haXbox

Dalton Banks, Noam D. Eisen
ESE 350 Spring 2012

Overview

Videogames have been demonstrated to be highly effective in administering physical therapy, but consumer game controllers are generally unsuited to therapeutic use. Dedicated therapeutic gaming systems exist, but are typically quite limited; tens of thousands of man-hours are typically required to develop top consumer videogames, and these resources are simply not invested in therapeutic systems due to the limited market. Existing therapeutic systems therefore tend to offer a lackluster gaming experience, and furthermore tend to be cumbersome and prohibitively expensive for home use. Our question was this: how might we harness the physical and psychological stimulation of high-quality consumer games for therapeutic use, both in the clinic and in the home?

For this, we have created what we call the haXbox. The haXbox allows any sort of input device (foot pedal, joystick, wheel, lever, sip-and-puff, etc.) to replace buttons and other inputs on any game controller (GameCube, Xbox, Play Station, etc.) By introducing controllers that incorporate desired user motions, the haXbox can be used to incorporate gameplay into any form of physical therapy, and can make a world of previously unplayable games accessible for home use. Bridging the gap between customized controllers and mass-market gaming systems, the haXbox couples the demands of therapy to the rewards of gameplay, harnessing consumer videogames to facilitate highly flexible game-based therapy, *with* the appeal and stimulation of mass-market games, and *without* the expense and limitations of standalone systems.

Goals

The object of this project was to implement all core functionalities that constitute the system described above. Our end goal was to have in hand a setup that would demonstrate the capabilities of the haXbox system, and which would allow us to immediately begin creating input devices for trial patients. To this end, we set out as mandatory objectives the following items:

- GameCube controller interface: serial input to controller output
- haXbox v 1.0: multiple inputs mapped to single serial controller interface
- Inputs: a few representative input devices, including non-standard inputs

Additionally, we outlined the following desirable goals, to be accomplished if time and resources should permit:

- Sleek enclosures incorporating necessary ports and connectors
- Laptop-based GUI for configuring and mapping haXbox channels
- Bit-banging GameCube controller interface (i.e. spoofing the protocol)
- Additional controller interface (e.g., Xbox, PS3)
- Any other promising features and ideas that reveal themselves along the way

Philosophy

From a design perspective, the haXbox system is based around three central objectives:

User-friendliness

Because the haXbox is intended for use in both clinical and home settings, it is absolutely essential that even the most minimally tech-savvy user feel capable and comfortable setting up and using the haXbox. Therefore the system must provide plenty of intuitive feedback, and must convey the experience of plug-and-play simplicity.

Configurability

In order to maximize the usefulness of the haXbox in connecting and mapping any variety of input devices to any sort of output system, the system must permit a broad range of setup configurations, and must therefore be highly adaptable in its connections, both physical and virtual.

Modularity

In order to maximize its adaptability, we aim to make the haXbox system accessible at multiple levels of abstraction. While it is absolutely essential that both the therapist and the home user be able to configure the haXbox at the basic level of input and output channel mapping, other levels of accessibility would greatly enhance the system's reach and flexibility. The hobbyist might develop features and functionalities at the slightly more technical level of channel thresholds, dead-bands, and control curves, etc.; the seasoned hacker could create output systems and sophisticated input devices; and the professional should be able to add new and novel functionalities with ease. This calls for modularity at every level.

Design Specification

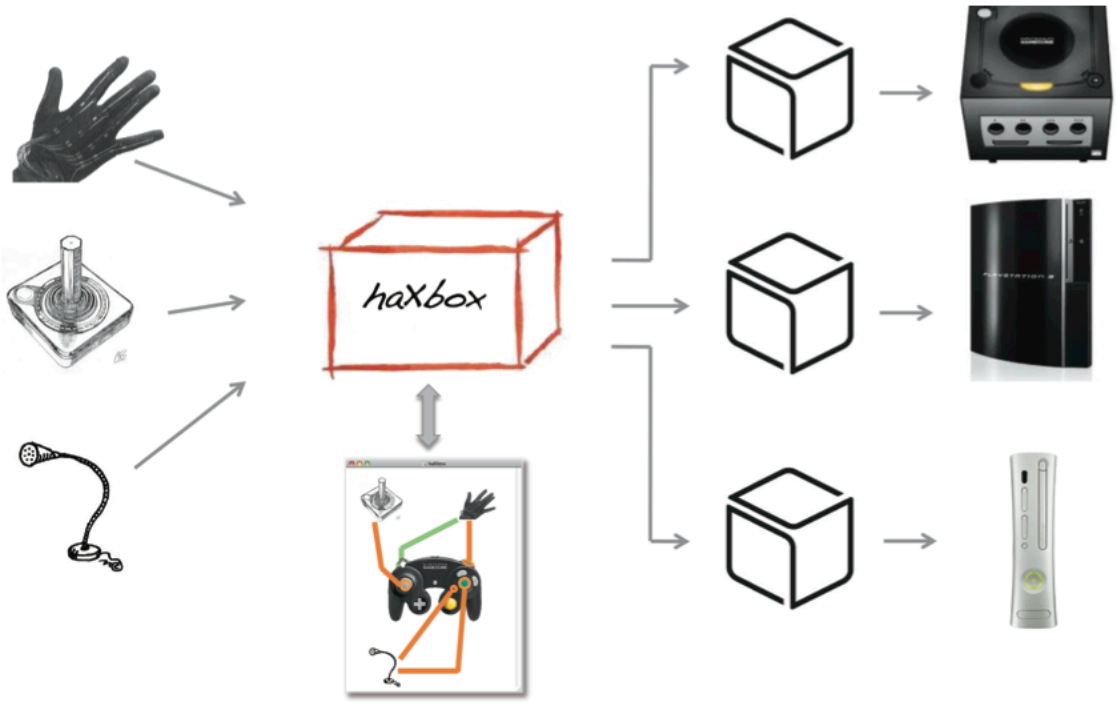
Prior to working on implementation details, we developed a fairly exhaustive specification of the system's ideal functionality. For the end-user (typically a therapist, parent, or capable patient), the following features are desired:

- Any input (foot pedal, neck sensor, lip suction, etc.) on any compatible input device can be configured as a replacement for any input (button, joystick, etc.) on any standard game controller
- Any number of input devices can be connected and configured simultaneously
- Inputs are transmitted to the console in real time with a refresh rate of 20-50 Hz
- A physical device (the haXbox) powered by a wall socket serves as a central router, accepting multiple input devices, an output game console, and USB access to its microcontroller, and provides user feedback based on its operation status (transmitting, configuring, error, etc.)
- The haXbox provides a power source for connected input devices
- Input mappings can be performed via a PC-based GUI connected by USB, with the following features:
    - The haXbox is automatically detected when online, and the option to connect is provided
    - When connected, devices plugged into the haXbox are automatically detected and recognized
    - A mapping interface is automatically populated with connected input devices, but mapped devices can be deleted and offline devices can be added from a list
    - Mappings can be saved and loaded
    - The mapping interface is simple and intuitive (drag-and-drop)
    - Input response thresholds can be configured (advanced user feature)
- The haXbox defaults to data transmission mode; it stores its current mapping configuration on shutdown, and reads the configuration on boot-up
- GUI configuration mode is entered once the haXbox is "connected" from the PC end, and returns to transmission mode after a "transmit" command is sent or the PC is disconnected

Input devices themselves should be "easy" to create; for a device designer, an API should be provided that allows for development with minimal knowledge of the haXbox implementation details.

Architecture

Broadly, the system as specified compels an architecture as depicted in the following diagram:



Input devices connect to the haXbox, which can output their data in real-time to any of multiple game systems via a device that translates input data to valid game commands for the given console. The input devices are mapped to game controller inputs via a PC-based GUI (bottom).

