Camera Capture of QR Code for Automated Door Unlock Using Raspberry Pi and Gumstix

Team OpenSaysPi
Shantanu Bobhate and Michael Z. Webster

Abstract

The goal of the project was to demonstrate a successful door-lock system with QR code based authentication. The QR code unlock leverages the greater processing power of the Raspberry Pi for decoding and low-level kernel modules for quick response from servos and status LEDs. The project resulted in a complete, compact, and enclosed system that could easily be switched from independent operation on battery for several hours or direct power via a USB power supply. All components of the system are designed to be as low power as possible while maintaining the core functionality.

Introduction

Our product is an automated door lock and unlock system that uses QR codes for authentication. The motivation was to provide easy access to homes while maintaining or exceeding the current level of security. The rise of companies like Airbnb and HomeAway have made renting your home very easy. Since a major concern with renting out your apartment to strangers is to have a safe and reproducible way of giving them access, our project was aimed at just that. The system we wanted to provide was a hybrid between a lock control and video monitoring system.

Using QR codes as a means of authentication makes the process of giving access simple. QR codes have been becoming more and more popular, as a means of authenticating a purchased movie ticket or a means to check-in for a flight even in some cases being used directly for payment. We wanted to incorporate this simplicity in our locking system. The other advantage of having QR codes is that they also can help users to identify the person accessing their home providing a more manageable log and better sense of security. Added to that, with the availability of services that can provide embedded QR codes that can be remotely updated as needed, such as in passes within Apple Wallet, that allows for flexibility in keeping the database of authenticated users up to date while avoiding any compromised codes from overuse. This can also all be done without the need to have user generated QR codes that may not be unique enough to ensure proper security.

To achieve the core functionality, a Raspberry Pi Zero Wireless - offering better external connectivity and more processing power - was linked with a Gumstix. The Raspberry Pi had a hardware level attached camera that could reliably capture QR codes. Using the GPIO pins available on both devices a signal could be sent that triggered the Gumstix to enter an unlock state and report the status using LEDs. Both devices were powered by the same external 2500mAh battery as was the single servo. The battery was regulated using a 1000c PowerBoost that could also accept a 5V input from a microUSB for both power and charging. Using this setup, we were able to achieve a rapid unlock in less than 3 seconds with very little worry of how well the QR code was placed. This time was minimized as such by improving the communication speed between the devices as well as fully optimizing the kernel modules.

