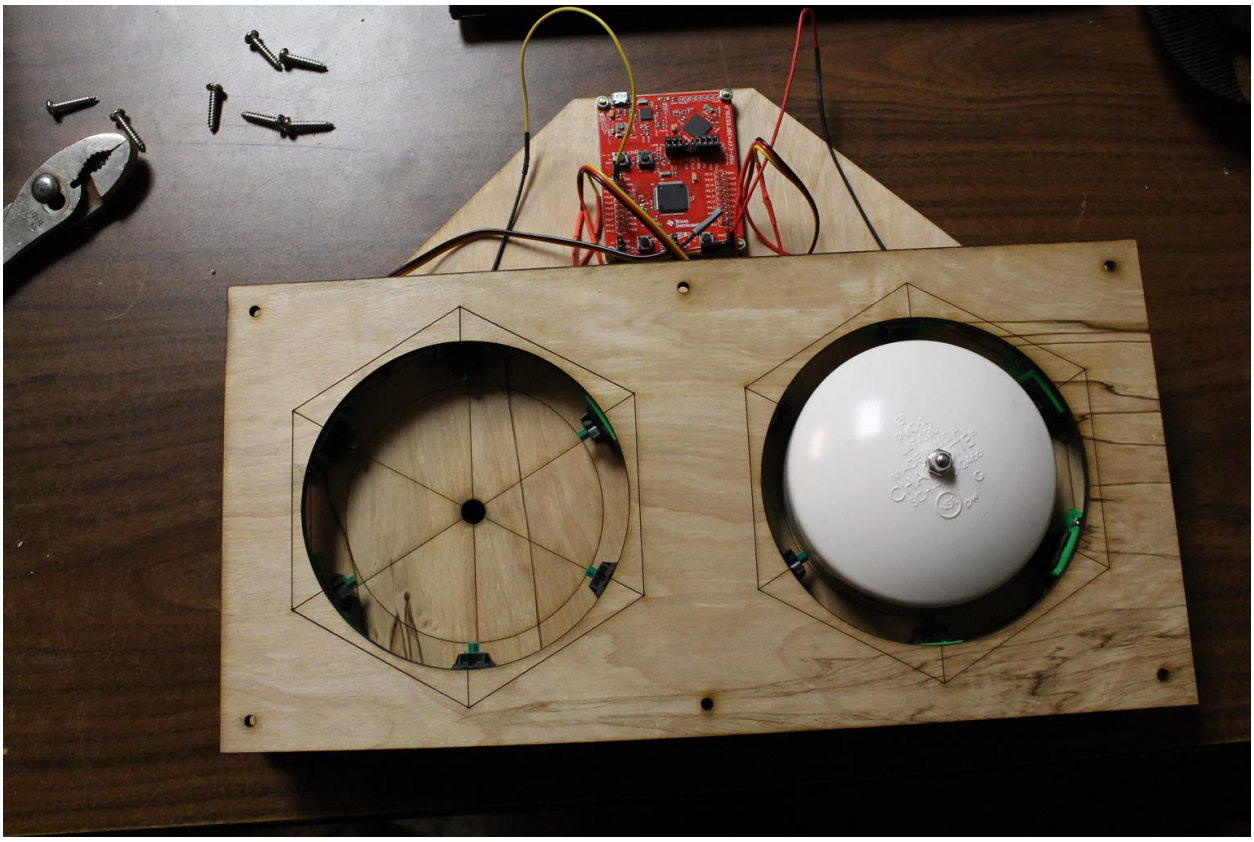


Further evaluation of the design led to an issue with giving consistent input. With the original placement of the switches, the user had to get the paddle onto the center of the switch to activate the key. Since the design is supposed to cater towards users with motor disabilities, there seemed to be room for improvement. The next major iteration involved rotating the position of the switches around the perimeter of the paddle. To further increase usability, the nature of what constituted valid input was changed. In the new setup, a user needed to maneuver the paddle into one of six “channels” created by the gap between two switches. This meant that valid input was now established by having the paddle make contact with two adjacent switches. The net result was making contact with the paddle became easier, and positioning of what counted as input became slightly more intuitive. Further improvements were made to better accommodate all of the required hardware: the wiring was compartmentalized in a layer below where the user interacts (being funneled out to the top where the MSP430 is placed) and the switch bracers are

concealed by a final layer, leaving the user with just the paddles to interact with. Posted below is the keyboard with the new positioning, paddles attached, and the bottom layer concealing most of the wiring:



Followed by a shot of the triple layer design that conceals any extraneous hardware from the user:



Detailed Analysis of Software Functionality

The application takes input from the user using a textbox with a listener attached. This serves the purpose of listening for keystrokes. When a key is pressed, the application looks for the text the user is currently typing and presumes it is a partially completed word. The program then takes this word and asks a SQLite database for the 5 most likely completions to this word. The selected candidates are chosen based on frequency from the user, as well as alphabetical proximity to the letters already input so far.

