Hands-free Computer Interaction Using a Sip/Puff Sensor as a Mouse

Daniel Williams

Computer Engineering Department

California Polytechnic State University, San Luis Obispo

December 2015

© 2015 Daniel Williams
Contents

1 Executive Summary 3

2 Introduction 4
  2.1 Formal Problem Definition ................................. 4
  2.2 Specifications .............................................. 4

3 Design Development 6
  3.1 System Black Box Diagrams ................................. 6
  3.2 Microcontroller ........................................... 7
    3.2.1 State Machine Diagram ................................. 8
  3.3 Computer Interpreter ...................................... 8

4 Final Design 11
  4.1 Final Microcontroller Design .............................. 11
    4.1.1 State Machine Design ................................. 11
  4.2 Final Computer Software Design ............................ 12
  4.3 Design Verification Plan ................................. 13
  4.4 Design Verification Plan Tests ............................ 14
    4.4.1 MSP powered by USB ................................. 14
    4.4.2 Python software connects to MSP ..................... 15
    4.4.3 Mouse direction X ................................. 15
    4.4.4 Mouse direction Y ................................. 15
    4.4.5 Mouse direction X - Reversed ........................ 15
    4.4.6 Mouse direction Y - Reversed ........................ 16
    4.4.7 Mouse direction reverse at bounds of screen .......... 16
    4.4.8 Left mouse click ................................... 16
    4.4.9 Right mouse click ................................... 17
    4.4.10 Double mouse click .................................. 17
    4.4.11 Sip/Puff switch sends ............................... 17
  4.5 Final System Assembly .................................... 18
  4.6 Design Verification Results ............................... 19

5 Conclusions 20

A Python Source Code 21

B MSP 430 Source Code 26
List of Figures

1. Level 0 Black Box Diagram ........................................ 6
2. Level 1 Black Box Diagram ........................................ 6
3. The circuit that was used for testing and development of the system behavior. . 7
4. Initial design for pattern matching state machine .......................... 8
5. The software flow for the client computer interpreter ....................... 9
6. The translation handler method flow .................................... 10
7. The final finite state machine design for the MSP 430 microcontroller. . . . 11
8. The state machine for the timer interrupt handler ....................... 12
9. The state machine for the pattern matcher that handles direction inversion. . . . 12
10. The next tick time equation ........................................... 13
11. The revised translation handler method flow ............................ 13
12. Design Verification Plan tests ......................................... 14
13. Troubleshooting circuit for sip/puff switch output .......................... 18
14. Development Circuit used for testing .................................... 32
15. Detail of pins wiring for development circuit ............................ 33
16. Origin Instruments Sip/Puff switch ..................................... 33
17. Detail of Sip/Puff switch output ports .................................. 34

List of Tables

1. System Requirements .................................................. 5
2. System Specifications .................................................. 5
3. Patterns for moving the mouse ........................................ 7
4. Patterns for using the mouse ........................................... 7
5. Command characters ................................................... 9
6. Table describing results of design verification tests ..................... 19
7. Bill of materials ....................................................... 32
Statement of Disclaimer

Since this project is a result of a class assignment, it has been graded and accepted as fulfillment of the course requirements. Acceptance does not imply technical accuracy or reliability. Any use of information in this report is done at the risk of the user. These risks may include catastrophic failure of the system or infringement of patent or copyright laws. California Polytechnic State University at San Luis Obispo and its staff cannot be held liable for any use or misuse of the project.

1 Executive Summary

This is the final project report for the Hands-Free Mouse, which details the system design with information regarding the design choices and justifications, testing verification and procedures, and the chosen components for the project’s implementation. These chapters describe background information about the project, hardware and software design, verification and testing, and concludes with the current status of the project and what future work may involve.
2 Introduction

The goal of this project is to create a system which can function as a mouse, and behaves as one transparently to the computer and operating system it is attached to. The motivating factor behind this project was to create a low cost system, and an alternative to existing products that have the same function. This system allows those that are unable to use a mouse to interact with a computer.

Currently, there are several classes of systems that perform this function. Broadly speaking, they are divided into two categories. The first, the sip-and-puff, which uses air pressure to send control signals to a computer, which then interprets those signals to modify the computer’s state. There are two types of inputs into these devices, sip (inhale) and puff (exhale). These types of devices are commonly used by quadriplegics and those who have experienced significant injury to their spinal cord.\(^1\)

The second class of products uses eye-tracking technology. This is a more recent technological development, that often uses computer vision to measure eye positions and movement. These products can involve elaborate rigs to hold the head steady so that the eye is always in view, goggle-like devices that fit over the face, or even consumer-level web cameras that are integrated with laptops. The last of these products holds the most promise for being low-cost (or free\(^2\)).

For this project, the sip-and-puff technique was chosen for its simplicity in implementation and non-intrusive use. There is no need for head rigs, only a straw that connects to the sip-puff sensor. Additionally, this solution works without a webcam.

### 2.1 Formal Problem Definition

The end objective for this project is to create a system that behaves as a mouse device when connected to a computer. This “mouse” system will be controlled with a straw that is connected to a sip/puff sensor switch. This sensor switch can detect either a sip or puff, and creates an electrical signal that is output to one of the channels of the integrated 3.5 mm stereo jack.

