Isotope

USB HID Emulation for Embedded Devices

Benjamin Matthew Pannell

STUDY LEADER Professor Thomas Niesler
DATE November 2014

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at the University of Stellenbosch.

I, the undersigned, hereby declare that the work contained in this report is my own original work unless indicated otherwise.

Signature **Date**

Abstract

English

Isotope is a project which addresses the need for an easy to use, low cost, USB HID emulation framework for use on embedded devices as an interface between these devices and any personal computer.

Applications include, but are not limited to, voice control of personal computers to aid performance and usability while ensuring universal compatibility. It is also possible that Isotope may be used to rapidly develop low cost simulator controls, remote control devices and administration tools.

Isotope has been designed to make use of the low cost ATmega32u4 chip which is readily available and can be sourced in small volumes for easy prototyping. Hardware integration has been kept as simple as possible, and maximum flexibility with respect to the host device has been sought to allow future expansion.

At the conclusion of this project a low cost design for a USB HID emulation interface which made use of a UART connection was developed, with a standardized set of communications protocols governing the use of this interface. Firmware and libraries were developed for the components specified, simplifying integration with new or existing software, and a number of demonstration applications were developed to illustrate the available functionality.

Afrikaans

Isotope is ‘n projek wat die tekort aan ‘n gemaklike, goedkoop, USB HID nydigheid raamwerk aanspreuk vir ingesluite toestelle. Die raamwerk kan gebruik word om ‘n koppelvlak op te stel tussen ingesluite toestelle en enige persoonlike rekenaar wat deur die alledaagse gebruiker benut sal word.

Isotope kan gebruik word in verskeie maniere, hierdie sluit ook in: Stem beheer van die persoonlike rekenaar om die prestasie en bruikbaarheid aan-te-hulp, terwyl dit ook die universele verenigbaarheid daarvan verseker. Dit is ook moontlik om Isotope te gebruik om lae koste simulator beheer sisteeme, afgeleë beheer toestelle en administrasie gereedskap, te ontiwikkel.

Isotope is ontwerp om gebruik te maak van die goedkoop ATmega32u4 chip, wat geredelik beskikbaar is, en kan in klein volumes aangekoop word vir gemaklike gebruik in die toets-fase van ‘n projek. Die integrasie van hardeware is so eenvoudig as moontlik gehou en maksimum buigsaamheid, ten opsigte van die gasheer toestel, is na gestreef om toekomstige uitbreiding toe te laat.

Met die gevolgtrekking van die hierdie projek was ‘n goedkoop ontwerp vir ‘n USB HID nydigheid koppelvlak, wat gebruik maak van ‘n UART konneksie, ontwikkel. Hierdie ontwerp benut ook ‘n gestandaardiseerde stel kommunikasie protokolle wat gebruik maak van die koppelvlak. Firmware en biblioteke is ontwikkel vir die komponente, soos wat daar vir hulle aangedui is, om die integrasie van nuwe asook bestaande sagteware te vereenvoudig. Daar is ook ‘n aantal programme ingesluit vir demonstrasie doeleindes, om die beskikbaare funksies van die projek te illustreer.

Table of Contents

Figures

Tables

Symbols

Adafruit Industries – An online electronics hobby store focusing on embedded device development and wearable computing.
Application Programming Interface - A set of methods, often provided in the form of a library, enabling 3rd party applications to make use of another application, device or service.
Application Specific Integrated Circuit – A chip designed to serve a specific purpose, in contrast to programmable devices which may be modified to suit a number of purposes.
Bipolar Junction Transistor – A form of switch activated by the flow of current through a base node. Similar in purpose to a FET.
Central Processing Unit – The primary component of a programmable chip, responsible for performing most logical operations as a result of the instructions it receives.
Field Effect Transistor – A form of switch activated by the voltage difference over its gate node. Similar in purpose to a BJT.
Future Technology Devices International – A hardware design company responsible for a number of USB interface chips used in a wide range of devices.
Human Interface Device – A subset of the USB protocol which makes allowance for the use of devices through which humans can interact with their computers, removing the need for dedicated drivers.
Master Device – The device responsible for generation of the emulation instructions. For example, the Raspberry Pi.
Printed Circuit Board – Circuit boards on which copper "tracks" are printed to simplify the construction of complex circuitry.
Random Access Memory – High speed memory used to store data which is currently in use on the system, including the instructions of running processes and the information they are interacting with.
Slave Device – The device responsible for interpretation of emulation instructions, and the emulation of user input on the Target Device. In this project, a Teensy 2.0.
Target Device – The device to which emulated input is directed, usually a personal computer of some kind.
Universal Asynchronous Receiver Transmitter – A standardised device responsible for serial communication between two devices over a two-wire connection.
Universal Serial Bus – A serial communication protocol, and accompanying hardware specification, which dictates the way in which a number of different device types may be connected to computers.

API ..... *See* Application Programming Interface
BJT ..... *See* Bipolar Junction Transistor
CPU ..... *See* Central Processing Unit
FET ..... *See* Field Effect Transistor
FTDI ..... *See* Future Technology Devices International
HID ..... *See* Human Interface Device
PCB ..... *See* Printed Circuit Board
RAM ..... *See* Random Access Memory

UART.....*See* Universal Asynchronous Reciever Transmitter
USB.....*See* Universal Serial Bus

Introduction

Modern voice recognition systems commonly fall into two primary categories, cloud based and native. Examples of cloud based speech recognition engines include Google's Voice Search, Apple's Siri virtual assistant and more recently, Microsoft's Cortana. The native implementations are best represented by Nuance's Dragon series of products and Microsoft's proprietary Speech API (MSSAPI).

Native solutions are generally built on learning Hidden Markov Models which adapt to the speaker and can achieve high accuracy levels once trained and paired with a high quality microphone. In most cases these systems are designed to assist people who would otherwise be required to perform a lot of typing, or the disabled, and as a result their implementations are often tailored towards single users.

Conversely, cloud solutions are vastly more complex and generally designed to be able to achieve good accuracy rates with little or no speaker specific adaptations. At the time of writing several state-of-the-art systems were built using a combination of advanced neural networks and HMMs to help improve feature detection across very large datasets. An example of this type of system is that developed by Google through their 1-800-GOOG-411 service [1]. There are a few restrictions to these cloud based services though, often determined by their target applications, these restrictions include limitations on the maximum length of a dictated statement and the inability to adapt to a user's pronunciation.

One of the major issues faced with both cloud and native approaches is that they rely on software on the target device to record, pre-process, recognise where necessary, and finally output the result – leading to platform restrictions which are often difficult to overcome. Another is the possibility of piracy, as this software is often extremely expensive with a low number of users it poses a major threat to the producer's revenue stream.

This project is tasked with the development of a device which addresses both issues by providing a means through which a hardware speech recognition device can emulate user input. This will allow speech recognition to be performed either on a hardware processor attached to the user's computer or in the cloud with this device as a proxy. As the solution is hardware based, piracy will be impossible and the revenue stream of the producer will be more secure.

In addition to this, the ability to easily connect the device to any computer, has the advantage of allowing learning algorithms to be applied – either on the device itself or using the device as an identifier - improving recognition rates for the device's user.

The goal of this project is to develop an interface which can be used by embedded devices to emulate a user's input devices – such as a keyboard or mouse – without the installation of custom drivers or software on the target machine. In light of this requirement, this project has taken the form of a USB HID emulation device which is controllable over a simple serial protocol over an UART, allowing it to be used on almost any embedded device platform with minimal, or in some cases no, hardware alterations.

As an adjunct to this, a series of communications protocols and libraries will be developed to make the use of the emulation hardware as straightforward as possible from a variety of different programming languages. To demonstrate this, a simple speech recognition engine will be implemented to allow basic commands to be given to the device and executed.

Universal Serial Bus Background

The Universal Serial Bus (USB) specification was developed in the mid-1990s to provide a common set of connectors and protocols through which a multitude of devices could be connected to computer systems. The USB specification defines four primary device classes – Mass Storage, Media Transfer Protocol, Human Interface Devices and Device Firmware Upgrade. These classes can be used to fulfil a large range of requirements. This project will focus on the Human Interface Device class as a means of emulating common USB input devices like the Mouse and Keyboard.

The USB HID specification was originally introduced to provide a standardized interface through which input devices could expose functionality over the USB protocol, and includes named support for a vast array of device types including Mice, Keyboards, Joysticks and Game Controllers. This standardization has enabled operating system developers to include generic device drivers for these devices as part of their distributions, reducing the need for custom driver development and allowing almost universal compatibility for common device classes. The result is that the USB device classes are well supported by all current major operating systems.

At a minimum, a USB HID device requires a USB connector - either in the form of an attached USB Standard-A Plug as seen in Figure 1, or through the use of a USB Mini-B (Figure 2) or USB Micro-B (Figure 3) connector and the appropriate connection cable – and a chip capable of reading from and writing to the serial data lines provided by these connectors.

Figure 1 USB Standard-A Plug. Image courtesy of Evan-Amos.

Figure 2 USB Mini-B Plug. Image courtesy of Winford Engineering LLC.

Figure 3 USB Micro-B Plug. Image courtesy of Winford Engineering LLC.

The HID specification requires that the USB client device declares itself, and its capabilities, to the host device through a series of reports which allow the host operating system to accurately predict the manner in which the client device will behave. These reports are formally defined within the HID Device Class Definition [2] and are integral to the correct detection and functioning of an HID device.

Pre-Design Investigation

Prior to beginning the design of the system it was important to investigate the possible approaches and determine which of them best suited the task of USB HID input emulation. During this phase a number of possible solutions were investigated, their advantages and disadvantages compared and finally a decision was made on the best option for this project.

Custom ASIC Design

The first option to be considered was the design of a custom ASIC for the purpose of USB emulation, implementing its own interface over either serial UART or i2c. Doing so would have allowed the manufacture of extremely small, energy efficient and cheap emulation chips and would have proven an ideal solution for mass production due to the potential cost and size savings involved. On the other hand, by virtue of the solution being entirely implemented in hardware, the design and testing phase would have been prolonged while simultaneously reducing the flexibility with which additional features could be added.

To summarise, a custom ASIC providing USB HID emulation would suit a large scale project. However, for prototyping purposes, it would present challenges. ASICs aredifficult to acquire due to the need for fabrication of the chips prior to acquisition. They also force developers to spend a large amount of time implementing support for their low level interfaces. In particular, the design and implementation of the low level interface would be time consuming, as would later modifications.

Programmable USB Slave Device

There are a number of USB interface chips available on the market for example, the well-known FT232R [3] by FTDI, but also more complex programmable devices. One such example is the Vinculum-II [4] which includes a built in 16-bit CPU and programmable code block, allowing you to easily modify it to suit any number of applications.

The primary reason for avoiding the use of these devices was the difficulty of obtaining a chip which made use of common slave-type (B class) connectors in a pre-packaged form. This would require the purchase of individual chips in their unpackaged state and – due to their form factors – the surface mounting on custom PCBs, restricting the ability to construct the device easily and potentially raising costs above acceptable levels for small runs.

Microprocessor with USB Interface

Another option was to make use of a microprocessor which included a built in USB interface, and reprogramming its firmware to allow USB HID emulation to take place while repurposing one of its IO channels for inter-device communication. One particularly promising candidate was the ATmega32u4 [5] which includes its own full speed USB controller which is fully programmable. Other advantages included the fact that it is the basis of the Arduino Nano [6] and as a result was commonly available in an easy to use package.

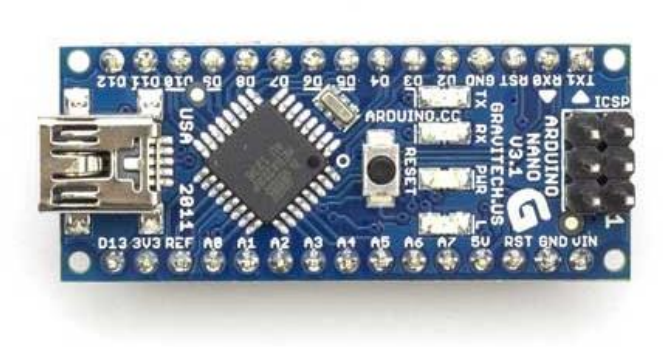

Figure 4 Arduino Nano. Image courtesy of Arduino SA [6]

In addition to this, the ATmega32u4 is designed to act as a controller for a number of USB peripherals. This proven functionality improved confidence in the chip's ability to fulfil the requirements at hand, while its small size and relatively low cost would ensure low costs in production. In addition to this, it would be easily possible to transition to either the ATmega16u2 or ATmega8u2 in production without major code changes – allowing costs to be reduced further.

Conclusion

After analysing three possible options, it was decided that centering the design around the ATmega32u4 would provide the best prototyping platform for the project with the best prospects for future expansion, while remaining accessible and low cost.

In selecting the ideal prototyping platform, it was noted that the Arduino Nano [6], which uses the ATmega32u4, provided adequate functionality. However the PJRC Teensy 2.0 [7] provided a more rounded set of features at a lower cost, while remaining equally accessible and using the same tooling. In addition to this, the Teensy 2.0's size – roughly one third smaller than the Arduino Nano – meant the final prototype would be smaller and more portable. As a result the decision was made to acquire the Teensy 2.0 as the prototyping board of choice.

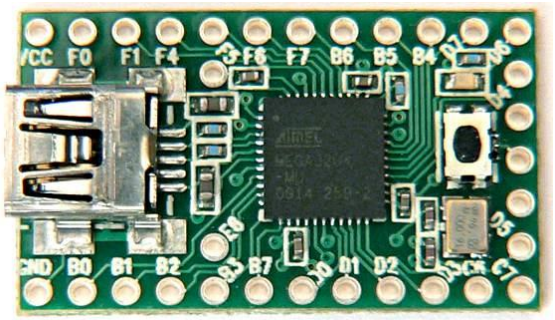

Figure 5 PJRC Teensy 2.0. Image courtesy of PJRC [7]

Demonstration Device Selection

In order to demonstrate the emulation device, it was necessary to acquire a master device similar to that which would be used in production or application prototyping environments. Due to Professor Thomas Niesler's experience with speech recognition, and interest in making use of the device in this field, it was decided that the demonstration would consist of a self-contained speech recogniser – requiring that the master device be capable of running the software required.

Following on the work performed by Christian Truter in implementing Automatic Speech Recognition on the Raspberry Pi in his 2013 Electrical & Electronic Final-Year project entitled "Pi Your Command" [8] it was decided that a Raspberry Pi class device would suffice as a low cost demonstration platform for a number of reasons discussed below.

There are a number of devices of similar performance occupying a similar price point to the Raspberry Pi [9] including the BeagleBone Black [10] and, although somewhat more expensive, the Intel Galileo Development Board [11]. More recently, Raspberry PI class boards such as the Banana Pi [12] and HummingBoard [13] have also begun to appear which offer a higher performance solution at a corresponding price point.

Of these devices, one of the primary concerns within the scope of this project was the cost effectiveness of the prototyping platform. While options such as the HummingBoard [13] are significantly more powerful than the Raspberry Pi [14], this additional processing power would not make any significant contribution to this project. In addition to this, the popularity of the Raspberry Pi has served to ensure that devices such as the HummingBoard and Banana Pi adopt the same physical layouts. This allows the design of this project's module to be specifically tailored towards a specific form factor, allowing drop-in replacement of the master board.

Availability, both of the physical hardware as well as support, examples and documentation also played an important role in the decision making process - a field in which the Raspberry Pi's popularity is again a factor.

Device Design

There were a number of device aspects which need to be taken into account when undertaking design of the board and its associated components. These would dictate the ways in which components were connected and have an effect on the communication protocols used to allow the Raspberry Pi [14] to communicate with the Teensy 2.0 [7].

One of the initial design considerations was based on the fact that the Raspberry Pi [14] operated at a core voltage of 3.3V while the Teensy 2.0 [7] – for lack of a voltage converter – operated at 5.0V supplied via its USB port. As a result of these different voltage levels, it would be dangerous to connect the Raspberry Pi [14] and Teensy 2.0 [7] directly to one another.

For this reason it would be necessary to include a level translator in the design to allow the Raspberry Pi [14] to communicate safely with the Teensy 2.0 [7]. Initially the use of a voltage divider and basic BJT or FET booster were considered [15], however upon further inspection it became clear that at very high switching rates the voltage divider may become unsuitable due to the output pin capacitances.