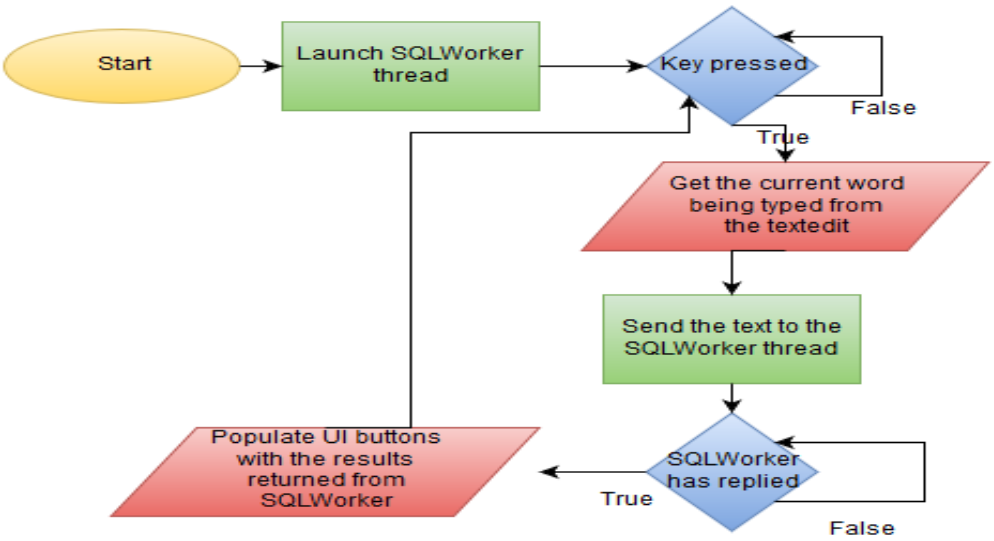
The program consists of two primary objects: the listener is in the main object, while sql queries are handled by another object. This object is run on a child thread, allowing the user to continue using the application even while the sql object is waiting for a reply from the database. Within the main object, once the word to be completed is obtained, it is forwarded to the sql object and waits for it to reply with the five results. Once the potential candidates are calculated,

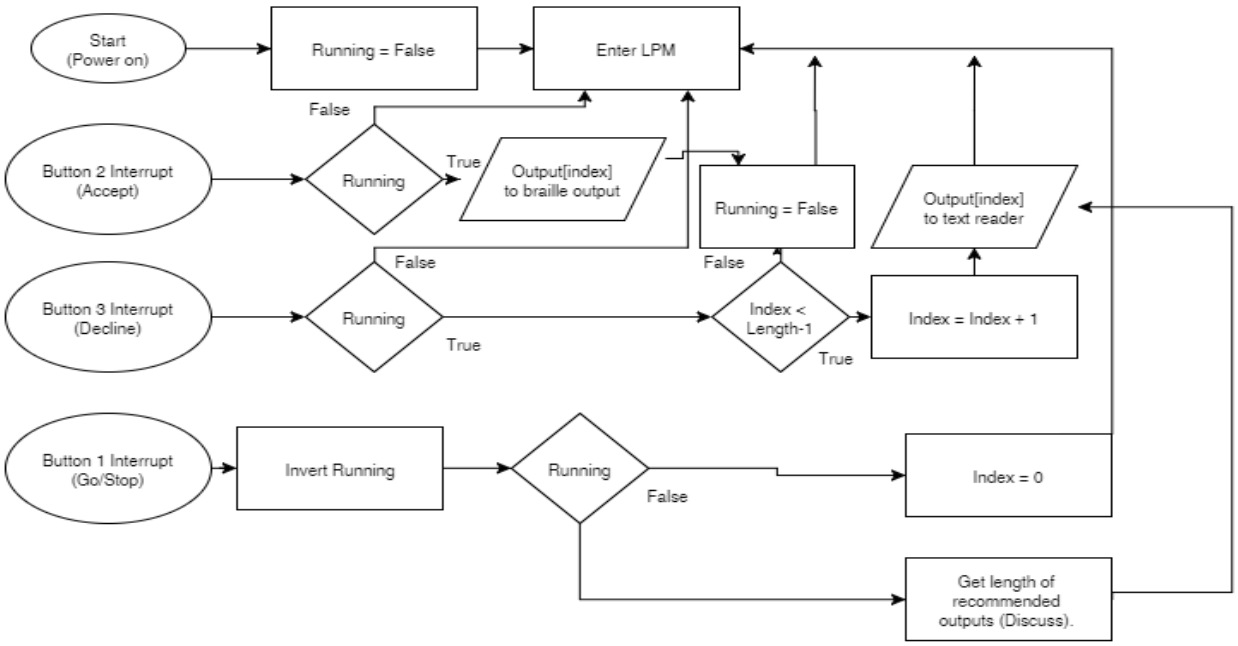
each answer is posted to a button for the user to choose from. The sql object receives the partially completed word and stores it in a variable 'arg' and runs the following query: "SELECT WORD FROM dictionary WHERE WORD LIKE '" + arg + "%' ORDER BY COUNT DESC LIMIT 5;". The result is then parsed into an array of 5 strings and is sent to the main object.