A stereo cable will connect this switch to a MSP 430 microcontroller. The two channels will be wired to two different GPIO pins on the microcontroller. The microcontroller will receive patterns of sips and puffs, and by using a state machine will determine the command that will be sent to the computer. The software on the computer will consist of a user-level (not a device driver) program that listens to a serial port and will modify the mouse position based on the commands it receives.

### 2.2 Specifications

The system requirements are tabulated in table 2.2 that are needed for the integrated system to function in its intended use. The nominal system specifications are listed in table 2.

---

\(^1\)http://atwiki.assistivetech.net/index.php/Alternative_wheelchair_control

\(^2\)http://www.cameramouse.org/ - Camera Mouse was developed at Boston College, and is able to use any webcam.
### 2.2 Specifications

#### Parameter Description

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control methods</td>
<td>Sip or puff</td>
</tr>
<tr>
<td>Horizontal Motion (X)</td>
<td>Long Sip (&gt; 0.25 s)</td>
</tr>
<tr>
<td>Vertical Motion (Y)</td>
<td>Long Puff (&gt; 0.25 s)</td>
</tr>
<tr>
<td>Invert X Direction</td>
<td>Puff, Sip</td>
</tr>
<tr>
<td>Invert Y Direction</td>
<td>Sip, Puff</td>
</tr>
<tr>
<td>Left Mouse Click</td>
<td>Short Sip (&lt; 0.25 s)</td>
</tr>
<tr>
<td>Right Mouse Click</td>
<td>Short Puff (&lt; 0.25 s)</td>
</tr>
<tr>
<td>Double Click</td>
<td>2 short sips</td>
</tr>
</tbody>
</table>

Table 1: System Requirements

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of mouse control</td>
<td>100% of any single monitor</td>
</tr>
<tr>
<td>Command Latency</td>
<td>0.25 s</td>
</tr>
<tr>
<td>Power</td>
<td>3.3 V</td>
</tr>
<tr>
<td>Communication</td>
<td>UART</td>
</tr>
<tr>
<td>Baud Rate</td>
<td>9600</td>
</tr>
</tbody>
</table>

Table 2: System Specifications
3 Design Development

The overall system consists of three components: the sensor, the MSP 430 Launchpad, and the computer. The sip puff sensor provides the MSP 430 controller with electrical signals indicating whether a sip or puff was just detected. The MSP 430 then processes the input according to the state machine in figure 4 to determine what command to send to the computer. The MSP 430 controller is connected to the computer via USB, and communicates using UART over that connection. The computer program receives the commands from the MSP 430 and changes the position of the mouse accordingly.

The system is divided such that the mouse behavior parameters (speed, etc.) can be modified without having to recompile the microcontroller source code. The microcontroller determines the user’s intent, and the client software translates those intentions into actions.

3.1 System Black Box Diagrams

The following black box diagrams show how the system is connected together.

In figure 1, the top level of the system is shown to illustrate how each component is connected to each other. The MSP 430 interfaces with the sip/puff switch and the computer. The computer receives commands from the MSP 430 on how to manipulate the current mouse position, which the MSP 430 generates by interpreting the signals received from the sip/puff switch.

![Figure 1: Level 0 Black Box Diagram](image1)

In figure 2, a more detailed description of the system is shown. The sip/puff switch has a 3.5 mm stereo jack that provides the sip and puff events with a single stereo cable. The stereo cable has the left and right channels broken out into separate wires, which connect to two distinct ports on the MSP 430. The MSP 430 and computer communicate over a USB connection, using UART. The communication between the MSP 430 and the computer is undirectional from the MSP 430 to the computer; the MSP 430 can only send. Any data the MSP 430 receives will be ignored.

![Figure 2: Level 1 Black Box Diagram](image2)

The circuit that was used for development and testing of the MSP 430 is shown in figure 3. The push buttons (PushButton1 and PushButton2) simulate the control signals that would be received from the output of the sensor’s 3.5mm stereo jack. Nodes P1.4 and P1.5 represent IO.
pin connections to the MSP 430. In their default state with no button pressed, the nodes are at $V_{cc}$. When one of the buttons is pushed, the node is connected via a short directly to ground. This falling edge triggers an interrupt in the MSP 430, which is handled by the state machine shown in figure 4.

![Circuit Diagram](image)

Figure 3: The circuit that was used for testing and development of the system behavior.

### 3.2 Microcontroller

The initial design for the microcontroller (MSP430) was to take a series of sips and puffs which would be processed by a pattern-matching finite state machine, as shown in figure 4. The threshold for a long sip or puff was set to be 1 seconds, meaning that at each state there would be a 1 second delay before transitioning to the next state based on that input.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sip, Long Sip</td>
<td>Left</td>
</tr>
<tr>
<td>Sip, Long Puff</td>
<td>Right</td>
</tr>
<tr>
<td>Puff, Long Sip</td>
<td>Down</td>
</tr>
<tr>
<td>Puff, Long Puff</td>
<td>Up</td>
</tr>
</tbody>
</table>

Table 3: Patterns for moving the mouse

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sip, Sip</td>
<td>Double Click</td>
</tr>
<tr>
<td>Puff, Puff</td>
<td>Right Click</td>
</tr>
</tbody>
</table>

Table 4: Patterns for using the mouse

The first action was designed to be a mode setting, setting X or Y motion, and the second action the direction to move. However, this design requires two discrete actions for each translation, and was determined to be too many actions to be usable.
A clock speed of 1 MHz was used to use the least amount of power since the MSP 430 is not performing any ongoing tasks or computations. The MSP 430 could be put into the low-power sleep mode, only to be woken by interrupts from the ports.

