Isotope

USB HID Emulation for Embedded Devices

Benjamin Matthew Pannell

**STUDY LEADER** Professor Thomas Niesler

**DATE** August 2014

# 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.

# Table of Contents

[Summaries 1](#_Toc395524737)

[Table of Contents 2](#_Toc395524738)

[Figures 3](#_Toc395524739)

[Tables 3](#_Toc395524740)

[Symbols 3](#_Toc395524741)

[Introduction 4](#_Toc395524742)

[Pre-Design Investigation 6](#_Toc395524743)

[Custom ASIC Design 6](#_Toc395524744)

[References 7](#_Toc395524745)

[Appendix A: Project Planning Schedule 8](#_Toc395524746)

[Appendix B: Project Specifications 9](#_Toc395524747)

[Non-Functional Specifications 9](#_Toc395524748)

[N-001 No Custom Drivers 9](#_Toc395524749)

[N-002 Keyboard Emulation 9](#_Toc395524750)

[N-003 Mouse Emulation 9](#_Toc395524751)

[N-004 Raspberry Pi Integration 9](#_Toc395524752)

[N-005 High Compatibility 9](#_Toc395524753)

[Functional Specifications 9](#_Toc395524754)

[F-001 High Performance 9](#_Toc395524755)

[F-002 Low Power Usage 9](#_Toc395524756)

[F-003 Low Cost 9](#_Toc395524757)

[F-004 Small Size 9](#_Toc395524758)

[Appendix C: Outcomes Compliance 10](#_Toc395524759)

[Appendix D: Circuit Diagram 11](#_Toc395524760)

[Appendix E: Performance Benchmarks 12](#_Toc395524761)

[Appendix F: Communications Protocol 13](#_Toc395524762)

[Requirements 13](#_Toc395524763)

[Design Decisions 13](#_Toc395524764)

[Protocol 14](#_Toc395524765)

[Packet 14](#_Toc395524766)

[OP Codes 14](#_Toc395524767)

[Expected Arguments 14](#_Toc395524768)

[Appendix G: C-Library API 18](#_Toc395524769)

[Appendix H: Source Code 19](#_Toc395524770)

# Figures

# Tables

[Table 1 UART Protocol Packet Structure 14](#_Toc395524733)

[Table 2 UART Protocol OP Codes 14](#_Toc395524734)

# Symbols

ASIC *See* Application Specific Integrated Circuit

# Introduction

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

Software based solutions are generally built on learning Markov models which adapt to the speaker and can often achieve astounding accuracy levels once trained, and when paired with a high quality microphone. In most cases these systems are designed to assist people who would otherwise be required to perform large amounts 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 very high accuracy rates with little or no speaker specific adaptations. As a result they are often built on advanced neural networks due to the problems of over-fitting commonly associated with Markov models when trained on massive 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, primarily 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 systems 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 a user’s 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 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 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 implementing 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.

# 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 at hand. Over this phase a number of possible solutions were investigated, their advantages and disadvantages weighed 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 chip 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 perfectly, however for prototyping purposes it would be difficult to modify, difficult to acquire and force developers to spend a large amount of time implementing support for its low level interface.

## Programmable USB Slave Device

There are a number of USB interface chips available on the market, from the venerable FT232R [2] by FTDI, a chip most hardware developers will have had experience with at one point or other in their careers, to vastly more complex programmable devices. One such example is the Vinculum-II [3] 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 this, or similar chips, 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 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, reprogramming its firmware to allow USB HID emulation to take place and repurposing one of its IO channels for inter-device communication. One particularly promising candidate was the ATmega32u4 [4] which included its own full speed USB controller which was fully programmable. Other advantages included the fact that it was the basis of the Arduino Nano [5] and as a result was commonly available in an easy to use package.

