Isotope

USB HID Emulation for Embedded Devices

Benjamin Matthew Pannell

**STUDY LEADER** Professor Thomas Niesler

**DATE** August 2014

**Previous Work**

Pi Your Command

*Automatic Speech Recognition using the Raspberry Pi*

Christian Truter - Project E (448) Report

# Summaries

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.

# Table of Contents

[Summaries 2](#_Toc401168255)

[Table of Contents 3](#_Toc401168256)

[Figures 4](#_Toc401168257)

[Tables 4](#_Toc401168258)

[Symbols 5](#_Toc401168259)

[Introduction 6](#_Toc401168260)

[Univeral Serial Bus Background 8](#_Toc401168261)

[Pre-Design Investigation 9](#_Toc401168262)

[Demonstration Device Selection 12](#_Toc401168263)

[Device Design 13](#_Toc401168264)

[Software Design 18](#_Toc401168265)

[Communications Protocol 24](#_Toc401168266)

[Future Expansion 30](#_Toc401168267)

[References 31](#_Toc401168268)

[Appendix A: Project Planning Schedule 33](#_Toc401168269)

[Appendix B: Project Specifications 34](#_Toc401168270)

[Appendix C: Outcomes Compliance 35](#_Toc401168271)

[Appendix D: Circuit Diagram 36](#_Toc401168272)

[Appendix E: Performance Benchmarks 37](#_Toc401168273)

[Appendix F: Communications Protocol 38](#_Toc401168274)

[Appendix G: Library Documentation 44](#_Toc401168275)

[Appendix H: Source Code 45](#_Toc401168276)

[Appendix I: Component Information 46](#_Toc401168277)

# Figures

[Figure 1 USB Standard-A Plug. Image courtesy of Evan-Amos. 8](#_Toc401168234)

[Figure 2 USB Mini-B Plug. Image courtesy of Winford Engineering LLC. 8](#_Toc401168235)

[Figure 3 USB Micro-B Plug. Image courtesy of Winford Engineering LLC. 9](#_Toc401168236)

[Figure 4 Arduino Nano. Image courtesy of Arduino SA [5] 10](#_Toc401168237)

[Figure 5 PJRC Teensy 2.0. Image courtesy of PJRC [6] 11](#_Toc401168238)

[Figure 6 TXB0104 Bi-Directional Level Shifter. Image courtesy of Adafruit Industries [11] 13](#_Toc401168239)

[Figure 7 Adafruit Prototyping Pi Plate Kit for Raspberry Pi. Image courtesy of Adafruit Industries [12] 14](#_Toc401168240)

[Figure 8 Raspberry Pi Model B. Image courtesy of Adafruit Industries [15] 16](#_Toc401168241)

[Figure 9 System Architecture 17](#_Toc401168242)

[Figure 10 Slave Device Command Process 18](#_Toc401168243)

[Figure 11 USB Emulation Example – Ctrl+A 19](#_Toc401168244)

[Figure 12 Master Device Library Process 20](#_Toc401168245)

[Figure 13 C-Library API Definition 21](#_Toc401168246)

[Figure 14 C-Library Example 21](#_Toc401168247)

[Figure 15 Node.js Library API 22](#_Toc401168248)

[Figure 16 Node.js Library Example 23](#_Toc401168249)

[Figure 17 Example Header Parsing 26](#_Toc401168250)

[Figure 18 Example Keyboard Command Breakdown 27](#_Toc401168251)

[Figure 19 Example Mouse Command Breakdown 28](#_Toc401168252)

[Figure 20 Axis Packing Algorithm (C/C++) 29](#_Toc401168253)

[Figure 21 Circuit Diagram 36](#_Toc401168254)

# Tables

[Table 1 Packet Structure 26](#_Toc401168117)

[Table 2 Packet Command Types 26](#_Toc401168118)

[Table 3 Keyboard Command Packet Format 27](#_Toc401168119)

[Table 4 Example Keyboard Emulation Packets 27](#_Toc401168120)