Design Overview

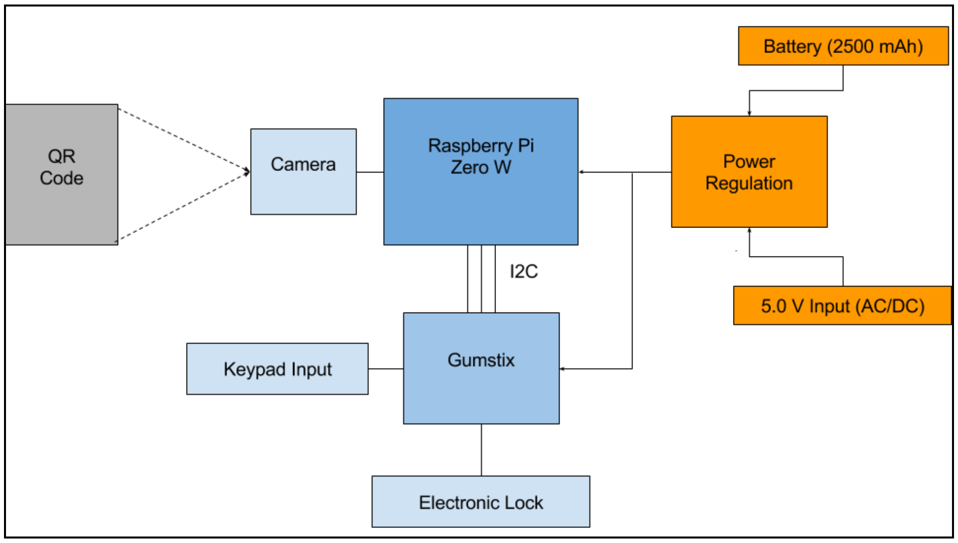

Figure 1. This diagram showcases the system-level diagram for our project. There are 2 main control nodes, the Raspberry Pi and the Gumstix Verdex Board. The Raspberry Pi is the higher level control module in charge of acquiring and decoding QR codes from the camera and authenticating the users. It's on-board wifi also makes it the perfect high-level control since it would make it easy to have it communicate with a backend database for the controlling valid QR codes from a dedicated web/mobile app.

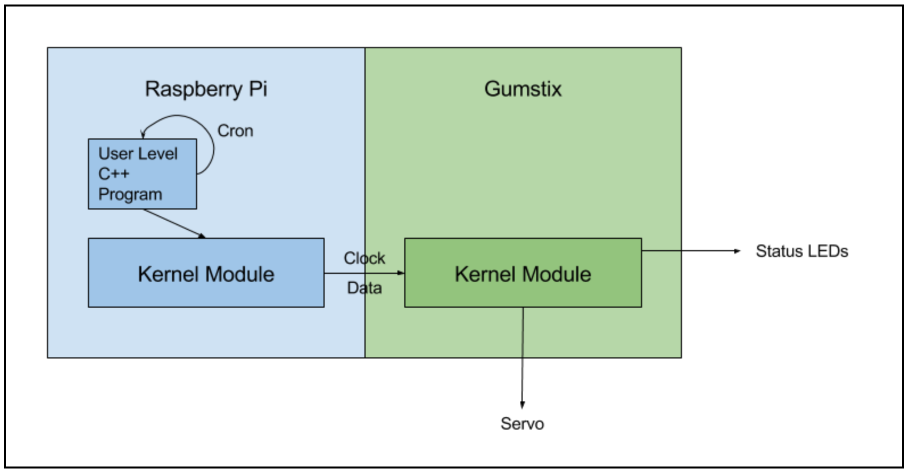

Figure 2. The diagram above gives a program-level overview of our system and highlights the interaction between the software and hardware in the system. The Raspberry Pi has a user level program that is continuously grabbing images from the PiCamera and using a utility tool called *zbar* to locate and decode the QR code in the image frame. It also uses the OpenCV-3.0.0 package to open and read frames from a video feed. Upon a successful authentication, the user level program communicates with an installed kernel module via a device file. The write invokes a signal to be sent to the Gumstix on a clock dictated by the Raspberry Pi. The gumstix acts on this by updating the status LEDs and actuating the servo.

Project Details

A. QR Code Unlock

The Raspberry Pi was in charge of decoding and authenticating the QR Codes. A user-level c++ program on the Pi user *OpenCV-3.0.0* to read images from the PiCamera. OpenCV is an open source computer vision library. Each frame was then decoded using a tool called *zbar* which is an open source software suite for reading and decoding barcodes.

On decoding, *zbar* extracts the embedded message from the code performs a comparison against stored passwords in a text file. A correct match with any of the entries signifies an authenticated user. A future upgrade to this method would be replace the text file with a database API call that queries the current passwords set by the user in the database.

Once the user is authenticated by the user-level program, it logs the entry to a log file and writes the message “open” to the device file */dev/mylockcontrol* loaded on the Pi. This device file is linked to the kernel module installed on the Pi called *mylockcontrol.ko*. This kernel module is the *master* module in the communication protocol between the Pi and the Gumstix. This module has a constant 20 Hz clock on pin G5 that signals the Gumstix when to expect signal and helps keep the 2 devices in sync. When a process writes the message “open” to the device file the kernel module signals the Gumstix to unlock the door by pulling the pin G6 high on the next positive edge of the clock and pulling it down again on the consecutive clock cycle. This signals the kernel module on the Gumstix to take action.

