JJEE
Volume 6, Number 3, 2020
Pages 179-203
Jordan Journal of Electrical Engineering
ISSN (Print): 2409-9600, ISSN (Online): 2409-9619

An Open Source IoT Garage Real Time Controller (GarageRTC)

Daniel A. Zajac[1], Jackson Harmer[2], Adnan Shaout[3*]

[1, 2, 3] The Department of Electrical and Computer Engineering, The University of Michigan, Dearborn, MI, USA
E-mail: shaout@umich.edu

Received: May 10, 2020 *Revised: June 2, 2020* *Accepted: June 7, 2020*

***Abstract*—** Internet-connected technologies have become mainstays of the modern household. From Internet of Things (IoT) connected coffee makers to sophisticated adaptive climate control systems, inexpensive wireless technology and the popularity of voice-activated digital assistants have enabled a wide variety of connected technology. Most of this technology remains closed source. However, many popular connected technologies such as Nest or ecoBee rely on closed source protocols and private cloud backends. What happens to the hardware when these companies go out of business or shut down older services? VueZone shut down its services leaving owners with severely crippled IoT cameras. This paper focuses on building a proof of the concept of open source IoT-connected garage real time controller (GarageRTC). It presents the features, design, and implementation of a reference architecture built on an ESP32 microcontroller and free real-time operating system (FreeRTOS). The results show that the GarageRTC meets most of the performance and design requirements as identified in the concept and requirements phase. The system is more than capable of proving responsive interactivity with a residential garage door system. Moreover, benchmarking - functionally - the system against other commercial offerings enables it – due to its platform flexibility - to outperform its commercial counterparts.

***Keywords*–** Garage real time controller; Internet of Things; free real-time operating system; ESP32 microcontroller.

1. INTRODUCTION

As Internet of Things (IoT) becomes more commonplace in our daily lives, we become increasingly dependent on the connected services that support their systems. Some of popular devices, such as the Nest thermostat, are the product of a small team operating as a startup. As these teams grow, they are sometimes bought out by larger corporations interested in entering into their respective market. For some less fortunate startups, they never breakthrough into profitability and slowly fade into obscurity.

What happens to the devices that depend on the cloud services previously maintained and supported by those teams? Sometimes the equipment can continue to function but with absent features or limited performance [1]. Often, they are rendered useless as registration and web-based configuration tools become obsolete [2]. For inexpensive devices, they can be cannibalized for parts or simply disposed of. However, for more expensive equipment, such as the Juicero, the founding company dissolved, leaving many users with useless $400 IoT juice machines [3].

The only guaranteed way, to ensure these devices being indefinitely supported, is to implement an open source methodology including web application programming interfaces (API). Preferably, a completely open source hardware and software reference design would be created and published. Startups, wishing to develop derivative works, could leverage the technology and extend the designs. If the startup dissolves or no longer support their design and compatibility with the reference being maintained, an internet community of users would be able to continue to support it. This concept has been proven to be profitable and

* Corresponding author
Article's DOI: 10.5455/jjee.204-1594032108

sustainable in large software efforts such as IBM purchasing Red Hat Linux [4]. In this context and to prove this concept, our research team designed and built a reference design for a real-time IoT connected garage control system, the GarageRTC.

The GarageRTC, shown in Fig. 1, is an automated system for a consumer garage connected as an IoT device. The system makes the status of the garage available to the user connected remotely through a web-based interface or locally via a display and control panel. Inside the garage, the system is connected to the garage door opener, a garage light, and an alarm. The system uses sensors to detect the door position, concentration of carbon monoxide, temperature, and objects in the path of the door. The user can check the status or manipulate the controls from the control panel or the web interface.

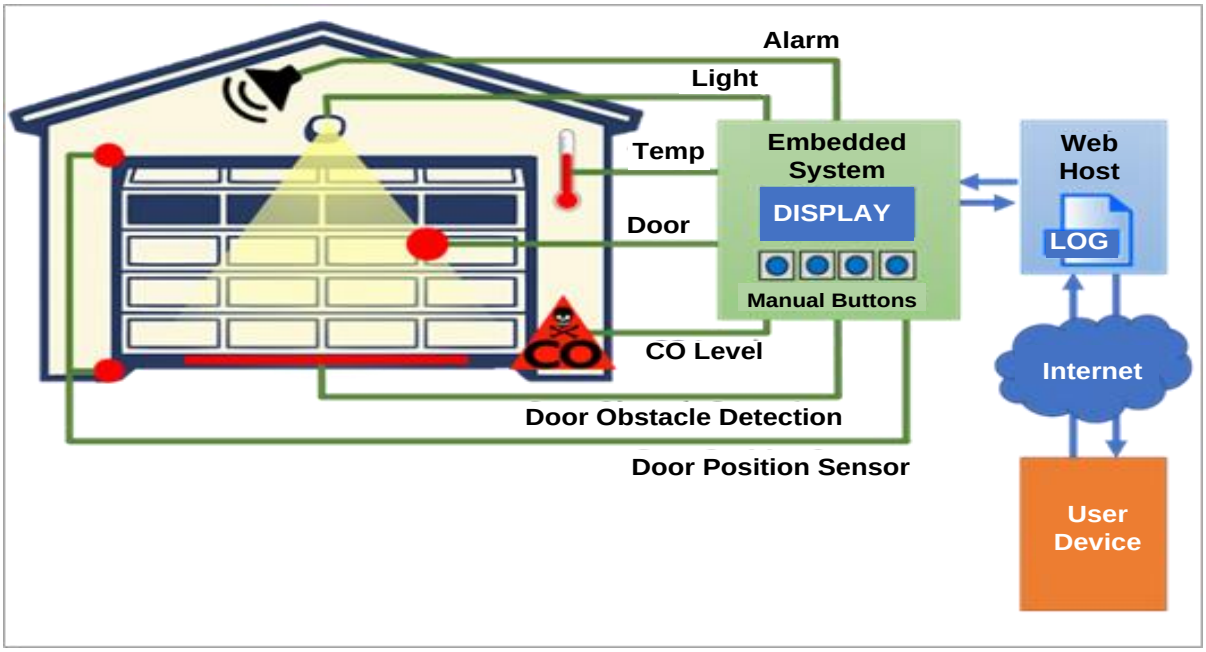

Fig. 1. GarageRTC concept.