The database consists of two columns: WORD and COUNT. The WORD column contains each word that can be recommended to the user, while the COUNT column determines which word is used more often than another. More specifically COUNT contains how many times that word has been chosen as the intended word and the word with the highest COUNT is considered to be the most likely to be used. Using this algorithm, the application will begin to learn what words really are more common for each user and will become more and more useful. The database is populated with the English dictionary and COUNT is initiated using various online articles so even on initial launch the application has a general idea of what words a random sampling of users prefers as their most used words.

Software Design Flow Chart

Below are flowcharts documenting the interface for the predictive text application, as well as the logical flow of the program when interacting with the user:





Software Development Obstacles

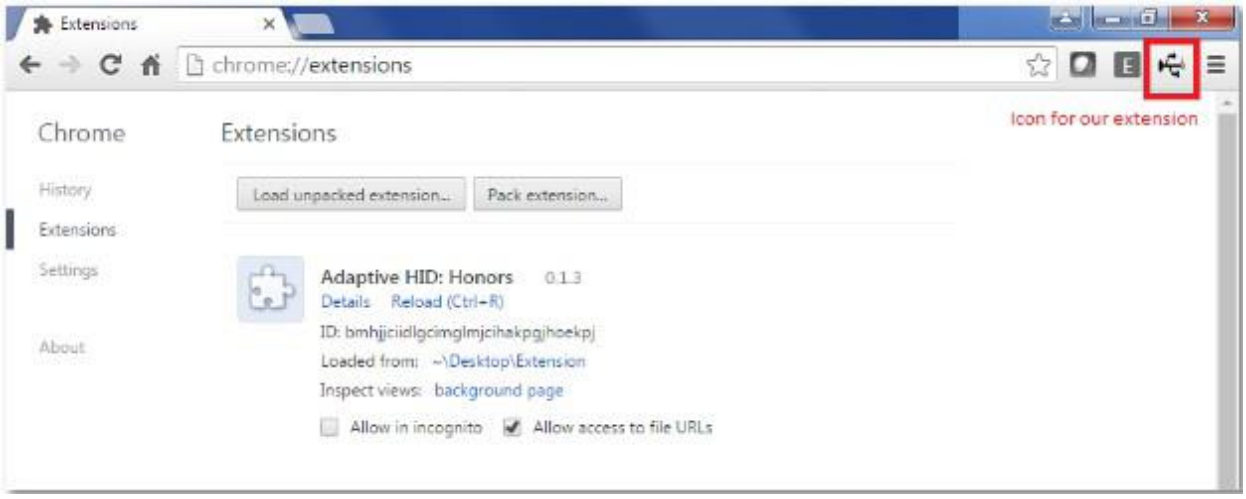
One of the more difficult aspects faced in creating the software package was the programming language. A key aspect of the project design was adaptability. To that end, it was imperative that the software package be compatible with whatever setup the user had for their personal computer. The prime choice for this goal was JavaScript; this was the first time either team member had interacted with JavaScript. This caused significant issues in the early stages of development, as each adapted to the new foibles of a foreign language. Once the adjustment was over, progress began to pick up.