### 3.2.1 State Machine Diagram

The following state machine show how the state of the controller changes according to the input it receives, and when commands are sent. Commands to start moving the mouse position are sent once; the client-side computer program will continue to move the mouse until a stop command has been received.

![State Machine Diagram](image)

**Figure 4:** Initial design for pattern matching state machine

### 3.3 Computer Interpreter

The MSP 430 sends individual uppercase characters to represent the motion commands expected by the client software, terminated by a carriage return and a newline. Based on the received characters, the software takes action to manipulate the current position of the mouse.

This software is written in Python to allow for cross-platform compatibility. The mouse control is achieved using the PyUserInput library\(^3\), which has bindings for Windows, Mac, and Linux operating systems.

\(^3\)https://github.com/SavinaRoja/PyUserInput - A module for cross-platform control of the mouse and keyboard.
<table>
<thead>
<tr>
<th>Letter</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>move mouse up</td>
</tr>
<tr>
<td>D</td>
<td>move mouse down</td>
</tr>
<tr>
<td>L</td>
<td>move mouse left</td>
</tr>
<tr>
<td>R</td>
<td>move mouse right</td>
</tr>
<tr>
<td>C</td>
<td>left mouse click</td>
</tr>
<tr>
<td>G</td>
<td>right mouse click</td>
</tr>
<tr>
<td>S</td>
<td>stop all motion</td>
</tr>
</tbody>
</table>

Table 5: Command characters

The client software is designed to open a serial port and block while waiting for a new character to be received. There is an event handler for each of the commands listed in table 5, and the event is dispatched using the received character.

![Software flow diagram]

The high level software flow is shown above in figure 5. The received character is used as a look-up into a hash table of functions; if the received character has an attached function, then

Figure 5: The software flow for the client computer interpreter.
the function is executed with the character as its parameter. If the received character has no associated event handler, no action is taken.

![Diagram of the translation handler method flow.](image)

Figure 6: The translation handler method flow.

The same event handler is used for all mouse translation methods (up, down, left, and right), and the software flow for this method is shown in figure 3.3. A series of vectors were created to hold the x and y amounts to add to the current mouse position. While the stop command has not been received from the MSP 430, the software will start a new timer for the move method to be raised again in the near future. This avoids inefficient busy waiting and requires no additional handling to receive commands during this process.

Further changes to the final design are discussed in section 4.2.
4 Final Design

The design changes between the initial and final designs are documented in the following section, along with design verifications plans for testing the final design.

4.1 Final Microcontroller Design

The final microcontroller design was significantly modified to reduce the number of user actions required to control the mouse. This was accomplished by eliminating the notion of the mode setting, and replacing it with a time-based system of control. The time that a sip or puff occurs changes the action that is taken; however, this does introduce a latency before an action can be taken. To minimize this latency without introducing errors, the threshold for the decision was set to 250 milliseconds. The redesigned (and simplified) state machine for handling this aspect of the mouse behavior can be seen in figure 7.

A timer interrupt is used to handle this threshold time. However, because the timer register is only 16 bits, it is impossible to use a single timer event to count up to the needed time. To solve this, a second counter variable is used to track the number of the times the timer interrupt is raised. The timer interrupt counter now only counts up to 50,000 (instead of the previous 65,535), and the second counter counts up to 5. When the second counter reaches 5, the microcontroller sends out a mouse move command. This behavior can be seen in figure 8.

The final source code for the microcontroller can be seen on page 26.

4.1.1 State Machine Design

![State Machine Design](image)

Figure 7: The final finite state machine design for the MSP 430 microcontroller.

\[ \text{At a 1 MHz clock speed, there would 250,000 clock cycles to count (250 ms * 1 MHz) - far more than the 16 bit register can track.} \]
4.2 Final Computer Software Design

There were two primary changes to the design of the software from the initial design phase.

The first of these changes was the modifying of the mouse translation speed as a function of the movement time. Previously, the mouse would move \( \frac{1 \text{ pixel}}{1 \text{ ms}} \); however, this was difficult to move small distances accurately. This was solved by using the equation shown below in figure 10, a piecewise function where \( n \) represents the number of ticks since the mouse translation began.

To handle small movements, the event is raised every 250 milliseconds for the first two seconds, every 10 milliseconds for the next two seconds, and 1 millisecond thereafter. This was designed to approximate mouse acceleration.\(^5\)

\(^5\)Mouse acceleration changes relationship between the on-screen crosshair and the mouse. If the mouse is moved slowly for 1 inch, the crosshair would move a set amount. However, if the mouse if moved quickly for 1 inch, the crosshair would move farther than when moved slowly.
\[ f(n) = \begin{cases} 
0.25 & : n < 8 \\
0.01 & : 8 < n < 16 \\
0.001 & : otherwise
\end{cases} \]

Figure 10: The next tick time equation.

