Raspberry Pi Keyboard Driver

M. Paul

December 9, 2016

1. Introduction

One of the more notable features of the raspberry pi 2 is its 40-pin male header located along the edge of the board. Of these 40 pins, 26 pins can be reconfigured as either input or output. These pins are called General Purpose Input Output or GPIO pins for short. By passing current from one of the GPIO pins configured as input to the ground line which is also accessible from the 40-pin header, the raspberry pi has a means of detecting input such as switches and levers. By changing the state of one of the pins configured as output from high to low or vice-versa the raspberry pi is able to send power and signals to external devices like a light bulb or display. Through the use of these properties, the raspberry pi can interface with the real world. [2](Raspberry Pi Foundation, GPIO: RASPBERRY PI MODELS A AND B)

A keyboard is a prime example of a peripheral used to obtain user input from the outside world. With the use of driver software, a computer system can decipher the signals being sent from the device and match them to a corresponding character or action which is carried out by the machine. Each key press corresponds to a different signal being sent so being able to differentiate between key presses is a crucial function of the driver.

The goal for this project is to produce a keyboard and driver that makes use of the GPIO pins. Ideally it will be able to effectively replace a conventional USB (Universal Serial Bus) keyboard for use with the raspberry pi. Since the pi only has 4 USB ports, it can be beneficial to free up a port to allow for additional peripherals to be utilized and considering how the retail price of a raspberry pi 2 is roughly 50$ Canadian, it is also understandable how someone might not want to spend an additional 20$ on a peripheral that is essential for the machine's operation. Driver software is often very restrictive and hard

to access the code for it. An open scource driver provides additional benefits such as the potential for customized key mappings and adjustments to input response times otherwise inaccessible to users. This project is effectively a test to see how close is it possible to recreate the keyboard experience using a do it yourself (DIY) solution that's made from parts included in most raspberry pi learner kits.

Using a prototyping board, 30 tact switches, a 40-pin male header, a ribbon cable, and a series of jumper cables, most of which was included in the kit that came with the raspberry pi, I was able to build a functional keyboard with an estimated price tag of about 6 or 7 $ which is noticeably cheaper than a real keyboard. The software used to operate the keyboard implements a few customization variables which can be altered in the code to select which predetermined key mapping to use as well as modify input trigger timings. By configuring all the GPIO pins as input, and then binding all the input pins to a character on a virtual keyboard the triggering of a tact switch will initiate a key press. The code also implements a delay variable which permits the user to select how much time needs to pass between actuations which can increase response time or decrease the likeliness of unintentional key presses. This code isn't locked away behind copywrite protection or encryption and is thus open source and easily modified by a user should they choose to do so.

Section 2 establishes all the background information required to understand the concepts involved with the creation of the keyboard and Drive.

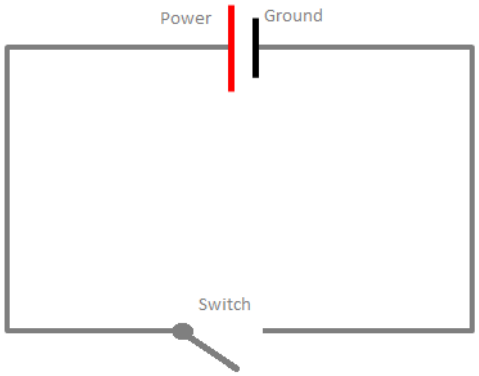
Section 3 illustrates the keyboard design and gives in depth descriptions on how the driver works.

Section 4 gives an honest self evaluation of what the project did well as well as some areas in need of improvement.

Section 5 summarizes the project, explains its relevance to the course and speculates ways to improve.

2. Background

Knowledge of the raspberry pi's hardware was crucial to this project as well as some technical understanding about circuitry. Past experiences performing basic input/output operations using tactile switches and Light-emitting diodes (LED) played a big role in the design of this driver as well as prior experience modifying and utilizing a driver provided by Adafruit (engineering website) for use with a DIY electronics project. The first thing that needs to be understood is how a basic circuit works.



The Figure to the left illustrates a simple circuit which is the foundation of the keyboard's design. Every key functions as a switch which breaks the flow while in its default state, but will complete the circuit when actuated. This allows power to flow directly to ground. [2](Raspberry Pi Foundation, GPIO: RASPBERRY PI MODELS A AND B)

The next most important thing to understand is the functionality of the raspberry pi's 40-pin header.