$C_{pin-ATmega} = 10\ pF$ [16] $\rightarrow \tau = RC = 500 \times 10^{-9} \rightarrow f_{max} = 2\ MHz$

$C_{pin-rpi} = 5pF$ [17] $\rightarrow \tau = RC = 250 \times 10^{-9} \rightarrow f_{max} = 4\ MHz$

These frequency limitations impose restrictions on the physical protocols which may be used, however does not affect the use of any common UART baud rate.

The BJT or FET booster and voltage divider would also need to be tailored to the voltages of each device, reducing the ease with which the system could be adapted to new host hardware.

Seeking an alternative solution, the Texas Instruments TXB0104 4-channel bi-directional level translator [18] was selected based on its ability to handle a wide range of voltages from 1.2V to 3.6V on the low side and 1.65V to 5.5V on the high side [19], with automatic direction detection on each channel. This, combined with the exceptionally high throughput (100Mbps [19]) meant that it would be possible to allow easy migration of the final device between different host devices with minimal, if any, modifications as well as allowing future extensions to the device's capabilities as necessary.

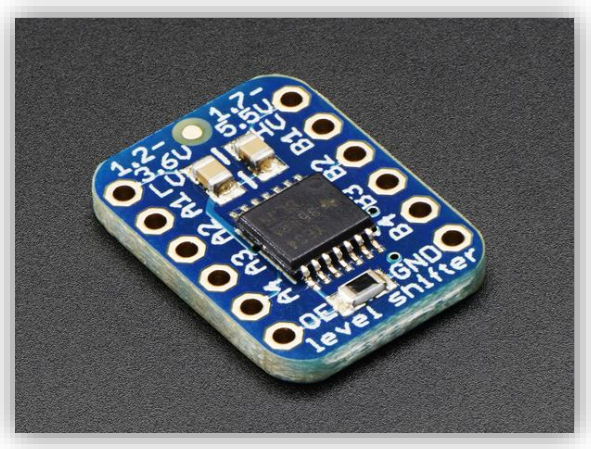

Figure 6 TXB0104 Bi-Directional Level Shifter. Image courtesy of Adafruit Industries [20]

Finally, to assist with prototyping on the Raspberry Pi [14] it was decided that a platform specific prototyping board, in the form of the Adafruit Prototyping Pi Plate Kit [21], would be used to allow easy attachment to the Raspberry Pi [14] and provide a stable platform on which to mount the Teensy 2.0 [7] and TXB0104 [18].

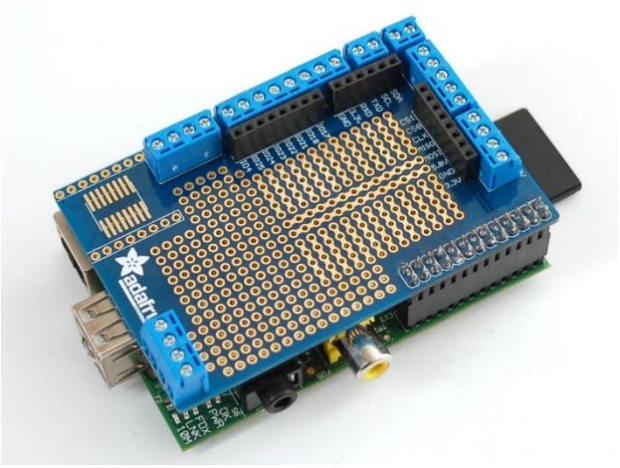

Figure 7 Adafruit Prototyping Pi Plate Kit for Raspberry Pi. Image courtesy of Adafruit Industries [21]

Electrical Interface

The primary hardware design challenge is the electrical interface between the master device (Raspberry Pi in this case) and the ATmega32u4. The design requires that a one-way command channel be established between these devices in a reliable and safe manner, requiring minimal setup on behalf of the user.

Hardware Protocol Selection

There was a strong incentive to make use of common standardized interface protocols like i^2c, SPI or UART in order to allow the design to easily be adapted to alternative master devices. When considering these protocols it was important to take a number of concerns into account, namely available bandwidth, voltage levels and switching frequencies.

In all cases, the differing core voltages utilized by the Raspberry Pi and ATmega32u4 (3.3V and 5V respectively) would require some form of voltage level switching circuit to be used.

When considering i^2c and SPI, one of the immediately apparent issues was the synchronous nature of the link – requiring the master device to continuously poll the client for new data – which would complicate attempts to extend the interface to operate in a duplex manner. The second significant hurdle was the switching rates used by these interfaces, often in the order of several MHz, this would make the task of level switching vastly more complex as a non-resistor based solution would need to be sought to avoid a high RC time constant or current drain.

In addition to these issues, i^2c requires a somewhat more complex pull-up arrangement which makes level translation more challenging for automated circuits, requiring specialist chips to ensure that it operates correctly. SPI, while not suffering from this issue, has the problem of a minimum of 3-wires, increasing the number of voltage level translators necessary and therefore the complexity of the design.

As a result, it was decided that UART offered a good compromise between available bandwidth (115200 baud would offer enough bandwidth to adequately convey commands with a binary protocol), simplicity (with only one wire for simplex, and two for duplex communication) and availability (with most integrated devices sporting at least one UART).

Voltage Level Translation

Due to different core operating voltages on the Raspberry Pi and ATmega32u4 (Teensy 2.0) it is necessary to perform voltage level translation on the interface lines so as to prevent damage to the master or slave devices.

Two solutions were considered to this problem, the first being a custom BJT/resistor ladder circuit which would raise and lower (respectively) the voltage levels on the transmit and receive lines, and the second being a Texas Instruments TXB0104 [18] level translation chip which was designed for this purpose.

The advantage of a BJT/resistor ladder configuration was a significant reduction in project cost, by approximately 90% for the level switcher circuit, however it would impose restrictions on the flexibility of the design. Specifically, the BJT's configuration would be tied to the master and slave voltage levels, and it would require replacement if either of these

operating voltages changed in future. The same issue applied to the resistor ladder as a step-down converter, as well as possible transient interference with higher UART baud rates if resistor selections were poor – a mistake someone unfamiliar with the project could easily make when attempting their own implementation.

The TXB0104 [18] conversely offers an increase in flexibility, automatically translating voltage levels between 1.2V and 5.5V without any circuit modifications, and with automatic direction detection. This would significantly simplify circuit design and allow the project to be easily implemented by even inexperienced hobbyists at the result of a higher unit cost.

Safety Precautions

In the interest of safety, it has been decided that the Raspberry Pi will not have its 5V connection coupled to the Teensy 2.0's 5V connection, which would potentially allow the Raspberry Pi to operate without the need of an external power source. This decision was made in the interest of caution as the Raspberry Pi is capable of drawing far more than the USB 2.0 standard 500mA of current which, if drawn through the ATmega32u4, has the potential to damage the Teensy 2.0.

The result is that it is necessary for the Raspberry Pi to be connected to its own power source, independently of the Teensy 2.0.

Raspberry Pi Selection

At the time of selection, there were two models of Raspberry Pi available. The Model A and the Model B. The Model B was selected due to the presence of an Ethernet Port, as well as the additional 256MB of RAM, with the intention of running a basic voice recognition framework on the device.

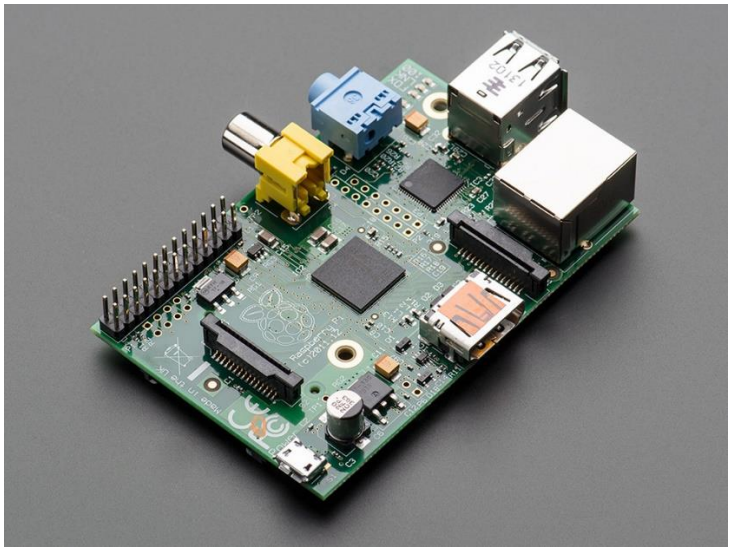

Figure 8 Raspberry Pi Model B. Image courtesy of Adafruit Industries [9]

System Overview

The system architecture is outlined in Figure 9 demonstrates the manner in which the device is structured. This approach allows the Master Device to emulate user input on the Target Device through the use of the Emulation Slave Device.

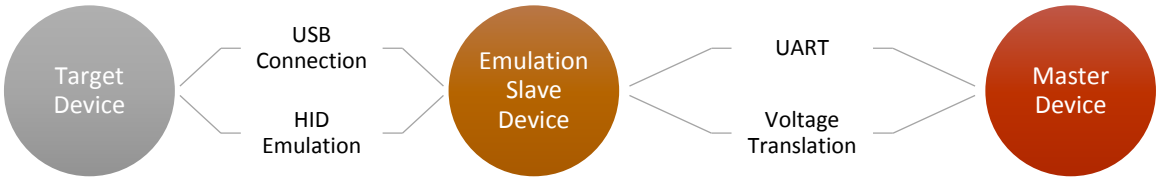

Figure 9 System Architecture

Within this architecture, the Teensy represents the Emulation Slave Device which receives commands from the Raspberry Pi (Master Device) over a UART connection, and is responsible for converting these commands into USB HID reports which may be submitted to the Target Device. In this manner, emulation implementation details are abstracted away from the Master Device to allow for simpler hardware and software design as well as the minimization of configuration conflicts and incompatibilities. The design also simplifies replacement of the Raspberry Pi should an improved alternative become available in the future, requiring only superficial changes be made.

Figure 10 shows the final emulation device circuit layout, demonstrating how the Teensy's UART is passed through the TXB0104 prior to being connected to the Master Device – providing bi-directional voltage level translation.

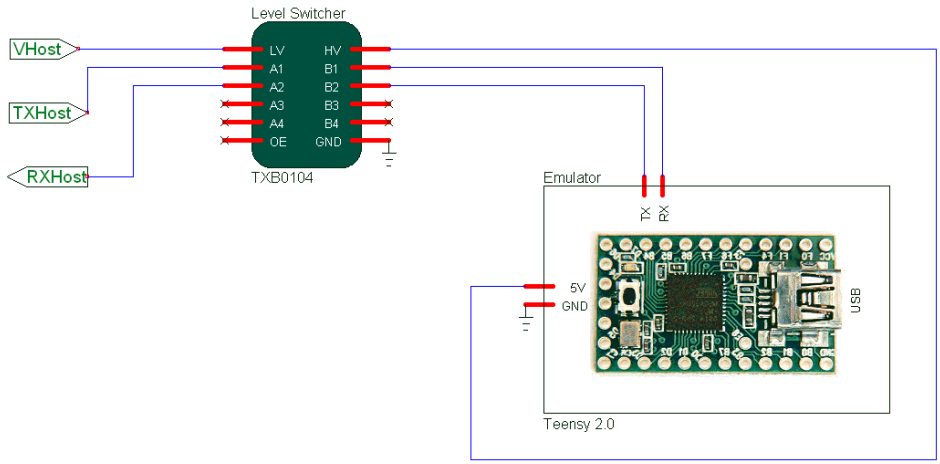

Figure 10 Emulation Slave Device Circuit Diagram

Software Design

Inter-Device Communication

There are two physical communications channels which are utilized in this project. The first, USB, is used to allow the Emulation Slave Device to connect to, and communicate with, the Target Device. This channel is a half-duplex serial communication path initialized as an HID device (class 3 according to the USB specification) allowing the emulation of keyboards, mice and joysticks. USB communication is handled through the Teensy's included USB HID helper methods which provide low level HID emulation functionality.

The second channel connects the Emulation Slave Device to the Master Device via a Serial UART operating at 38 400 bits/second. This channel operates in full-duplex mode, however at present only the downstream (Master Device to Slave Device) channel is utilized. It is the responsibility of this channel to convey emulation commands (as defined in the Performance Considerations

Performance of the Emulation Slave Device is dictated by three primary factors: the UART's bandwidth, the USB HID specification and the performance of the ATmega32u4 used for emulation tasks.

Of these three factors, the USB HID specification is the most restrictive in terms of performance due to the polling nature of the protocol. The limit affects communication between the Emulation Slave Device and Master Device as defined in Figure 9 – which is limited to a maximum command rate of 1000Hz. This limitation requires that the software is designed to never exceed a reporting rate of 1000 commands per second so as to avoid emulation issues. The effect of failure to adhere to this rate is investigated in the section "Effects of Exceeding USB HID Rate Limitations".

A secondary limit is imposed in the form of the UART's maximum reliable baud rate of 115200 bits per second. This limit affects communication between the Master Device and the Emulation Slave Device as defined in Figure 9. Due to the way in which serial communication libraries are implemented, as well as the configurable nature of the UART's baud rate, it is possible to utilize this limit as a means to reduce the likelihood of exceeding the USB HID protocol's polling rate limitation.

The final restriction, performance of the ATmega32u4, given its 16MHz core frequency and 256KB of RAM makes it imperative that any software implementations running on the device be optimized to make use of as few steps, and as little processor time, as possible. Testing has not demonstrated any appreciable issues with performance of the device beyond those limitations imposed by the USB HID protocol.

Target Device Performance

The performance of the Target Device also imposes restrictions on the maximum usable emulation command rate. This comes about as an effect of assumptions taken when developing applications which expect human input, specifically the rate at which a human is capable of providing input. By providing input at rates which drastically exceed this it is possible to cause applications to freeze or even crash.

Rate Limiters

Due to the USB HID rate limitation of 1000Hz it is necessary to enforce a command rate limit which is responsible for restricting the rate at which commands are dispatched to the Emulation Slave Device. Rate limiting is ideally achieved by restricting the rate at which the Master Device dispatches emulation commands, and can be implemented using two separate methods.

Serial Baud Rate

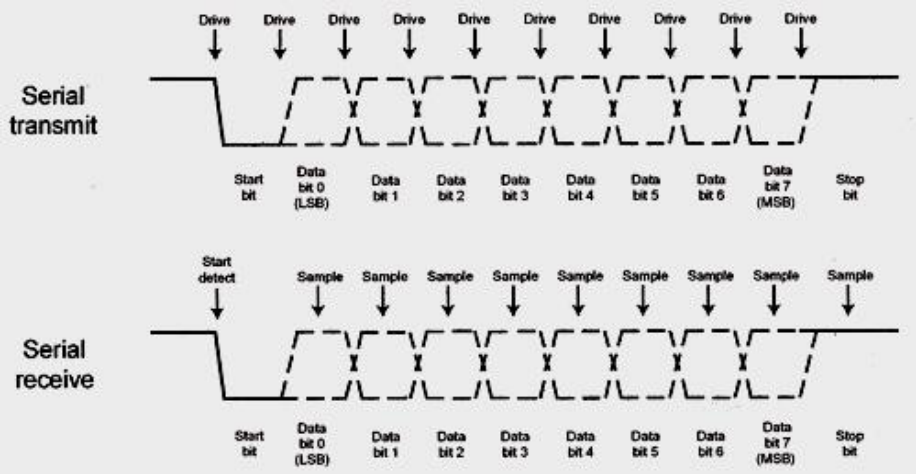

Figure 22 UART Byte Transmission Levels. Image courtesy of embedded .

The UART protocol operates through the use of start, stop and checksum bits on each transmitted byte of data – as show in **Error! Reference source not found.** – resulting in an effective byte length of 11 bits. By considering this, in conjunction with the minimum relevant packet size for the protocol as defined in **Error! Not a valid bookmark self-reference.**, it is possible to determine the ideal baud rate.

$$idealBaud = 11 \times 1000 \times packetSize$$

Making the assumption that the average common packet is between 2 and 3 bytes long for keyboard input, it makes sense to select a baud rate as close to 33000 as possible. As the nearest standard baud rate is 38400, it makes sense to select that as the primary transmission rate.

Library Integrated Rate Limiters

By implementing a rate limiter as part of the Master Device's software libraries it is possible to exert considerably more control over the rate at which commands are transmitted to the Emulation Slave device. This makes it possible to configure the rate at which commands are transmitted at runtime, without requiring modifications to the Emulation Slave Device's firmware.

