

Abstract:

This project implements an automated gate system that operates based on a barcode attached to a car. Network communication protocols are included for ease of management. This implementation can decrease operation costs, by doing number of entrances can be increased with minimal cost.

Introduction:

The objective of the project is to design a system that would automate a gated entrance. This would decrease operational costs. This way traffic throughput can be increased at minimal costs. System consists of a raspberry pi embedded system running on a Linux environment. There are two sensors connected to the board: distance sensor and a barcode scanner. The board is also connected to a network, where it can be managed, and provides information about gate users.

Background:

A perfect example would be a gated community with multiple entrances. Instead of having a guard posted at each entrance we can have the guard at one where all the visitors would be redirected and the rest would be automated for barcode owners by the proposed design. by installing new entrances operating with the aid of proposed implementation traffic throughput can be increased significantly compared to investment and operation costs. A single guard can easily manage multiple gate entrances running the embedded system. Using the network protocols real time data on traffic passing through is available. Also users can be added, removed, log data retrieved, and system restarted. Additional functionality can be implemented: voice communication, real time video, remote operation....

Proposed Method/System Description/Implementation:

Network
Communication

FIFO

Main program

Server
Thread

FIFO

Barcode Scanner
ISR



Barcode scanner scans a barcode attached to the car and passes that information to the board via a write to FIFO which triggered by an interrupt. Main program evaluates the code and sends an appropriate signal to the gate. If the code is valid the gate opens otherwise it remains closed. Once the gate is open the distance sensors data is continuously read in by the Gate Control portion of kernel and evaluated. Once it determines based on the data that cars has passed it send a signal to the gate to close it.

There are 2 files on the board memory (semaphore protected):

- Key.txt -> This files holds the barcode key numbers that are allowed to pass the through the gate. When program is launched this file is read into a dynamic array to speed up searching for key matches.
- Log.txt-> This file holds the log of all the barcodes scanned, where they allowed to pass thought, and the event timestamp. This file is constantly updated thought program execution.

Network communication between the client computer and the server thread in main program is done using connection based protocol (TCP) and sockets.

'Over the network' functionality includes:

- Addition and Removal of barcode keys.
Addition -> "@ + [barcode]"
Removal -> "@ - [barcode]"
Upon either of the above operations the Key.txt file is edited accordingly and the dynamic keys array updated.
- Printing the log file "@ log"
Upon sending the above operation to the board server thread, log.txt file is read and transferred thought the network to the client, where it can be printed out on a screen. Instead of printing, the incoming character stream can be redirected and written into a file.

Experiments and Results:

The module portion of the program has not been implemented, and is simulated using two different programs. Mod_sim.c program simulates the receiving information intended for kernel space gate management sent from the main program. Second program barcode.c simulates sending barcode data through a FIFO to the main from kernel Barcode ISR. User space portion and client has been implemented successfully and fully operational. Different number of barcodes has been processed by the main program without crashing it. Barcode information has been processed correctly if an integer number has been passed in as designed. There has been no success corrupting the log.txt and key.txt file because they are protected by semaphores. There is a sem_wait() before opening a file a sem_post() after closing it. If multiple tasks/processes/functions try to access the file at the same time, only one will access the file at a time.

Discussion and Conclusions:

The module portion of the program has not been implemented. The outcomes of experiments conducted above were anticipated based on working the bugs out in the code.

An interesting observation made during coding was that there were no library functions to remove a line of data from a file. In my case when user requested to remove a barcode key I had to remove the key from key.txt file and reload the dynamic array. I implemented this functionality by opening two different files, original one and duplicate. I would be scanning the original file line by line copying the lines that did not match remove key criteria to the duplicate file and just not copying a if it mated the remove key criteria. Once done, deleting the original file and renaming the duplicate in place of original. This implementation could get very resource consuming if the number of entries in a file is large.

Implementation of the TCP protocol was not too bad however some learning how to was involved. In the TCP protocol a server listens and the client need to connect to it before any exchange of information can be done. In addition after connection was established the server assigns a new socket descriptor through which all communication will be done. In my implementation I did not use a multi client model, but the server code can be modified to fork or create a thread once the connection and new socket descriptor was assigned. This way thread/threads manage client communication and main process/thread is listening for new clients. I ran into a problem when I was sending the log data from the user space main/server to the client. I was sending multiple lines of data from the server without any read() in-between them, and doing multiple reads on the client without any writes in-between.

This produced unpredictable behavior and random output on the client side. I fixed this by synchronizing the information exchange using acknowledge writes in-between the reads on client side, and acknowledge reads on the server side.

Other small everyday coding problems were encountered and overcome while coding the file read/write, dynamic key array and program flow.

The kernel module was not implemented due to because I ran out of time. Instead of working on the project throughout the semester I put it off till the break. However on the break I let to town to see my parents. Then end of semester work and finals piled up and

I started looking at the data sheet for raspberry pi to figure out the register mapping info, but it proved more time consuming then I thought, so eventually I ran out of time.