This paper documents the design process, development, implementation, and testing of a Wi-Fi connected garage monitoring system. It implements a basic JavaScript object notation (JSON) based API and includes a reference web application that can easily be hosted in a personal cloud or on a Linux based development board.

1.1. Organization of this Paper

This paper is organized into nine parts:

- Introduction – introduces the motivations and organization of this project
- Objectives and functional description – outlines the high-level objectives for the project and reviews the design process being employed
- Requirements – reviews the high-level requirements that GarageRTC is designed to meet
- High-level design – describes the process used to decompose and partition the system as well as the generation of high-level designs and test plans
- Detail design – discusses the detailed design decisions including the selection of hardware, software, and web applications

- Testing – reviews the tests and stages used to ensure continued proper functioning of the system
- Accomplishments – reviews the objectives met, limitations of the current design, possible improvements, and lessons learned
- Comparison of commercial offerings – includes a comparative discussion about similar commercially available products
- Conclusions and closing remarks.

2. OBJECTIVE AND FUNCTIONAL DESCRIPTION

The objective of the GarageRTC project is to develop an open source, real-time, embedded system that can be integrated with an existing door opener system. The GarageRTC collects data from obstruction detectors, limit switches, and carbon monoxide (CO) and temperature sensors. It maintains a connection to a basic web server.

From the control panel or web interface, the user can observe the following:

- Status of the garage door (open, moving, or closed)
- Status of the light (on or off)
- Level of CO
- Temperature in the garage

From either interface, the user can execute commands including:

- Open or Close the door
- Stop door movement
- Turn on/off the Light
- Silence the alarm.

The system monitors the environmental conditions of the garage and can sound an alarm if any of the following issues are detected:

- CO level is too high
- Low temperature/freezing risk
- High temperature/possible fire
- Movement of the door being interrupted by an object.

This system provides the user with an easy way to operate, monitor, and control the status of his garage. The scope of this effort is to create the garage interface control, status architecture, and basic web interface.

This project was developed using a waterfall software development lifecycle. Adherence to a classic waterfall model was attempted, but during implementation, it was discovered that the selected 2.31 cm OLED original display was poorly suited. It had only partially implemented drivers and the display was small and more difficult to read than expected. The authors returned to the design phase briefly to select another display that met the requirements and had better driver support. The waterfall software development lifecycle with the feedback from the design phase is illustrated in Fig. 2.

The maintenance phase is not applicable in this instance as this paper represents the final stage of this effort. The authors did make several recommendations for future improvements and the source is available in the public GitHub repository. The proposed improvements can be found in Section 7 of this paper.

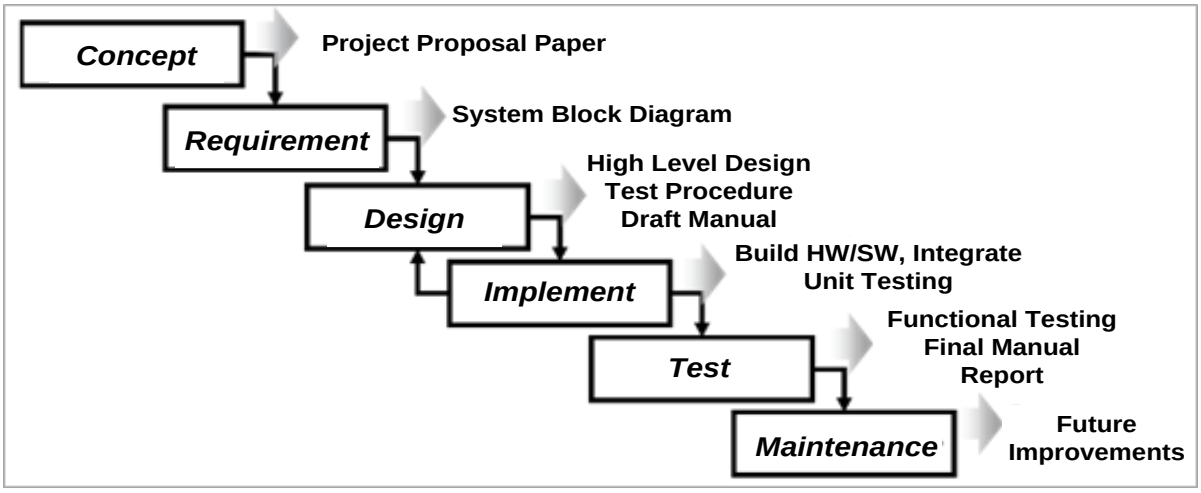

Fig. 2. Waterfall software development lifecycle.

3. REQUIREMENTS

The high-level requirements were developed as part of the initial project proposal paper during the concept portion of the design process. Functionally, the system needed to have the following interfaces:

- Electrical interface with a garage door opener
- Electrical interface with a mains voltage light
- Web interface for remote operation
- CO and temperature monitoring
- Limit switches and obstacle detection
- Local display and buttons.

Design details -within the system- were mostly left up to the design activity but were loosely defined as a conventional web server interfaced to the embedded system. Further details were provided on the specific functions of each button and the display content. The web display would mirror the local display and provide buttons to interact with the embedded system as if the user was local. Response timing goals were also provided and are summarized in Table 1.

Table 1. Response timing goals.