This project makes use of two primary rate limiter types – blocking and asynchronous – depending on the platform. The C library makes use of a blocking rate limiter which will prevent a transmission request from completing until a specific delay has been achieved. The Node.js library on the other hand, to support the asynchronous nature of the Node.js

runtime, makes use of a limiter which stores pending commands in memory and dispatches them at the configured rate.

Optimal Command Execution Rate

After testing a number of rates, it has been concluded that a maximum command rate of 500Hz should be used for keyboard input. At this rate it is possible for basic text editors to receive large amounts of text in a performant manner – however editors which perform spell checking or syntax highlighting have in certain cases exhibited freezing at this emulation speed. For this reason, it is recommended that applications which intend to make use of emulation on a universal basis should instead use 125Hz as their maximum reporting rate – allowing for up to 62 characters of text entry per second.

For more information on the tests that were conducted to determine these rates please consult the Testing section of this report.

Communications Protocol) to the Emulation Slave Device in a timely and reliable manner. Serial communication will be handled through the termios interface on the Master Device, a standard Linux serial interface, while communication on the Teensy will be handled by the included Serial1 wrapper.

USB Emulation Slave Device

The slave device, an ATmega32u4, is responsible for the conversion of emulation commands into USB HID packets which are conveyed to a computer over a USB connection. The PJRC Teensy 2.0 [7] selected for this purpose is bundled with a basic USB Mouse and Keyboard emulation library which is capable of emulating simple key presses and mouse movements.

Figure 11 Slave Device Command Process

The slave device's software is therefore responsible for the parsing of a received packet, delegation of the parsed packet to the relevant logic function and finally the calling of the emulation functions required to fulfil the command.

In order to maximize throughput, minimize processing time and boost flexibility it was decided that a binary protocol represented the best approach to command structuring. The design of this protocol, its structure and examples are available in **Error! Reference source not found.**.

USB HID Emulation Capabilities

The Teensy 2.0 [7] selected for this project is bundled with a USB HID emulation library capable of generating the HID reports necessary for basic input device emulation. This library includes support for USB Mice, Keyboards and Joysticks – providing a very low level API through which emulation of these devices can be achieved. In all cases, it is necessary to call the **`usb_init`** function prior to using any of the emulation API methods. This method is responsible for ensuring that the correct feature reports are sent to the host PC, allowing the Teensy to identify itself as an HID device.

The Keyboard emulation layer is exposed through a **`keyboard_modifier_keys`** variable which holds flags indicating the active modifier keys, as well as a **`keyboard_keys`** array which holds the six keys which may be sent at a time. Upon populating these variables, the **`usb_keyboard_send`** function is executed to transmit this information to the host PC.

The Mouse emulation layer exposes itself in the form of an **`usb_mouse_buttons`** and **`usb_mouse_move`** function which allow the pressed mouse buttons and movement deltas to be specified respectively.

USB Emulation Example

In this short example, the process of emulating a Ctrl+A (select all text on Windows and *NIX devices) is demonstrated through the use of the Teensy's built in USB emulation layer.

It should be clear that there are a large number of steps involved which, if presented through a Remote Procedure Call system, would result in a large bandwidth overhead for every command to be executed. This was the primary reason for implementation of the binary protocol outlined in **Error! Reference source not found.**.

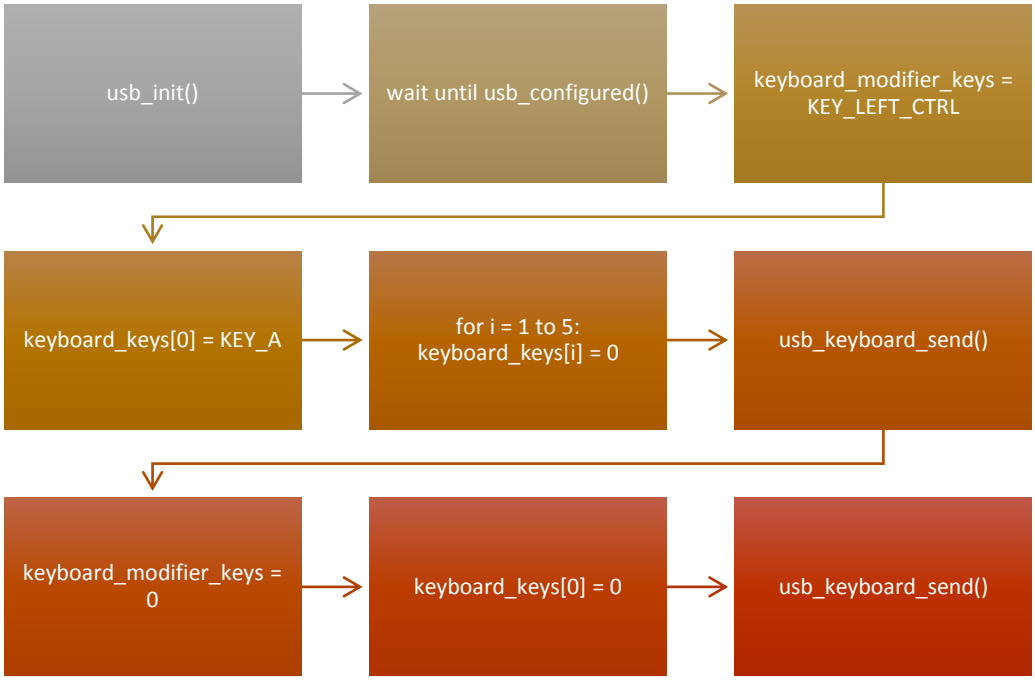

Figure 12 USB Emulation Example – Ctrl+A

Master Device

The master device software consists of libraries responsible for wrapping of emulation commands in their relevant protocol level representations and transmission of these to the slave device. These libraries were developed for number of languages and, at a minimum, provide a means to submit mouse and keyboard emulation commands to the slave device for emulation.

Figure 13 Master Device Library Process

At a base level, these libraries provide functions for emulating mouse key presses, movement and scrolling through a function like `isotope_mouse(buttons, deltaX, deltaY, deltaScroll)` and the emulation of keyboard input through a function like `isotope_keyboard(modifiers, [] keys)`. These functions are intended to be used by higher-level wrappers which extend their behaviour through the use of state machines, key maps and other application specific logic to provide more advanced functionality.

This approach allows complex functionality to be implemented at a library level while abstracting the details of the emulation away from the Master Device – vastly simplifying the task of maintenance and future extension, as well as allowing differentiation within libraries and customization of their output.

For example, it would be possible for a custom library to provide a transcription service through which text passed to it in string form would be emulated, effectively transcribing the text onto a target computer.

Library APIs

This section covers the various software libraries developed to simplify the use of Isotope in projects, as well as providing examples of their use. For more information on each API it is recommended you consult their bundled documentation which has been made available in **Error! Reference source not found.**.

Interface libraries have been developed independently in both C and JavaScript, and a number of command line utilities to provide emulation have also been created for use from a terminal shell.

C-Library

The C library provides a low level interface through which interactions with the Isotope device can be managed. It is designed to simplify the creation of higher level software wrappers by providing platform specific communication logic and native packet generation functions through a standardized API.

It is recommended that, where possible, application developers refrain from using the C library directly, as its functionality is substantially more basic than that provided by some of the other higher level wrapper libraries.

```
#include <libisotope.h>
void main() {
      int isotope = isotope_open("/dev/ttyAMA0");
      char keys[] = { KEYS_A };
      isotope_keyboard(isotope, 0, keys, 1);
      isotope_keyboard(isotope, 0, keys, 0);
      isotope_close(isotope);
}
```

Figure 14 C-Library Keyboard Code Example

```
#include <libisotope.h>
void main() {
      int isotope = isotope_open("/dev/ttyAMA0");
      const char *text = "Hello World!\n";
      isotope_maxRate = 125;
      isotope_text(isotope, text);
      isotope_close(isotope);
}
```

Figure 15 C-Library Text Code Example

The full API is described in Figure 16, including parameter descriptions and return values.

```c
/**
 * Isotope C Library
 * Provides a C interface between the Isotope emulation chip
 * and the local system, as well as a number of useful command
 * wrappers.
 *
 * Copyright Â© Benjamin Pannell 2014
 */

#ifndef ISOTOPE_H
#define ISOTOPE_H

/**
 * Configures the maximum transmission rate in commands per second (Hz)
 * to be used by this library. By default this is 500Hz to prevent issues
 * with the USB HID protocol.
 * May be set to 0 to disable rate limiting.
 */
int isotope_maxRate;

/**
 * Opens a new Isotope using the specified device to communicate, e.g. /dev/ttyUART0
 * @param device The device to open for interaction between the Isotope and this library
 * @returns isotope The unique Isotope ID used for submission of emulation requests
 */
int isotope_open(const char* device);
/**
 * Closes an Isotope connection given the Isotope's ID, this should be done before closing your application.
 * @param isotope The ID of the connection to terminate
 * @returns status Returns 0 if the connection was terminated successfully
 */
char isotope_close(int isotope);

/**
 * Sends a mouse emulation request using the given Isotope.
 * @param isotope The opened Isotope connection to send the emulation request over.
 * @param buttons The depressed mouse buttons to emulate, a set of MOUSE_ flags.
 * @param deltaX The x-axis movement of the mouse -127 to 127.
 * @param deltaY The y-axis movement of the mouse -127 to 127.
 * @param deltaScroll The scroll delta, usually +/- 3 to emulate a standard scroll wheel.
 */
char isotope_mouse(int isotope, char buttons, char deltaX, char deltaY, char deltaScroll);

/**
 * Sends a keyboard emulation request using the given Isotope.
 * @param isotope The opened Isotope connection to send the emulation request over.
 * @param modifiers A combination of modifier keys to be transmitted
 * @param keys[] The physical keyboard keys who's depression should be emulated.
 * @param keys_count The number of physical keyboard keys to be emulated (max of 6).
 */
char isotope_keyboard(int isotope, char modifiers, const char keys[], char keys_count);

/**
 * Converts the given text into a series of keyboard emulation requests for transmission
 * to a remote Isotope.
 * @param isotope The opened Isotope connection to send the emulation requests over.
 * @param text The null terminated string to send to the remote machine.
```

```
 */
int isotope_text(int isotope, const char* text);

/**
 * Sends a joystick emulation request using the given Isotope.
 * @param isotope The opened Isotope connection to send the emulation request over.
 * @param buttons The 32 button's depression states to emulate 0x1, 0x2 etc.
 * @param x The x-axis reporting value (from 0 to 1023), 512 represents center.
 * @param y The y-axis reporting value (from 0 to 1023), 512 represents center.
 * @param z The z-axis reporting value (from 0 to 1023), 512 represents center.
 * @param rz The z-
axis rotation reporting value (from 0 to 1023), 512 represents center.
 * @param sliderLeft The left slider reporting value (from 0 to 1023), 512 represent
s center.
 * @param sliderRight The right slider reporting value (from 0 to 1023), 512 represe
nts center.
 * @param hat The hat switch position. 0xff represents center, 0 to 7 represent angl
es in multiples of 45 degrees.
 */
char isotope_joystick(int isotope, int buttons, short x, short y, short z, short rz,
 short sliderLeft, short sliderRight, char hat);

/**
 * Writes a packet to the given Isotope, used for low level commands
 * @param isotope The Isotope connection over which to send the packet
 * @param packet The packet to transmit to the given Isotope
 * @param length The number of bytes in the packet
 */
char isotope_write(int isotope, const char* packet, char length);

/**
 * Reads a number of bytes from the remote device into the given buffer
 * @param isotope The Isotope connection over which to retrieve data
 * @param packet The buffer into which data will be read
 * @param length The number of bytes available within the buffer for writing
 * @returns The number of bytes read into the buffer
 */
int isotope_read(int isotope, char* buffer, int length);

#endif
```

Figure 16 C-Library API Definition

Node.js Library

The Node.js library provides a high level wrapper around the Isotope protocol, with a number of useful helper functions and an easy to use asynchronous message based API. This library is intended for rapid prototyping and lightweight application development across a wide range of devices.

```
interface Isotope {
	Isotope(string device);
	Isotope(SerialPort device);
	void flush();
	void close();

	Keyboard keyboard;
	Mouse mouse;
	static KeyboardKeys keyboard;
	static MouseButtons mouse;
}
interface Device {
	Device then;
	Device queueUpdate();
	Device now();
}
interface Keyboard : Device {
	Keyboard ctrl, alt, shift, releaseAll;
	Keyboard press(byte[] keys), release(byte[] keys);
	Keyboard pressModifiers(byte modifiers), releaseModifiers(byte
modifiers);
	Keyboard write(string text);
}
interface Mouse : Device {
	Mouse left, right, middle;
	Mouse press(byte buttons),  release(byte buttons);
	Mouse scroll(sbyte delta);
	Mouse move(sbyte deltaX, sbyte deltaY);
}
interface MouseButtons {
	static byte left = 0x1;
	static byte right = 0x2;
	static byte middle = 0x4;
}
interface KeyboardKeys {
	static KeyboardModifiers modifiers;
	static KeyboardStandard keys;
}
interface KeyboardModifiers {
	static byte ctrl = 0x01; // And so on…
}
interface KeyboardStandard {
	static byte a = 4; // And so on…
}
```

Figure 17 Node.js Library API

The library's API has been designed to enable the development of highly readable and natural code for the emulation of input on a target system through the use of the Mouse and Keyboard device helper classes. These classes enable commands such as **`isotope.keyboard.shift.press(4).then.releaseAll.then.press(5).then.releaseAll`** which types "Ab" on the host machine.

```
var Isotope = require('libisotope');
var isotope = new Isotope('/dev/ttyAMA0');
isotope.keyboard.write("Hello World!");
isotope.mouse.move(-10,0);
isotope.close();
```

Figure 18 Node.js Library Example

The Node.js library is available through the NPM (Node.js Package Manager) repository and can be installed locally by running **npm install libisotope** from a terminal, provided the platform has been configured to include NPM in the path.

Command Line Toolkit

To further supplement the C and Node.js API libraries, a set of command line tools for Mouse and Keyboard input emulation was also developed. This enables the use of Isotope from a terminal or shell script and is a useful way to test functionality prior to writing code.

```
isokey [OPTIONS] KEYS
       OPTIONS:
       -C/--ctrl     - Control modifier key
       -S/--shift    - Shift modifier key
       -A/--alt      - Alt modifier key
       -W/--win      - Windows/Command modifier key
       -H/--hold     - Hold down keys until next command
       Long options (--ctrl) can additionally be prefixed with ‘r’ or ‘l’
       for right or left (--rctrl or --lctrl)

       KEYS:
       A B C D E F...
       F1..24
       ,./[];’\`
       ENTER ESC BACKSPACE SPACE DEL INSERT PGUP PGDN EQUALS MINUS...

       EXAMPLES:
       isokey –S h
       isokey e
       isokey l
       isokey l
       isokey o
```

Figure 19 Isotope Keyboard Emulation Command Line API

```
isomouse [OPTIONS] MOVEMENT
       OPTIONS:
       -L/--left     - Press left mouse button
       -R/--right    - Press right mouse button
       -M/--middle   - Press middle mouse button
       -H/--hold     - Hold buttons until next command

       MOVEMENT:
       X[delta]      - Move [delta] units to the right
       Y[delta]      - Move [delta] units upwards
       S[delta]      - Scroll [delta] lines upwards

       EXAMPLES:
       isomouse –L --hold
       isomouse –L –H X10 Y1
       isomouse –L –H X10 Y1
       isomouse
```

Figure 20 Isotope Mouse Emulation Command Line API

```
isowrite [TEXT]
       Accepts text in the form of command line arguments, or STDIN if no
       command line arguments are provided.

       EXAMPLES:
       isowrite Hello World
       isowrite “Hello World”
       echo “Hello World” | isowrite
```

Figure 21 Isotope Text Transcription Command Line API

Performance Considerations

Performance of the Emulation Slave Device is dictated by three primary factors: the UART's bandwidth, the USB HID specification and the performance of the ATmega32u4 used for emulation tasks.

Of these three factors, the USB HID specification is the most restrictive in terms of performance due to the polling nature of the protocol. The limit affects communication between the Emulation Slave Device and Master Device as defined in Figure 9 – which is limited to a maximum command rate of 1000Hz. This limitation requires that the software is designed to never exceed a reporting rate of 1000 commands per second so as to avoid emulation issues. The effect of failure to adhere to this rate is investigated in the section "Effects of Exceeding USB HID Rate Limitations".