The second change was to reflect the mouse if an edge of the screen was encountered. This was done to prevent unnecessary direction inversion commands from being issued. Each time the bounds of the screen are reached, the direction vector is inverted. This inverted behavior lasts until the stop command is received, and then is reset to the normal behavior.

The final source code for this program is shown on page 21.

4.3 Design Verification Plan

This section describes the tests used for verifying the system operation and design.
<table>
<thead>
<tr>
<th>No</th>
<th>Specification</th>
<th>Test Description</th>
<th>Acceptance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MSP powered by USB</td>
<td>Tests USB power is sufficient for MSP 430</td>
<td>MSP green LED turns on</td>
</tr>
<tr>
<td>2</td>
<td>Python software connects to MSP</td>
<td>Software will start and connect automatically to the MSP 430</td>
<td>Software receives commands from the MSP</td>
</tr>
<tr>
<td>3</td>
<td>Mouse direction X</td>
<td>Mouse moves when sip occurs</td>
<td>Mouse begins to move in X direction</td>
</tr>
<tr>
<td>4</td>
<td>Mouse direction Y</td>
<td>Mouse moves when puff occurs</td>
<td>Mouse begins to move in Y direction</td>
</tr>
<tr>
<td>5</td>
<td>Mouse direction reversed X</td>
<td>Mouse direction reversed when pattern for reversing entered</td>
<td>Mouse begins to move in opposite X direction</td>
</tr>
<tr>
<td>6</td>
<td>Mouse direction reversed Y</td>
<td>Mouse direction reversed when pattern for reversing entered</td>
<td>Mouse begins to move in opposite Y direction</td>
</tr>
<tr>
<td>7</td>
<td>Mouse direction reversed when edges of screen reached</td>
<td>Mouse direction reversed when edge of screen is encountered</td>
<td>Mouse begins to move in opposite X or Y direction</td>
</tr>
<tr>
<td>8</td>
<td>Left click</td>
<td>Single left click occurs when pattern for left click entered</td>
<td>Mouse left clicks at current mouse position</td>
</tr>
<tr>
<td>9</td>
<td>Right click</td>
<td>Single right click occurs when pattern for right click entered</td>
<td>Mouse right clicks at current mouse position</td>
</tr>
<tr>
<td>10</td>
<td>Double click</td>
<td>Two consecutive left clicks are entered</td>
<td>Mouse double clicks at current mouse position</td>
</tr>
<tr>
<td>11</td>
<td>Sip/Puff switch sends signals</td>
<td>Sip &amp; puff events are observed electrically on the output of the 3.5 mm jack</td>
<td>Oscilloscope views electrical output of signals</td>
</tr>
</tbody>
</table>

Figure 12: Design Verification Plan tests

4.4 Design Verification Plan Tests

For all following tests, the MSP 430 is assumed to be on serial port 3 (COM3). This may change on other systems, so the `–port` parameter may change as well. The sip port is P1.4, and the puff port is P1.5.

4.4.1 MSP powered by USB

**Determine:** Tests USB power is sufficient for MSP 430

**Preconditions:** USB 2.0 cable, MSP 430, and computer with USB port

**Procedure:**
4.4 Design Verification Plan Tests

1. Plug-in USB 2.0 cable to computer.
2. Plug-in USB 2.0 mini-connector to MSP 430.
3. Observe that green power LED (PWR LED0) is lit on MSP 430.

4.4.2 Python software connects to MSP

**Determine:** Tests if Python software can connect to MSP

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit

**Procedure:**

1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: `python mouse.py -port COM3`.
3. Push one of the buttons in the development circuit.
4. Observe that the Python program has received the command character.

4.4.3 Mouse direction X

**Determine:** Tests if mouse movement commands in the X direction are recognized correctly

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit or sip/puff switch

**Procedure:**

1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: `python mouse.py -port COM3`.
3. Push and hold for longer than 250 ms the sip button in the development circuit or sip using the sip/puff switch.
4. Observe that the mouse has begun to move in the X direction, left if freshly started.
5. Release the button and observe that the mouse has stopped moving.

4.4.4 Mouse direction Y

**Determine:** Tests if mouse movement commands in the Y direction are recognized correctly

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit

**Procedure:**

1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: `python mouse.py -port COM3`.
3. Push and hold for longer than 250 ms the puff button in the development circuit or puff using the sip/puff switch.
4. Observe that the mouse has begun to move in the Y direction, up if freshly started.
5. Release the button and observe that the mouse has stopped moving.

4.4.5 Mouse direction X - Reversed

**Determine:** Tests if mouse movement in the X direction is reversed

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit

**Procedure:**
4.4 Design Verification Plan Tests

1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: python mouse.py –port COM3.
3. Push the sip button, puff button, and sip button again in sequence, each for less than 250 ms.
4. Push and hold the sip button for longer than 250 ms.
5. Observe that the mouse is now moving right.