Parameter	Timing requirement
Local display	Every 500 ms or better
Obstacle detection	Toggle opener within 150 ms from detection
Limit switch	Toggle opener within 150 ms from detection
Door movement	Toggle opener within 300 ms from press
Light	Latch light within 300 ms from press
Idle post to server	Post to the server every 5000 ms
Event push to server	Post to the server within 500 ms from event
Check server	Check remote messages every 1000 ms

From the timing goals and requirements, a system block diagram, shown in Fig. 3, was composed to illustrate the major partitioning of the system as well as the system inputs and outputs. The block diagram was annotated to show the type of signal (resistive, digital, voltage, user datagram protocol (UDP) data, etc.) and the response times for the outputs.

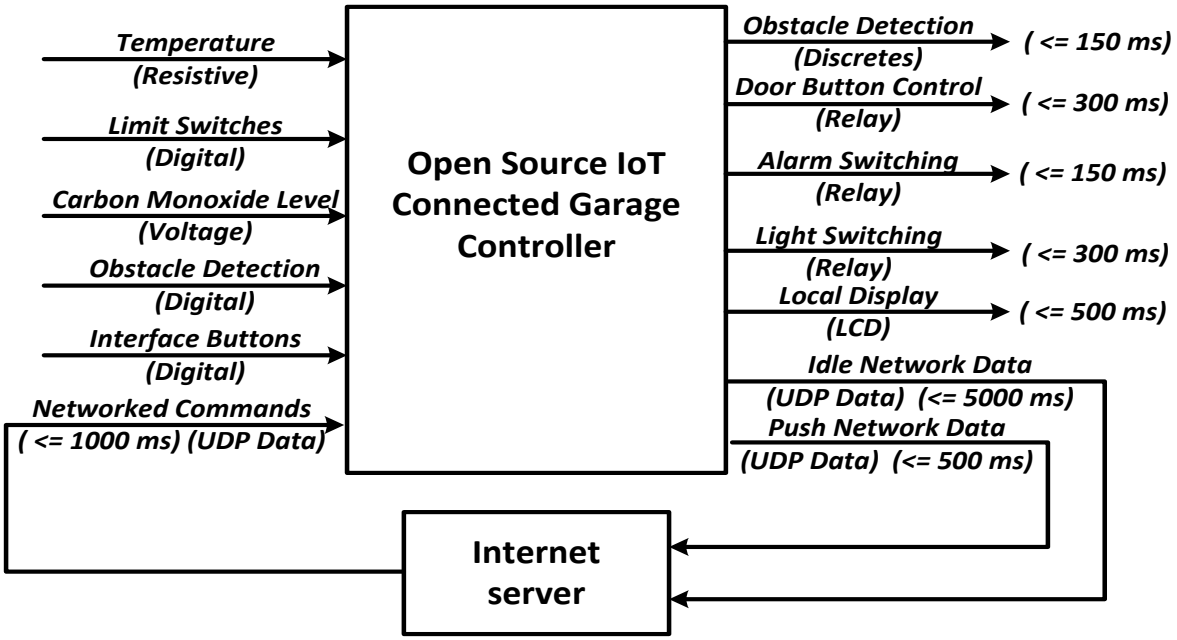

Fig. 3. Block diagram of GarageRTC system.

4. HIGH-LEVEL DESIGN

From the block diagram, the major partitioning of the system could be determined. The functions of the internet server could be developed independently from the embedded system assuming that good interface constraints could be established. Fig. 4 shows the decomposition of the functions and responsibilities of the entire system. This section presents the activities performed as part of the high-level design for the GarageRTC project.

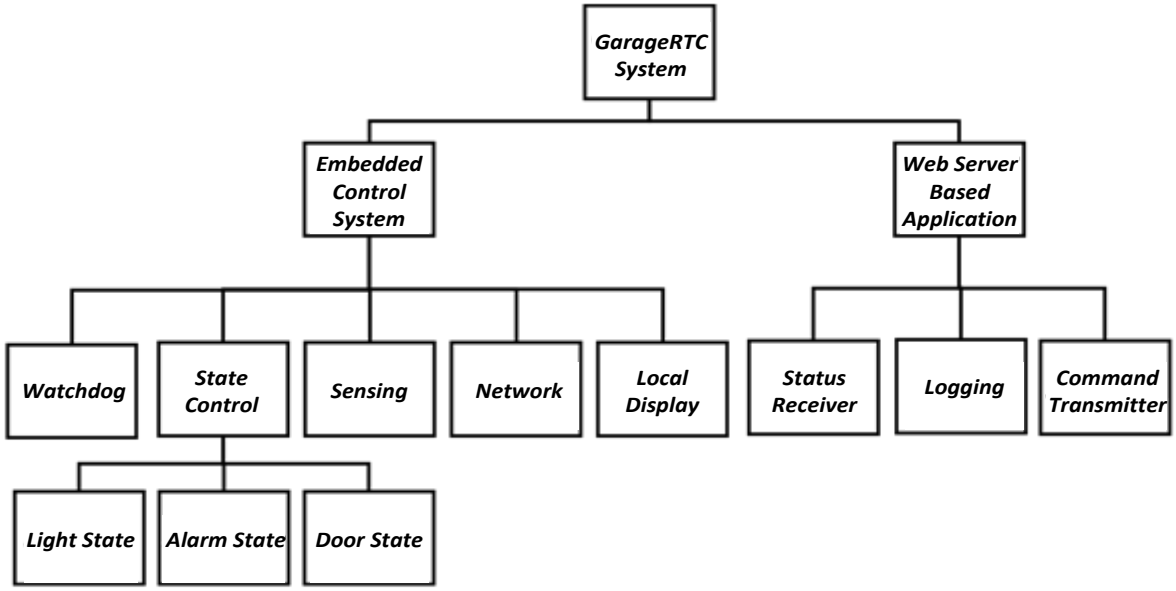

Fig. 4. GarageRTC system organization.