A secondary limit is imposed in the form of the UART's maximum reliable baud rate of 115200 bits per second. This limit affects communication between the Master Device and the Emulation Slave Device as defined in Figure 9. Due to the way in which serial communication libraries are implemented, as well as the configurable nature of the UART's baud rate, it is possible to utilize this limit as a means to reduce the likelihood of exceeding the USB HID protocol's polling rate limitation.

The final restriction, performance of the ATmega32u4, given its 16MHz core frequency and 256KB of RAM makes it imperative that any software implementations running on the device be optimized to make use of as few steps, and as little processor time, as possible. Testing has not demonstrated any appreciable issues with performance of the device beyond those limitations imposed by the USB HID protocol.

Target Device Performance

The performance of the Target Device also imposes restrictions on the maximum usable emulation command rate. This comes about as an effect of assumptions taken when developing applications which expect human input, specifically the rate at which a human is capable of providing input. By providing input at rates which drastically exceed this it is possible to cause applications to freeze or even crash.

Rate Limiters

Due to the USB HID rate limitation of 1000Hz it is necessary to enforce a command rate limit which is responsible for restricting the rate at which commands are dispatched to the Emulation Slave Device. Rate limiting is ideally achieved by restricting the rate at which the Master Device dispatches emulation commands, and can be implemented using two separate methods.

Serial Baud Rate

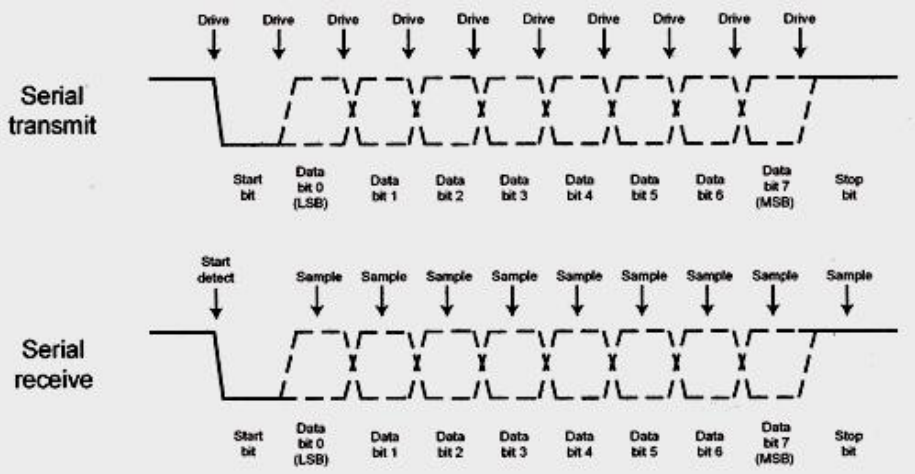

Figure 22 UART Byte Transmission Levels. Image courtesy of embedded [22].

The UART protocol operates through the use of start, stop and checksum bits on each transmitted byte of data – as show in **Error! Reference source not found.** – resulting in an effective byte length of 11 bits. By considering this, in conjunction with the minimum relevant packet size for the protocol as defined in **Error! Not a valid bookmark self-reference.**, it is possible to determine the ideal baud rate.

$$idealBaud = 11 \times 1000 \times packetSize$$

Making the assumption that the average common packet is between 2 and 3 bytes long for keyboard input, it makes sense to select a baud rate as close to 33000 as possible. As the nearest standard baud rate is 38400, it makes sense to select that as the primary transmission rate.

Library Integrated Rate Limiters

By implementing a rate limiter as part of the Master Device's software libraries it is possible to exert considerably more control over the rate at which commands are transmitted to the Emulation Slave device. This makes it possible to configure the rate at which commands are transmitted at runtime, without requiring modifications to the Emulation Slave Device's firmware.

This project makes use of two primary rate limiter types – blocking and asynchronous – depending on the platform. The C library makes use of a blocking rate limiter which will prevent a transmission request from completing until a specific delay has been achieved. The Node.js library on the other hand, to support the asynchronous nature of the Node.js

runtime, makes use of a limiter which stores pending commands in memory and dispatches them at the configured rate.

Optimal Command Execution Rate

After testing a number of rates, it has been concluded that a maximum command rate of 500Hz should be used for keyboard input. At this rate it is possible for basic text editors to receive large amounts of text in a performant manner – however editors which perform spell checking or syntax highlighting have in certain cases exhibited freezing at this emulation speed. For this reason, it is recommended that applications which intend to make use of emulation on a universal basis should instead use 125Hz as their maximum reporting rate – allowing for up to 62 characters of text entry per second.

For more information on the tests that were conducted to determine these rates please consult the Testing section of this report.

Communications Protocol

Communication between the Teensy and Raspberry Pi is an integral component of the project, requiring a structured protocol which ensures devices are able to interface in a reliable and efficient manner.

Given the nature of the target devices, the low level interconnects being used and the types of data being transmitted it is important to design a protocol that imposes a very low command overhead, minimizes generation and parsing load – both memory and CPU time - and maximizes simplicity. To address these requirements a basic binary protocol has been developed which operates within a fixed memory space, removing the need for heavy libraries such as malloc, allows the use of in-place memory type-casting rather than conversions and maximizes effective bandwidth usage through the implementation of variable length packets with optional parameters.

Protocol Requirements

The protocol was required to make use of short packets for each command to allow useful command rates over the UART – which is severely bandwidth limited when compared to other interconnect types.

The low processing power of the Teensy also imposed restrictions on the amount of processing which could be performed for each packet – ruling out any type of compression algorithm and encouraging a design which removed the need for value conversion. In the same vein, memory restrictions on the Teensy made it preferable to design a protocol which permitted fixed address-space parsing – removing the need for dynamic memory allocation and potential memory leaks.

It was also necessary to accommodate future expansions to the protocol's command set should it become necessary to emulate other HID device classes, or expand the range of features provided by the API.

Design Decisions

Packet Level Design

In order to maximize bandwidth availability the choice was made to allow variable length packets, with all parameters being optional. This, combined with a header field within each packet which was responsible for indicating the command type and number of parameters supplied, made it possible to design a protocol which offers full access to the emulation functionality of the Teensy while reducing the average command length to approximately 2-3 bytes.

In addition to this, the choice was made to combine the command type and parameter length information into a single byte, thereby restricting the number of parameters (and therefore the maximum packet length) to 31 bytes (with an additional byte for the header).

Parameters take the form of unsigned 1 byte integers, with the provision that larger values be composed by concatenating subsequent parameter's bits. This allows memory access typecasting to be used without any additional processor or memory overhead when working with almost any value type.

Implementation Level

It was also necessary to decide on how emulation functionality would be implemented within the protocol. Two primary options were available, with each offering advantages and disadvantages.

The first option was to offer the low level HID emulation API through the protocol, requiring the master device to generate the raw HID reports and using the Teensy as a proxy through which these reports would be transmitted. This approach would offer the ability to represent any HID device without requiring modifications to the Teensy's firmware. However it would require that master device libraries be responsible for creating valid HID reports and would significantly complicate their relevant codebases.

The second option was to create a very specific remote procedure call interface through which predefined emulation functions on the Teensy could be called. This approach would rely on the master device to report what emulation it wished to take place, and allow the Teensy's firmware to determine how this is achieved.

After considering both options, it was decided that the second would offer a more lightweight protocol – with the first option requiring more bandwidth to work correctly – as well as simpler master library implementations. As it is likely that there will be significantly more master libraries than firmware versions this makes sense as it allows updates to the Teensy's firmware to be used by any master library without significant changes. This choice also separates the master libraries – and by extension, applications on the Master Device – from implementation details. This hypothetically allows changes to the emulation hardware in future (for example, the transition to an ASIC, or even a software implementation on the host PC) without requiring changes to the software developed for the Master Device.

Packet Structure

Packets consist of a single 8-bit header field, followed by a variable number of 8-bit parameter fields. The header field is responsible for reporting the command type and number of parameters included within the packet, allowing simple parser implementations.

Table 1 illustrates the basic packet structure, with the horizontal axis representing bit-indices and the vertical axis representing byte-indices; in both cases utilizing a zero-based-index.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	OP_CODE			ARG_COUNT				
1	ARG_1							
…	…							
N	ARG_N							

Table 1 Packet Structure

This approach allows the command type (OP_CODE) and number of parameters (ARG_COUNT) to be extracted using a simple bit mask command such as the following.

```
int8 header = packet[0];
int8 command = (header & 0xe0) >> 5;
int8 parameters = header & 0x1f;
```

Figure 23 Example Header Parsing

Packet Types

Packet types, specified as the OP_CODE within a packet, are used to determine the functional endpoint to which commands are directed and, by extension, the emulation type to take place.

Table 2 lists the reserved OP_CODEs defined in this protocol, with both *0x0* and *0x7* reserved for end user customization and *0x4*, *0x5* and *0x6* reserved for future protocol extensions. Custom Operations are intended for customer specific modifications to the protocol, while Reserved operation codes are meant for future generic protocol expansion.

OP_CODE	Description
0x0 000	Custom Operation
0x1 001	Keyboard
0x2 010	Mouse
0x3 011	Joystick
0x4 100	Reserved For Future Expansion
0x5 101	Reserved For Future Expansion
0x6 110	Reserved For Future Expansion
0x7 111	Custom Operation

Table 2 Packet Command Types

Keyboard Commands

This command type is designed to enable keyboard emulation covering the full scope of the USB HID keyboard interface through a relatively simple, compact and flexible packet structure.

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Field	HEADER	MODIFIERS	KEY1	KEY2	KEY3	KEY4	KEY5	KEY6

Table 3 Keyboard Command Packet Format

In the interest of minimizing packet size, commands of this type make use of the variable packet length functionality provided by the protocol to allow the smallest amount of relevant data possible to be transmitted. This is achieved by placing the modifier keys argument prior to any key arguments, allowing only active keys to be sent and removing the need for zero-padding, thereby reducing the packet length for a single key press from 8 bytes to 3 bytes.

Packets which submit more than the maximum number of arguments will have extraneous arguments ignored in association with the functionality available through the implementing device. This allows devices which implement N-Key Roll Over (NKRO) [23] to support a higher number of active keys should they wish.

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Release All Keys	0x20							
Press A	0x22	0x00	0x04					
Press Shift+A	0x22	0x02	0x04					
Press Ctrl+Shift+A+B+C	0x24	0x03	0x04	0x05	0x06			

Table 4 Example Keyboard Emulation Packets

Example Command Breakdown

This section will breakdown the components of the “Press Shift+A” example command from Table 4. Specifically, the reasoning behind the selection of each byte in the packet.

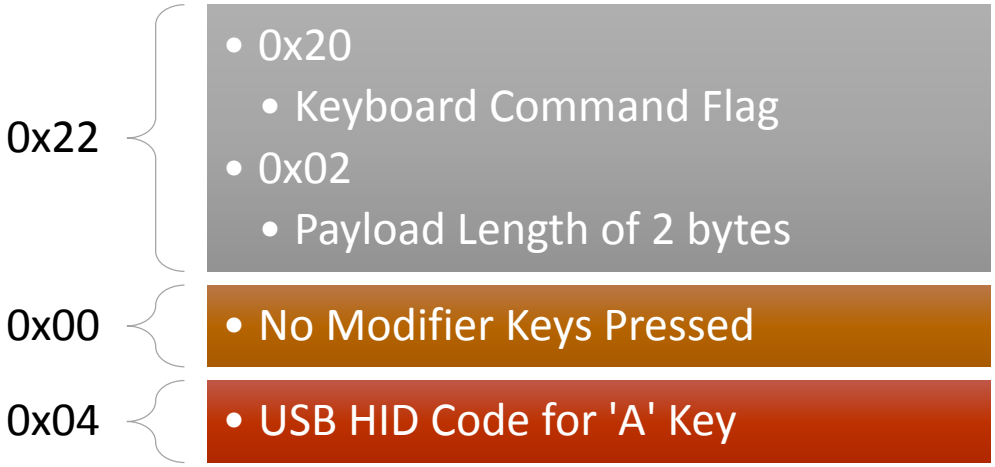

Figure 24 Example Keyboard Command Breakdown

Mouse Commands

The mouse command type is intended to allow flexible mouse control through the use of variable length packets which allow button, movement and scroll emulation (or a subset thereof). Mouse commands are represented through packets with the structure shown in Table 5.

Byte	**0**	**1**	**2**	**3**	**4**
Field	HEADER	BUTTONS	DX	DY	DSCROLL

Table 5 Mouse Command Packet Format

Button	**Bit Mask**
Left	**0x1** 0000 0001
Right	**0x2** 0000 0010
Middle	**0x4** 0000 0100

Table 6 Mouse Button Flags

To allow for the shortest possible command length for common operations, commands may be submitted with fewer than the maximum number of arguments. Extraneous arguments will be ignored by the implementation.

Examples of a number of different mouse emulation packets are listed in Table 7, indicating the hexadecimal bytes to be sent for each command.

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Release All Buttons	0x40							
Press Left Mouse Button	0x41	0x01						
Move Mouse Right 8 Units	0x42	0x00	0x08					
Scroll Up Two Clicks	0x44	0x00	0x00	0x00	0x02			

Table 7 Example Mouse Emulation Packets

Example Command Breakdown

Figure 25 breaks down the “Scroll Up Two Clicks” command from Table 7, explaining what each byte represents and how it is constructed.

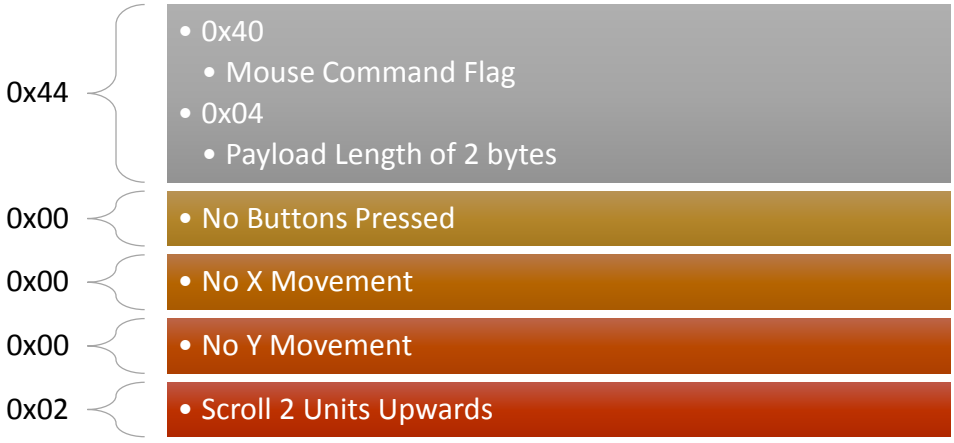

Figure 25 Example Mouse Command Breakdown

Joystick Commands

The joystick emulation command has been made available to allow the emulation of game input devices for the operation of simulators and other specialist applications. Unlike the keyboard and mouse commands, this command requires a degree of data manipulation in the form of packing.

The emulated joystick provides 32 buttons, 6 axes and a single 8-way hat switch. The joystick command packet has been designed to enable fully independent control over all of these inputs simultaneously. The packet structure can be seen in Table 8.

Byte	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Field	HEADER	BUTTONS				X_Y_Z				RZ_SL_SR				HAT

Table 8 Joystick Command Packet Format

In an effort to reduce the overall packet size, and given the 10-bit accuracy provided by the various axes, axis values are packet into 32-bit integer values following the algorithm defined in Figure 26.

```
int32 pack(int16 axis1, int16 axis2, int16 axis3) {
  int32 packed = 0;
  packed |= axis1; // Logical OR of packed and axis1
  packed <<= 10; // Left shift by 10-bits
  packed |= axis2; // Logical OR of packed and axis2
  packed <<= 10; // Left shift by 10-bits
  packed |= axis3; // Logical OR of packed and axis3
  return packed;
}
```

Figure 26 Axis Packing Algorithm (C/C++)

The hat switch's position is defined according to the layout provided in Table 9.

7	0	1
6	255	2
5	4	3

Table 9 Joystick Hat Switch Position Values

It is important to note that this command type does not, in general, respond well to submission of partial packets due to the way in which axes are handled. For this reason, it is advisable that you only transmit fully formed commands to this API to avoid strange behaviour.

Testing

Performance Limitations

Maximum Emulation Device Command Execution Rate

In this section tests were conducted to determine the amount of time taken by the Teensy to execute a keyboard emulation instruction. Times were measured using an oscilloscope, and by running a custom firmware image which set one of the Teensy's output pins to LOW prior to execution of the command and returned its state to HIGH at the conclusion of the command.