4.4.6 Mouse direction Y - Reversed

**Determine:** Tests if mouse movement in the Y direction is reversed

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit

**Procedure:**
1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: python mouse.py –port COM3.
3. Push the puff button, sip button, and puff button again in sequence, each for less than 250 ms.
4. Push and hold the puff button for longer than 250 ms.
5. Observe that the mouse is now moving down.

4.4.7 Mouse direction reverse at bounds of screen

**Determine:** Tests if mouse movement direction changes at the edges of the screen.

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit or sip/puff switch

**Procedure:**
1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: python mouse.py –port COM3.
3. Push and hold the sip button, waiting until the mouse pointer has reached the edge of the screen.
4. Observe that the mouse continues to move, but in the reverse X direction.
5. Release the sip button.
6. Push and hold the puff button, waiting until the mouse pointer has reached the edge of the screen.
7. Observe that the mouse moves in the reverse Y direction.
8. Release the puff button.

4.4.8 Left mouse click

**Determine:** Tests if a left mouse click can be generated.

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit or sip/puff switch

**Procedure:**
1. Open a terminal and navigate to the directory where mouse.py is located.
2. Execute the following command: python mouse.py –port COM3.
3. Push the sip button (holding for less than 250 ms).
4. Observe that the computer has received a left click.
(a) This can be confirmed by moving the mouse over an inactive window (the window would appear gray or muted in appearance). When performing step 3, observe that the window is no longer gray or muted in appearance.

4.4.9 Right mouse click

**Determine:** Tests if a right mouse click can be generated.

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit or sip/puff switch

**Procedure:**
1. Open a terminal and navigate to the directory where `mouse.py` is located.
2. Execute the following command: `python mouse.py -port COM3`.
3. Push the sip button (holding for less than 250 ms).
4. Observe that the computer has received a right click, by seeing a right click contextual menu appear.

4.4.10 Double mouse click

**Determine:** Tests if a two left mouse clicks are treated as a double click.

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit or sip/puff switch

**Procedure:**
1. Open a terminal and navigate to the directory where `mouse.py` is located.
2. Execute the following command: `python mouse.py -port COM3`.
3. Push the sip button twice in sequence, each for less than 250 ms.
4. Observe that a double click has been received.
   (a) This can be done by moving the mouse over a program icon on the computer desktop. Double clicking one of these will launch the program.

4.4.11 Sip/Puff switch sends

**Determine:** Tests if a two left mouse clicks are treated as a double click.

**Preconditions:** USB 2.0 cable, MSP 430, computer with USB port, and development circuit or sip/puff switch

**Procedure:**
1. Open a terminal and navigate to the directory where `mouse.py` is located.
2. Execute the following command: `python mouse.py -port COM3`.
3. Push the sip button twice in sequence, each for less than 250 ms.
4. Observe that a double click has been received.
   (a) This can be done by moving the mouse over a program icon on the computer desktop. Double clicking one of these will launch the program.
4.5 Final System Assembly

Development was done using a simple two button circuit connected to the MSP 430, in lieu of the sip/puff switch. This was done due to ease of assembly and facilitated rapid development. However, when the sip/puff switch was arrived for integration into the overall system. Due to the lack of a complete datasheet regarding the operation of the Origin Instruments Sip/Puff switch\(^6\), the only data that could be found was that the two pressure switches are closed when an event is observed, and the switches are passive requiring no external power. Additonally, the switches carry 40 mA of current during operation.

Unfortunately, due to the lack of a complete datasheet and time constraint, integration of the switch was not achieved. Initially, the one of the stereo output channels was wired directly to a LED test circuit that would turn the LED on when the switch was closed. This did not work as expected. Additionally, the circuit shown figure 13 to attempt to have the output of the switch turn on the base-emitter junction, thereby turning on the LED (not shown in this figure). This also did not work.

Using an oscilloscope attached to one of the channels and ground on the cable, there was no seen change in the voltage output when one of the test buttons was pressed on the device. Connections were checked multiple times, and on both channels of the output cable, to no avail. It appears that the sip/puff switch used was either defective or being intergrated incorrectly. Without a complete datasheet, it will be difficult to know which of these is the correct case.

---

\(^6\)http://www.orin.com/access/sip_puff/ - Sip/Puff Switch™ section header
### Design Verification Results

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Dev Circuit</th>
<th>Sip/Puff Switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>6</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>7</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>8</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>9</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>10</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>11</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 6: Table describing results of design verification tests.
5 Conclusions

The goal of the project was to create a system for controlling a mouse for a user who is unable to use a computer mouse. The idea is a low-cost, flexible solution that allows for configuration of its parameters, a separation between the microcontroller and software components, and non-intrusive - needing only a straw to control. The separation of tasks was desired to separate the intent of the user from the execution of that intent, and make configuration simpler without needing to recompile and reprogram the microcontroller.

Recommendations for possible future work involve fully determining the operation of the sip/puff switch. Due to time constraints, the switch was not able to successfully integrated into the overall system. Testing and verification of electrical components can be time-consuming, so time allocated for performing these tasks is essential early in the project. The modular nature of the software design would aid in modifying the system, if needed, to support the sip/puff switch. The microcontroller’s limited tasks allow for increased flexibility in the client software. One possible feature that was considered that could be implemented would be varying the resolution of movement based on the dimensions of the monitors, reducing movement time. Additionally, during system idle time, the MSP430 could be placed into the lowest power mode achievable. The microcontroller would then be woken up into a higher power to handle any interrupts.