4.1. Interface Specification

To enable independent development of the embedded system and web service, a simple JSON based messaging interface was developed. Three different packets were

identified: embedded status, status request, and send command. These commands would be sent over UDP and encoded in a plaintext JSON format. Examples for the EmbeddedStatus, StatusRequest, and SendCommand are shown in Figs. 5-7.

{
{"name": "alarmStatus", "value": "False"},
{"name": "doorStatus", "value": "2"},
{"name": "lightStatus", "value": "OFF"},
{"name": "tempStatus", "value": " 63.4"},
{"name": "coStatus", "value": "LOW"}
}

Where doorStatus value equals:

0x0 = OPEN 0x1 = CLOSE

0x2 = STOP 0x3 = MOVE

Fig. 5. EmbeddedStatus packet.

{"cmd": "getStatus", "arg": ""}

Fig. 6. StatusRequest Packet.

The SendCommand and EmbeddedStatus packets used a bitwise encoding of the command value for enumerating the door position and the command type. This simplified the design on the embedded side and required less string parsing using the limited resources of the microcontroller.

{CMD:<value>}

Where <value> equals:

0x1 – Door Command

0x2 – Light Command

0x4 – Alarm Command

Fig. 7. SendCommand Packet.

The downside of this formatting is that it required the embedded system to compose strings to send and receive from the server. The embedded system turned out to have ample resources, both memory and speed, to accomplish the creation and processing of these messages. The benefits of using this format included the easy consumption of the JSON format by the web server and the reduced chance of interpretation error.

4.2. Embedded System

For the embedded system, a real-time operating system (OS) was necessary to provide predictable responses to inputs while simultaneously executing multiple tasks in a timely fashion. While development of a custom real-time operating system (RTOS) was considered, ultimately the FreeRTOS project was selected. FreeRTOS implements a real-time kernel and schedulers targeted as low resource microcontrollers. The project is professionally developed and is available for free use in open source and commercial embedded systems. It supports wide variety of hardware and offers excellent documentation and, most importantly, is freely available including its source [5].

The Espressif ESP series of embedded microcontrollers was selected as the primary controller for the embedded system. This controller has rich library support and, most

importantly, an integrated Wi-Fi transceiver. This would enable the embedded system to be built physically separate from the web server. The embedded application would be built in C/C++ as this is the language that the FreeRTOS and Espressif libraries are developed to support [6]. The Arduino integrated development environment (IDE) was also selected as it offers integration with the ESP platform, supports linking with FreeRTOS, and provides a variety of development support tools.

4.3. Web Application

The web application is needed to implement a method to process the packets to and from the embedded system and post them to a web page. While conceptually this sounds very simple, the team ended up merging several different projects to achieve the desired result. The team did not want a page that was continually being refreshed and made the decision that the page would be asynchronously updated. The web interface should be updated with the current status without forcing the browser to go through refresh cycles. In the background, a combination of JavaScript and HTML would fetch and update the components of the page dynamically.

The web subsystem would need to present a page in HTML format, process interactions from the user and packets sent via UDP from the embedded system, and convert user clicks into commands to the GarageRTC. The simplest toolchain to achieve this function would be a combination of a web server for hosting, cascading style sheets (CSS), and JavaScript for dynamically updating. Python would act as a go-between, consuming and converting data, storing the data structures and invoking behaviors such as UDP communication.

Since the web application was not the primary focus of this effort, several decisions were made to simplify the design. First, instead of a complete web server stack, Flask was selected to implement the Python, JavaScript, and HTML frontend. Flask is a microframework that is useful for rapid prototyping of Python-based transactions [7]. It dynamically creates web content based on Python models and incorporates an integrated web server.

The second design decision was to use web sockets for handling the asynchronous behavior of the web page. Web sockets enable client-server interactive behavior built on JavaScript [8]. SocketIO is a web socket framework that integrates well with Flask and JavaScript and enabled the resulting page to be dynamically updated with the system status without constant refreshing.

4.4. Documentation

During the design phase, two documents were initiated: the initial draft of the product manual, and the test plan. The draft manual helped to outline how the team expected the user to interact with the GarageRTC. The manual was then iteratively updated through the design, implementation, and testing phases. The final manual is available on the project website.

The test plan was a simple matrix that listed the various stages of testing to be performed and mapped them to the established requirements. The test plan is summarized

in Tables 2 and 3. Testing was broken into three parts: embedded, web, and complete system. The test level (L) indicates U, I, F, or A which are defined as follows:

- U – Unit level testing
- I – Integration testing
- F – Functional testing
- A – Acceptance testing.

Table 2. Embedded system test plan.

#	L	Description	Tests function
1E	U	Board functional	Basic I/O, IDE, toolchain
2E	U	Display	LCD Driver, text, display task
3E	U	A/D sensing	AD conversion, scaling
4E	U	CO	CO sensor, calibration
5E	U	Temperature	Temperature, calibration
6E	U	Relay	Relay module, I/O driver
7E	U	Limit switches	Switch input
8E	U	Buttons	Button input
9E	U	Network	Associate, receive, send UDP
10E	U	Alarm logic	State machine for alarm control
11E	U	Light logic	State machine for light control
12E	U	Door logic	State machine for door control
13E	U	FreeRTOS	Scheduler, library compatibility
14E	U	Shared memory	Globals, mutexes, semaphores
15E	I	Network commands	Send/Receive commands
16E	I	Display	Integrate all display elements
17E	F	Benchtop simulation	System performance on bench
18E	F	Functional	System performance installed
19E	A	Timing	System response time

Table 3. Web interface test plan.