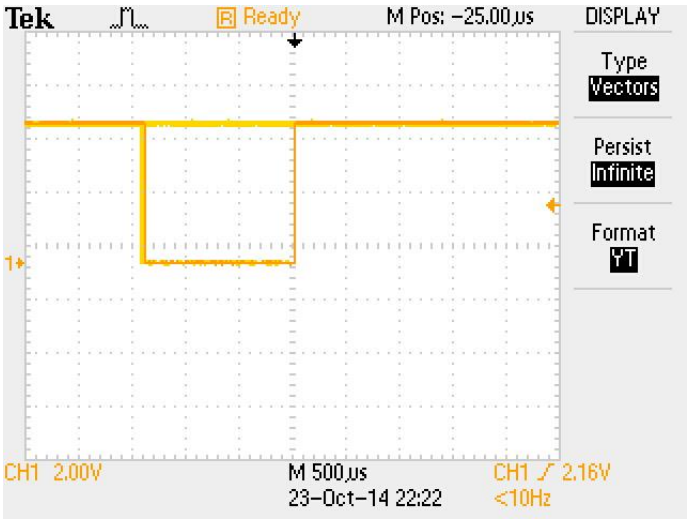

Figure 27 Command Execution Timing

Figure 27 shows the results of this measurement, indicating a total command execution time of 1500μs, representing a maximum command throughput of 650 commands per second. The following operations are performed during this time, starting with the receipt of the header byte.

- Test whether bytes available
- Clear receive buffer
- Receive header byte
- Determine validity of header byte
- Parse header into OP_CODE and ARGS_LENGTH
- Receive remaining arguments
- Setup emulation state
- Transmit emulation packet over USB

Of these operations, the reception of the header and arguments account for between 20-30% of the operation time with a roughly equivalent amount of time dedicated to the generation and transmission of the USB HID packet. The result is approximately 600μs spent on packet parsing and command processing, a reasonable amount of time given the Teensy's 16MHz core clock, and 8-bit processor.

Standard Text Transcription

In this section tests were conducted to determine the accuracy and performance of text input under standard operating conditions. The C library was configured to operate at a command rate of 500Hz and the UART to operate 115200 baud. A 3517 character text file was then piped to the isowrite command line application and the time taken to complete emulation of the file's typing was measured.

```
pi@isotope ~/Code/Isotope $ wc -c README.md
3517 README.md
pi@isotope ~/Code/Isotope $ time cat README.md | ./build/apps/rpi/isowrite

real    0m15.057s
user    0m0.120s
sys     0m0.420s
```

Figure 28 Text Input Command Execution Timing

Figure 28 shows the outputs of Linux's word count and time commands as used to measure these metrics. From these results it is possible to determine the average number of characters "typed" per second, giving a value of $\frac{3517}{15.057} = 233.579$. It is important to take into account the algorithm used for generating keyboard commands from a string of text when analysing this value – namely that a key release command is transmitted after each character, resulting in two emulation commands per character to be emulated.

When considering this, the time overhead of the system can be determined by the following equations.

$$idealRate = \frac{500}{2} = 250$$
$$actualRate = 233.58$$
$$totalOverhead = \frac{1}{actualRate} - \frac{1}{idealRate} = 6.703ms$$
$$averageOverhead = \frac{totalOverhead}{actualRate} = 28.69\mu s$$

Text Transcription Accuracy

In order to measure the accuracy with which text is entered, the same 3517 character text file was used to create a transcribed file which was then compared to the original to determine byte level differences.

The Linux commands shown in Figure 29 were used to run this test, and a short explanation of their purpose follows. A new interpreter is forked who's task it is to sleep for 5 seconds, and subsequently begin transcription of the README.md file. Upon completion of the transcription, the key commands Ctrl+O and ENTER are used to save the file and Ctrl+X is used to close nano. On the primary interpreter the nano text editor is then opened to accept the transcribed text, and the save commands once transcription is completed. Once the files have been saved and nano closed, diff is used to determine whether the files differ in any way.

```
(sleep 5 && cat README.md |  ./build/apps/rpi/isowrite && \
./build/apps/rpi/isokey -C o && \
sleep 1 && \
./build/apps/rpi/isokey ENTER && \
sleep 1 && \
./build/apps/rpi/isokey -C x) &
nano README_transcribed.md && \
diff -q README.md README_transcribed.md
```

Figure 29 Linux Commands Executed for Text Transcription Accuracy Test

Testing determined no appreciable differences between the files in cases where standard keyboard characters were used, however the ability of the system to emulate complex Unicode input – or lack thereof – hindered attempts at transcribing binary files or those in non-ASCII formats.

Effects of Exceeding USB HID Rate Limitations

In this section an investigation into the results of higher-than-specified command rates was investigated to determine the effect, if any, on emulation accuracy. This test was performed by configuring the UART to operate at a baud rate of 115200 and disabling rate limiting in the C library. A simple executable file which printed the alphabet five times was then run and the output captured in **Error! Reference source not found.**.

Expected Output

abcdefghijklmnopqrstuvxyzabcdefghijklmnopqrstuvxyzabcdefghij
klmnopqrstuvxyzabcdefghijklmnopqrstuvxyzabcdefghijklmnopqrs
tuvxyz

Actual Output

abcdefghijklmnopqrstuvwxyabcdefgjnnnnnnnnnnnnnnnn…

Figure 30 Test Output for Exceptional Emulation Rate

As can be seen in Figure 30**Error! Reference source not found.**, failure to adhere to the standard USB HID reporting rate can result in garbled output for even short pieces of text.

Electrical Coupling

Due to the different core operating voltages of the ATmega32u4, in the form of a Teensy 2.0, and the Raspberry Pi it is necessary to ensure safe coupling of the devices to ensure damage is not caused. There are two primary aspects of the design which fall under this restriction, namely the power supplies for the devices and the serial UART connection.

Power Supply Isolation

Power supply isolation dictates that at no point do the power supplies of the Raspberry Pi and Teensy interact in any way. Specifically that the Raspberry Pi may not draw power from the Teensy, and that the Teensy may not draw power from the Raspberry Pi.

Testing of this isolation was undertaken by supplying power to the Raspberry Pi and determining whether the Teensy received power. The Raspberry Pi was selected at the initial device due to its lower core voltage of 3.3V which would prevent damage to the Teensy. Following this test, the Teensy's emulation board was connected to a power supply and the Raspberry Pi's 3.3V and 5V rails were measured to determine whether any voltage was present – findings indicated that all Raspberry Pi related rails remained in a "hanging" state – unconnected.

Voltage Level Translation

The Serial UARTs on the Teensy and Raspberry Pi both operate at the chips respective core voltages – 5V and 3.3V respectively. To prevent damage to the UARTs it is necessary to perform voltage level translation from 3.3V to 5V and vice versa. This task is undertaken by the Texas Instruments TXB0104 [18] Bi-Directional Level Shifter.

To test the functioning of this device the Teensy and Raspberry Pi were connected to it and voltage measurements were taken, both on the Teensy and Raspberry Pi sides.

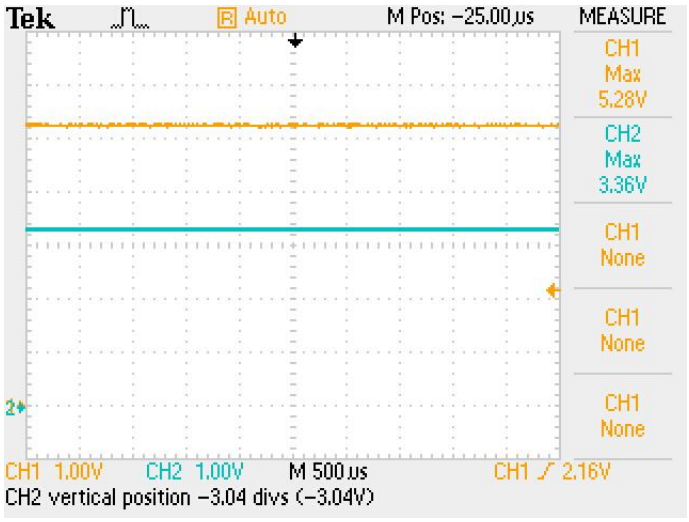

Figure 31 Voltage Level Translation Measurements

Figure 31 shows the voltage levels measured, yellow representing the Teensy side of the TXB0104 and cyan representing the Raspberry Pi side. Expected values are 5.0V and 3.3V respectively, representing a 5.6% and 1.8% deviation from normal in each case, well within design tolerances.

Future Expansion

This initial revision of the device represents a basic level of functionality required to allow USB input device emulation in an embedded device for speech recognition. There are a number of additional features which could be added at a later stage including a built in display and/or menu to allow customization of the device and/or the display of contextual information like processed commands.

It would also be possible to integrate the text-to-keypress conversion functionality on the emulation slave device, with a simple protocol extension allowing for transition between different keyboard layouts, thus removing the need to include this in implementation libraries and therefore reducing the workload on the master device.

It would also be possible to make use of PS/2 emulation as an alternative to USB HID if functionality such as NKRO (N-Key Roll Over) [23] was required, allowing for more than the 6 keys available through USB HID to be pressed at a time.

Possible Usage Scenarios

Aside from the intended usage as a speech control interface, the ability to emulate human input on a hardware level in a manner which is transparent to the target device's operating system presents a number of interesting usage scenarios.

Of particular interest is the application of this device in the bypassing of security systems such as Microsoft's User Access Control (UAC) since Windows Vista. By design, the Windows UAC prompt (shown in Figure 32) disables all software based mouse and keyboard emulation to prevent programmatic bypassing of the prompt. By resorting a hardware based solution it would be possible to avoid this security measure, allowing a malicious party to more easily bypass the security systems in place on secure systems.

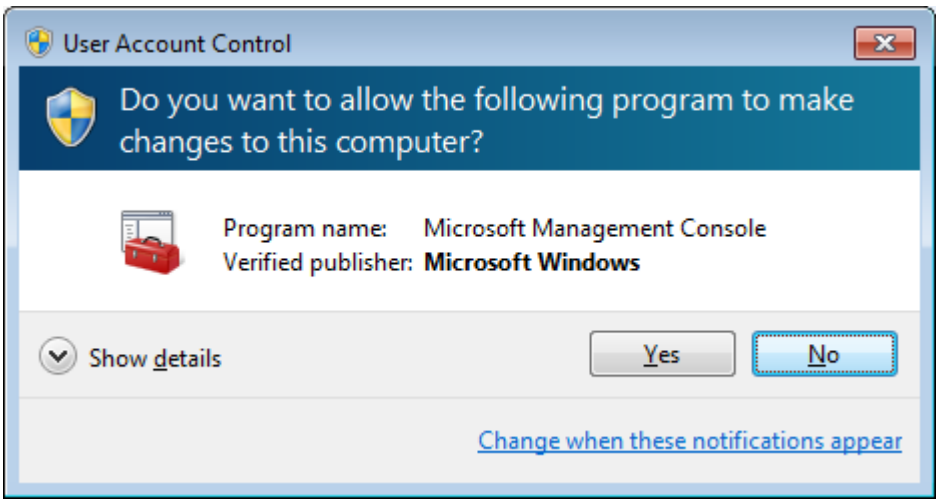

Figure 32 Microsoft Windows 7 UAC Prompt. Image courtesy of Michael Manley [24]

Another possible use, when combined with a VGA/DVI forwarder, is to allow the remote control of a system from the BIOS level upwards in much the same way as costly enterprise Out-of-band management options such as DELL's DRAC and IBM's Remote Supervisor Adapter.

Conclusion

The scope of this project covered the design, implementation and delivery of a USB HID emulation layer for embedded devices such as the Raspberry Pi. The primary requirement being an ability to emulate mouse and keyboard input on a target device without the requirement for custom drivers or software running on the target machine. Secondary requirements put an emphasis on simplicity, adaptability and ease of acquisition in order to permit usage within the academic community for research and prototyping of voice recognition and control systems.

Of the different options considered, it was determined that an ATmega32u4 in the form of a PJRC Teensy 2.0 [7] would present the best combination of low cost, small footprint and ready availability to satisfy these requirements. The issue of voltage level conversion between the master device's core voltage and that of the Teensy was solved through the use of a Texas Instruments TXB0104 automatic level translator which enables the completed device to quickly be adapted for use with master devices whose core voltages range from 1.2V to 3.6V without any physical changes to the device.

In addition to the physical device design, an efficient protocol for inter-device communication was developed and implemented in the form of a custom firmware image for the ATmega32u4 and a number of libraries for the Raspberry Pi. These libraries provide straightforward access to the different emulation options provided by the Teensy, including Mouse, Keyboard and Joystick emulation – as well as a number of helper functions to simplify common tasks such as text entry. Common prototyping requirements were addressed through the creation of a number of command line utility functions for keyboard and mouse emulation, permitting use of the device from shell scripts without the need for any actual programming.

Testing has revealed a maximum safe text transcription performance of approximately 230 characters per second, compared to the maximum human rate of approximately 12 characters per second [25]. This represents almost a 50-fold increase over standard human input rate. This, combined with the ease of implementation, makes the device an ideal candidate for the development of speech recognition and control interfaces for consumer computers.

References

[1] J. C. Perez, “Google wants your phonemes: InfoWorld,” InfoWorld, 23 October 2007. [Online]. Available: http://www.infoworld.com/t/data-management/google-wants-your-phonemes-539. [Accessed 26 07 2014].

[2] USB Implementers Forum, Inc., “Device Class Definition for HID 1.11,” USB Implementers Forum, Inc., 27 June 2001. [Online]. Available: http://www.usb.org/developers/hidpage/HID1_11.pdf. [Accessed 8 October 2014].

[3] FTDI, “FT232R,” Future Technology Devices International Ltd., [Online]. Available: http://www.ftdichip.com/Products/ICs/FT232R.htm. [Accessed 07 August 2014].

[4] FTDI, “Vinculum II,” Future Technology Devices International Ltd., [Online]. Available: http://www.ftdichip.com/Products/ICs/VNC2.htm. [Accessed 07 August 2014].

[5] Atmel Corporation, “ATmega32u4,” Atmel, [Online]. Available: http://www.atmel.com/devices/atmega32u4.aspx. [Accessed 11 August 2014].

[6] Arduino, “Arduino Nano,” Arduino, [Online]. Available: http://arduino.cc/en/Main/arduinoBoardNano. [Accessed 2014 August 11].

[7] PJRC, “Teensy 2.0,” PJRC, [Online]. Available: https://www.pjrc.com/store/teensy.html. [Accessed 2014 August 11].

[8] C. Truter, “Pi Your Command,” Stellenbosch University, Stellenbosch, 2013.

[9] Adafruit Industries, “Raspberry Pi Model B 512MB RAM,” Raspberry Pi Foundation, [Online]. Available: https://www.adafruit.com/product/998. [Accessed 18 August 2014].

[10] Adafruit Industries, “BeagleBone Black Rev C,” BeagleBone, [Online]. Available: http://www.adafruit.com/products/1876. [Accessed 2 September 2014].

[11] Adafruit Industries, “Intel Calileo Development Board,” Intel Corporation, [Online]. Available: http://www.adafruit.com/products/1637. [Accessed 2 September 2014].

[12] Banana Pi, “Banana Pi,” Banana Pi, [Online]. Available: http://www.bananapi.org/p/product.html. [Accessed 06 10 2014].

[13] SolidRun, “HummingBoard,” SolidRun, [Online]. Available: http://www.solid-run.com/products/hummingboard/. [Accessed 06 10 2014].

[14] Raspberry Pi Foundation, “Raspberry Pi,” Raspberry Pi Foundation, [Online]. Available: http://www.raspberrypi.org/. [Accessed 11 August 2014].

[15] IanH, “RPi GPIO Interface Circuits,” eLinux.org, 30 August 2013. [Online]. Available: http://elinux.org/RPi_GPIO_Interface_Circuits. [Accessed 11 August 2014].

[16] Atmel, “ATmega32u4 Datasheet,” [Online]. Available: http://magicshifter.net/static/datasheets/atmega32u4.pdf. [Accessed 18 August 2014].

[17] Mosaic Industries, “Raspberry Pi GPIO Pin Electrical Specifications,” Broadcom, [Online]. Available: http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications. [Accessed 18 August 2014].

[18] Texas Instruments, “TXB0104,” Texas Instruments, [Online]. Available: http://www.ti.com/product/txb0104. [Accessed 11 August 2014].

[19] Texas Instruments, “TXB0104 Technical Documentation,” May 2012. [Online]. Available: http://www.ti.com/lit/gpn/txb0104. [Accessed 11 August 2014].