As our system relies on device-to-device communication, the primary architectural decision was to determine what communication protocol to use between devices. The primary contenders were I^2C and SPI, as they are ideal for high-speed, short-range wired communication, and most microcontrollers support them. Since we did not want the number of devices connected to be limited by the physical ports available on the haXbox, I^2C at first seemed more attractive, since it supports far more addressable devices on a single two-wire bus than we would ever need to accommodate for one user, which would simplify both the wiring and code. However, we also wanted to be able to dynamically detect and identify connected devices, which I^2C would make quite difficult due to its lack of dynamic addressing. The best option we could envision using I^2C and preserving these goals would have been to specify an I^2C address that devices always used upon connection, and have the haXbox continually poll that address for a connected device and immediately reassign it to a free address The drawback is that devices would have to be unplugged on startup and connected sequentially, which goes against our philosophy of user-friendliness. (Note: to support device recognition

devices must have standard unique IDs, so an ID-based I^2C identification method would have been logical, but this would limit the number of possible unique devices to the range of the I^2C address space, and would also prohibit two devices of the same model from being connected at the same time.) Using SPI, device differentiation is straightforward as each device has a hard-wired selection line; the addition of a few more wires is a small price to pay for the simplification of the dynamic addressing problem. Furthermore, extending the number of usable devices beyond the physical ports provided can be accomplished in software, allowing connected devices to either identify themselves as input devices or expansion ports, whose ports would be treated as normal haXbox ports.

Finally, we had to choose which microcontrollers to use for the various components of our platform. Because of its powerful ARM-based LPC1768 processor, its SPI and USB capabilities and the ease-of-use of its high-level C++ API, we decided to use the mbed as the microcontroller for the haXbox itself. To use the mbed for input and output devices, however, would not have been cost-effective, so for these we used the MAEVARM M2 (ATmega32U4 processor), which also features SPI and USB modules.

Six-Week Project

For the purposes of the class project, we had to select and fabricate specific peripheral devices to demonstrate the capabilities of the haXbox. For the game console, we chose the Nintendo GameCube, as the gameplay of several of its popular games (e.g., Super Smash Bros. Melee, Metroid Prime, Mario Kart DoubleDash) are straightforward to understand and would be engaging both for therapy and for demonstration. Ideally, we would hack the communication protocol used by the game controller and spoof it directly, but for simplicity's sake we decided to hold off on bit-banging and instead interface directly with the controller's electronics.

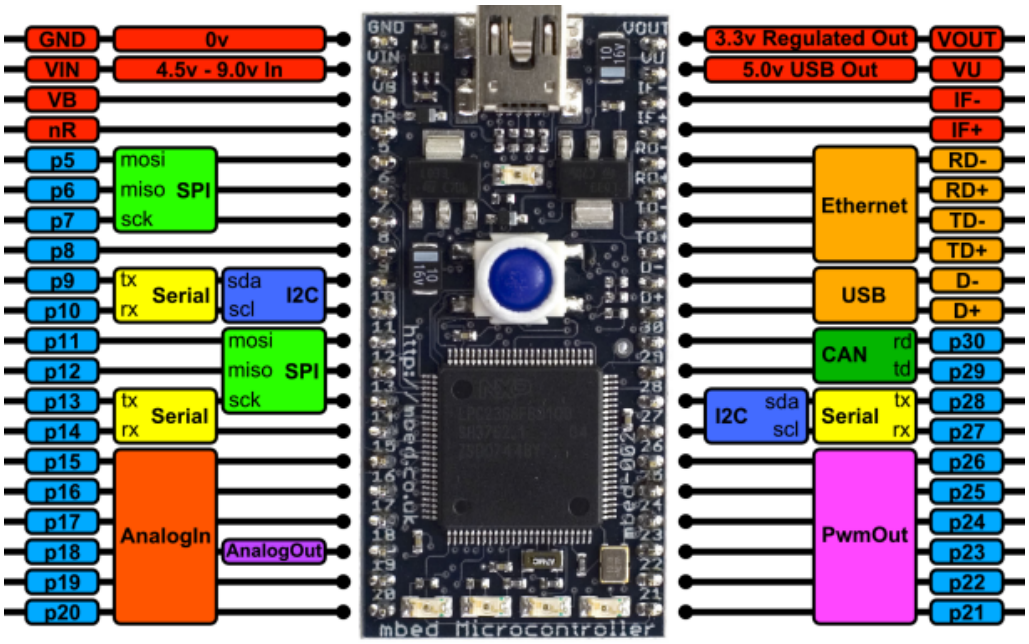
We carefully chose an array of input devices that would reflect the capabilities and versatility of the haXbox system. For development and testing, we used keyboard input over USB and a mini-joystick output to an analog-to-digital converter. We included a sip-n-puff input as a quintessential alternative input device already in use by paraplegics, a custom-designed wrist gimbal (per a physical therapists recommendation) as an example of the feasibility and limitless possibilities of customized device design, and a pair of pre-existing foot pedals to demonstrate the usability and the configurability of adapting devices not designed for the haXbox.

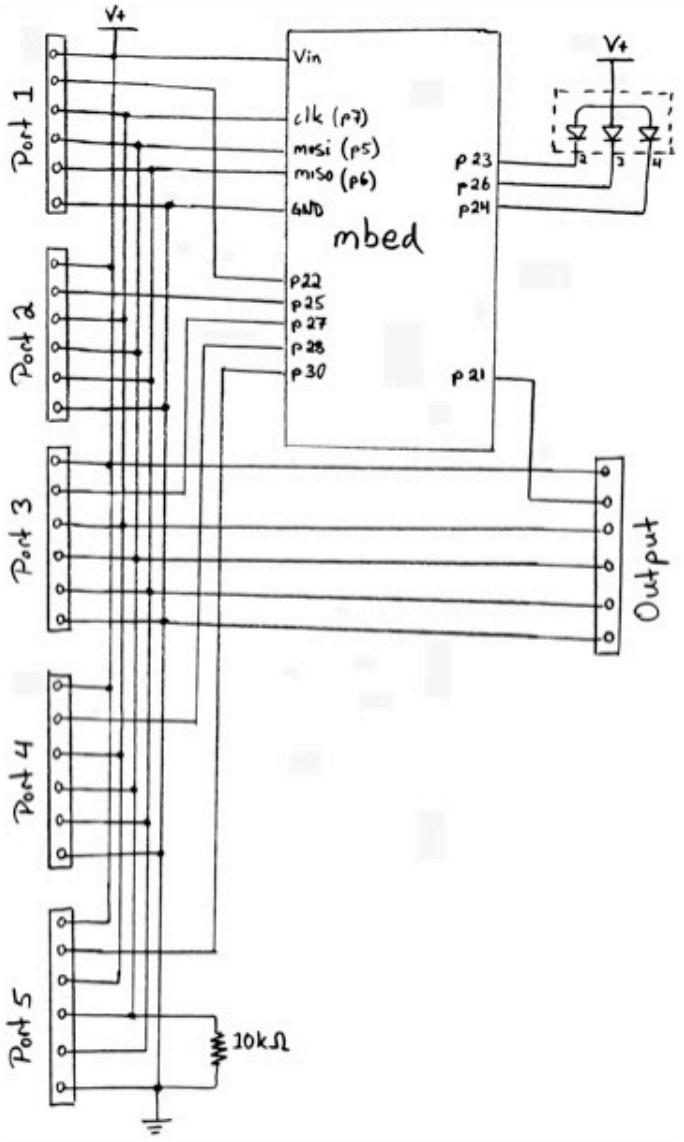
haXbox

The haXbox itself comprises the bulk of the engineering involved in the haXbox project, serving as a central hub for the system both in software (data transmission and routing, configuration) and in hardware (device connection, power). The hardware, software and electronics are tightly integrated to form a polished, versatile final product.