[Table 5 Mouse Command Packet Format 28](#_Toc401168121)

[Table 6 Mouse Button Flags 28](#_Toc401168122)

[Table 7 Example Mouse Emulation Packets 28](#_Toc401168123)

[Table 8 Joystick Command Packet Format 29](#_Toc401168124)

[Table 9 Joystick Hat Switch Position Values 29](#_Toc401168125)

[Table 12 UART Protocol Packet Structure 39](#_Toc401168126)

[Table 13 UART Protocol OP Codes 39](#_Toc401168127)

[Table 10 Component List 46](#_Toc401168128)

[Table 11 Component Websites 46](#_Toc401168129)

# 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.

**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.

**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

ASIC *See* Application Specific Integrated Circuit

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. As a result they are often built using a combination of advanced neural networks and HMMs to help improve feature detection across very large datasets, such as those which Google acquired 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. In practice however, these limitations are minor in nature.

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 hopes to enable the development of a device which addresses both issues in an elegant and universally compatible manner, namely through the emulation of input devices. This will allow speech recognition to be performed either on a hardware processor attached to the user’s computer or on 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 move the device between 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 chip 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.

# Univeral 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 – which 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.

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 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.

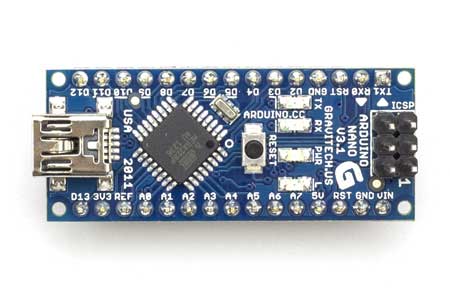


Figure 4 Arduino Nano. Image courtesy of Arduino SA [6]

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.

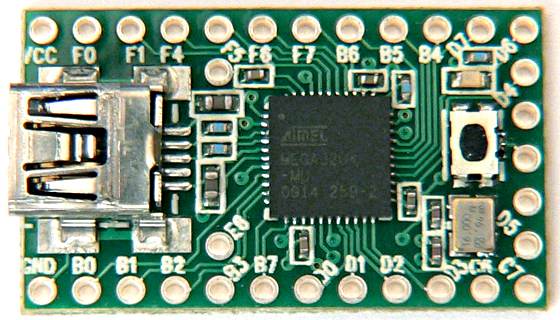


Figure 5 PJRC Teensy 2.0. Image courtesy of PJRC [7]

# 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. 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.

[16]

[17]

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.