[20] Adafruit Industries, “TXB0104 Bi-Directional Level Shifter,” Texas Instruments, [Online]. Available: https://www.adafruit.com/products/1875. [Accessed 18 August 2014].

[21] Adafruit, “Adafruit Prototyping Pi Plate Kit for Raspberry Pi,” Adafruit, [Online]. Available: https://www.adafruit.com/products/801. [Accessed 11 August 2014].

[22] M. I. P. I. Aaron Minor, “Implementing your MCU-based system's serial UART in software,” 02 Feburary 2007. [Online]. Available: http://www.embedded.com/design/other/4025995/Implementing-your-MCU-based-system-s-serial-UART-in-software. [Accessed 21 October 2014].

[23] “Rollover (key),” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Rollover_%28key%29. [Accessed 15 October 2014].

[24] M. Manley, “User:Mmanley,” [Online]. Available: http://en.wikipedia.org/wiki/User:Mmanley. [Accessed 27 October 2014].

[25] V. Gable, “How Fast Do People Type?,” 5 December 2007. [Online]. Available: https://imlocation.wordpress.com/2007/12/05/how-fast-do-people-type/. [Accessed 25 October 2014].

[26] Adafruit Industries, “Teensy (ATmega32u4 USB dev board) 2.0,” PJRC, [Online]. Available: https://www.adafruit.com/products/199. [Accessed 18 August 2014].

[27] C. Williams, “node-serialport,” GitHub, 5 September 2014. [Online]. Available: https://github.com/voodootikigod/node-serialport. [Accessed 20 October 2014].

[28] B. Goodwine, “RS-232 Serial Protocol,” 29 September 2002. [Online]. Available: http://controls.ame.nd.edu/microcontroller/main/node24.html. [Accessed 11 August 2014].

Appendix A: Project Planning Schedule

1. Functional Specification
2. Existing Implementation Research
3. Initial Design
4. Feasibility and Limitation Study
5. Refinement of Design, Parts List
6. Ordering of Components
7. Initial Software Design
8. Assembly
9. Software Testing
10. Platform Expansion – new libraries, demos, documentation

Appendix B: Project Specifications

Non-Functional Specifications

N-001 No Custom Drivers

Device must make use of the USB HID protocol for communication to ensure that it does not require the development and installation of custom drivers.

N-002 Keyboard Emulation

Device must be able to emulate a keyboard, allowing the pressing and releasing of individual keys, as well as combinations of keys and/or modifiers like Ctrl, Alt or Shift.

N-003 Mouse Emulation

Device must be able to emulate a mouse, supporting movement of the mouse cursor in the X and Y axis, pressing and releasing of the Left, Middle and Right mouse buttons and scrolling up or down by emulation of a scroll wheel.

N-004 Raspberry Pi Integration

Device must be able to be used in conjunction with a Raspberry Pi Model B, with no modifications to the Raspberry Pi itself so as to minimize the risk of damage.

N-005 High Compatibility

Device must make use of a commonly available communication interface available on most embedded systems to allow adaptation to different platforms in the future.

Functional Specifications

F-001 High Performance

The device and protocol should ensure that any performance limitations are dictated by the USB HID protocol rather than the implementation. Specifically, the inter-device communication bus should be able to transmit at least 1000 commands per second.

F-002 Low Power Usage

The device should not require more power than can be supplied through the USB port of the system it is connected to – 500mA on USB2.0 and 1200mA on USB2.0 High-Power or USB3.0.

F-003 Low Cost

The device should be cheap to prototype as well as having low cost components such that mass production profit margins may be maximized. Maximum cost of the emulation components for prototyping may not exceed R500 while production costs shall not exceed R100.

F-004 Small Size

The device should require a minimum of PCB surface area such that production devices may be built to be extremely small and lightweight so as to be easily portable.

Appendix C: Outcomes Compliance

ECSA Outcome	Description	Addressed In
Problem Solving	The process of USB HID emulation on an embedded device was investigated, a number of solutions were considered and the best option for the project selected and implement.	• Introduction • Pre-Design Investigation • Appendix B: Project Specifications
Application of Scientific & Engineering Knowledge	Implementation of the hardware interfaces, software libraries and communications protocols through which emulation takes place – as well as a considered approach to performance and functionality testing.	• Device Design • Software Design • **Error! Reference source not found.**
Engineering Design	A structured approach to the system design consisting of research into feasible solutions, the design of communications protocols based on the system's requirements	• Universal Serial Bus Background • Pre-Design Investigation • Demonstration Device Selection • Device Design • Software Design • Performance Considerations Performance of the Emulation Slave Device is dictated by three primary factors: the UART's bandwidth, the USB HID specification and the performance of the ATmega32u4 used for emulation tasks. Of these three factors, the USB HID specification is the most restrictive in terms of performance due to the polling nature of the protocol. The limit affects communication between

	and a structured approach to cross platform software development was used to ensure that the project was completed on time and with all requirements met.	the Emulation Slave Device and Master Device as defined in Figure 9 – which is limited to a maximum command rate of 1000Hz. This limitation requires that the software is designed to never exceed a reporting rate of 1000 commands per second so as to avoid emulation issues. The effect of failure to adhere to this rate is investigated in the section “Effects of Exceeding USB HID Rate Limitations”. A secondary limit is imposed in the form of the UART’s maximum reliable baud rate of 115200 bits per second. This limit affects communication between the Master Device and the Emulation Slave Device as defined in Figure 9. Due to the way in which serial communication libraries are implemented, as well as the configurable nature of the UART’s baud rate, it is possible to utilize this limit as a means to reduce the likelihood of exceeding the USB HID protocol’s polling rate limitation. The final restriction, performance of the ATmega32u4, given its 16MHz core frequency and 256KB of RAM makes it imperative that any software implementations running on the device be optimized to make use of as few steps, and as little processor time, as possible. Testing has not demonstrated any appreciable issues with performance of the device beyond those limitations imposed by the USB HID protocol. Target Device Performance The performance of the Target Device also imposes restrictions on the maximum usable emulation command rate. This comes about as an effect of assumptions taken when developing applications which expect human input, specifically the rate at which a human is capable of providing input. By providing input at rates which drastically exceed this it is possible to cause applications to freeze or even crash. Rate Limiters Due to the USB HID rate limitation of 1000Hz it is necessary to enforce a command rate limit which is responsible for restricting the rate at which commands are dispatched to the Emulation Slave Device. Rate limiting is ideally achieved by restricting the rate at which the Master Device dispatches emulation commands, and can be implemented using two separate methods.

Serial Baud Rate

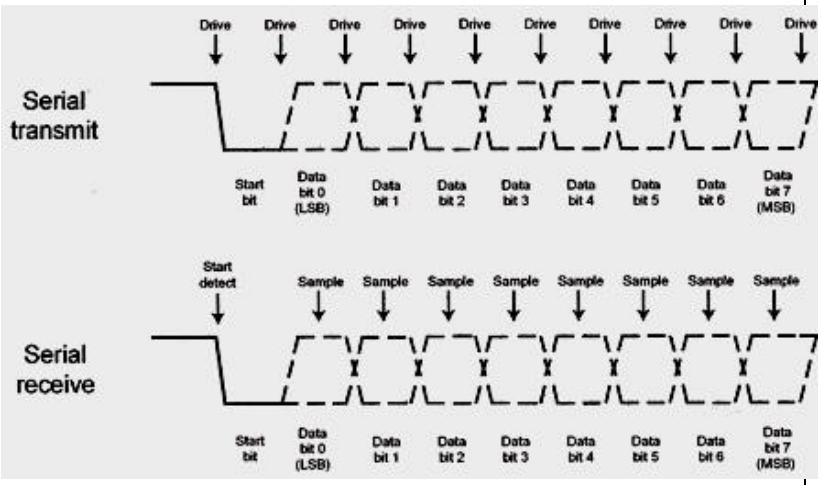

Figure 22 UART Byte Transmission Levels. Image courtesy of embedded .

The UART protocol operates through the use of start, stop and checksum bits on each transmitted byte of data – as show in **Error! Reference source not found.** – resulting in an effective byte length of 11 bits. By considering this, in conjunction with the minimum relevant packet size for the protocol as defined in **Error! Not a valid bookmark self-reference.**, it is possible to determine the ideal baud rate.

$$idealBaud = 11 \times 1000 \times packetSize$$

Making the assumption that the average common packet is between 2 and 3 bytes long for keyboard input, it makes sense to select a baud rate as close to 33000 as possible. As the nearest standard baud rate is 38400, it makes sense to select that as the primary transmission rate.

Library Integrated Rate Limiters

By implementing a rate limiter as part of the Master Device's software libraries it is possible to exert considerably more control over the rate at which commands are transmitted to the Emulation Slave device. This makes it possible to configure the rate at which commands are transmitted at runtime, without requiring modifications to the Emulation Slave Device's firmware.

This project makes use of two primary rate limiter types – blocking and asynchronous – depending on the platform. The C library makes use of a blocking rate limiter which will prevent a transmission request from completing until a specific delay has been achieved. The Node.js library on the other hand, to support the asynchronous nature of the Node.js runtime, makes use of a limiter which stores pending commands in memory and dispatches them at the configured rate.

Optimal Command Execution Rate

		After testing a number of rates, it has been concluded that a maximum command rate of 500Hz should be used for keyboard input. At this rate it is possible for basic text editors to receive large amounts of text in a performant manner – however editors which perform spell checking or syntax highlighting have in certain cases exhibited freezing at this emulation speed. For this reason, it is recommended that applications which intend to make use of emulation on a universal basis should instead use 125Hz as their maximum reporting rate – allowing for up to 62 characters of text entry per second. For more information on the tests that were conducted to determine these rates please consult the Testing section of this report. • Communications Protocol • **Error! Reference source not found.**
Investigations, Experiments and Data Analysis	Research into available hardware and software to perform the requisite tasks, as well as testing of the final device to determine its adequacy for the adopted task – both through the use of automated testing scripts and electronic measuring devices.	• Pre-Design Investigation • Demonstration Device Selection • **Error! Reference source not found.**
Engineering Methods, Skills and Tools, Including Information Technology	The use of a number of programming languages and frameworks to allow flexible use of the system. A design focused on future extensibility,	• Software Design • Performance Considerations Performance of the Emulation Slave Device is dictated by three primary factors: the UART's bandwidth, the USB HID specification and the performance of the ATmega32u4 used for emulation tasks. Of these three factors, the USB HID specification is the most restrictive in terms of performance due to the polling nature of the protocol. The limit affects communication between the Emulation Slave Device and Master Device as defined in Figure 9 – which is limited to a maximum command rate of 1000Hz. This limitation requires that the software is

	cross platform compatibility and ease of construction so as to simplify academic use.	designed to never exceed a reporting rate of 1000 commands per second so as to avoid emulation issues. The effect of failure to adhere to this rate is investigated in the section “Effects of Exceeding USB HID Rate Limitations”. A secondary limit is imposed in the form of the UART’s maximum reliable baud rate of 115200 bits per second. This limit affects communication between the Master Device and the Emulation Slave Device as defined in Figure 9. Due to the way in which serial communication libraries are implemented, as well as the configurable nature of the UART’s baud rate, it is possible to utilize this limit as a means to reduce the likelihood of exceeding the USB HID protocol’s polling rate limitation. The final restriction, performance of the ATmega32u4, given its 16MHz core frequency and 256KB of RAM makes it imperative that any software implementations running on the device be optimized to make use of as few steps, and as little processor time, as possible. Testing has not demonstrated any appreciable issues with performance of the device beyond those limitations imposed by the USB HID protocol. Target Device Performance The performance of the Target Device also imposes restrictions on the maximum usable emulation command rate. This comes about as an effect of assumptions taken when developing applications which expect human input, specifically the rate at which a human is capable of providing input. By providing input at rates which drastically exceed this it is possible to cause applications to freeze or even crash. Rate Limiters Due to the USB HID rate limitation of 1000Hz it is necessary to enforce a command rate limit which is responsible for restricting the rate at which commands are dispatched to the Emulation Slave Device. Rate limiting is ideally achieved by restricting the rate at which the Master Device dispatches emulation commands, and can be implemented using two separate methods.

Serial Baud Rate

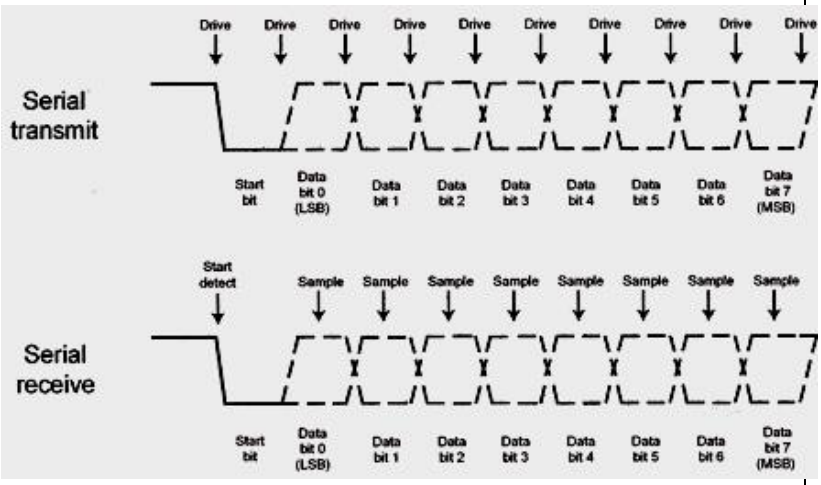

Figure 22 UART Byte Transmission Levels. Image courtesy of embedded .

The UART protocol operates through the use of start, stop and checksum bits on each transmitted byte of data – as show in **Error! Reference source not found.** – resulting in an effective byte length of 11 bits. By considering this, in conjunction with the minimum relevant packet size for the protocol as defined in **Error! Not a valid bookmark self-reference.**, it is possible to determine the ideal baud rate.

$$idealBaud = 11 \times 1000 \times packetSize$$

Making the assumption that the average common packet is between 2 and 3 bytes long for keyboard input, it makes sense to select a baud rate as close to 33000 as possible. As the nearest standard baud rate is 38400, it makes sense to select that as the primary transmission rate.

Library Integrated Rate Limiters

By implementing a rate limiter as part of the Master Device's software libraries it is possible to exert considerably more control over the rate at which commands are transmitted to the Emulation Slave device. This makes it possible to configure the rate at which commands are transmitted at runtime, without requiring modifications to the Emulation Slave Device's firmware.

This project makes use of two primary rate limiter types – blocking and asynchronous – depending on the platform. The C library makes use of a blocking rate limiter which will prevent a transmission request from completing until a specific delay has been achieved. The Node.js library on the other hand, to support the asynchronous nature of the Node.js runtime, makes use of a limiter which stores pending commands in memory and dispatches them at the configured rate.

Optimal Command Execution Rate

		After testing a number of rates, it has been concluded that a maximum command rate of 500Hz should be used for keyboard input. At this rate it is possible for basic text editors to receive large amounts of text in a performant manner – however editors which perform spell checking or syntax highlighting have in certain cases exhibited freezing at this emulation speed. For this reason, it is recommended that applications which intend to make use of emulation on a universal basis should instead use 125Hz as their maximum reporting rate – allowing for up to 62 characters of text entry per second. For more information on the tests that were conducted to determine these rates please consult the Testing section of this report. • Communications Protocol
Professional and Technical Communication	This report and the presentation of the device.	• Software Design
Independent Learning Ability	The student was responsible for performing his own research, making design decisions and implementing the system with minimal assistance from 3rd parties. Additional skills were developed as necessary to complete the project.	• Universal Serial Bus Background • Pre-Design Investigation • Conclusion The scope of this project covered the design, implementation and delivery of a USB HID emulation layer for embedded devices such as the Raspberry Pi. The primary requirement being an ability to emulate mouse and keyboard input on a target device without the requirement for custom drivers or software running on the target machine. Secondary requirements put an emphasis on simplicity, adaptability and ease of acquisition in order to permit usage within the academic community for research and prototyping of voice recognition and control systems. Of the different options considered, it was determined that an ATmega32u4 in the form of a PJRC Teensy 2.0 would present the best combination of low cost, small footprint and ready availability to satisfy these requirements. The issue of voltage level conversion between the master device's core voltage and that of the Teensy was solved through the use of a Texas Instruments TXB0104 automatic level translator which enables the completed device to quickly be adapted for use with master devices whose core voltages range from 1.2V to 3.6V without any physical changes to the device. In addition to the physical device design, an efficient protocol for inter-device communication was developed and implemented in the form of a custom firmware image for the ATmega32u4 and a number of libraries for the

		Raspberry Pi. These libraries provide straightforward access to the different emulation options provided by the Teensy, including Mouse, Keyboard and Joystick emulation – as well as a number of helper functions to simplify common tasks such as text entry. Common prototyping requirements were addressed through the creation of a number of command line utility functions for keyboard and mouse emulation, permitting use of the device from shell scripts without the need for any actual programming. Testing has revealed a maximum safe text transcription performance of approximately 230 characters per second, compared to the maximum human rate of approximately 12 characters per second . This represents almost a 50-fold increase over standard human input rate. This, combined with the ease of implementation, makes the device an ideal candidate for the development of speech recognition and control interfaces for consumer computers. • References

Appendix D: Source Code

Certain parts of the implementation, including the Node.js library, have been released under permissive open source licences and are publicly available on the following websites.