Overall, this project provided many opportunities for understanding how to develop interdependent embedded systems and computer software. The work that has been completed provides a solid foundation for continuing work in the future.
Listing 1: Client-Side Command Interpreter

```python
#!/usr/bin/python

# Client-side software for controlling mouse based on
# commands received from MSP 430 interpreter and sip/puff
# switch

# Author: Daniel Williams

from __future__ import print_function
import sys
import threading
import serial
from functools import wraps
import socket
import argparse
from pymouse import PyMouse

# Global state variables
REPEATER = None
RESOLUTION = 2
handlers = {}
mouse = PyMouse()
WIDTH, HEIGHT = mouse.screen_size()
TICK_COUNT = 0
REVERSE = RESOLUTION

class Vector(object):
    # Class for vector operations

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, oth):
        return Vector(self.x + oth.x, self.y + oth.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __rmul__(self, scalar):
        return self * scalar

    def tuple(self):
        return self.x, self.y

DIR_VECTOR = {
    'U': Vector(0, -1),
    'D': Vector(0, 1),
    'L': Vector(-1, 0),
    'R': Vector(1, 0),
    'U+R': Vector(1, -1),
    'U-L': Vector(-1, -1),
    'D+R': Vector(1, 1),
    'D-L': Vector(-1, 1),
    'U+L': Vector(1, 1),
    'U-R': Vector(-1, 1),
    'D+L': Vector(-1, -1),
    'D-R': Vector(1, -1),
    'L+R': Vector(1, 0),
    'L-D': Vector(0, 1),
    'R+U': Vector(1, 1),
    'R-D': Vector(0, -1),
    'R+L': Vector(-1, 0),
    'R-U': Vector(1, 0),
    'L+D': Vector(0, -1),
    'L-U': Vector(0, 1)}
```

A PYTHON SOURCE CODE
def handler(*commands):
    ""
    decorator to hold command handlers to be called when command is received
    ""
    def wrap(func):
        @wraps(func)
        def enter_exit(cmd):
            return func(cmd)
        for cmd in commands:
            handlers[cmd] = func
        return enter_exit
    return wrap

def wait_timer(tick):
    ""
    Implements piecewise function used for controlling mouse translation speed
    
    \[
    f(n) = \begin{cases} 
    0.25 & : n < 8 \\
    0.01 & : 8 < n < 16 \\
    0.001 & : \text{otherwise}
    \end{cases}
    \]
    ""
    print(' ', tick)
    if tick < 8:
        return 0.25
    if tick < 16:
        return 0.01
    return 0.001

def translate(direction):
    ""
    Moves mouse pointer in the requested direction, and then starts a thread timer to re-raise the event.
    
    If the bounds of the screen are encountered, then the motion direction is reversed from the current direction.
    
    The event is raised continually until 'S' is received.
    ""
    global REPEATER
    global TICK_COUNT
    global REVERSE
    pos = Vector(*mouse.position())
    new_pos = pos + REVERSE * DIR_VECTOR[direction]
    mouse.move(*new_pos.tuple())
stop = (new_pos.x < 0 or new_pos.y < 0 or
        new_pos.x > WIDTH or new_pos.y > HEIGHT)

if stop:
    REVERSE *= -1

REPEATER = threading.Timer(wait_timer(TICK_COUNT), lambda:
    translate(direction))
REPEATER.start()
print('move direction: %s' % direction, file=sys.stderr)
TICK_COUNT += 1

@handler('U', 'D', 'L', 'R')
def move_handler(cmd):
    global REPEATER

    # kill current event if another one is
    # received, this could be the a runaway
    if REPEATER is not None:
        REPEATER.cancel()
        REPEATER = None
    translate(cmd)

@handler('S')
def stop_handler(cmd):
    ""
    Stops all mouse motion by cancelling the current event,
    and resetting the tick counters and direction.
    ""
    global REPEATER
    global TICK_COUNT
    global REVERSE

    TICK_COUNT = 0
    REVERSE = RESOLUTION

    if REPEATER is not None:
        REPEATER.cancel()
        REPEATER = None
    print('stop translation', file=sys.stderr)

@handler('C')
def click_handler(cmd):
    ""
    Gets current mouse position, and
    presses and release mouse to create
    left mouse click.
    ""
    mouse.press(*mouse.position())
    mouse.release(*mouse.position())
    print('left click', file=sys.stderr)

@handler('G')
def rclick_handler(cmd):