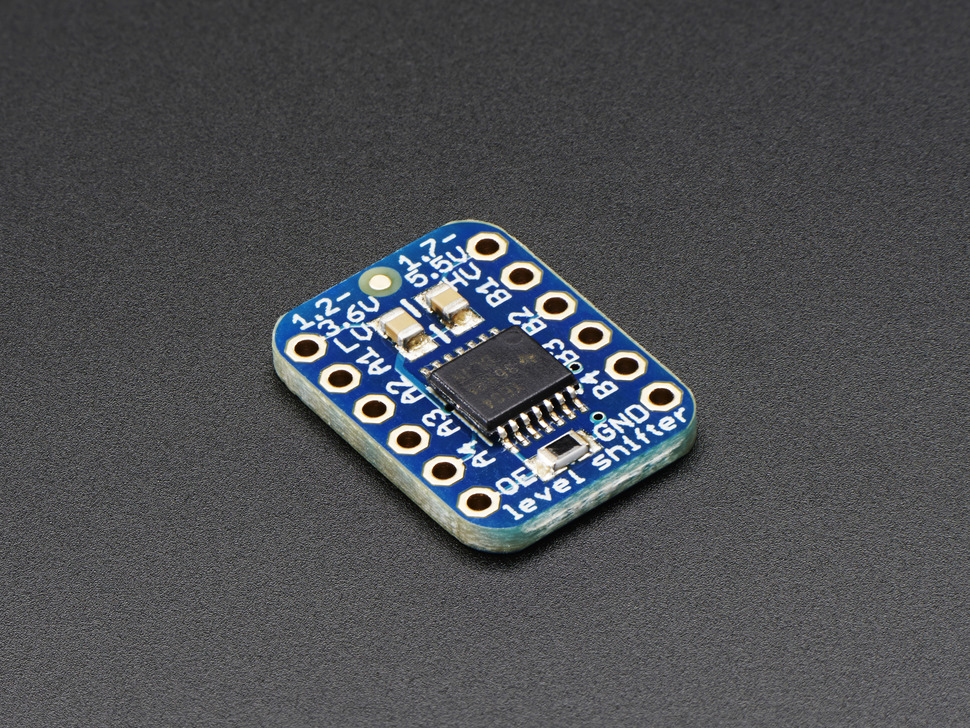


Figure 6 TXB0104 Bi-Directional Level Shifter. Image courtesy of Adafruit Industries [20]

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].

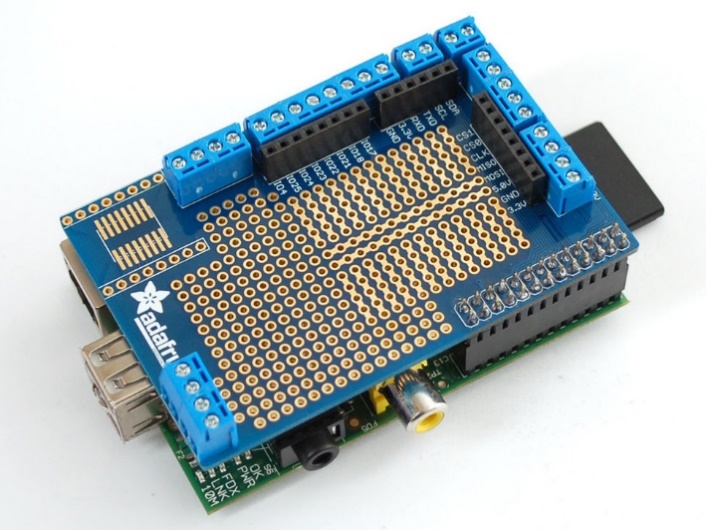


Figure 7 Adafruit Prototyping Pi Plate Kit for Raspberry Pi. Image courtesy of Adafruit Industries [21]

## 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 i­2c, 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 i2c 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, i2c 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.

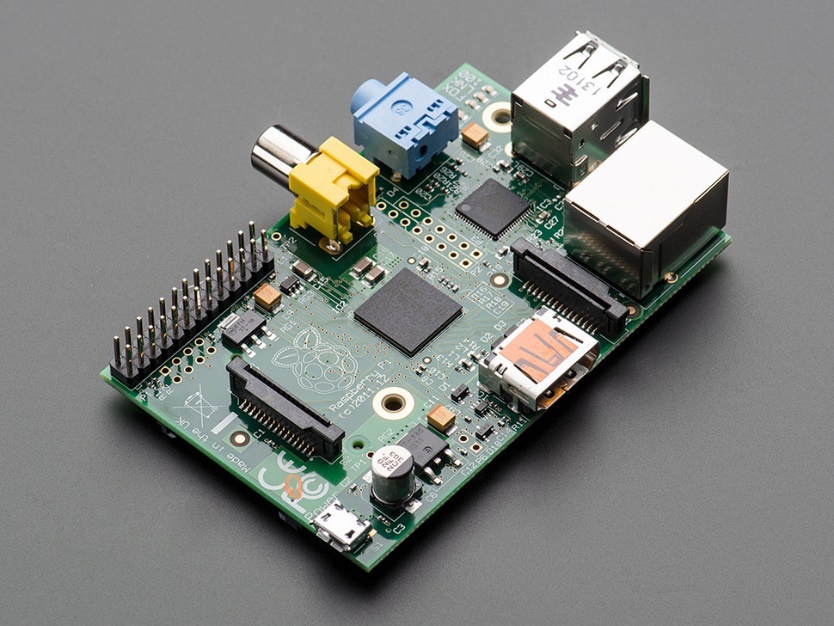


Figure 8 Raspberry Pi Model B. Image courtesy of Adafruit Industries [9]

## System Overview

The system architecture is outlined in Figure 9 demonstrates the manner in which the device is structured to allow the Application Logic to direct input to the host PC.