#	L	Description	Tests function
1W	U	Flask	Flask framework
2W	U	SocketIO	SocketIO interactivity
3W	U	Database	Save/Retrieve data
4W	U	Network	Receive/Send UDP
5W	I	Settings page	Settings functions
6W	I	Status page	Status page
7W	I	Network commands	Send/Receive commands
8W	I	Embedded	Embedded send/receive
9W	F	Functional	System performance installed

5. DETAILED DESIGN

5.1. Traffic Simulators

Python scripts were written to simulate the traffic between the embedded system and the web interface. They simulated the messaging so that both subsystems could be built independently. The scripts enabled unit testing of the network sending and receiving with only one subsystem available. This enabled both the web and the embedded system to be developed in decoupled timelines.

5.2. Embedded System Software

The embedded system was divided into five periodic, concurrent, prioritized (P) tasks as shown in Table 4. The FreeRTOS uses a priority-based preemptive scheduler. Scheduling frequency was selected based on the timing requirements identified earlier.

Table 4. Tasks, priority, and frequency (3 is the highest).

#	Task	Priority	Frequency [Hz]
1	TaskReadSensors	3	100
2	TaskUpdateDisplay	1	2
3	TaskPriorityMachines	2	10
4	TaskNetwork	1	2
5	TaskWatchdog	1	0.67

When registered with the scheduler, a task control block and an individual stack are allocated. The scheduler handles context switching and pulling tasks in and out of execution. The developer is responsible for handling access to shared resources using mutexes or queues. The GarageRTC employs three global mutexes for shared resources:

- *g_sharedMemMutex* – protects shared memory and global variables.
- *g_serialMutex* – protects access to the serial port.
- *g_wdMutex* – protects access to the watchdog bowl.

5.2.1. Task Read Sensors

This task is responsible for fetching, converting values from the sensors and making the results available to other tasks. The task runs every 10 ms and has the highest priority in the system. It buffers data as it is being collected and then writes it back, protected by the *g_sharedMemMutex*. In general, this task implements the following functions:

- Fetching values from sensors
- Scaling/converting values
- Reading/debouncing switches
- Storing values to memory
- Updating shared variables.

5.2.2. Task Update Display

This task is responsible for updating the local display. The task runs every 500 ms and has the lowest non-idle priority in the system. This task performs the following functions:

- Fetching values from memory
- Posting readings or status to the display
- Calculating the door position and system state for the web
- Updating the displays, which includes: temperature, CO level, connection status and system state.

5.2.3. Task Priority State Machines

This task is responsible for monitoring door position, obstacle detection, and starting or stopping door movement. Light control and alarm state were also added into this task.

The functions of each state machine are described in details below. In general, this task performs the following functions:

- Maintaining door state
- Starting/Stopping door movement
- Monitoring obstacle detection
- Operating the light state machine
- Operating the alarm state machine.

The light control state machine ensures proper latching of the associated relay. The state machine takes button press state as an input. It includes dwell states, where the user is holding down the button. Using the state machine shown in Fig. 8, the button functions as a press-on and press-off latching output. The state machine prevents the relay from fluttering if the relay in the button is pressed and held for more time than necessary.

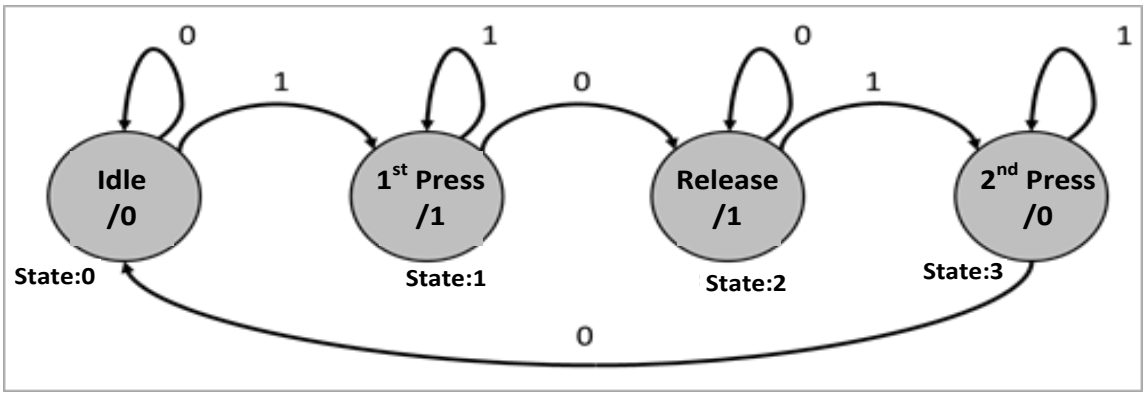

Fig. 8. Light control state machine.

The door control state machine operates the relay attached to the opener button. When the DOOR button is pressed on the GarageRTC or door web command, the state machine triggers the door switch for a period. Then it idles as the door moves. If the door strikes a limit switch or obstacle, another pulse is sent to the switch to stop the movement. If the user presses the stop button or a stop web command is sent, the machine moves to the stopping state. The state machine is shown in Fig. 9.