Once the primary language was decided, the next step was to decide where to place the majority of the program's logic. The decision boiled down to attempting to place the core functionality in JavaScript and simply pipe information through the interface (be it a web browser, or when it is running within the keyboard), or dedicate the majority of work to the interface. To decide on this, a prototype for each method was developed: one using the Google API, the other focusing purely on JavaScript. Once each was evaluated, we decided to stick with

placing the key functionality within JavaScript, as it would allow the program to ported as an extension to multiple web-browsers with very little difficulty, which more closely adheres to the design goal of adaptability.

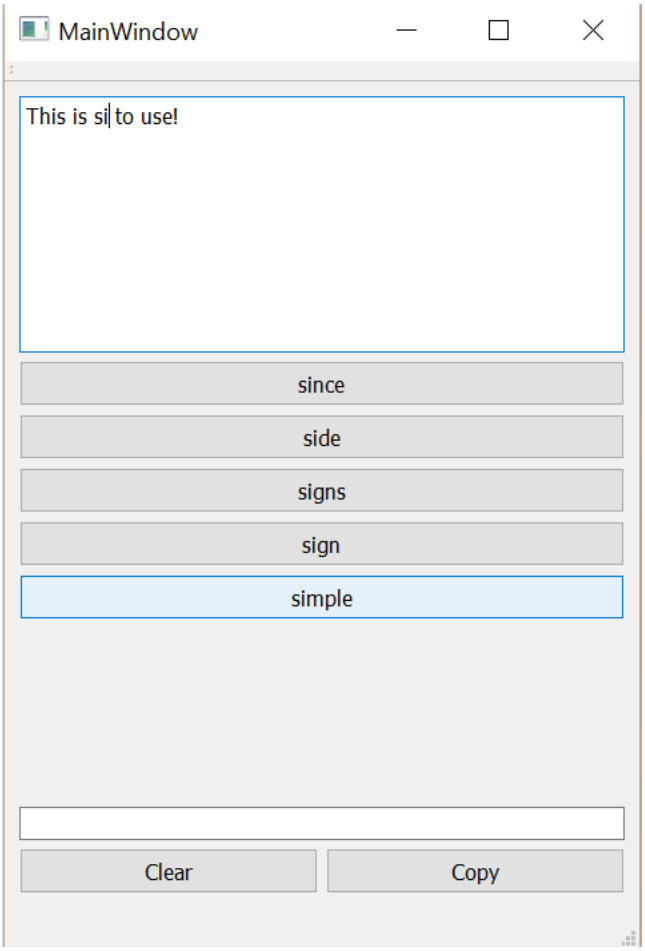
Once an html page was working, the next step was needing a way for the application to funnel data from a SQL database towards the application layer. The original attempt had the web interface reach into the SQL table and grab the information (the predicted word choice). While this was simple and straightforward, it presented a somewhat major security hazard in allowing the web browser to access data stored by the computer. To compensate for this, an open source JavaScript file called “Websql” was utilized. Using this allowed the program to create what amounted to a SQL database within the web browser, preventing the security hazard. One issue using this method is that the database created within the web-browsers *sometimes* gets deleted upon closing. To compensate for this required an overhaul of the code to support re-creating the framework with records of the user's habits still intact.

One of the prime features desired for the program was to create an easy to implement software package. The ideal target therefore was a web extension: naturally this led to difficulty to interfacing with Google API. A Chrome extension was chosen as the test-bed, as it was the most likely candidate to be used by our targeted user base. Although there were setbacks, the final version was successfully attached to a Chrome application. Listed below is a small photograph of the finished extension as it is viewed when accessing it from the web.



Combined Product Operation

The goal of the honors project was to enhance the Adaptive HID device. Below shows a user typing a sentence and the application automatically suggesting words for the user to input.



The program allows for the user to increase the speed at which words are inputted while learning their vocabulary habits and attempting to improve delivery speed by guessing user input. The final hardware product has two paddles, each with six points of contact. Both of the paddles slide to type a character as keyboard input. The paddles do not rotate about the z-axis (they do not twist or turn). Each paddle slides independent of the other paddle, though both paddles need to make contact for valid input to be considered. Making contact with the left paddle allows the right paddle to select from a group of six valid characters. Making a different selection on the left paddle causes different options to be made by the right paddle.

Global and Societal Impact of Project

Environmental Factors

After a thorough review of the source materials, the environmental impact of this project has been deemed negligible. Components of the system are all readily available for purchase in the United States and have passed all pertinent consumer regulations. Hardware, when discarded, should be disposed of based on the relevant regulations. No notably hazardous materials are involved with any of the proposed builds, and none of the methods of manufacturing impose considerable risk.