Figure 9 System Architecture

Within this architecture, the Teensy represents the emulation device which receives commands from the Raspberry Pi over a UART connection, and is responsible for converting these commands into USB HID reports which may be submitted to the host PC. In this manner, emulation implementation details are abstracted away from the Raspberry Pi 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.

# Software Design

## USB Emulation 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 10 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 Appendix F: Communications Protocol.

### 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 Appendix F: Communications Protocol.

Figure 11 USB Emulation Example – Ctrl+A

## Emulation 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 are available in a number of languages and, at a minimum, provide a means to submit mouse and keyboard emulation commands to the slave device for emulation.

Figure 12 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 client – vastly simplifying the task of maintenance and future extension, as well as allowing differentiation within client 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 available to developers wishing to make use of Isotope in their 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 Appendix G: Library Documentation.

### 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.

*// Opens a new Isotope device connection*

**int** isotope\_open(**const char\*** device);

*// Closes an Isotope device connection*

**char** isotope\_close(**int** isotope);

*// Formats and sends a mouse command to the specified Isotope*

**char** isotope\_mouse(**int** isotope, **char** buttons, **char** deltaX, **char** deltaY, **char** deltaScroll);

*// Formats and sends a keyboard command to the specified Isotope*

**char** isotope\_keyboard(**int** isotope, **char** modifiers, **const char** keys[], **char** keys\_count);

*// Formats and sends a joystick command to the specified Isotope*

**char** isotope\_joystick(**int** isotope, **int** buttons, **short** x, **short** y, **short** z, **short** rz, **short** sliderLeft, **short** sliderRight, **char** hat);

*// Sends the specified packet to the specified Isotope*

**char** isotope\_write(**int** isotope, **const** **char\*** packet, **char** length);

Figure 13 C-Library API Definition

#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 Example

### 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 15 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.

varIsotope = require(‘libisotope’);

var isotope = new Isotope(‘/dev/ttyAMA0’);

isotope.keyboard.write(“Hello World!”);

isotope.mouse.move(-10,0);

isotope.close();

Figure 16 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.

# 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 high performance 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

When designing the protocol, a number of decisions were made. 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 the how.

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, the developer’s code – 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 developer’s code.

## Packet Structure

Packets consist of a single 8-bit header field, as well as 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 17 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.

|  |  |
| --- | --- |
| 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) [22] 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 18 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 6, 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 15 breaks down the “Scroll Up Two Clicks” command from Table 6, explaining what each byte represents and how it is constructed.

Figure 19 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 16.

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 20 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.







# Future Expansion

This initial revision of the device represents the most 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-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 offloading the primary 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) [22] was required, allowing for more than the 6 keys available through USB HID to be pressed at a time.

# 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] | “Rollover (key),” Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Rollover\_%28key%29. [Accessed 15 October 2014]. |
| [23] | Adafruit Industries, “Teensy (ATmega32u4 USB dev board) 2.0,” PJRC, [Online]. Available: https://www.adafruit.com/products/199. [Accessed 18 August 2014]. |
| [24] | 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** | **Addressed In** |
| Problem Solving | * Introduction * Pre-Design Investigation * Appendix B: Project Specifications |
| Application of Scientific & Engineering Knowledge | * Device Design * Appendix E: Performance Benchmarks |
| Engineering Design | * Univeral Serial Bus Background * Pre-Design Investigation * Demonstration Device Selection * Device Design * Software Design * Communications Protocol * Appendix F: Communications Protocol |
| Investigations, Experiments and Data Analysis | * Pre-Design Investigation * Demonstration Device Selection * Appendix E: Performance Benchmarks |
| Engineering Methods, Skills and Tools, Including Information Technology | * Software Design * Communications Protocol |
| Professional and Technical Communication | * Software Design |
| Independent Learning Ability | * Univeral Serial Bus Background * Pre-Design Investigation * References |