- Isotope for Node.js – https://npmjs.org/package/libisotope

Required Tools

- Git (**apt-get install git**)
- GCC and GNU Make (**apt-get install gcc make**)
- Arduino IDE (http://www.arduino.cc/en/Main/Software)
- PJRC Teensyduino (https://www.pjrc.com/teensy/td_download.html)

Process of Acquisition

Once a git client has been installed on your development machine, simply run **git clone https://github.com/SierraSoftworks/Isotope.git** to download the full source tree.

Compilation of Libraries

Preparation of the development environment and compilation of the libraries, examples and applications is undertaken through the use of GNU Make. To compile the libraries simply run **make** from within the source tree.

Additional make options include the following:

1. **make all** – Executes clean, c and test in sequence – the default option.
2. **make clean** – Cleans the build directory to prepare for a new build
3. **make c** – Builds the C libraries for both UART and File IO.
4. **make js** – Prepares the Node.js library by resolving all dependencies.
5. **make test** – Builds the various testing applications which can be used to ensure aspects of the code work as expected.
6. **make rpi** – Builds the Raspberry Pi libraries, examples and applications – ignoring the File IO alternatives.
7. **make publish** – Publishes the latest version of the Node.js JavaScript library to the NPM repository.

Directory Structure

- **build** – Target directory for compiled libraries and binaries. Results are organized into **{class}/{target}** where class can represent apps, examples or libs and target is either rpi or file.
- **docs** – Various documentation relating to the device and this report, includes configuration instructions for the Raspberry Pi's UART.
- **examples** – Source code for various example applications which demonstrate the use of the device.
- **src** – Source files for the various libraries and firmwares used for the project. Organized into apps, libs and teensy.
- **utils** – Various utility scripts which simplify development, includes a script which is responsible for sharing internet on a Linux device with the Raspberry Pi.

Appendix E: Component Information

All components were sourced from Adafruit Industries, through their online store, and shipped to South Africa using United Parcel Service Worldwide Expedited shipping.

Component Name	Price	Quantity
Raspberry Pi Model B 512MB RAM [9]	$39.95	1
Teensy 2.0 – ATmega32u4 [26]	$15.95	1
Adafruit Prototyping Pi Plate Kit for Raspberry Pi [21]	$15.95	1
TXB0104 Bi-Directional Level Shifter [20]	$3.50	1
Extra-long break-away 0.1” 16-pin strip male header (5-pieces)	$3.00	1
USB cable – 8” A to Mini B Charging and Micro B Data	$3.95	1

Table 10 Component List

Component Name	Website
Raspberry Pi Model B	http://www.raspberrypi.org/product/model-b/
Teensy 2.0	https://www.pjrc.com/teensy/
TXB0104	http://www.ti.com/product/txb0104
Prototyping Plate	https://www.adafruit.com/products/801

Table 11 Component Websites

Appendix F: C-Library Source Code

The following code represents the latest C library code available for use with an Isotope device. It includes support for the Raspberry Pi's UART through the **termios.h** header, and falls back on file IO on devices which lack a UART connection, or for execution of unit tests.

```
1.  /**
2.   * Isotope C Library
3.   * Provides a C interface between the Isotope emulation chip
4.   * and the local system, as well as a number of useful command
5.   * wrappers.
6.   *
7.   * Copyright Â© Benjamin Pannell 2014
8.   */
9.  #ifndef ISOTOPE_C
10. #define ISOTOPE_C
11.
12. #include "isotope.h"
13. #include "keylayouts.h"
14. #include <stdio.h>
15. #include <string.h>
16. #include <time.h>
17.
18. #ifdef RPI
19. #define ISOTOPE_IO
20.
21. #include <fcntl.h>
22. #include <unistd.h>
23. #include <termios.h>
24.
25. #else
26. #define ISOTOPE_FIO
27.
28. #endif
29.
30. /*
31.  * Internal Functions
32.  */
33. int _isotope_pack(short axis1, short axis2, short axis3) {
34.     return 0x00000000 | ((axis1 & 0x3ff) << 20) | ((axis2 & 0x3ff) << 10) | (axis3 &
     0x3ff);
35. }
36.
37. clock_t _isotope_lastwrite = 0;
38. int isotope_maxRate = 500;
39. void _isotope_ratelimit() {
40.     clock_t now;
41.     double delay;
42.     double minDelay;
43.
44.     if(!isotope_maxRate) return;
45.
46.     now = clock();
47.     delay = (now - _isotope_lastwrite) * 1.0 / CLOCKS_PER_SEC;
48.     minDelay = 1.0 / isotope_maxRate;
49.
50.     if(delay < minDelay)
51.         usleep(1e6 * (minDelay - delay));
52.     _isotope_lastwrite = clock();
53. }
54.
55. /*
56.  * Public API Functions
57.  */
58.
```

```
59. int isotope_open(const char* device) {
60.     int uart;
61. 
62.     #ifdef ISOTOPE_FIO // f Prefixed IO operations (fopen, fclose etc.)
63.     uart = (int)fopen(device, "w");
64.     if(!uart) return 0;
65.     #endif
66. 
67.     #ifdef ISOTOPE_IO // Standard IO operations
68.     uart = open(device, O_RDWR | O_NOCTTY | O_NDELAY);
69.     if(-1 == uart) return 0;
70.     #endif
71. 
72.     /**
73.      * Raspberry Pi specific UART configuration
74.      * Adapted from http://www.raspberry-projects.com/pi/programming-in-c/uart-
    serial-port/using-the-uart
75.      */
76. 
77.     #ifdef RPI
78.     struct termios options;
79. 
80.     tcgetattr(uart, &options);
81.     options.c_cflag = B115200 | CS8 | CLOCAL;       //<Set baud rate
82.     options.c_iflag = PARENB;
83.     options.c_oflag = 0;
84.     options.c_lflag = 0;
85.     tcflush(uart, TCIFLUSH);
86.     tcsetattr(uart, TCSANOW, &options);
87.     #endif
88. 
89.     return uart;
90. }
91. 
92. char isotope_close(int handle) {
93.     #ifdef ISOTOPE_FIO
94.     return (char)fclose((FILE*)handle);
95.     #endif
96.     #ifdef ISOTOPE_IO
97.     return close(handle);
98.     #endif
99. }
100. 
101. char isotope_write(int isotope, const char* packet, char packet_length) {
102.         _isotope_ratelimit();
103. 
104.     #ifdef ISOTOPE_FIO
105.     return fwrite(packet, sizeof(char), packet_length, (FILE*)isotope);
106.     #endif
107.     #ifdef ISOTOPE_IO
108.     return write(isotope, packet, packet_length);
109.     #endif
110. }
111. 
112. int isotope_read(int isotope, char* buffer, int length) {
113.         #ifdef ISOTOPE_FIO
114.         return fread(buffer, sizeof(char), length, (FILE*)isotope);
115.         #endif
116.         #ifdef ISOTOPE_IO
117.         return read(isotope, buffer, length);
118.         #endif
119. }
120. 
121. char isotope_mouse(int isotope, char buttons, char deltaX, char deltaY, char deltaSc
    roll) {
122.     char packet[5] = { 0x40 }, length = 4;
```

```
123.
124.     packet[1] = buttons;
125.     packet[2] = deltaX;
126.     packet[3] = deltaY;
127.     packet[4] = deltaScroll;
128.
129.     // Compress the packet to only send the required data
130.     if(!deltaScroll) {
131.         length--;
132.         if(!deltaY) {
133.             length--;
134.             if(!deltaX) {
135.                 length--;
136.                 if(!buttons) length--;
137.             }
138.         }
139.     }
140.
141.     // Populate the header's length field
142.     packet[0] |= length;
143.
144.     return isotope_write(isotope, packet, length + 1);
145. }
146.
147. char isotope_keyboard(int isotope, char modifiers, const char keys[], char keys_coun
    t) {
148.     char packet[8] = { 0x20 }, i = 0;
149.     packet[1] = modifiers;
150.
151.     if(!keys_count && !modifiers) return isotope_write(isotope, packet, 1);
152.
153.     for(i = 0; i < keys_count; i++) packet[i + 2] = keys[i];
154.     packet[0] |= keys_count + 1;
155.
156.     return isotope_write(isotope, packet, keys_count + 2);
157. }
158.
159. char isotope_joystick(int isotope, int buttons, short x, short y, short z, short rz,
     short sliderLeft, short sliderRight, char hat) {
160.     char packet[14] = { 0x60 | 13 };
161.
162.     // Copy buttons into packet
163.     *((int*)(packet + 1)) = buttons;
164.
165.     // Copy X, Y, Z axis pack into packet
166.     *((int*)(packet + 5)) = _isotope_pack(x, y, z);
167.
168.     // Copy rZ, sL, sR axis pack into packet
169.     *((int*)(packet + 9)) = _isotope_pack(rz, sliderLeft, sliderRight);
170.
171.     // Copy hat switch position into packet
172.     packet[13] = hat;
173.
174.     return isotope_write(isotope, packet, 14);
175. }
176.
177.
178. const char* _isotope_immutable = "\n\t ";
179. const char* _isotope_mutable_normal = "abcdefghijklmnopqrstuvwxyz1234567890-
    =[]\\;'`,./";
180. const char* _isotope_mutable_shifted = "ABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()_+{}|:\"
    ~<>?";
181. const char _isotope_map_immutable[] = {40, 43, 44};
182. const char _isotope_map_mutable[] = {
183.     4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
184.     30,31,32,33,34,35,36,37,38,39,
```

```
    45,46,47,48,49,51,52,53,54,55,56
};

int isotope_text(int isotope, const char* text) {
    int written = 0;
    char current, *index;
    char key = 0, modifiers = 0;

    while(current = *text++) {
        modifiers = 0;
        key = 0;
        if(index = strchr(_isotope_immutable, current))
            key = _isotope_map_immutable[index - _isotope_immutable];
        else if(index = strchr(_isotope_mutable_normal, current))
            key = _isotope_map_mutable[index - _isotope_mutable_normal];
        else if(index = strchr(_isotope_mutable_shifted, current)) {
            key = _isotope_map_mutable[index - _isotope_mutable_shifted];
            modifiers = MODIFIERKEY_SHIFT;
        } else continue;


        written += isotope_keyboard(isotope, modifiers, &key, 1);
        written += isotope_keyboard(isotope, 0, 0, 0);
    }
    written += isotope_keyboard(isotope, 0, 0, 0);
    return written;
}

#endif
```

Appendix G: Teensy Firmware Source Code

The following represents the latest Teensy 2.0 firmware for use with Isotope, it includes support for Keyboard, Mouse and Joystick emulation and will flash the LED whenever a command is received.

```
#define READ_TIMEOUT 500

#define OP_CUSTOM 0x00
#define OP_KEYBOARD 0x1
#define OP_MOUSE 0x2
#define OP_JOYSTICK 0x3

char rx_buffer[32] = {0};

// Handler functions
void on_custom();
void on_keyboard();
void on_mouse();
void on_joystick();

// Protocol utility functions
char read_blocking(char* target, int max_attempts);
char proto_opcode(char header);
char proto_args(char header);
short proto_short(char start);
int proto_int(char start);

void setup() {
  // Initialize the UART
  Serial1.begin(38400);

  // Enable manual mode for Joystick timing
  Joystick.useManualSend(true);

  // Turn on the LED
  pinMode(11, OUTPUT);
  digitalWrite(11, LOW);
}

void loop() {
  char i, opcode, args;

  if(!Serial1.available()) return;

  digitalWrite(11, HIGH);

  // Reset the buffer
  for(i = 0; i < 32; i++) rx_buffer[i] = 0;

  // Read the header
  if(!read_blocking(rx_buffer, READ_TIMEOUT)) {
    digitalWrite(11, LOW);
    return;
  }


  // Process the header values and store their outputs
  opcode = proto_opcode(rx_buffer[0]);
  args = proto_args(rx_buffer[0]);

  // Read the arguments
  for(i = 1; i <= args; i++)
    if(!read_blocking(rx_buffer + i, READ_TIMEOUT)) {
      digitalWrite(11, LOW);
```

```
60.       return;
61.     }
62.
63.   // Call the function responsible for the relevant op-code
64.   switch(opcode) {
65.     case OP_CUSTOM:
66.       on_custom();
67.       break;
68.     case OP_KEYBOARD:
69.       on_keyboard();
70.       break;
71.     case OP_MOUSE:
72.       on_mouse();
73.       break;
74.     case OP_JOYSTICK:
75.       on_joystick();
76.       break;
77.      default:
78.       break;
79.   }
80.
81.   digitalWrite(11, LOW);
82. }
83.
84. // Handler functions
85. void on_custom() {
86. }
87.
88. void on_keyboard() {
89.   Keyboard.set_modifier(rx_buffer[1]);
90.   Keyboard.set_key1(rx_buffer[2]);
91.   Keyboard.set_key2(rx_buffer[3]);
92.   Keyboard.set_key3(rx_buffer[4]);
93.   Keyboard.set_key4(rx_buffer[5]);
94.   Keyboard.set_key5(rx_buffer[6]);
95.   Keyboard.set_key6(rx_buffer[7]);
96.   Keyboard.send_now();
97. }
98.
99. void on_mouse() {
100.   Mouse.set_buttons(!!(rx_buffer[1] & 0x1), !!(rx_buffer[1] & 0x2), !!(rx_buffer[1]
    & 0x4));
101.   Mouse.move(rx_buffer[2], rx_buffer[3], rx_buffer[4]);
102. }
103.
104. void on_joystick() {
105.   char i;
106.   int pack;
107.
108.
109.   // Load the buttons
110.   pack = proto_int(1);
111.   for(i = 0; i < 32; i++) Joystick.button(i, !!(pack & (0x00000001 << i)));
112.
113.   // X, Y, Z
114.   pack = proto_int(5);
115.   Joystick.Z(pack & 0x3FF);
116.   pack = pack >> 10;
117.   Joystick.Y(pack & 0x3FF);
118.   pack = pack >> 10;
119.   Joystick.X(pack & 0x3ff);
120.
121.   // rZ, sL, sR
122.   pack = proto_int(9);
123.   Joystick.Zrotate(pack & 0x3FF);
124.   pack = pack >> 10;
```

```
  Joystick.sliderLeft(pack & 0x3FF);
  pack = pack >> 10;
  Joystick.sliderRight(pack & 0x3ff);

  // Hat Switch
  i = rx_buffer[13];
  if(i == 0xFF) Joystick.hat(-1);
  else Joystick.hat(45 * i);

  Joystick.send_now();
}

// Utility functions
char read_blocking(char* target, int max_attempts) {
  byte tmp;
  while(max_attempts--) {
    if(!Serial1.available()) continue;
    tmp = Serial1.read();
    if(tmp == 0xf8) continue;

    *target = tmp;
    return 1;
  }
  return 0;
}

char proto_opcode(char header) {
  return (header & 0xE0) >> 5;
}

char proto_args(char header) {
  return header & 0x1F;
}

short proto_short(char start) {
  // Returns a 2-byte short from the given starting position
  return *((short*)(rx_buffer + start));
}

int proto_int(char start) {
  // Returns a 4-byte integer from the given starting position
  return *((int*)(rx_buffer + start));
}
```

Appendix H: JavaScript Library Source Code

The following code represents aspects of the core JavaScript library for use from within Node.js. It is composed of three primary components, a low level Isotope interaction layer, a high level Keyboard wrapper and a high level Mouse wrapper.

The interaction layer makes use of the **serialport** module [27] for serial communications over the device's UART.