The kernel module installed on the Gumstix has a interrupt handler that is fired for every rising edge of the clock signal from the Pi. This allows the Gumstix to check if an unlock signal is sent from the Pi. On receiving a HIGH signal on pin I2C-4 the kernel module changes the state of the lock to the unlock state. This causes the RED LED to turn off and the GREEN LED to turn on signifying that the door is locked. Ideally, this is where the Servo would be commanded to move into its unlocked state, i.e. 160 degrees.

One of the refinements for the future would be improving the efficiency of the OpenCV component in the user-level application. During our testing phase we discovered that the user-level program consumes about 96% of the processing power during its image capturing and QR decoding phase. We would like to reduce this so that its impact on the battery life of the system is less. Perhaps there would be a method by which the user-level component could be called on demand while still maintaining the current latency.

B. Keypad Unlock

The keypad unlocking mechanism was a fail-safe implemented in case the PiCamera failed to detect the QR code. The mechanism provides a simple manual way of entering a passcode that authenticates the user and allows entry to the house.

The kernel module has an interrupt handler for each individual button. A 4-bit array keeps track of the pattern entered. The passcode is evaluated when the 4th button in the sequence is pressed. The passcode is hardcoded in the current version of the code (3214) but the module could implement a *setup* state where the user can use the buttons to reset the passcode to a different pattern. Since the device is headless, we added a yellow LED to blinks thrice to signify that a wrong sequence was entered. This allows the user to know when to re-enter the passcode.

The first button has an alternate function, that is to lock the door. The module has an internal 50 ms timer that is constantly checking for coherence between the level of the first button and its state on the previous clock cycle. When the level has stayed HIGH across 2 consecutive clock cycles the kernel module register the press as a lock and locks the door. The timer expiration time could be extended to require a longer press in order to lock the door, eliminating a accidental lock occurring on pressing the button. A 1000 ms timer should work.

C. Servo Lock/Unlock

We decided to replace the electronic deadbolt lock with a servo motor to add some complexity to our project and allow us to gain some more knowledge. We began by trying to control the servo using the PWM header file from Lab 5. The lack of documentation on the implementation of this setup posed a challenge. The main challenge was getting the right signals to drive the servo. The details on how a Servo works are given in the image below. The servo expects a pulse every 20 ms and the width of the pulse determines the angle of the motor. A 1 ms long pulse sets the angle to 0 degrees while a 2 ms pulse sets the angle to 160 degrees.

To get around this method we created our own kernel module capable of controlling a servo using High Resolution Timers in linux. We create a timer that had a total period of 20 milliseconds and we changed the duty cycle to drive the servo into the correct state. The GPIO pins of the Gumstix board posed a challenge as they seemed to be having some limitations. The kernel module worked seamlessly on the Raspberry Pi, however the same module gave a very unstable output on the Gumstix. We tested the output using an oscilloscope and the results are given below.