Electronics

The haXbox is built around an mbed microcontroller platform, which drives the functionality of the haXbox. The microcontroller is configured as an SPI master, and communicates with all input devices and with the output system through a common SPI bus, with a separate 'slave-select' line for each device. In order to guarantee a predictable signal from a port with no device connected, a pull-down resistor is added on the 'master-out, slave-in' line. Additionally, the mbed's USB programming and communication port is broken out to be accessible in-circuit. A single tri-color LED is used to indicate system status to the user. Following is a the pin layout of the embed, followed by a schematic of the haXbox core:





The schematic above is a generic version of the haXbox circuitry. The system does not depend upon specific components aside from the mbed, and, furthermore, component choices are driven mostly by mechanical, rather than electronic considerations. Ports and other interface components were chosen that offer wire connections, rather than DIP or SMT, and which lend to the crisp, rugged feel sought in the haXbox. The following components were used in the current build of the haXbox:

- 1 x Perfboard: 2" x 3" (custom GM Lab board)
- 6 x Mini DIN Connector, 6-position, Digikey CP-2960-ND (Ports 1-5 + Output)
- 1 x DC Power Jack, 2.5x5.5mm, Digikey CP-065B-ND
- 1 x Tri-Color LED, common anode (mLab stock)
- 1 x SPST Slide Switch (Detkin Lab Stock) (power switch, in series with jack)
- 1 x USB Jack, type-B, Digikey MUSBD1130-ND
- 1 x mbed microcontroller
- 1 x 10kΩ Resistor, 1/8W carbon

Software

There are four major software components in the haXbox system: the main haXbox code, the input device SPI communication API, the output console SPI communication API, and the PC-based GUI for mapping and configuring input devices. The haXbox software was developed in conjunction with the other three components and works with them tightly; for clarity, each component will be described in this section.

As an overview, most of the haXbox functionality is written in an object-oriented manner (taking advantage of the mbed's C++ compiler), encoding input and output data and communication in a haXbox object containing several IOPort objects. In the main() function, an instance of the haXbox class is initialized (using pin parameters specific to the electronics for the given haXbox design), and the haXbox enters configuration mode, where it listens for and responds to configuration commands sent over USB. After ending configuration, it enters transmission mode, repeatedly collecting input data and sending it to the output device. Each input device has a predefined number of "channels", or discrete inputs, as does each output device (discrete buttons, joystick inputs, etc.). During data transmission, the one byte of data from each input channel is collected and sent to an output channel based on the user-defined mapping.

In the initial configuration, a PC connected via USB can query currently connected devices to aid the user in specifying a mapping configuration, can set haXbox ports to listen for a specified device having a specified number of channels, can map individual input channels to individual output channels, and can finally send the haXbox into transmission mode. Devices are identified using a unique (to the model) two-byte device ID. To avoid string comparison, each command from the PC takes the form of a single character, which is #defined as a unique byte to be acted upon as it is received. Responses to the PC are sent as newline-delimited strings. During setup, a blue LED is blinked continually (accomplished by repeatedly attaching a "blink" function to an mbed::Timeout of 0.5 s) indicating that the haXbox is in configuration mode.

One challenge for collecting input data is that the input devices cannot notify the haXbox when ready to send data, as SPI communication is entirely master-driven. The communication scheme we settled on is as follows: for each transmission, the master continually polls each connected input device until it responds with a (one-byte) "data-ready" message, then successively collects as many bytes as the input device has channels before moving on to the next device. The API for the input devices therefore expects the input device to alternate between data collection and SPI transmission modes; as soon the data is ready (an ADC conversion completes, etc.) the haxbox_listen() function is called, which waits for a data request from the haXbox and then transmits its data before returning to data collection mode.

We decided that mappings would be done by port rather than by device ID because of the reasonable scenario of two connected input devices of the same model (e.g., two foot pedals) but different user-defined mappings. Therefore, the haXbox checks initially to verify that the correct device is connected to each port, turning the LED to red if any device is in the wrong location or disconnected (signifying that the user should plug in the PC and check the mapping using the GUI). A 'disconnected' state is determined if during data transmission the device does not respond within a specified brief timeout period, or an ID query returns 0 (indicating that device is not connected and the value of the pulled-down MISO line was read).

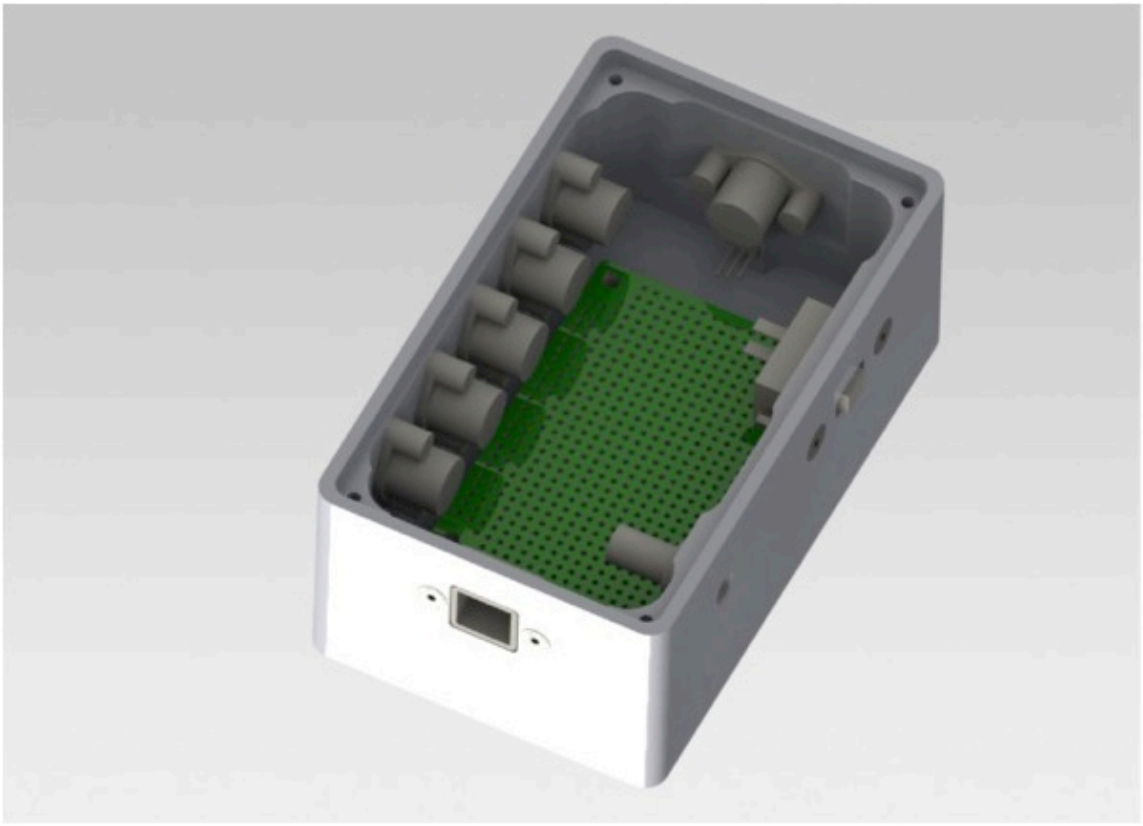
We decided that data transmission should continue to function seamlessly for devices that are correctly connected even if others are disconnected, so after a data request times out, the input device requested is marked disconnected and no data from it is sent or requested, while all other connected inputs function as normal. Disconnected devices are polled on each transmission cycle to see if they have come back online, and upon reconnection their data is transmitted as normal.