A PYTHON SOURCE CODE

170
171     ""
172     Gets current mouse position, and presses
173     and releases right mouse button to create
174     a right mouse click.
175     ""
176     mouse.press(*mouse.position(), button=2)
177     mouse.release(*mouse.position(), button=2)
178     print('right click', file=sys.stderr)
179
180
def handle_data(data):
181     ""
182     Retrieves event handler for specified command
183     (stripped of newline) and executes it.
184     ""
185     method = handlers.get(data, lambda c: None)
186     method(data)
187
188
def read_from_port(port):
189     return port.readline()
190
191
def read_from_sock(sock):
192     return sock.recv(3)
193
194
def main(args):
195     # can debug by listening on UDP socket 50000, and sending
196     # characters commands directly
197     if args.socket:
198         port = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
199         port.bind(('localhost', 50000))
200         read_from = read_from_sock
201         print('DEBUG MODE (UDP port 50000)...', file=sys.stderr)
202     else:
203         try:
204             port = serial.Serial(args.port)
205         except serial.serialutil.SerialException as e:
206             print(e, file=sys.stderr)
207             sys.exit(1)
208         read_from = read_from_port
209         print('LIVE MODE (%s)...' % args.port, file=sys.stderr)
210
211     # center mouse before starting
212     mouse.move(HEIGHT / 2, WIDTH / 2)
213
214     # block and read forever
215     while True:
216         cmd = read_from(port).strip()
217         print('received: %s' % cmd, file=sys.stderr)
218         handle_data(cmd)
219
220     if __name__ == '__main__':
221         # creates argument parser to receive requested
222         # configuration parameters
223         #
```python
args = argparse.ArgumentParser()
args.add_argument('-s', '--socket',
    help='debug using UDP socket 50000',
    action='store_true'
)
args.add_argument('-p', '--port',
    help='port to listen on',
    default='COM3'
)
options = args.parse_args()
main(options)
```
#ifndef SIPPUFF_H_
#define SIPPUFF_H_

#define SIPP BIT4
#define PUFFP BIT5
#define SIP_SHORT BIT0
#define SIP_LONG BIT1
#define PUFF_SHORT BIT2
#define PUFF_LONG BIT3

/* (CC0_NUM * CLICK_THRESH) / (CLK_SPEED) = threshold time
   * (50000 * 5) / (1 MHz) = 250 ms = 0.25 s
*/
#define CC0_NUM 50000
#define CLICK_THRESH 5

#define RXD BIT1
#define TXD BIT2
#define BUTTON BIT3
#define LED BIT0

#define EnableTimerA() { CCR0 = CC0_NUM; CCTL0 = CCIE; }
#define DisableTimerA() { CCR0 = CC0_NUM; CCTL0 &= ~CCIE; }

typedef struct {
    char x;
    char y;
} Direction;

typedef enum {
    MOVE_LEFT='L',
    MOVE_RIGHT='R',
    MOVE_UP='U',
    MOVE_DOWN='D',
    CLICK='C',
    RIGHT_CLICK='G',
    STOP='S'
} Command;

typedef enum {
    WAITING,
    SIP_F,
    PUFF_F
} State;

typedef enum {
    INIT,
    /* sippuff.h*/
    * Author: Daniel Williams
*/
#endif

Listing 2: Header file for MSP 430 source code
Listing 3: MSP 430 Microcontroller Source Code

/*
 * MSP 430 microcontroller software for processing
 * signals received from the sip/puff switch for use
 * in the hands-free mouse project. Processes patterns
 * of sips and puffs using state machine to determine
 * which command to send.
 * Author: Daniel Williams
 */

#include <msp430g2553.h>
#include <stdbool.h>
#include "sippuff.h"

/* Global state variables */
static State CurrentState = WAITING;
static int InterruptCount;
static Direction Dir = {false, false};
static PatternState PState = INIT;

void UART_TX(char * tx_data);
void send_command(Command);

void main(void)
{
    /* Disable watchdog timer */
    WDTCTL = WDTPW + WDTHOLD;
    /* 16 Mhz SMCLK */
    if (CALBC1_16MHZ == 0xFF) /* If calibration constant erased */
    {
        while (1)
            /* do not load, trap CPU!! */
    }
    /* Set DCO to 1 MHz */
    BCSCTL1 = CALBC1_1MHZ;
    DCOCCTL = CALDCO_1MHZ;
    /* setup timer for later use */
    CCR0 = 65000;
    TACTL = TASSEL_2 + MC_2;
    /* enable LED for control */
    P1DIR |= LED;
    /* setup up interrupt parameters for input ports

P_I,
P_II,
S_I,
S_II
} PatternState;
#endif /* SIPPUFF_H_ */
During transmission, a break character is sent to indicate a break in communication. After receiving a break character, the receiver waits for the end of the break period, and then sends a break character to indicate the end of the break. The receiver then waits for the next receive interrupt. If the break period is not detected, the receiver sends a break character to indicate a break in communication. After receiving a break character, the receiver waits for the end of the break period, and then sends a break character to indicate the end of the break. The receiver then waits for the next receive interrupt.

The receive and transmit functions use the UART module to handle communication. The UART is configured to use a 9600 baud rate and to use the TxD and RxD pins. The UART is also configured to use a 1 MHz system clock. The receive function checks for a break character and sends a break character when necessary. The transmit function sends a break character when necessary, and then sends the command character.