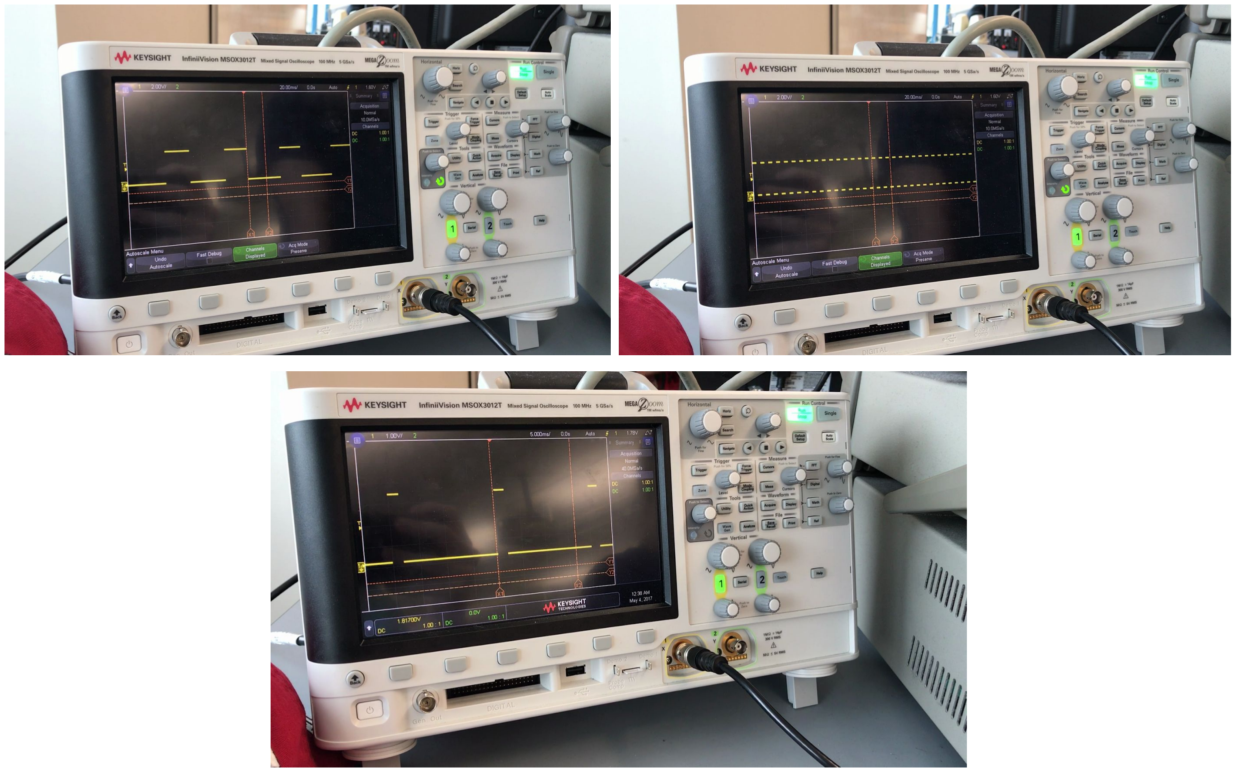

Figure 3. The two images above show the output intended for a servo coming from the Gumstix as measured on an oscilloscope. The same kernel module, running on a Raspberry Pi, gave the proper output as seen in the bottom image. Only with a stable signal and pulse as expected will the servo actuate as expected.

Another problem we ran into with the servo was the current drawn by the servo was causing it to disrupt the power connections to the Pi and the Gumstix, forcing them to reboot. We suspect that this issue is caused because we try to power all the devices on the same battery.