Output data transmission is more straightforward, as the haXbox simply sends a "begin-transmission" message, then successively sends as many bytes of data as there are output channels. Correspondingly, the output device listens for the "begin-transmission" message, reads as many bytes as it has channels, configures the corresponding analog and digital channels accordingly, and returns to listening for a new "begin-transmission" message.

Hardware

The hardware was designed to suit the context in which the haXbox will operate. The system will be used by patients and practitioners who generally have little technical background and who may be only minimally computer literate. As such, it is vital to present the user with a well-structured hierarchy of information that simultaneously provides the cues they need to use the system and hides details irrelevant to their interaction with the device. It is also essential that the device convey the image and feeling of crispness and simplicity to avert any intimidation that might otherwise be sparked by such a non-trivial electronic system.

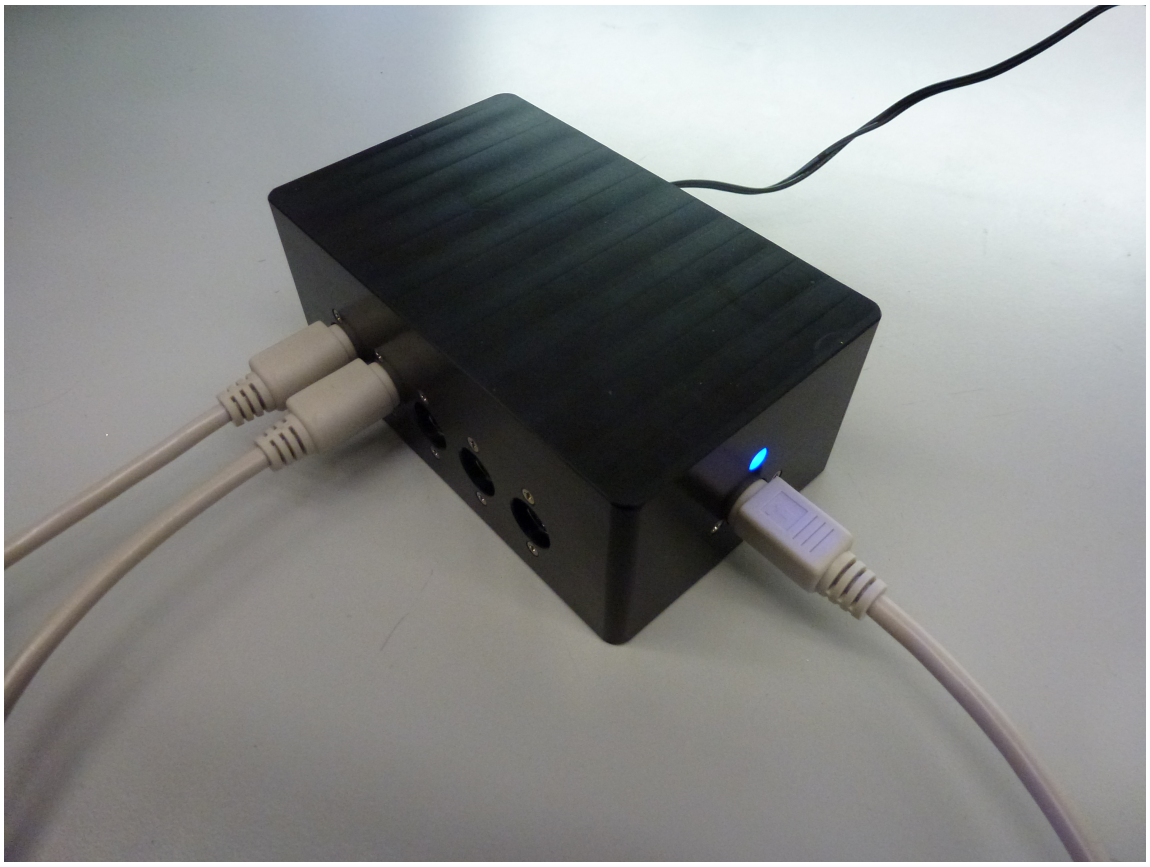
In light of these considerations, the haXbox was designed to be a self-contained object, an entity of its own. Its form is simple on the outside, exposing only the ports and jacks necessary for use, along with a single power switch, while inside it hides away the many details of circuitry, wiring, and assembly. Ports and jacks were chosen for their crisp and substantial feel, further contributing to the overall impression of sturdiness and simplicity. Light from a single LED is exposed to the user to provide straightforward feedback cues. Care was also taken in the design process to assure that the final plans would be simple to manufacture, thereby keeping the price of the system to a minimum.



Above is a rendering of the case as viewed from rear-left underneath with many of the components mounted inside of it. The haXbox case contains and organizes all electronic components, and presents the connectors and power switch to the user in an accessible and useful manner. It consists of three parts. The main shell is machined from Delrin (acetal resin) using a combination of manual and CNC methods. A translucent LED insert is turned from polycarbonate plastic, and press-fit into the shell to filter and diffuse light from the LED. The bottom cover is machined from 1/8" aluminum plate, and is fastened to the shell with four cap-head screws. Each type of component contained in the case is mounted with appropriate fasteners. A complete list of fasteners in the haXbox and their functions is as follows:

- 4 Cap-head screw 6-32 x 3/8" (secure base plate)
- 4 Standoff Female-Male 6-32 x 1/4" (mount protoboard)
- 4 Cap-head screw 6-32 x 1/4" "
- 12 Flat-head screw M-2 x 16mm (mount SPI ports)
- 12 Nut M-2 "
- 2 Flat-head screw 4-40 x 3/8" (mount USB connector)
- 2 Flat-head screw 6-32 x 1/4" (mount power switch)
- 2 Nut 6-32 "

Below is a photograph of the finished and assembled product as viewed from a vantage point above and to the front-right.



Configuration Software

The underlying functionality of the PC-based GUI is a Python "HaXbox" class managing connection to, and communication with the haXbox. When initialized, the object attempts to establish a connection to a given COM port, and if successful, sends a device query message. If this message is acknowledged, the device is connected and class methods can be used to take advantage of all of the PC-haXbox functionality (querying connected devices, initializing ports, mapping channels, ending configuration mode).

As previously discussed, the commands it can send to the device are encoded as bytes— it maintains the same list of command-to-byte mappings as the main haXbox code. Arguments are sent as a predefined number of bytes following the command, and if a response is expected from the haXbox, bytes are read until a newline character is reached.

The GUI is written using wxPython, and is still in development. At present, it displays a main window containing two sub-displays, one for input device mapping and one for device connection and status display. On startup, the bottom display indicates the haXbox is "Disconnected", populates a list displaying all currently detected USB devices, and when a device is selected the user has the option to attempt connection to the devices. Upon connection, an instance of the Python HaXbox class attempts to use the provided serial connection to initialize communication with the haXbox, and if successful the status is changed to "Connected" and the serial device list is exchanged for a list displaying the IDs of all devices currently plugged in to the connected haXbox. At the same time, a mock-mapping selection list associated with each detected devices is displayed in the upper input device mapping display.