The interrupt function handles timer interrupts and sends the command based on the current state, direction, and the current interrupt count. The interrupt function is called when the timer expires and the interrupt is enabled. The interrupt function checks the current state and direction, and sends the appropriate command character.
break;
}

DisableTimerA();
}
#endif
#endif

_disable_interrupts();
}

/* starts timer and begins counting the elapsed */
void start_timer() {
    InterruptCount = CLICK_THRESH;
    CCTLO = CCIE;
}
#endif
#endif
#endif

/* handles receiving intentions from the user */
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR() {
    _disable_interrupts();
    /* toggle LED when interrupt occurs */
    P1OUT ^= LED;
    if (P1IFG & SIPP) {
        switch (CurrentState) {
        case WAITING:
            CurrentState = SIP_F;
            start_timer();
            P1IES &= ~SIPP;
            break;
        case SIP_F:
            CurrentState = WAITING;
            if (InterruptCount > 0) {
                send_command(CLICK);
            } else {
                send_command(STOP);
            }
            P1IES |= SIPP;
            break;
        default:
            CurrentState = WAITING;
            break;
        }
    P1IFG &= ~SIPP;
    } else if (P1IFG & PUFFP) {
        switch (CurrentState) {
        case WAITING:
            CurrentState = PUFF_F;
            start_timer();
            P1IES &= ~PUFFP;
            break;
        }
case PUFF_F:
    CurrentState = WAITING;
    if (InterruptCount > 0) {
        send_command(RIGHT_CLICK);
    }
    else {
        send_command(STOP);
    }
    P1IES |= PUFFP;
    break;
default:
    CurrentState = WAITING;
    break;
    }
    P1IFG &= ~PUFFP;
    }
    _enable_interrupts();
}
/*
* State machine for detecting if one
* of the mouse axis should be reversed
* * Sip, Puff, Sip -> inverts mouse.X direction
* * Puff, Sip, Puff -> inverts mouse.Y direction
*/
PatternState next_pstate(Command c) {
    PatternState next;
    next = INIT;
    switch (PState) {
    case INIT:
        if (c == CLICK) {
            next = P_I;
        }
        else if (c == RIGHT_CLICK) {
            next = S_I;
        }
        break;
    case P_I:
        if (c == RIGHT_CLICK) {
            next = P_II;
        }
        break;
    case P_II:
        if (c == CLICK) {
            Dir.x = !Dir.x;
        }
        next = INIT;
    case S_I:
        if (c == CLICK) {
            next = S_II;
        }
        break;
    case S_II:
        if (c == RIGHT_CLICK) {
            }
Dir.y = !Dir.y;

    next = INIT;
    break;
    default:
    next = INIT;
}

return next;

/* Handles sending command and updating direction
* inversion pattern matching state machine */
void send_command(Command c) {
    char buf[3];
    buf[0] = (char)c;
    buf[1] = '';
    buf[2] = '
';
    UART_TX(buf);
    PState = next_pstate(c);
}

/* Sends array over UART connection
* | data | is expected to be a null-terminated string */
void UART_TX(char * tx_data) {
    unsigned int i=0;
    while(tx_data[i]) {
        /* Wait if line TX/RX module is busy with data */
        while ((UCA0STAT & UCBUSY)) ;

        /* Send out element i of tx_data array on UART bus */
        UCA0TXBUF = tx_data[i];
        i++;
    }
}
Appendix C. Bill of Materials

<table>
<thead>
<tr>
<th>Item</th>
<th>Unit Price ($)</th>
<th>Quantity</th>
<th>Total ($)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI Launchpad MSP430G2553</td>
<td>21.50</td>
<td>1</td>
<td>20.75</td>
</tr>
<tr>
<td>Origin Instruments Sip/Puff Sensor*</td>
<td>160.00</td>
<td>1</td>
<td>160.00</td>
</tr>
<tr>
<td>USB 2.0 Cable - A-Male to Mini-B</td>
<td>5.09</td>
<td>1</td>
<td>5.09</td>
</tr>
<tr>
<td>Push Buttons</td>
<td>0.68</td>
<td>2</td>
<td>1.36</td>
</tr>
<tr>
<td>Resistors (1 kΩ)</td>
<td>0.10</td>
<td>2</td>
<td>0.20</td>
</tr>
<tr>
<td>Misc. jumpers</td>
<td>0.30</td>
<td>10</td>
<td>3.00</td>
</tr>
<tr>
<td>Breadboard</td>
<td>9.69</td>
<td>1</td>
<td>9.69</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>201.09</td>
</tr>
</tbody>
</table>

Table 7: Bill of materials

* The Origin Instruments Sip/Puff sensor was provided by Dr. Kevin Taylor of the Kinesiology for use in this project.

Appendix D. Development Pictures

Figure 14: Development Circuit used for testing.
Figure 15: Detail of pins wiring for development circuit.

Figure 16: Origin Instruments Sip/Puff switch.
Figure 17: Detail of Sip/Puff switch output ports.

Appendix E. Reference Materials

URLs were accessed on December 4, 2015.

- MSP 430G2x53, MSP 430G2x13 Mixed Signal Microcontroller (Rev. H)
  http://www.ti.com/lit/ds/symlink/msp430g2553.pdf
- Origin Instruments Sip/Puff User’s Guide
- PyUserInput
  https://github.com/SavinaRoja/PyUserInput