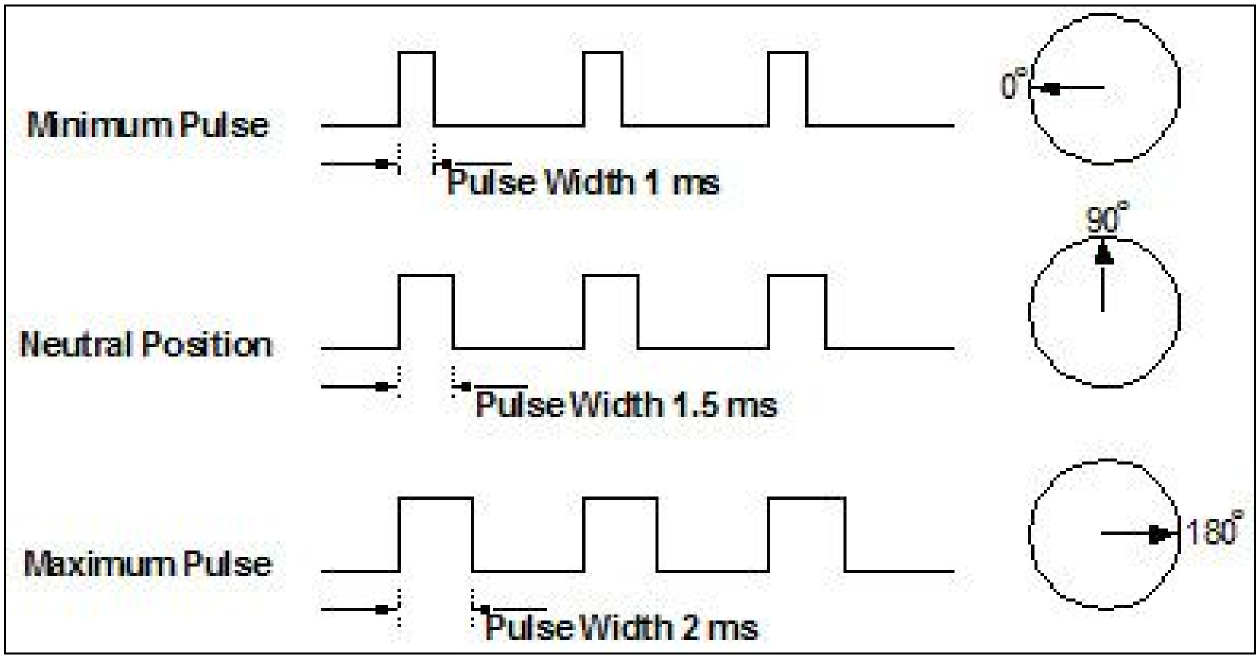

Figure 4. The servo response is tied to a specific pulse width provided over a certain time frame. Given the accuracy needed, we were able to use high resolution timers (HRTimers) to actuate the servo to a desired angle [4].

Summary

For this project we truly achieved a very fast unlocking mechanism that recognizes QR codes and acts on them in a way that would feel very seamless to users. Some video monitoring was also enabled therein by recording a time stamped image of the authenticated QR code for better security. Plenty of challenges remain, several of which highlight that this project serves as an excellent foundation on which an even more realized system could be built. An interesting component of this is that for a relatively low cost a system can be assembled that is overall small, powerful, and easy to interconnect.

Additional challenges lie primarily in refining the servo use, perhaps transferring the project to entirely to run on the Raspberry Pi. One of the first steps would first be to isolate the power provided to the servo as often its use would cause a low voltage event across the other devices. Furthermore, if a second embedded linux device such as the Gumstix is used in future iterations a more robust communication protocol would need to be built to enable two-way communication with return states sent in reply. There may be some advantage in maintaining a second device as it adds redundancy in case of any failure on either device. Additionally, having a second device obscures either intentional or unintentional activations of the servo (or other locking device) that could be protected both physically and with a more secure communication.

To extend the capabilities of the project, we could give the system a stronger software backend. A database could be implemented for more control over the passcodes and QR codes. The greater access options on the Raspberry Pi also give the option of adding a streaming server to attach to a webpage or mobile application. A final link to the growing AI industry would be to have Amazon's Alexa connected to our system and have it do most of the heavy-lifting.

References

[1] Watchdog Script to Keep Process Running with Cron.
https://community.webfaction.com/questions/6157/watchdog-script-to-keep-process-running-with-cron
[2] High-resolution kernel timers.
http://www.ece.cmu.edu/~ee349/gumstix/opt/include/linux/hrtimer.h
[3] The high-resolution timer API. https://lwn.net/Articles/167897/
[4] Servo Motor. http://blogspot.tenettech.com/servo-motor.html
[5] Timers and lists in the 2.6 kernel: https://www.ibm.com/developerworks/library/l-timers-list/
[6] The Linux Kernel Module Programming Guide: www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf
[7] User space memory access from the Linux kernel:
www.ibm.com/developerworks/library/l-kernel-memory-access/
[8] Writing a Linux Kernel Module - Part 2: A Character Device:
http://derekmolloy.ie/writing-a-linux-kernel-module-part-2-a-character-device/
[9] How compile a loadable kernel module without recompiling kernel:
https://raspberrypi.stackexchange.com/questions/39845/how-compile-a-loadable-kernel-module-without-recompiling-kernel