In addition to this, the ATmega32u4 is designed to act as a controller for a number of USB peripherals, this proven background improved confidence in the chip’s ability to fulfil the requirements at hand, while its small size and relatively low cost would ensure that future production runs wouldn’t be hampered by its use. 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 the possible options, it was decided that the ATmega32u4 would provide the best out of the box 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 decided that the Arduino Nano [5] provided adequate functionality however the PJRC Teensy 2.0 [6] provided a more rounded set of features at a lower cost, while remaining equally accessible and using the same tooling. As a result the decision was made to acquire the Teensy 2.0 as the prototyping board of choice.

# 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 [7] to communicate with the Teensy 2.0 [6].

One of the initial design considerations was based on the fact that the Raspberry Pi [7] operated at a core voltage of 3.3V while the Teensy 2.0 [6] – 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 [7] and Teensy 2.0 [6] directly to one another.

The result was that it would be necessary to include a level switcher in the design to allow the Raspberry Pi [7] to communicate safely with the Teensy 2.0 [6]. Initially the use of a voltage divider and basic BJT or FET booster were considered [8], however upon further inspection it became clear that at very high switching rates the voltage divider could cause issues due to the output pin capacitances.

Seeking an alternative solution, the Texas Instruments TXB0104 4-channel bi-directional level translator [9] 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, with automatic direction detection on each channel. This, combined with the exceptionally high throughput which could be achieved of up to 100Mbps [10] 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.

# 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] | FTDI, “FT232R,” Future Technology Devices International Ltd., [Online]. Available: http://www.ftdichip.com/Products/ICs/FT232R.htm. [Accessed 07 August 2014]. |
| [3] | FTDI, “Vinculum II,” Future Technology Devices International Ltd., [Online]. Available: http://www.ftdichip.com/Products/ICs/VNC2.htm. [Accessed 07 August 2014]. |
| [4] | Atmel Corporation, “ATmega32u4,” Atmel, [Online]. Available: http://www.atmel.com/devices/atmega32u4.aspx. [Accessed 11 August 2014]. |
| [5] | Arduino, “Arduino Nano,” Arduino, [Online]. Available: http://arduino.cc/en/Main/arduinoBoardNano. [Accessed 2014 August 11]. |
| [6] | PJRC, “Teensy 2.0,” PJRC, [Online]. Available: https://www.pjrc.com/store/teensy.html. [Accessed 2014 August 11]. |
| [7] | Raspberry Pi Foundation, “Raspberry Pi,” Raspberry Pi Foundation, [Online]. Available: http://www.raspberrypi.org/. [Accessed 11 August 2014]. |
| [8] | IanH, “RPi GPIO Interface Circuits,” eLinux.org, 30 August 2013. [Online]. Available: http://elinux.org/RPi\_GPIO\_Interface\_Circuits. [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
11. Voice Recognition Implementation

# 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. Total chip surface area may not exceed 1cm2.

# Appendix C: Outcomes Compliance

The following ECSA outcomes have been satisfied in the listed sections.

# Appendix D: Circuit Diagram

Need to add a circuit diagram showing how the Teensy/ATmega32u4 is connected through the TB0104 to the Raspberry Pi’s UART.

# 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.

# 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 requring 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 UART Protocol Packet Structure

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **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 UART Protocol OP Codes

|  |  |
| --- | --- |
| OP\_CODE | Description |
| 0x0 000 | Custom Operation |
| 0x1 001 | Keyboard |
| 0x2 010 | Mouse |
| 0x3 011 | Joystick |
| 0x4 100 | Reserved |
| 0x5 101 | Reserved |
| 0x6 110 | Reserved |
| 0x7 111 | Reserved |

### 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 hexadecimal representations of packets for performing some basic operations.

* **Press A** 22 00 04
* **Press Shift+A** 22 02 04
* **Press Ctrl+Shift+A+B+C** 24 03 04 05 06
* **Release All Keys** 20

##### 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: C-Library API

This is a quick breakdown of the libisotope C library and its API methods, for further information please consult the examples bundled with the libisotope package.

# 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>