Security Concerns

Security vulnerabilities can be introduced when a person decides to build our design with modifications to the microcontroller firmware. Since many existing keyboard kits available on the market use similar microcontrollers and firmware approaches, this keyboard is adequately secure in respect to the scope of modern keyboards. Our project is open source, and if another person was to use our design and implement a key logger into the source code for the keyboard, they could potentially steal information inputted by another user. We plan to counteract this

possibility by providing source and binary checksums while requesting all potential developers build the project directly from source or from a trusted authority.

Compatibility

We are designing our hardware to be interoperable to conform to existing standards (USB 1.1, 2.1 CDC and HID). Legacy and emerging computer systems should allow for the keyboard to be operational via the USB interface.

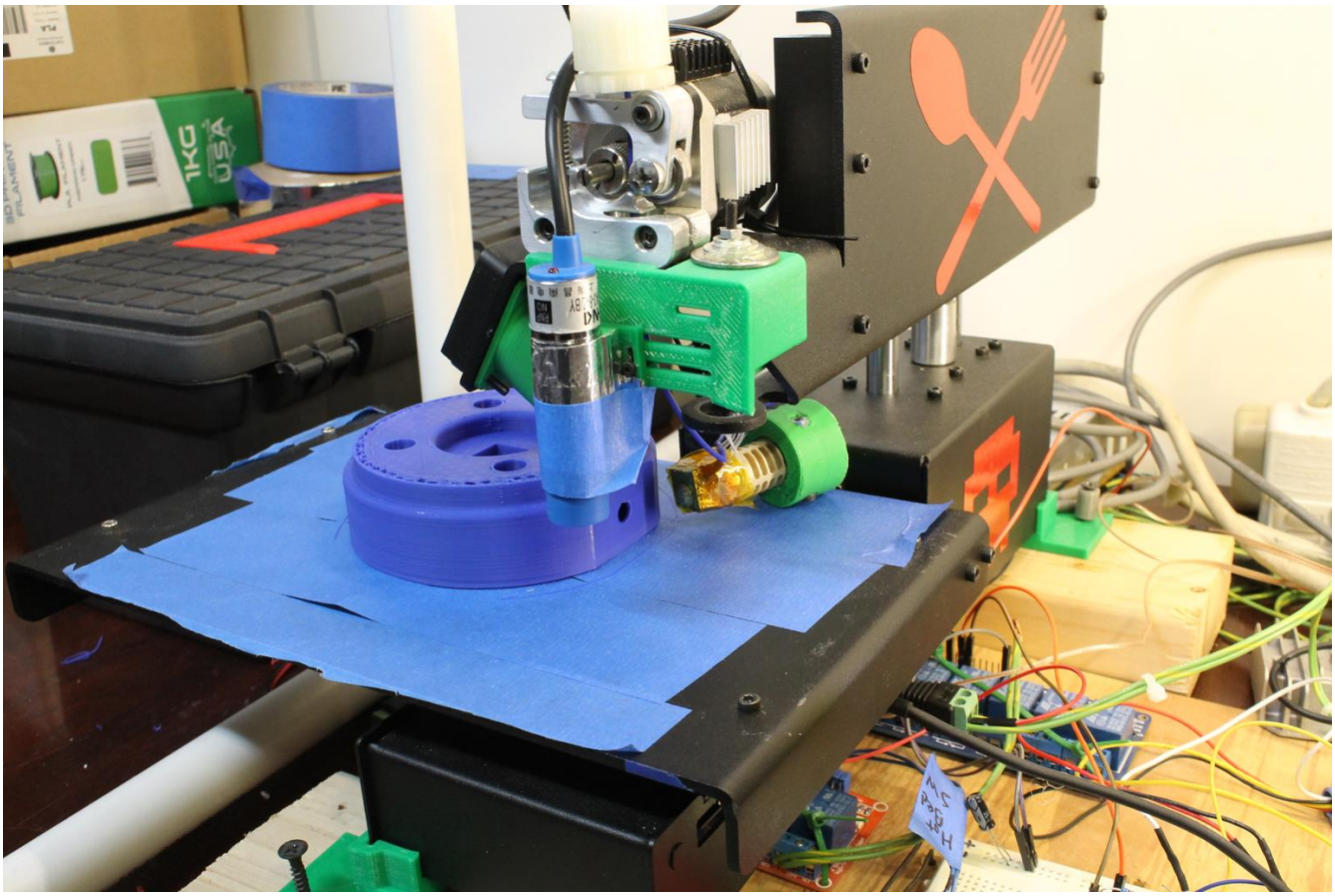
Privacy Concerns

With respect to personal privacy, our prospective desktop software design plans be a learning program that will remember most commonly used phrases in order to help predict the user's word input and sentences. We will have to take ethical measures to ensure the raw or generated data taken is with consent and not distributed to unauthorized parties. We will also review later designs to implement fuzzing techniques which prevent the extraction of valid writings and communications of the user.