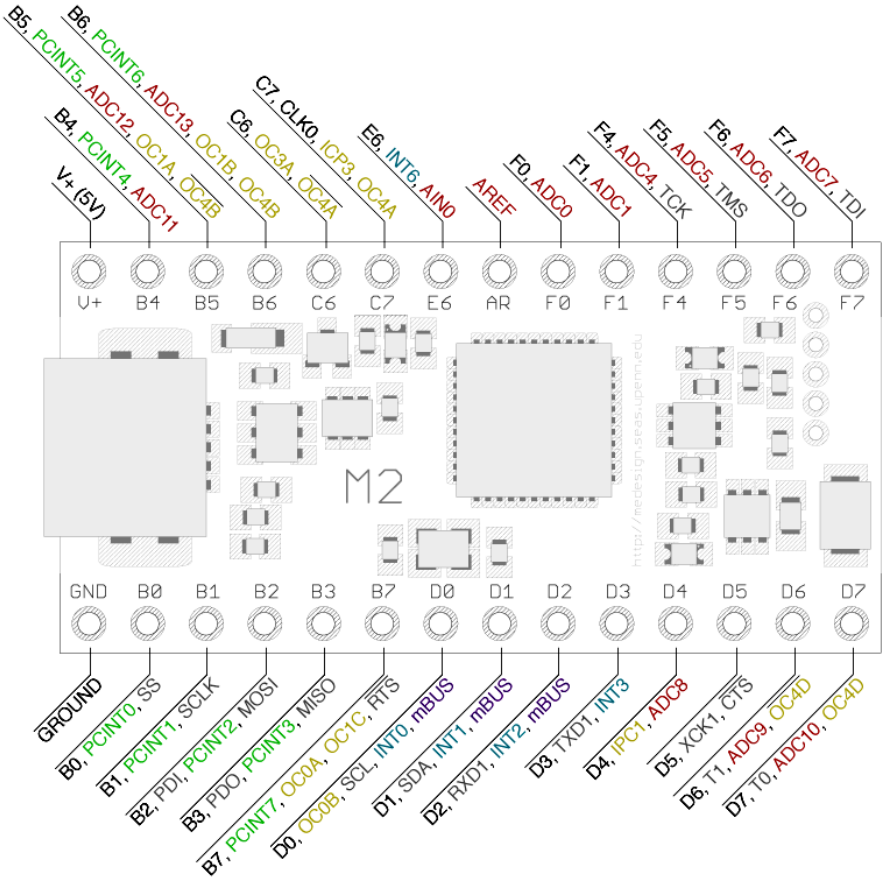
Additionally, a specification for device profiles was developed; these are files representing input and output devices that associate a device description and a description of each channel with a two-byte device ID. Using these profiles, the GUI can display the connected device names and automatically populate the mapping display with device channels and descriptions. The functions to do this have been successfully tested but not yet integrated into the GUI. Once device lookup is integrated, the mapping display will be able to be linked to the mapping functions in the HaXbox class, allowing simple point-and-click configuration of connected devices with no knowledge of haXbox internals, device internals, or the HaXbox class. Furthermore, a feature will be added allowing a user to also incorporate offline devices in mappings, using a device browser populated with devices in the device profile folder. Users will have the option of saving input mappings, or loading past mappings from disk using a standard mapping format. These capabilities will allow users to create custom device mappings whether the haXbox is online or offline, and will be able to load frequently used configurations instantly rather than remapping each time.

Real-time device detection is implemented using a thread that monitors haXbox connection and communication, and uses a semaphore-based locking system to integrate safely with the main wxApp thread. When connected to the haXbox the thread constantly polls the haXbox for device connection data, updating the main window.

Finally, a mock haXbox interface was developed for testing the GUI and the haXbox class. The HaXboxMock class can simulate disconnection and reconnection of devices with specified IDs, and uses a pseudo-teletype serial port to communicate with an instance of HaXbox connected to the corresponding teletype serial port.

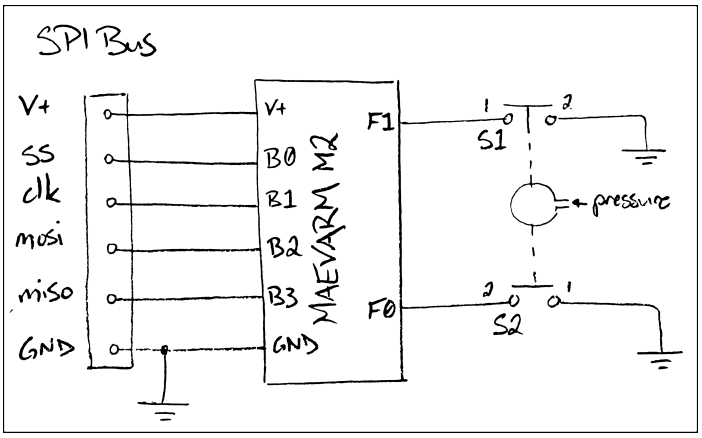
Input Devices

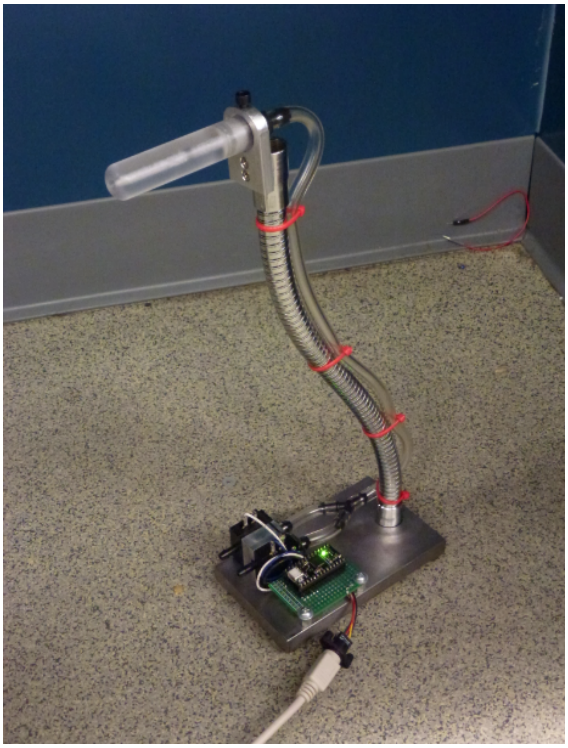
Three representative input devices were created to demonstrate the capabilities of the haXbox system as well as the wide range of possibilities for input devices. All input devices use the MAEVARM M2 microcontroller platform, a pinout description of which follows:



Sip-n-Puff

The sip-n-puff input device takes input from the user in the form of gentle suction and pressure applied to a straw using the mouth. This device targets quadriplegic patients, who lack most other means of physical interaction. Two differential-pressure-sensitive switches are used to capture user input. An M2 microcontroller is used to collect input signals and send them via SPI to the haXbox. A schematic and photograph follow:

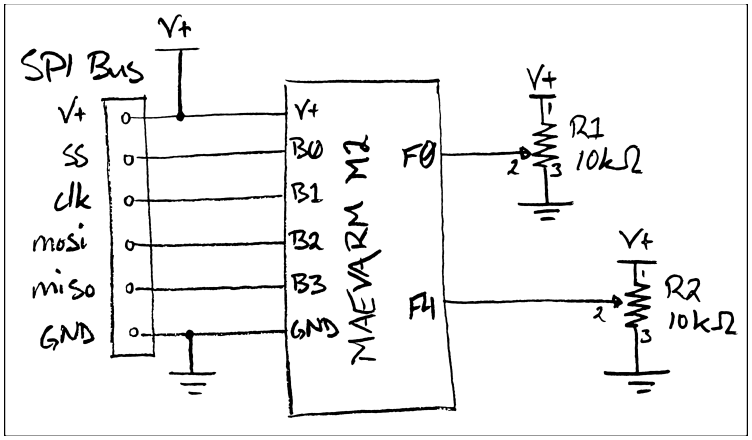




The sip-n-puff created for this project is custom-designed and machined. The mouthpiece is turned from polycarbonate plastic, and is mounted on a flexible gooseneck for ease of configuration. It is plumbed to the switches with 3/8"-ID flexible rubber tubing and barb-type connectors, and the entire system is mounted to a heavy steel base with rubber feet to allow for easy desktop use.