# Appendix D: Circuit Diagram

The circuit diagram for the Isotope emulation device, as connected to the Raspberry Pi’s UART, is show below. Take note that at no point are the Raspberry Pi and Teensy 2.0’s power supplies coupled, ensuring that the Raspberry Pi cannot draw current through the Teensy 2.0 – as doing so has the possibility of damaging the Teensy 2.0’s circuitry.

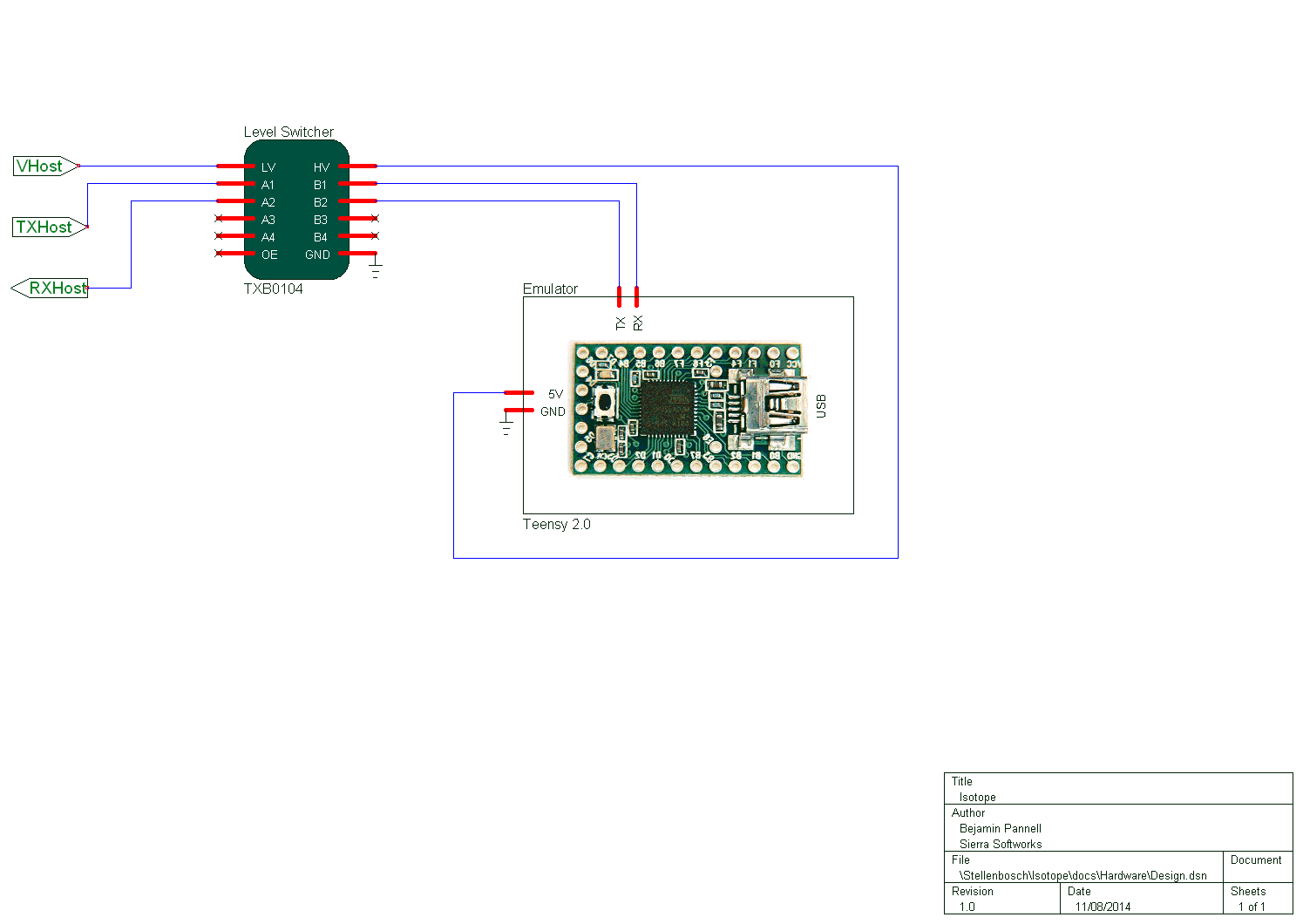


Figure 21 Circuit Diagram

# Appendix E: Performance Benchmarks

Need to do performance testing to determine the maximum data throughput, as well as the best baud rate to operate the UART at to help prevent data-loss due to buffer overflows.

Due to the Teensy’s firmware using a circular buffer to store messages, buffer overflows do not pose a security issue.

## USB Protocol Limitations

The USB HID protocol specifies that for each HID device connected to the host, provision must be made to accept up to 1000 reports per second (1000Hz polling rate). As this is a protocol specific limitation, we will use it as the upper limit on our performance benchmarks (exceeding this limit should not be permitted).

## UART Limitations

The UARTs used on the Raspberry Pi and Teensy 2.0 are capable of operating at baud rates of up to 115200 baud, as we are using 8-bit packets with even parity we are in fact sending 11-bits per packet. [24] From this it is possible to deduce that our maximum packet throughput is equal to .

## Isotope Protocol Limitations

Due to the flexible design of the Isotope protocol, in which packet lengths can vary from 1-byte to 32-bytes depending on their context, it is important to analyse performance given different usage scenarios.

The most common usage scenario is text input, in which one key is pressed at a time (with optional modifier keys) and then released. Optimizations to the way in which key presses are sent (only sending the key release command if the key remains the same, or at the end of the text) can help improve throughput, however for the sake of our calculations we will assume that each key is pressed, followed by a release command, and then the next is pressed.