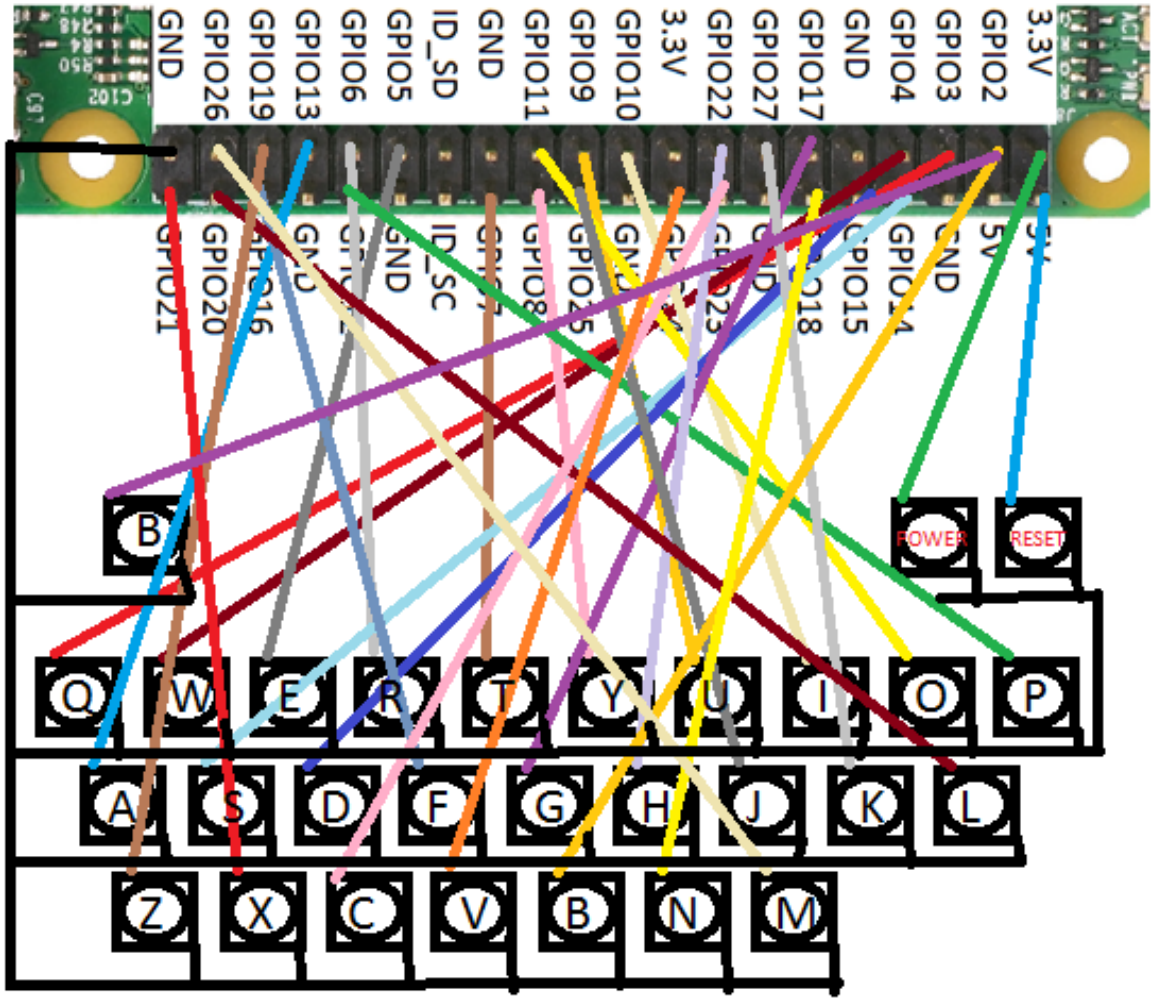
The figure to the left is a detailed pinout of the Raspberry Pi 2 B+'s 40-pin header. Included in the 40 pins is 4 pins used for DC (direct current) power labelled as either 3.3 or 5 volts, 26 pins used for IO (input/output) labelled as GPIO2 through 27, 8 pins which connect to ground labelled as GND and 2 pins reserved for Electrically Erasable Programmable Read-Only Memory (EEPROM) which are only used by raspberry pi compatible hardware in order to communicate device information needed to identify the hardware and run the appropriate drivers. They are labelled as ID_SD and ID_SC in the diagram. [3] (University of Cambridge, Raspberry Pi)

The Adafruit driver [1](P.Burgess, Adafruit-Retrogame, 2013) that was used as the basis of this project was originally written as a means to map hard wired videogame controls to the GPIO pins of any Raspberry Pi board for use with RetroPie, an arcade emulation system built off of MAME (Multiple Arcade Machine Emulator). It is an excellent program that is open sourced, well documented and encourages community modifications. It is however extremely bloated as it uses thousands of lines of code spanning multiple files and performs a lot of subjectively unnecessary tasks pertaining to error checking and hardware compatibility. It was also designed specifically for RetroPie and therefore includes a multitude of functions that only serve a purpose on that system. A criticism I have for most driver software in general is its lack of user transparency. Adafruit's driver is a clear exception to this, however most drivers are copywrite protected and/or encrypted with no intention of ever being seen by people not employed to make them.

Keyboards are a necessity to the computing experience and a raspberry pi is no exception. While purchasing a keyboard is generally a small expense when used with a full-scale desktop PC (personal computer), when the price of a keyboard starts to rival the price of the computer it becomes far less reasonable. Fortunately, the primary demographic of consumers for the raspberry pi is programmers and DIY enthusiasts who probably have little to no problem with implementing a cheaper alternative for a keyboard.