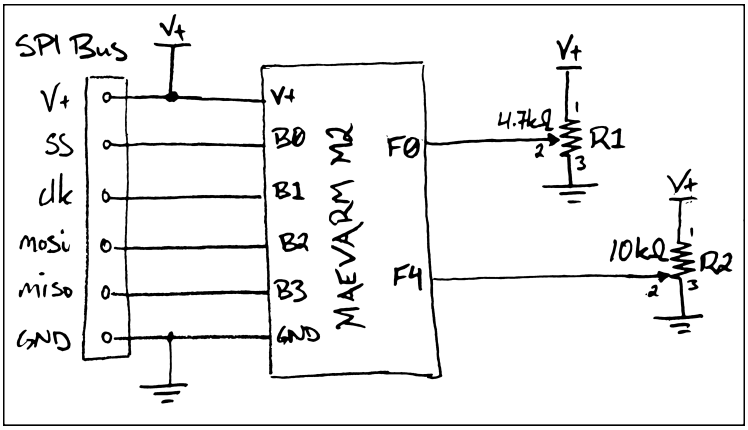
Foot Pedals

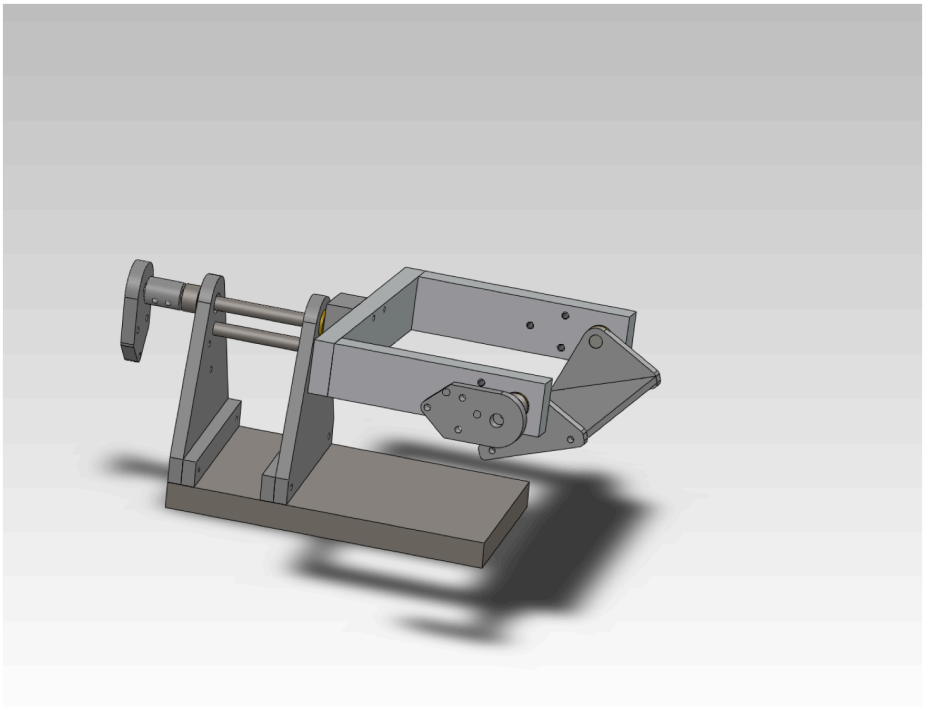
A pair of analog foot pedals is used as input to demonstrate the flexibility and ease of creating and configuring input devices. Pedal position is measured using potentiometer voltage dividers, and the resulting voltages are read by an M2 through onboard ADC channels. A threshold is set on each channel causing the output signal to behave as a digital device. Additionally logic is applied in code to create a third output channel that is the result of a logical AND on two pedal channels. As with all the input devices, the M2 packages the channel data and sends it via SPI to the haXbox upon request. A schematic follows:



Wrist Gimbal

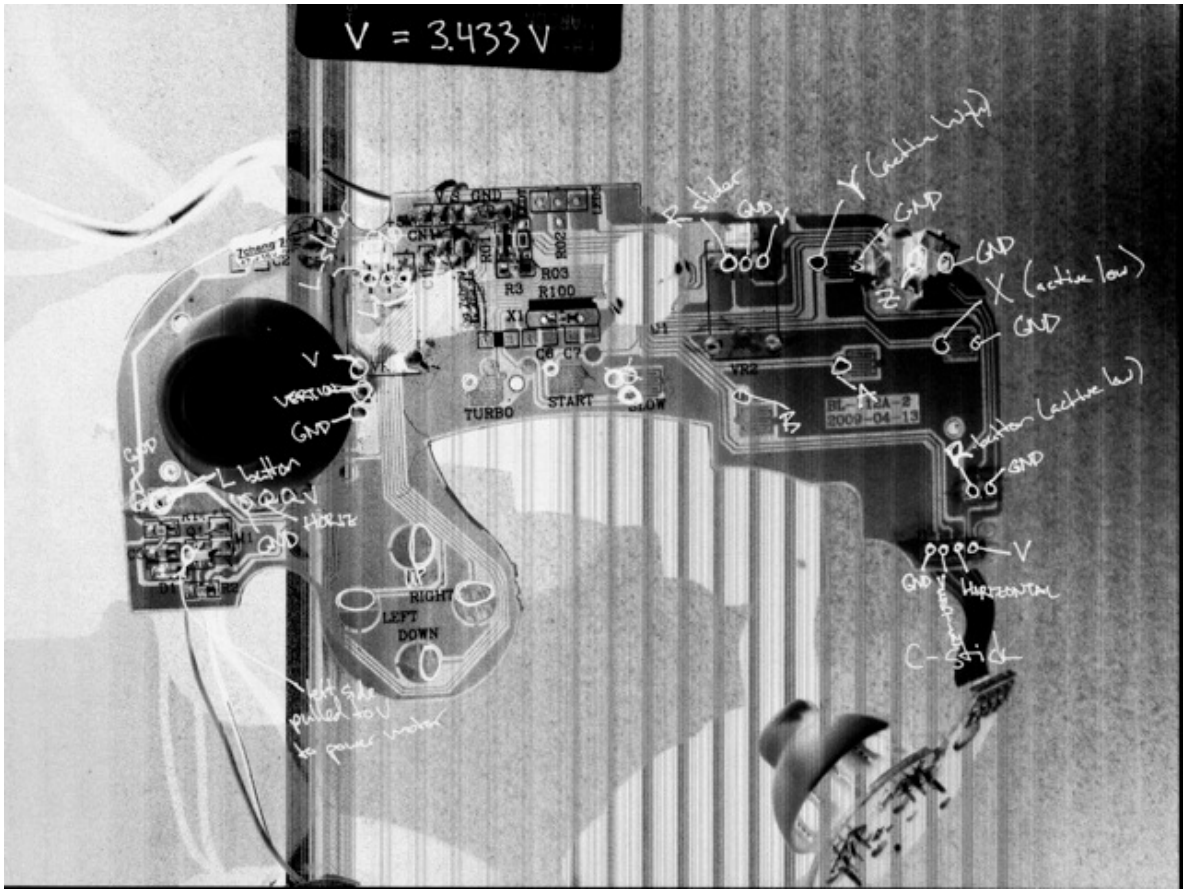
The wrist gimbal is an input device designed for this project to target patients limited in the mobility, strength, or dexterity of their wrists. It accepts two axes of input: supination/pronation ('roll'), and flexion/extension ('pitch'). Both axes are constrained to pivot around steel shafts by axially mounted bronze bushings. A mounting plate is provided as a means to connect handles or other manipulators suitable to patients' needs. Both axes are measured using potentiometer voltage dividers, the signals from which are read into an M2 through onboard ADC channels. These signals are passed through the M2 to the haXbox via SPI. Below is a schematic of the device, followed by a rendering of the design:



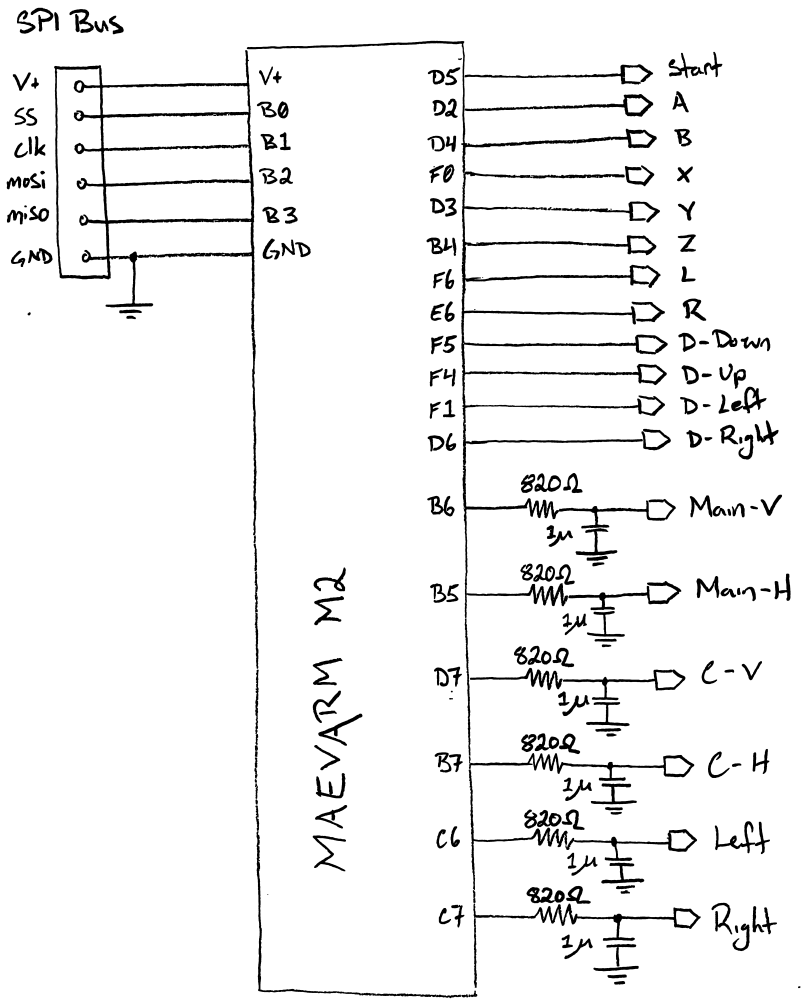


GameCube Controller

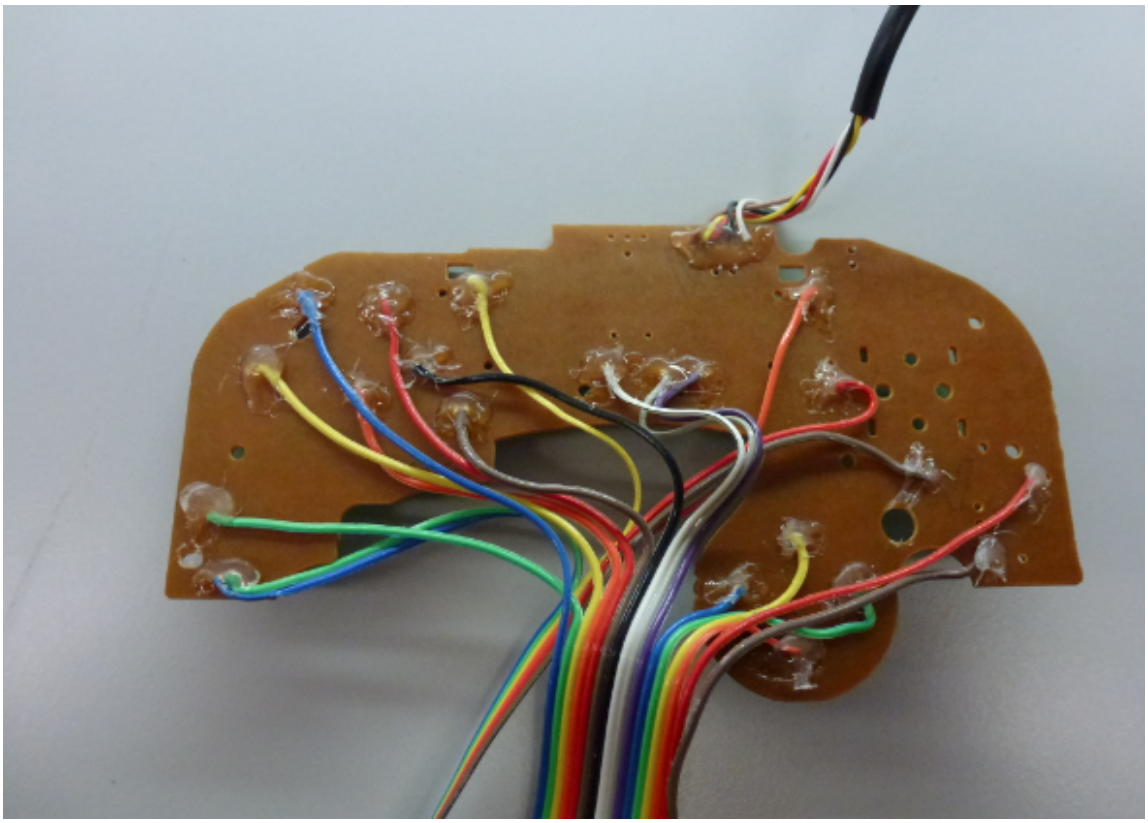
In order to control the GameCube, a generic controller was hacked to accept signals from the haXbox and feed these to the gaming system. Following is a map of channel locations on the controller's PCB:



The controller takes 6 analog and 12 digital inputs, plus two digital inputs ('turbo' and 'slow') that are not exposed to the user. The controller operates at 3.41 V, with an auxiliary 5v supply to drive the vibrate motor. Its digital inputs are active-low with internal pull-ups to 3.41 V, and its analog inputs operate rail-to-rail. The GameCube controller is driven by an M2 microcontroller, which accepts commands from the haXbox via SPI and applies the commanded signals to the controller. A full schematic follows:



Since the M2 microcontroller operates at 5 V, a non-standard method is used to control the GameCube. To leave a button in its normal 'un-triggered' state, the corresponding I/O pin of the M2 is configured as an input, thereby providing a high-Z connection, and allowing the controller's built-in pull-up resistor to control the line. To trigger a button, the corresponding I/O pin is configured as an output, and is driven low, thereby overriding the pull-up resistor to pull the line to ground. By solving the problem in code rather than employing some form of level-shifting or isolation, a great deal of unwieldy circuitry was avoided. Analog lines are driven by low-passed PWM lines, with care taken never to exceed the 3.41 V limit. The PWM is driven at 62.5 kHz, and the low-pass filter is tuned to 194.1 Hz, leaving negligible ripple. Following is a photograph of the controller wiring:



Device Creation

In developing our input devices, it became obvious, given certain details such as the use of SPI communications, that it would be beneficial to simplify the device creation process and thus enable less technical users to create their own devices. We therefore created a pair of command-line utilities – one to specify the channel types (analog, digital, logic-driven, etc.) and corresponding input pins of an input device, and the other to turn this information into finished C code to be compiled and flashed to the M2. Together these utilities reduce the coding process from implementing non-trivial C code by hand to simply following a set of procedures.