In this scenario, the key press packets are each 3-bytes long while key release packets are each 1-byte long (see Example Packets). This gives an average packet length per command of 2-bytes – leading to an optimum baud rate (to ensure we do not exceed the 1000Hz USB protocol limitation) of 45000 baud.

As most other commands will generally result in equal, or longer, packets it is safe to assume that a maximum baud rate of 38400 (the nearest lower standard baud rate) will ensure the USB specification is never exceeded.

# Appendix F: Communications Protocol

Version 1.0

## Requirements

This protocol is required to include support for Mouse, Keyboard and Joystick emulation in a robust and high efficiency manner. In addition to this, the protocol should aim to be as understandable as possible and avoid complex behaviour which complicates implementations wherever possible.

For performance reasons this requires that the protocol be binary in nature, reducing (and in many cases removing) the need for packet parsing. In addition to this, attempts will be made to reduce the amount of data which will be transmitted over the UART connection to improve performance as much as possible.

## Design Decisions

There are two approaches to the protocol which we are able to take. The first is to attempt to design a protocol which is as faithful to the USB HID specification as possible - effectively causing the ATmega to act as a relay device, however while this will certainly minimize packet size to a large degree and faithfully allow emulation of any USB HID device it also has the distinct disadvantage of requiring the master implementation to handle the creation of all HID packets - a complex task which is prone to errors.

The simpler alternative is to rely on the included HID emulation libraries and instead declare a protocol which acts to perform RPC (remote procedure calls) on the ATmega chip. This, if well designed, could result in smaller packets for most common operations and would significantly simplify protocol implementations. The obvious disadvantage of this approach is that in order to emulate additional devices it would be necessary to extend the functionality of the ATmega's firmware as well as (possibly) adding additional op-codes to the protocol.