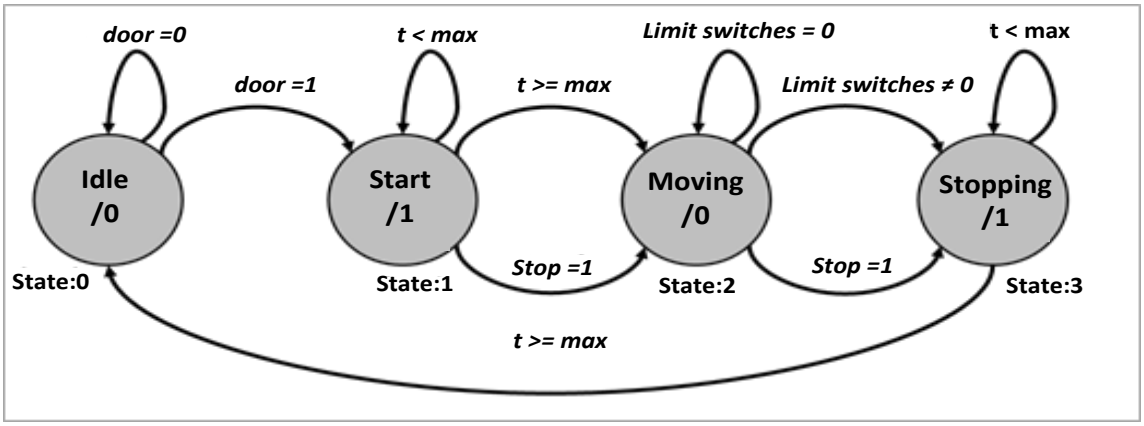

Fig. 9. Door control state machine.

The alarm control state machine monitors the temperature and CO sensors against predetermined set points. When the limits are exceeded, the state machine sets the alarm

relay. If the user presses the alarm button or alarm web command is received, the alarm is silenced by opening the relay. The alarm state will persist until the value falls into the acceptable range with hysteresis. The state machine for the alarm control is shown in Fig. 10.

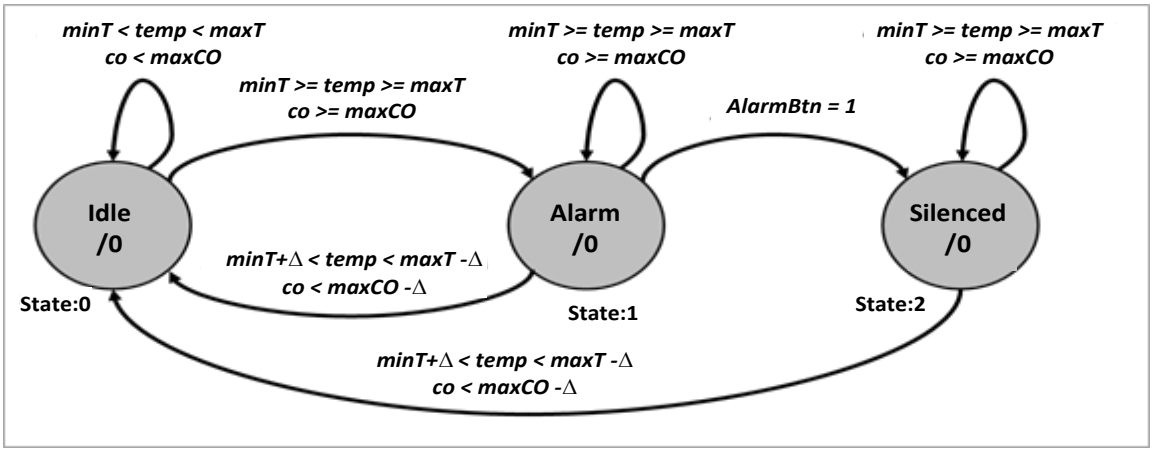

Fig. 10. Alarm control state machine.

5.2.4. Task Network

The network needs to update the task that is scheduled to run based on the time that was allocated to it.

5.2.5. Task Watchdog

The watchdog needs to monitor all tasks and ensures that they are periodically operating. This is achieved by creating a bit-wise "bowl" where each of the tasks is required to set a flag at completion. An interrupt is registered on a timer set to 5 s. In the interrupt service routine (ISR), a system reset is triggered. Periodically, the watchdog task will check the bowl. If all bits are set, then the watchdog resets the timer and empties the bowl. If the bowl is never completely full, such as when one task is starving or stuck, then the watchdog never resets the timer, as seen in Fig. 11. Eventually, the timer exhausts and the system is reset, hopefully clearing the fault condition.

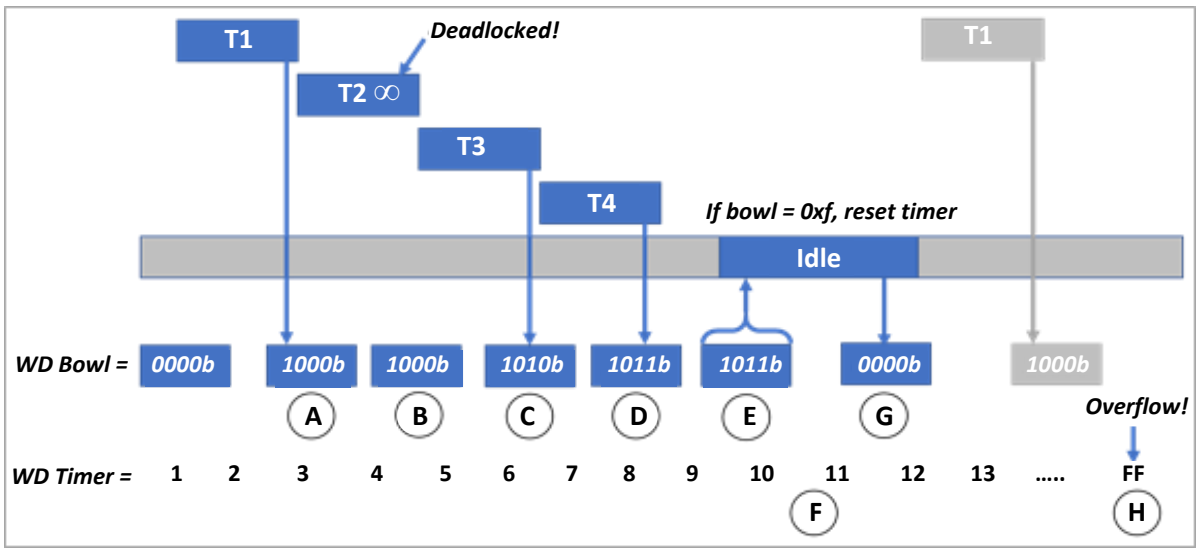

Fig. 11. Watchdog operation.