Safety During Assembly

Health and safety concerns are minimal as the keyboard is constructed of nontoxic material. As far as the development of our product, we are prototyping with a chemical laser bed and we have to ensure proper ventilation and eye protection is used during operation. The 3D printers are printing with polylactic acid (PLA) thermoplastics that are food grade, thus posing limited risk and no noxious fumes during extrusion. Furthermore, we are soldering with leadless solder and are ensuring that proper safety measures are taken in the lab.

Shown below is a photograph of the 3D printer maintained during the project as it prints out one of the AT buttons. The setup is fairly basic, one that can reasonably be expected at most any makerspace.



Legal Concerns

In regard to regulatory and legal issues, we are using MSP430 TI USB API Stack which includes the driverlib package. Both are clearly licensed under BSD compliant licenses via the following [http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430_USB_Developers_Package/latest/index_FDS.html]. MSP430 hardware is subject to export restrictions for certain packages, but we are not crossing state or country lines and are not mailing the hardware systems.

Code Appendix

Standalone Predictive Text Application

Mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <windows.h>
#include <QClipboard>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QThread *thread = new QThread();
    myWorker = new SQLWorker;
    myWorker->moveToThread(thread);
    thread->start();
    tc = new QTextCursor;
    connect(ui->plainTextEdit, SIGNAL(cursorPositionChanged()), this, SLOT(mySlot()));
    connect(myWorker, SIGNAL(finished(QString*, int)), this, SLOT(print(QString*, int)));
    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(press0()));
    connect(ui->pushButton_2, SIGNAL(clicked()), this, SLOT(press1()));
    connect(ui->pushButton_3, SIGNAL(clicked()), this, SLOT(press2()));
    connect(ui->pushButton_4, SIGNAL(clicked()), this, SLOT(press3()));
    connect(ui->pushButton_5, SIGNAL(clicked()), this, SLOT(press4()));
    connect(ui->pushButton_6, SIGNAL(clicked()), this, SLOT(pressE()));
    connect(ui->pushButton_7, SIGNAL(clicked()), this, SLOT(pressC()));


    myWorker->doWork("");
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::print(QString *argv, int argc)
{
    ui->pushButton->setText("");
    ui->pushButton_2->setText("");
    ui->pushButton_3->setText("");
    ui->pushButton_4->setText("");
    ui->pushButton_5->setText("");
    switch(argc)
    {
    case 5:
    ui->pushButton_5->setText(argv[4]);
    case 4:
    ui->pushButton_4->setText(argv[3]);
    case 3:
    ui->pushButton_3->setText(argv[2]);
```

```
    case 2:
    ui->pushButton_2->setText(argv[1]);
    case 1:
    ui->pushButton->setText(argv[0]);
    }
}

void MainWindow::mySlot()
{
    *tc = ui->plainTextEdit->textCursor();
    tc->select(QTextCursor::WordUnderCursor);
    myWorker->doWork(tc->selectedText());
}

SQLWorker::SQLWorker()
{
    db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("test.db");
}

SQLWorker::~SQLWorker(){}

void SQLWorker::doWork(QString arg)
{
    if (!db.open())
        QMessageBox::warning(0, "Failed to open", "Failed to open");
    QString *out = new QString[5];
    const char *str = arg.toStdString().c_str();
    bool trm = false;
    for (int i = 0; i < arg.length(); i=i+1)
    {
        switch (str[i])
        {
        case '\'':
            qDebug("Boom");
            break;
        }

    }
    QString sql = "SELECT WORD FROM dictionary WHERE WORD LIKE '" + arg + "%' ORDER BY
COUNT DESC LIMIT 5;";
    QSqlQuery query(db);
    query.exec(sql);
    if (!query.isActive())
        QMessageBox::warning(0, "Bad Query", QString("Bad Query, It's
inactive:%1").arg(query.lastError().text()));
    else
    {
        int i = 0;
        while (query.next())
        {
            out[i] = query.value(0).toString();
            i = i + 1;
        }
        finished(out, i);
    }
    db.close();
}
```