Version 1.0 of this protocol will adopt the second approach, attempting to implement a very specific RPC system built around USB HID emulation on the ATmega chip. Packets will consist of an op-code, packet length field and a number of 8-bit arguments to be passed to the corresponding functions. If needed, these 8-bit arguments can be combined to create 16-bit or 32-bit values where those are necessary.

## Protocol

### Packet

All protocol operations are wrapped in a packet structure similar to the following. Packets consist of a 3-bit **OP\_CODE** field, a 5-bit **ARG\_COUNT** field as well as a number of 8-bit arguments. There is a protocol imposed limit of 31 arguments, limiting the total packet size to 32-bytes.

Table 12 UART Protocol Packet Structure

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Index | **0x0** | **0x1** | **0x2** | **0x3** | **0x4** | **0x5** | **0x6** | **0x7** |
| **0x00** | OP\_CODE | | | ARG\_COUNT | | | | |
| **0x10** | ARG\_1 | | | | | | | |
| **0x20** | … | | | | | | | |
| **0x30** | ARG\_N | | | | | | | |

### OP Codes

There are a number of basic op codes which cover the spectrum of available functions which may be performed by the emulation layer. These codes define the way in which their received arguments are treated and allow future extensions to the protocol through the use of the **000** op code.

Table 13 UART Protocol OP Codes

|  |  |
| --- | --- |
| 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 | Reserved For Future Expansion |

### Expected Arguments

Each op code expects a certain set of arguments to be provided, and their presence dictates the behaviour undertaken by the emulation layer when the op code is received. In all cases, transmission of a packet with **ARG\_COUNT=0** will be used to release all active keys or buttons.

#### Keyboard

The keyboard operation is used to trigger the emulation of KeyDown operations. It is important to note that unlike platform native emulation libraries like SendKey() or Win32 API calls it is not necessary to send a KeyUp message when performing USB emulation, rather the KeyUp state will be detected when a packet is sent without the key listed as depressed. This is an important distinction and one which will in many ways dictate the way this protocol is designed.

In addition to this, the ATmega32u4's keyboard emulation is limited to 6 keys + 4 modifiers at any one time, and due to the way that HID emulation is performed it is impossible to "trick" the operating system into believing that more than that number are depressed at any one time.

As a result of these restrictions, the Keyboard (**0x20** flag) OP\_CODE requires the following argument structure.

* **ARG\_1** MODIFIERS : **uint8**
* **ARG\_2..7** KEY : **uint8**

It is important to note that the protocol and implementation allow the transmission of partial packets - meaning that it is not necessary to send additional arguments for keys which are not in use. Therefore to send the 'A' key it is simply necessary to send 2 arguments.

##### Example Packets

The following are examples of packets for performing some basic operations. Numbers represent hexadecimal bytes sent to the emulation device

* **Press A** 22 00 04
  + 0x20 – Keyboard Operation + 0x02 – 2 byte packet length
  + 0x00 – No modifier keys
  + 0x04 – Key code for A key
* **Press Shift+A** 22 02 04
  + 0x20 – Keyboard Operation + 0x02 – 2 byte packet length
  + 0x02 – Shift modifier key
  + 0x04 – Key code for A key
* **Press Ctrl+Shift+A+B+C** 24 03 04 05 06
  + 0x20 – Keyboard Operation + 0x04 – 4 byte packet length
  + 0x03 – Ctrl and Shift modifier key codes
  + 0x04 – Key code for A key
  + 0x05 – Key code for B key
  + 0x06 – Key code for C key
* **Release All Keys** 20
  + 0x20 – Keyboard Operation

##### Known Issues

Because of the way emulation is implemented in the Teensy firmware, it is impossible to send the full **uint16\_t** key codes to the emulation layer. This means that it is not possible to emulate certain special keys like **VOLUME\_UP**, **MUTE** etc. at this time. In the future, if this functionality becomes available it may be possible to tweak this implementation to support sending the full key codes in which case modifiers and keys will need to be handled differently.