A limitation with this design is that the watchdog may not notice, if a task is missing its deadlines due to running slowly. As long as the task periodically puts a bit in the bowl and so do all the other tasks within the TaskWatchdog cycle, then the timer will be reset. For time-critical tasks, a watchdog with tighter control on timing should be set up and mark that the system has failed.

5.3. Embedded System Hardware

The proof of concept hardware was built and assembled using off-the-shelf modular components, breadboard, and a backing board. The components were sourced and assembled using discrete wiring. The schematics for the completed design are available on the project website. The complete bill of materials is listed in Table 5. The following subsections describe the notable components including the microcontroller, CO detector, and display.

Table 5. Bill of materials.

REF	Part	PN	Quantity
U1	ESP32_NodeMCU	B07F877YZQ	1
U2	20x4 character LCD Display	030003LA	1
D1	Laser diode assy	1172	1
D2	Laser detector	B01M8PFZRC	1
S1-S5	Mom PCB switch	B06XT3FLVM	4
S6-S8	Limit switch	V-153-1C25	2
K1	4x Relay assy	4450182	1
J1	USB cable	7T9MV4	1
CO	CO detector	FTC010-MQ-7	1
R1	10k Thermistor	MF52-103	1
R2	10k ohm ±1% 1/4W	MFP-25BRD52-10K	1
-	Breadboard	B01EV6LJ7G	1

5.3.1. Microcontroller

The microcontroller selected is an Espressif ESP32 system-on-a-chip micro-controller that utilizes the Tensilica Xtensa microprocessor. The specific development board selected is a NodeMCU-32S, shown if Fig. 12, which includes a USB to serial interface for re-programming and serial monitoring. The ESP32 is supported by the FreeRTOS project and a board support package is available for the Arduino IDE.

The ESP32's features include:

- Dual-core 32-bit processor
- Ultra-low power co-processor
- 25 GPIO, 6 analog input, and 2 analog output
- 3 UART, 3 I2C, and 2 SPI
- Cryptographic and security ASICS
- 4 MB of flash, with 520 KB of SRAM.
- Built-in 802.11n Wi-Fi and bluetooth LE
- Software, hardware, and timer interrupts.

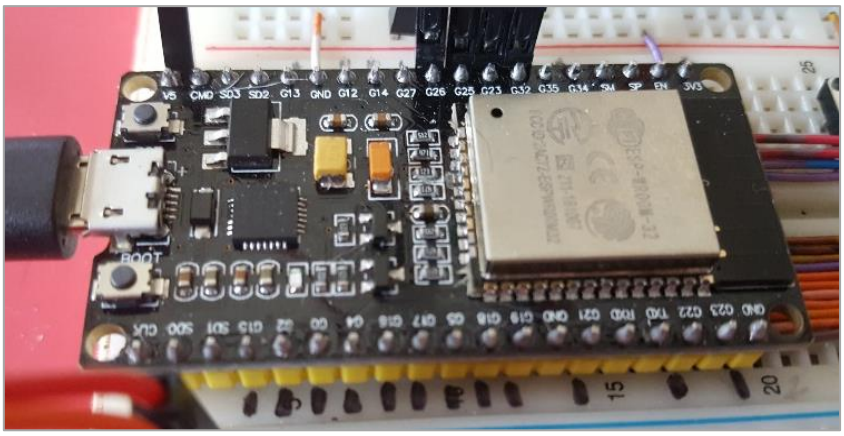

Fig. 12. NodeMCU-32S compatible ESP32 module.

5.3.2. CO Detector

For CO detection, MQ-7 CO detector integrated onto a development module that includes a comparator pre-amp was used (see Fig. 13). The module has an adjustable digital alarm output (not used) and a scaled analog output reporting parts per million of CO.

Fig. 13. MQ-7 carbon monoxide detector.

The challenge with the CO detector was calibration and susceptibility to temperature and moisture variation. The research team did not have a calibrated source for correcting the CO concentrations and the sensor was found to be sensitive to fluctuations in temperature and humidity. The sensor was ultimately roughly calibrated by using ambient in-house CO readings as "LOW" and then "HIGH" on the tailpipe of a cold start automobile. The "WARN" threshold was set at approximately 20% of the observed full range and then adjusted to not trigger any warnings when operating in the home.

5.3.3. Display

The display was initially selected to be a 2.31 cm 128x32 SPI OLED monochrome graphics display. There were several sources of drivers available for this type of display for the ESP toolchain. However, when the team attempted to integrate the drivers, they found that they were only partially functioning. After attempting to integrate the display, manually updating drivers, and struggling through poor documentation, the team decided to return to the high-level design phase and select a different display.

A simpler way to integrate a display was using a 20x4 character LCD display as shown in Fig. 14. This display is a parallel matrix interfaced through a PCF8574 chip. The PCF8574 is an 8-bit I/O expander for the I²C bus. Drivers already existed for communicating with this display over I²C and the display was quickly integrated. However, the display does have somewhat poor contrast and is slow to update.

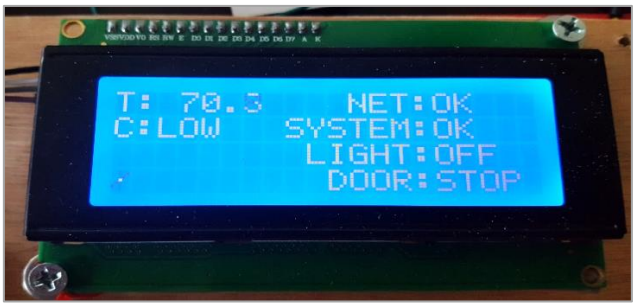

Fig. 14. 20x4 character LCD SPI display.