```
void MainWindow::press0()
{
    if (ui->pushButton->text() != "")
        tc->insertText(ui->pushButton->text());
    ui->plainTextEdit->setFocus();
}

void MainWindow::press1()
{
    if (ui->pushButton_2->text() != "")
        tc->insertText(ui->pushButton_2->text());
    ui->plainTextEdit->setFocus();
}

void MainWindow::press2()
{
    if (ui->pushButton_3->text() != "")
        tc->insertText(ui->pushButton_3->text());
    ui->plainTextEdit->setFocus();
}

void MainWindow::press3()
{
    if (ui->pushButton_4->text() != "")
        tc->insertText(ui->pushButton_4->text());
    ui->plainTextEdit->setFocus();
}

void MainWindow::press4()
{
    if (ui->pushButton_5->text() != "")
        tc->insertText(ui->pushButton_5->text());
    ui->plainTextEdit->setFocus();
}

void MainWindow::pressE()
{
    ui->plainTextEdit->clear();
}

void MainWindow::pressC()
{
    QClipboard *temp = QApplication::clipboard();
    temp->setText(ui->plainTextEdit->toPlainText());
    ui->plainTextEdit->clear();
}

bool MainWindow::eventFilter(QObject *target, QEvent *event)
{
    if (target == ui->lineEdit) // This is the input line
    {
        if (event->type() == QEvent::KeyPress) // This is a keypress
        {
            QKeyEvent *key = static_cast<QKeyEvent*>(event);
            switch(key->key())
            {
            case 39:
```

```
                break;
            }
        }
        else
        {
            return false;
        }
    }
    else
    {
        return QMainWindow::eventFilter(target, event);
    }
}
```

Main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

Mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QtSql/QtSql>
#include <QMessageBox>
#include <QKeyEvent>
#include <QDebug>
#include <QTextCursor>

namespace Ui {
class MainWindow;
}

class SQLWorker : public QObject
{
    Q_OBJECT
public:
    SQLWorker();
    ~SQLWorker();
public slots:
    void doWork(QString arg);
signals:
```

```
    void finished(QString *argv, int argc);
private:
    QSqlDatabase db;
};

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    SQLWorker *myWorker;
    QTextCursor *tc;

protected:
    bool eventFilter(QObject *target, QEvent *event);

private slots:
    void print(QString *argv, int argc);
    void press0();
    void press1();
    void press2();
    void press3();
    void press4();
    void pressE();
    void pressC();
    void mySlot();
};

#endif // MAINWINDOW_H
```

Mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>476</width>
    <height>665</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <layout class="QVBoxLayout" name="verticalLayout">
    <item>
     <widget class="QPlainTextEdit" name="plainTextEdit">
```

```
   <property name="plainText">
    <string/>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QPushButton" name="pushButton">
   <property name="text">
    <string/>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QPushButton" name="pushButton_2">
   <property name="text">
    <string/>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QPushButton" name="pushButton_3">
   <property name="text">
    <string/>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QPushButton" name="pushButton_4">
   <property name="text">
    <string/>
   </property>
  </widget>
 </item>
 <item>
  <widget class="QPushButton" name="pushButton_5">
   <property name="text">
    <string/>
   </property>
  </widget>
 </item>
 <item>
  <spacer name="verticalSpacer">
   <property name="orientation">
    <enum>Qt::Vertical</enum>
   </property>
   <property name="sizeHint" stdset="0">
    <size>
     <width>20</width>
     <height>64</height>
    </size>
   </property>
  </spacer>
 </item>
 <item>
  <widget class="QLineEdit" name="lineEdit"/>
 </item>
 <item>
  <layout class="QHBoxLayout" name="horizontalLayout">
```

```
      <item>
       <widget class="QPushButton" name="pushButton_6">
        <property name="text">
         <string>Clear</string>
        </property>
       </widget>
      </item>
      <item>
       <widget class="QPushButton" name="pushButton_7">
        <property name="text">
         <string>Copy</string>
        </property>
       </widget>
      </item>
     </layout>
    </item>
   </layout>
  </widget>
  <widget class="QMenuBar" name="menuBar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>476</width>
     <height>31</height>
    </rect>
   </property>
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
  </widget>
  <widget class="QStatusBar" name="statusBar"/>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Textcompletion.pro

```
#-------------------------------------------------
#
# Project created by QtCreator 2015-12-07T12:12:47
#
#-------------------------------------------------

QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4):
QT += widgets
QT += sql
```

```
TARGET = TextCompletion
TEMPLATE = app


SOURCES += main.cpp\
        mainwindow.cpp

HEADERS  += mainwindow.h

FORMS    += mainwindow.ui
```