Isotope Interaction Layer

```
var SerialPort = require('serialport').SerialPort,
    EventEmitter = require('events').EventEmitter,
    util = require('util');

var Keyboard = require('./helpers/Keyboard'),
    Mouse = require('./helpers/Mouse');

module.exports = Isotope;

function Isotope(device) {
    this.open = false;
    this.buffer = [];
    this.writeInterval = null;

    if(typeof device == "string")
        this.uart = new SerialPort(device, {
            baudrate: 38400,
            parity: 'even'
        });
    else this.uart = device;

    this.keyboard = new Keyboard(this);
    this.mouse = new Mouse(this);

    this.uart.on('open', (function() {
        this.open = true;
        if(!this.writeInterval) {
            this.writeInterval = setInterval(this.flush.bind(this), 1);
            this.writeInterval.unref();
        }
        this.emit('open');
    }).bind(this));

    this.uart.on('data', (function(data) {
        console.log("%s", data.toString('hex'));
        this.emit('data', data);
    }).bind(this));

    this.uart.on('close', (function() {
        this.emit('close');
        if(this.writeInterval) {
            clearInterval(this.writeInterval);
            this.writeInterval = null;
        }
    }).bind(this));

    this.uart.on('error', (function(err) {
        this.emit('error', err);
    }).bind(this));
}

util.inherits(Isotope, EventEmitter);

```

```
Isotope.keyboard = require('./keycodes/keyboard');
Isotope.mouse = require('./keycodes/mouse');

Isotope.prototype.send = function(packet) {
    this.buffer.push(packet);
};

Isotope.prototype.flush = function() {
    while(this.buffer.length) {
        var packet = this.buffer.shift();
        this.uart.write(packet);
    }
};

Isotope.prototype.mouseRaw = function(buttons, deltaX, deltaY, deltaScroll) {
    var packet = zeros(5), length = 4;
    packet[0] = 0x40;
    packet[1] = buttons;
    packet[2] = deltaX;
    packet[3] = deltaY;
    packet[4] = deltaScroll;

    if(!deltaScroll) {
        length--;
        if(!deltaY) {
            length--;
            if(!deltaX) {
                length--;
                if(!buttons) length--;
            }
        }
    }

    packet[0] |= length;
    this.send(packet.slice(0, length + 1));
};

Isotope.prototype.keyboardRaw = function(modifiers, keys) {
    var packet = zeros(8), length = 0;
    packet[0] = 0x20;
    if(!modifiers && (!keys || keys.length == 0)) return this.send(packet.slice(0, 1
));

    if(!Array.isArray(keys)) throw new Error("Keys should be an array");
    if(keys.length > 6) throw new Error("A maximum of 6 keys can be pressed at any t
ime.");

    packet[1] = modifiers;
    for(var i = 0; i < keys.length; i++)
        packet[i + 2] = keys[i];

    packet[0] |= keys.length + 1;
    this.send(packet.slice(0, 2 + keys.length));
};

Isotope.prototype.close = function() {
    this.uart.close();
};

function zeros(n) {
    var o = [];
    for(var i = 0; i < n; i++)
        o.push(0);
    return o;
}
```

Keyboard Wrapper

```
var keyCodes = require('../keycodes/keyboard');

var charMap = {
    immutable: "\t ",
    normal: "abcdefghijklmnopqrstuvwxyz1234567890-=[]\\;'`,./",
    shifted: "ABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()_+{}|:\"~<>?"
}
var codeMap = {
    immutable: [43,44],
    mutable: [
        4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,
        30,31,32,33,34,35,36,37,38,39,
        45,46,47,48,49,51,52,53,54,55,56
    ]
};

module.exports = Keyboard;

function Keyboard(isotope) {
    this.isotope = isotope;

    this.updateTimeout = null;

    this.activeKeys = [];
    this.activeModifiers = 0;
    this.temporaryModifiers = 0;
}

Keyboard.prototype = {
    get then() {
        if(this.updateTimeout) {
            clearTimeout(this.updateTimeout);
            this.updateTimeout = null;
        }

        this.isotope.keyboardRaw(this.activeModifiers | this.temporaryModifiers, this.activeKeys);
        this.temporaryModifiers = 0;
        return this;
    },
    get ctrl() {
        this.temporaryModifiers |= keyCodes.modifiers.ctrl;
        return this.queueUpdate();
    },
    get alt() {
        this.temporaryModifiers |= keyCodes.modifiers.alt;
        return this.queueUpdate();
    },
    get shift() {
        this.temporaryModifiers |= keyCodes.modifiers.shift;
        return this.queueUpdate();
    },
    get releaseAll() {
        this.activeKeys = [];
        this.temporaryModifiers = 0;
        this.activeModifiers = 0;
        return this.queueUpdate();
    }
};

Keyboard.prototype.queueUpdate = function() {
    if(!this.updateTimeout)
        this.updateTimeout = setTimeout((function() {
            this.updateTimeout = null;
```

```
            this.now();
        }).bind(this), 0);
    return this;
};

Keyboard.prototype.press = function(keys) {
    if(!Array.isArray(keys))
        keys = Array.prototype.slice.call(arguments, 0);
    for(var i = 0; i < keys.length; i++)
        if(!~this.activeKeys.indexOf(keys[i]))
            this.activeKeys.push(keys[i]);
    if(this.activeKeys.length > 6)
        this.activeKeys = this.activeKeys.slice(this.activeKeys.length - 6);

    this.queueUpdate();
    return this;
};

Keyboard.prototype.release = function(keys) {
    if(!Array.isArray(keys))
        keys = Array.prototype.slice.call(arguments, 0);
    for(var i = 0; i < keys.length; i++)
        if(~this.activeKeys.indexOf(keys[i]))
            this.activeKeys.splice(this.activeKeys.indexOf(keys[i]), 1);

    this.queueUpdate();
    return this;
};

Keyboard.prototype.pressModifiers = function(modifiers) {
    if(!Array.isArray(modifiers))
        modifiers = Array.prototype.slice.call(arguments, 0);
    for(var i = 0; i < modifiers.length; i++)
        this.activeModifiers |= modifiers[i];

    this.queueUpdate();
    return this;
};

Keyboard.prototype.releaseModifiers = function(modifiers) {
    if(!Array.isArray(modifiers))
        modifiers = Array.prototype.slice.call(arguments, 0);
    for(var i = 0; i < modifiers.length; i++) {
        var compliment = 0xff ^ modifiers[i];
        this.activeModifiers &= compliment;
    }

    this.queueUpdate();
    return this;
};

Keyboard.prototype.write = function(text) {
    var index, lastIndex, c, m;
    for(var i = 0; i < text.length; i++) {
        lastIndex = index;

        // Handle the same character
        if(c == text[i]) {
            this.isotope.keyboardRaw();

            // And disable handling of the shift-changed keys
            lastIndex = -1;
        }

        c = text[i];

```

```
        if(~(index = charMap.immutable.indexOf(c))) this.isotope.keyboardRaw(0, [codeMap.immutable[index]]);
        else {
            m = 0;
            if(~(index = charMap.normal.indexOf(c))) m = 0;
            else if(~(index = charMap.shifted.indexOf(c))) m = keyCodes.modifiers.shift;
            else {
                console.warn("Unknown characer '%c'", c);
                continue;
            }

            // Handle the same key (with shift changed)
            if(index == lastIndex) this.isotope.keyboardRaw();

            // Send the new key
            this.isotope.keyboardRaw(m, [codeMap.mutable[index]]);
        }
    }

    this.isotope.keyboardRaw();
    return this;
};

Keyboard.prototype.now = function() {
    return this.then;
};
```

Mouse Wrapper

```
var mouse = require('../keycodes/mouse');

module.exports = Mouse;

function Mouse(isotope) {
    this.isotope = isotope;

    this.buttons = 0;
    this.deltaX = 0;
    this.deltaY = 0;
    this.deltaScroll = 0;

    this.updateTimeout = null;
}

Mouse.prototype = {
    get then() {
        if(this.updateTimeout) {
            clearTimeout(this.updateTimeout);
            this.updateTimeout = null;
        }

        this.isotope.mouseRaw(this.buttons | this.tempButtons, this.deltaX, this.deltaY, this.deltaScroll);
        this.tempButtons = 0;
        this.deltaX = 0;
        this.deltaY = 0;
        this.deltaScroll = 0;
        this.updateTimeout = null;

        return this;
    },
    get left() {
        this.tempButtons |= mouse.left;
        return this.queueUpdate();
    },
    get right() {
        this.tempButtons |= mouse.right;
        return this.queueUpdate();
    },
    get middle() {
        this.tempButtons |= mouse.middle;
        return this.queueUpdate();
    }
};

Mouse.prototype.queueUpdate = function() {
    if(!this.updateTimeout)
        this.updateTimeout = setTimeout((function() {
            this.updateTimeout = null;
            this.now();
        }).bind(this), 0);
    return this;
};

Mouse.prototype.now = function() {
    return this.then;
};

Mouse.prototype.press = function(buttons) {
    if(!Array.isArray(buttons))
        buttons = Array.prototype.slice.call(arguments, 0);
    for(var i = 0; i < buttons.length; i++)
        this.buttons |= buttons[i];
```

```
64.
65.     this.queueUpdate();
66.     return this;
67. };
68.
69. Mouse.prototype.release = function(buttons) {
70.     if(!Array.isArray(buttons))
71.         buttons = Array.prototype.slice.call(arguments, 0);
72.     for(var i = 0; i < buttons.length; i++) {
73.         var compliment = 0xff ^ buttons[i];
74.         this.buttons &= compliment;
75.     }
76.
77.     this.queueUpdate();
78.     return this;
79. };
80.
81. Mouse.prototype.scroll = function(delta) {
82.     this.deltaScroll += delta;
83.     this.queueUpdate();
84.     return this;
85. };
86.
87. Mouse.prototype.move = function(deltaX, deltaY) {
88.     this.deltaX += deltaX;
89.     this.deltaY += deltaY;
90.     this.queueUpdate();
91.     return this;
92. };
```

Appendix I: Command Line Tools Source Code

isokey

```c
#include "../libs/c/isotope.h"
#include "../libs/c/keylayouts.h"

#include "cmdline.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char parseKey(char* keys, int* keysCount);

int main(int argc, const char** argv) {
    int isotope;
    char modifiers = 0;
    char keys[6] = {0};
    int keysCount = 0;
    char release = 1;

    cmd_init(argc, argv);

    if(cmd_hasFlag('?', "help")) {
        printf("Isotope Keyboard Emulator\n");
        printf("Usage: %s -C -shift -A -win A SPACE BACKSPACE DEL F11 NUM7\n", cmd_application());
        return -1;
    }

    if(cmd_hasFlag('H', "hold")) release = 0;

    if(cmd_hasFlag('C', "ctrl")) modifiers |= MODIFIERKEY_CTRL;
    if(cmd_hasFlag(0, "lctrl")) modifiers |= MODIFIERKEY_LEFT_CTRL;
    if(cmd_hasFlag(0, "rctrl")) modifiers |= MODIFIERKEY_RIGHT_CTRL;

    if(cmd_hasFlag('A', "alt")) modifiers |= MODIFIERKEY_ALT;
    if(cmd_hasFlag(0, "lalt")) modifiers |= MODIFIERKEY_LEFT_ALT;
    if(cmd_hasFlag(0, "ralt")) modifiers |= MODIFIERKEY_RIGHT_ALT;

    if(cmd_hasFlag('S', "shift")) modifiers |= MODIFIERKEY_SHIFT;
    if(cmd_hasFlag(0, "lshift")) modifiers |= MODIFIERKEY_LEFT_SHIFT;
    if(cmd_hasFlag(0, "rshift")) modifiers |= MODIFIERKEY_RIGHT_SHIFT;

    if(cmd_hasFlag('W', "win")) modifiers |= MODIFIERKEY_GUI;
    if(cmd_hasFlag(0, "lwin")) modifiers |= MODIFIERKEY_LEFT_GUI;
    if(cmd_hasFlag(0, "rwin")) modifiers |= MODIFIERKEY_RIGHT_GUI;

    while(parseKey(keys, &keysCount));

    if(isotope = isotope_open("/dev/ttyAMA0")) {
        isotope_keyboard(isotope, modifiers, keys, keysCount);
        if(release) isotope_keyboard(isotope, 0, 0, 0);
        isotope_close(isotope);
        return 0;
    } else {
        printf("Error: Unable to connect to Isotope device, please ensure the UART is available.\n");
        return -2;
    }
}

typedef struct KEYBIND {
    char shortcut[15];
    int code;
```

```
} KEYBIND;

KEYBIND keymap[] = {
    { "0", KEY_0 },
    { "1", KEY_1 },
    { "2", KEY_2 },
    // and so on...
    { "NUMASTERIX", KEYPAD_ASTERIX },
    { "NUM+", KEYPAD_PLUS },
    { "NUMPLUS", KEYPAD_PLUS },
    { "NUM-", KEYPAD_MINUS },
    { "NUMMINUS", KEYPAD_MINUS },
    { "NUM.", KEYPAD_PERIOD },
    { "NUMPERIOD", KEYPAD_PERIOD }
};

char parseKey(char* keys, int* keysCount) {
    const char* key;
    char* keyUpper;
    int i;

    key = cmd_nextValue();
    if(!key) return 0;
    keyUpper = cmd_strupr(key);

    for(i = 0; i < sizeof(keymap)/sizeof(KEYBIND); i++) {
        if(!strcmp(keyUpper, keymap[i].shortcut)) {
            keys[(*keysCount)++] = keymap[i].code;
            return 1;
        }
    }

    printf("WARN: Failed to find a binding for key '%s'\n", keyUpper);
    return 1;
}
```

isomouse

```c
#include "../libs/c/isotope.h"

#include "cmdline.h"

#include <stdio.h>
#include <string.h>

int main(int argc, const char** argv) {
    int isotope;
    char buttons = 0;
    char x = 0, y = 0, scroll = 0;
    const char* value;
    int val;
    char release = 1;


    cmd_init(argc, argv);

    if(cmd_hasFlag('?', "help")) {
        printf("Isotope Keyboard Emulator\n");
        printf("Usage: %s -L -right -M X10 Y0 S-1\n", cmd_application());
        return -1;
    }

    if(cmd_hasFlag('L', "left")) buttons |= 0x1;
    if(cmd_hasFlag('R', "right")) buttons |= 0x2;
    if(cmd_hasFlag('M', "middle")) buttons |= 0x4;
    if(cmd_hasFlag('H', "hold")) release = 0;

    while(value = cmd_nextValue()) {
        sscanf(value + 1, "%d", &val);

        if(val > 127 || val < -127) {
            if(val > 127) val = 127;
            else val = -127;
            printf("WARN: %c value was outside maximum bounds (-
127 to 127), used %d instead.\n", value[0], val);
        }
        switch(value[0]) {
            case 'X':
            case 'x':
                x = (char)val;
                break;
            case 'Y':
            case 'y':
                y = (char)val;
                break;
            case 'S':
            case 's':
                scroll = (char)val;
                break;
        }
    }

    if(isotope = isotope_open("/dev/ttyAMA0")) {
        isotope_mouse(isotope, buttons, x, y, scroll);
        if(release) isotope_mouse(isotope, 0, 0, 0, 0);
        isotope_close(isotope);
    } else {
        printf("Error: Unable to connect to Isotope device, please ensure the UART i
s available.\n");
        return -2;
    }
}
```

isowrite

```
/**
 * Isotope Keyboard Emulation Command Line Applet
 * This command line executable allows emulation of
 * basic keyboard commands through a shell terminal
 * or the system() function and its equivalents in
 * other frameworks.
 */

/**
 * Examples
 * Type "Hello World!"
 * > isowrite Hello World!
 */

#include "../libs/c/isotope.h"
#include "../libs/c/keylayouts.h"

#include "cmdline.h"

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, const char** argv) {
    int isotope;
    char* text;
    char hasWritten = 0;

    cmd_init(argc, argv);

    if(cmd_hasFlag('?', "help")) {
        printf("Isotope Keyboard Writer\n");
        printf("Usage: %s Hello World!\n", cmd_application());
        return -1;
    }

    if(isotope = isotope_open("/dev/ttyAMA0")) {
        if(cmd_length())
            while(text = (char*)cmd_nextArgument()) {
                if(hasWritten) isotope_text(isotope, " ");
                isotope_text(isotope, text);
                hasWritten = 1;
            }
        else {
            text = (char*)malloc(sizeof(char) * 512);
            memset(text, 0, 512);
            while(fgets(text, 511, stdin))
                isotope_text(isotope, text);
        }
        isotope_close(isotope);
        return 0;
    } else {
        printf("Error: Unable to connect to Isotope device, please ensure the UART i
s available.\n");
        return -2;
    }
}
```