Results

As discussed, we have accomplished all the core goals set out for the project. We have constructed a modified GameCube controller that translates game commands received via SPI to valid GameCube inputs. We have built a first implementation of the haXbox, which takes multiple input devices and maps their inputs to an output game console according to a configuration commanded through a PC via USB. We have developed two non-standard game inputs for use with the system (sip-n-puff, and wrist gimbal), one of which (the gimbal) is an entirely novel design, and have demonstrated them working along with multiple pre-existing input devices, including a mini-joystick, a pair of foot pedals, and a computer keyboard.

Furthermore, we accomplished several of our reach goals, including useful features we conceived of during development. We developed a protocol for the haXbox that allows a PC to set up and configure input device mappings, as well as a Python class that handles haXbox connection, communication and configuration; this allows our input configurations to be done dynamically from a laptop rather than hard-coded on the haXbox controller. We implemented real-time device detection and recognition; when a user unplugs a connected device in run mode, the device is seamlessly dropped by the haXbox and picked back up when reconnected, and a PC-based mapper program can query ports for the identity of currently connected devices. We designed and machined a sleek enclosure for the haXbox; this gives the device a suitably finished appearance and the feel of a finished product; input and output ports, a USB port for PC configuration and programming, a power switch, a power supply connection, and a tri-color status indicator LED are all integrated into the package.

Additionally, we have made substantial progress on a Python-based GUI to extend the Python class that handles haXbox configuration. The GUI currently detects and displays connected USB devices, and on connection displays all connected devices and their associated device IDs, updating in real time, along with a dropdown mapping list. We have also developed a system for storing device profiles and retrieving device and channel information based on the two-byte device ID. This brings us well on the way to the drag-and-drop interface we envision.

For device designers (including our own demo devices), we developed an API to handle the SPI setup and communication with the haXbox, allowing the developer to simply write a main execution loop that collects input data and passes it to the haXbox communication routine. To further simplify the device creation process, we wrote a pair of command line tools that would allow a user to generate the C code for a device automatically, simply specifying the types of input channels and their connections.

One notable difficulty encountered has been with the GameCube controller interface; the main analog joystick started behaving erratically shortly before our first demonstration, and since the correct voltages were still being applied to the controller board, we assumed the microcontroller on the GameCube board had been destroyed. We hacked a second controller board, and this worked correctly for the second demonstration but again showed the same erratic behavior while setting up for our third demonstration. We realized that since the laptop being used for mapping (via USB) was plugged in to charge, it was likely creating a grounding loop with the power supply for the GameCube, thus causing the strange behavior. Indeed, the behavior returned to normal a few minutes after unplugging the laptop from its charger.

Reflections

The haXbox is complete, and is a finished product. It was immensely satisfying to take an initial kernel of an idea, refine it until it was an exciting design, and carry it through to a functional, polished product. This was largely facilitated by an excellent team dynamic, which we both found to be the best we have ever experienced. We both share a similar sense of aesthetic in virtually all aspects of design, from code to hardware to user experience, which is a rare match. Furthermore, we both strongly desired to create a *well-designed,* usable product with potential for real and significant impact, taking into account the many design considerations that entails. We thus found the hours of nit-picking and rabbit trails during the lengthy design phase, and the many days and nights in various labs turning the concept into a reality quite fulfilling, if at times frustrating.

One of the most significant lessons we have been learning is how to instill in an audience an accurate idea of what is important in a project. We were sometimes frustrated and often bemused by viewers' fixation on the input devices (particularly the sip-and-puff), since the really exciting, groundbreaking result of the project is the sleek haXbox platform that allows *any* input device to be easily adapted and configured for use in *any* game on *any* supported console. Many interesting alternative game input devices have been created already, even for this class, but they are typically very involved and limited projects since the designers must go through the trouble of adapting them for a single particular game system. The haXbox, by contrast, allows even a moderately capable electronics hobbyist to develop an input suite fully customized to a given user's needs—limited only by his or her capability at designing input mechanisms, and not by any subtleties of electronic communications and systems integration. Unfortunately, to the average person that is not an adequately compelling vision in and of itself, at least initially— but the sip-n-puff makes it *real*.

> "Ah. So the haXbox can be used for *paraplegics*."
>
> *Sigh. It can be used for anyone! Any need! Seriously! Input devices are only limited by your imagination!* . . ."Yes, exactly. And not just paraplegics, but you could make input devices for someone with limited hand dexterity— foot pedals, for instance. Or make a device for someone paralyzed below the shoulders, so they can use shrugs for input. The point is that it's very flexible and adaptable…"

Anyway, one must both communicate the vision and make it concrete.

Future Work

Discussion is underway with a group of therapists at Good Shepherd Penn Partners to develop and test suitable input devices, and to do research and validation on the system as a whole using real patients. The haXbox system has reached a stage of maturity where focusing on user testing will be appropriate and beneficial, and will significantly validate the project and shape the future direction of development. In the immediate future we must complete the configuration GUI, so that the system will be usable by untrained, non-technical therapists.

In the short term, to address the issue of the ground looping affecting the GameCube controller, we plan to opto-isolate GameCube communications from the rest of the haXbox circuitry; this should fully resolve the issue. We would also like to make an alternative GameCube translator using bit-banging, as it would be faster and easier to produce en masse, and less prone to hardware malfunction.

Taking advantage of the mbed's built-in file system, we hope to store mapping configurations in the haXbox before power-down, and if a configuration is stored, to restore it to use by default on startup rather, than entering configuration mode. Configuration mode would by initiated by a message from the PC, rather than being the default startup state it is currently.

Two input utilities we would like to create are expansion ports and analog input boards. Expansion ports would accommodate more devices than there are physical ports for on the haXbox, as discussed previously. Analog input boards would have many input channels, each directly transmitting voltages from any connected electronics. This would be very useful for testing during device development and would eliminate the need for dedicated microcontrollers on simple analog input devices.

One particular input device we would like to develop is an IMU-equipped controller similar to the WiiMote that could be trained to recognize user gestures using machine learning. The idea is that a handicapped patient may not be able to do a precise action required by a certain game (particularly motion based games, such as those supported by the Wii), due, for example, to limitations in range of motion, dexterity, or speed, but such users could make a recognizable attempt. A controller that could translate that attempt into a valid game input would open up new opportunities in gaming for such patients, and could also incorporate a teaching mode that would challenge the patient to improve desired capability (e.g., range of motion) over time.

Perhaps most importantly, our broader vision is to create an open source community around the project, which would greatly advance our goal of making the haXbox accessible both to hospitals and to home users. On one hand, this would allow people to publish their designs for useful input devices and upload matching device profiles to be downloaded with standard haXbox software distributions, and perhaps to offer to produce the devices themselves for a fee. On the other hand, talented software designers could develop and maintain console drivers for various game systems, by hacking the communication format and spoofing it via bit-banging. With these available, a reasonably tech-savvy user could simply reprogram a microcontroller configured for one console and connect it to the controller cable for a new console whenever he or she wanted to use a different game system.

In summary, by connecting therapy-tailored devices to industry-standard games, we are opening new modes of therapy that are accessible, affordable, engaging, and effective.