Predictive Text Web extension

Popup.html

```
<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title>My Popup!</title>
</head>
<body>
    <h1>Text Completion</h1>
    <input type="text" name="txtIn" id="txtIn" value="start"><br>
    <p>1:<button id="demo"></button></p>
    <p>2: <button id="demo1"></button></p>
    <p>3: <button id="demo2"></button></p>
    <p>4: <button id="demo3"></button></p>
    <p>5: <button id="demo4"></button></p>
    <script src="WebSQL.js"></script>
    <script type="text/javascript" src="mypopup.js"></script>
</body>
       </html>
```

Mypopup.js

```
document.addEventListener("DOMContentLoaded", function () {
var db = WebSQL('test.db');
console.log(db);
db.query("DROP TABLE Dictionary");
var init = "CREATE TABLE IF NOT EXISTS Dictionary(WORD varchar(255) UNIQUE, COUNT int)";
db.query(init)
        .fail(function (tx, err) {
            console.log(err);
        }).done(function (products) {
            len = products[0];
            db.query(init)
                .fail(function (tx, err) {
                    console.log(err);
                }).done(function (products) {
                    len = products[0];
                    myKeypress();
                });
```

```
        });
document.getElementById("txtIn").addEventListener("keyup", function () {
    myKeypress();
});
function myKeypress() {
    var len = 0;
    var input = "SELECT COUNT(*) count FROM Dictionary WHERE word LIKE '" +
document.getElementById("txtIn").value + "%'";
    db.query(input)
        .fail(function (tx, err) {
            console.log(err);
        }).done(function (products) {
            len = products[0].count;
        });
    input = "SELECT word FROM Dictionary WHERE word LIKE '" +
document.getElementById("txtIn").value + "%' ORDER BY COUNT DESC LIMIT 5";
    console.log(input);
    db.query(input)
        .fail(function (tx, err) {
            console.log(err);
        }).done(function (products) {
            console.log(products[0]);
            document.getElementById("demo").innerHTML = "NULL";
            document.getElementById("demo1").innerHTML = "NULL";
            document.getElementById("demo2").innerHTML = "NULL";
            document.getElementById("demo3").innerHTML = "NULL";
            document.getElementById("demo4").innerHTML = "NULL";
            if (len > 0) {
                document.getElementById("demo").innerHTML = products[0].WORD;
                if (len > 1) {
                    document.getElementById("demo1").innerHTML = products[1].WORD;
                    if (len > 2) {
                        document.getElementById("demo2").innerHTML = products[2].WORD;
                        if (len > 3) {
                            document.getElementById("demo3").innerHTML =
products[3].WORD;
                            if (len > 4) {
                                document.getElementById("demo4").innerHTML =
products[4].WORD;
                            }
                        }
                    }
                }
            }
        });
}
function executeCopy(text) {
    console.log("Hello World");
    var input = document.createElement('textarea');
    document.body.appendChild(input);
    input.value = text;
    input.focus();
    input.select();
    document.execCommand('Copy');
    input.remove();
}
document.getElementById("demo").addEventListener("click", function () {
    executeCopy(document.getElementById("demo").innerHTML);
```

```
});
document.getElementById("demo1").addEventListener("click", function () {
    executeCopy(document.getElementById("demo1").innerHTML);
});
document.getElementById("demo2").addEventListener("click", function () {
    executeCopy(document.getElementById("demo2").innerHTML);
});
document.getElementById("demo3").addEventListener("click", function () {
    executeCopy(document.getElementById("demo3").innerHTML);
});
document.getElementById("demo4").addEventListener("click", function () {
    executeCopy(document.getElementById("demo4").innerHTML);
});
       });
```

Manifest.json

```
{
    "manifest_version": 2,
    "name": "Adaptive HID: Honors",
       "permissions": [
                "browsingData",
          "unlimitedStorage",
                "clipboardRead",
                "clipboardWrite"
    ],
    "version": "0.1.3",
    "browser_action": {
      "default_icon": "icon.png",
      "default_popup":  "popup.html"
    },
    "content_scripts": [
        {
            "matches": [
                "<all_urls>"
            ],
            "js": [ "WebSQL.js", "mypopup.js" ]
        }
    ]
       }
```