3. Result

By Using all 26 of the GPIO pins as well as a ground pin and 2 power pins, I managed to construct a keyboard which incorporated 26 different input values as well as a reset switch. On the software side of things, I produced a driver that binds a virtual key press to the actuation of a physical switch there by retrieving input from a user and carrying out an action within the machine. The following schematic is a model of the keyboard in question.



When an input pin passes power to ground, it goes from high to the low state. When the pi detects this change in state, and recognizes that that specific pin has been has been triggered. So with all the GPIO pins set up as inputs and connected to switches, as detailed above, we have a functioning means of obtaining user input.

The driver works by creating a series of keysets using a struct that pairs every pin with a different key press. It then picks one of these keysets based on the value of the state variable and modifies a system config file appropriately to reflect the choice. It reads the file it modified to alter the state of the pins. From there it starts an infinite loop which waits for an input event to occur. When an

event occurs, it stores the info in the instate array as a flipped bit and then issues a keystroke that uses the pin's paired key setting depending on how the comparison between the instate array and the defaulted state of the pin goes. It then starts a timeout loop which freezes the input for however long the delay is set to. When the timeout ends it returns to its waiting for input events state.

The code also includes a plethora of error checking and handling. Right off the bat, the only way to access the system config files is with the root privilege so trying to run the code without it will throw an error. Trying to read from or write to the system file can also fail, so properly handling those errors is needed. Resetting the pin header is a very important step as failing to do so will have lasting effects on the raspberry pi's hardware. Thus, it is a good practice to insure a proper cleanup procedure gets called whenever the program closes. To help with this, a global running variable exists to abort the main loop should the program need to, and a signal handler is utilized to flip it off when needed.

4. Evaluation

With the goal of producing a product that can replace an existing USB keyboard at a much lower cost in mind, the keyboard and driver I designed for this project fairs well. However, there is a substantial list of drawbacks that come from using my prototype. For starters, the hardware limitation of the GPIO pins limits the keyboard to only 26 different inputs per mapping. This is enough for a full alphabet but not enough to include other valuable keys like backspace and enter. There are ways around this problem such as requiring a switch to bridge 2 or more pins to ground in order to trigger a keystroke, but this would require a more intricate wiring solution. With the value factor of this project relying on people being able to make their own keyboard, a more complicated design would be troublesome. Finally, a big thing that takes away from the user's typing experience is the keyboard's quality, no one could argue that a real keyboard isn't a more polished experience compared to a cheaply

implemented series of switches and the work that goes into wiring up such a thing takes time and energy that very few people would find worth while.

The driver software aims at being user accessible and customizable. As such my driver isn't hidden behind encryption or copy write. Furthermore, I included a few variables in the code that users might want to tune to match their preferences. My code is arguably less intimidating than the Adafruit driver used as a reference, since I only use about 150 lines of code compared to the thousands used by them. My driver doesn't include nearly as much functionality in terms of hardware compatibility, but for anyone who's merely looking to map pins to key strokes they'd probably prefer my code as it is far less bloated making it simpler to understand. A few problems might occur from giving users complete access to a driver that modifies system files, but anyone who's looking to implement a DIY keyboard is probably proficient enough to avoid damaging their hardware.

5. Conclusion

For this project, I managed to create a driver that modifies the GPIO pins of a raspberry pi 2 and allows them to interface between a DIY keyboard fashioned from a collection of tactile switches and the machine. It works as a cheaper alternative to a real keyboard which would potentially cost about half the price of the computer it was used with. The driver is open source making it accessible to any user who wishes to modify it, and it even includes a couple customization variables they may wish to change.

Devices, drivers, and user IO were all prominent topics featured in this course. By designing a driver, I was able to operate a device which obtained user input and produced an output. A substantial part of this project was spent learning and designing circuits which are an underlying principle of all electronics and perhaps an understated topic within the course. The coding segment of my project makes use of signals/handlers as well as error handling which were the focus of some of the earlier

assignments. An operating system is only accessible to a user if they have a way to interact with it, and that is primarily through devices like a keyboard.

Future ways to improve this project could include a redesign of the driver to require 2 pins to be bridged instead of just one. This would allow for a much greater number of unique inputs, which would permit larger key maps. A printed circuit board (PCB) would also be a beneficial change to the keyboard's design as it would simplify the construction and potentially lower the per unit material costs if it could be manufactured in bulk. The software could also benefit from a user-friendly interface which would permit sanitized user modifications instead of just letting them alter the code.

Contributions:

This project was done solo, so all work involved with the hardware, software and report was done exclusively by M. Paul.

References:

[1] P.Burgess, "Adafruit-Retrogame" https://github.com/adafruit/Adafruit-Retrogame: 2013

[2] Raspberry Pi Foundation, "GPIO: RASPBERRY PI MODELS A AND B" https://www.raspberrypi.org /documentation/usage/gpio/

[3] University of Cambridge, "Raspberry Pi" https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials /turing-machine/two.html