#### Mouse

The mouse operation type is used to emulate mouse button presses, movement and scrolling. As with the keyboard operation type, transmitting a Mouse packet with no arguments has the effect of releasing all pressed buttons.

Button presses are encoded into the first argument using a set of flags, namely the following. The button flags are OR-ed together to give the resulting button code.

* **Left** 0x1
* **Right** 0x2
* **Middle** 0x4
* **ARG\_1** BUTTONS : **uint8** flags
* **ARG\_2** DELTA\_X : **uint8**
* **ARG\_3** DELTA\_Y : **uint8**
* **ARG\_4** DELTA\_SCROLL : **uint8**

It is important to note that it is possible to send "partial" packets, in which case the subsequent values will be assigned a default value of 0. This means that a mouse button press emulation doesn't need to send the DELTA\_X, DELTA\_Y or DELTA\_SCROLL components. Similarly, a Y movement doesn't need to send the DELTA\_SCROLL component.

##### Example Packets

* **LMB Down** 41 01
* **Right 8px** 42 00 08
* **Scroll Up 2 Lines** 44 00 00 00 02
* **Reset Buttons** 40

##### Known Limitations

Due to the USB HID specification not supporting Mouse Button 4 or 5 (used commonly to provide Forward/Backward navigation) it is not possible to emulate these. In addition to this, the HID specification provides no way to move the mouse to an absolute position on the display (given a set of X, Y coordinates). This behaviour can be emulated by moving the mouse to the bottom left corner (repeated -127, -127 movements) followed by movements to the desired location. The number of movements required to move the mouse to (0,0) will depend on the target display's resolution.

#### Joystick

The joystick emulation layer is slightly more complex than that of the mouse or keyboard - as it is necessary to pack relatively more information into the packet than would otherwise be necessary. The ATmega32u4 is capable of emulating a joystick with 32 buttons, 6 axes and a single 8-way hat switch. In order to provide full accuracy (10-bit axis reporting) it is necessary to "pack" sets of 3 axes together such that one 4-byte integer contains axis information for 3 axes.

Packing is achieved by applying the following algorithm. Take note that 2-bits are lost for each set of 3 packed axes, resulting in a packed efficiency of 93.75%, compared to an efficiency of 62.5% if packing is not used.

int32\_t pack(int16\_t axis1, int16\_t axis2, int16\_t axis3) {

return (((axis1 << 10) | axis2) << 10) | axis3;

}

In addition to this, the hat switch is handled differently to the standard Arduino implementation to allow its data to be contained within a single 8-bit argument. The special value 0xff is used to represent center, while all other values are multiplied by 45 to give the number of degrees from north.

The resulting packet is in the form

* **ARG\_1..4** BUTTONS : **int32\_t**
* **ARG\_5..8** pack(X, Y, Z) : **int32\_t**
* **ARG\_9..12** pack(rZ, sL, sR) : **int32\_t**
* **ARG\_13** HAT : **int8\_t**

It is important to note that as with all other op-codes it is possible to send empty packets, however due to the way in which axes are handled this is not recommended under any circumstances as strange values will be reported. In future, the upper bit of a pack may be used to indicate that it is a valid value and should be updated, however that is currently beyond the scope of this implementation.

# Appendix G: Library Documentation

This appendix includes excerpts from the documentation for the various libraries and wrappers available for Isotope. For the most up to date documentation it is recommended that you consult the libraries relevant project pages directly.

## C Library – libisotope

## Node.js Library – libisotope

# Appendix H: Source Code

All project source code, documentation and development has been undertaken on a private git repository available at <https://git.sierrasoftworks.com/stellenbosch/isotope>. For access to the repository, please contact Benjamin Pannell at [admin@sierrasoftworks.com](mailto:admin@sierrasoftworks.com) and reference this document.

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>

# Appendix I: 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 [23] | $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