5.3.4. Obstacle Detection

For obstacle detection, the team considered interfacing with the break beam eyes supplied with conventional openers. The research revealed that these sensors typically emit an AC waveform that the opener looks at to ensure the sensor has not been bypassed. Ultimately, the team selected a laser module and a laser detector diode pair as shown in Figs. 15 and 16.

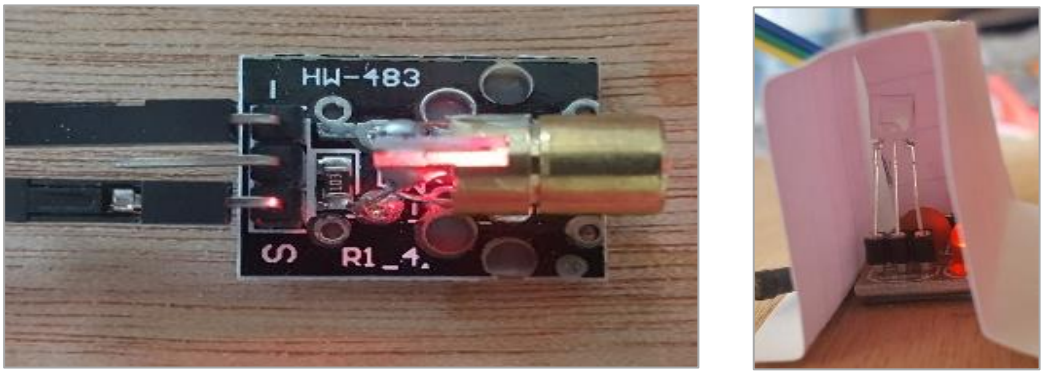

Fig. 15. Laser transmitter and diode detector.

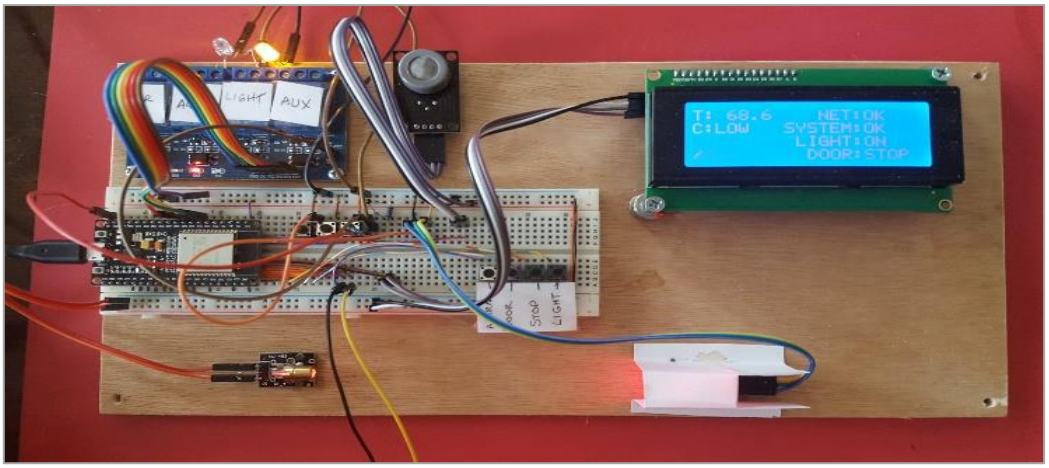

Fig. 16. Laser transmitter and diode detector interfaced with the embedded system.

5.3.5. Completed Proof of Concept GarageRTC Assembly

The fully integrated assembly was attached to a wooden mount to provide rigidity and portability. Mounting holes were added to allow the assembly to be fixed to the wall for testing. The system used a standard 4 gang relay to provide mains switching and opener button control. Power was provided by means of an AC-DC switching power supply with a USB adapter or from the laptop.

5.4. Web Service

The web service was constructed with three main pages:

- Menu page for navigating
- Status page for monitoring/commanding
- Settings page for configuring the system.

The menu page, shown in Fig. 17, allows the user to navigate between the landing page (Home), status, and settings pages. It is accessed through the settings button in the upper right corner of the display.

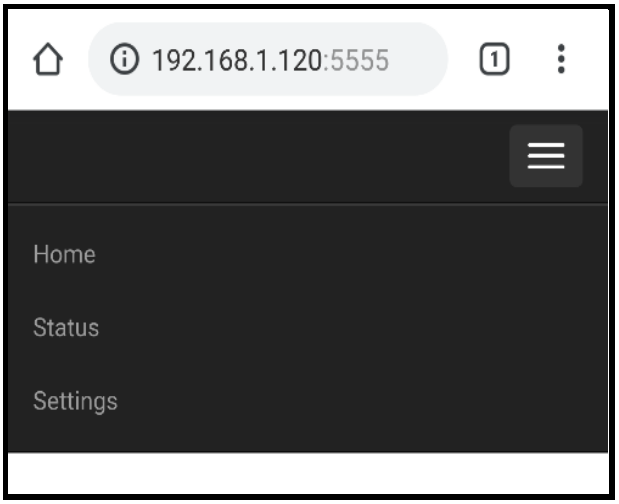

Fig. 17. Menu page.

For the status page, the web service used Python to capture and process JSON packages from the network interface. These were then stored to dictionaries within the Flask framework. JavaScript would then periodically read the data structure and update the user interface status page. An interval timer and button clicks were processed by the JavaScript and fed to the Python data structure via SocketIO and Flask. The status variables stored on the server were buffered to ensure the data was not overwritten during processing. When the event was processed, the Python code would create either a command or a "get status" packet and write it to the network adapter to be sent to the embedded system. A data flow graph of the web server subsystem is shown in Fig. 18, illustrating the process. The interactive status page is shown in Fig. 19.

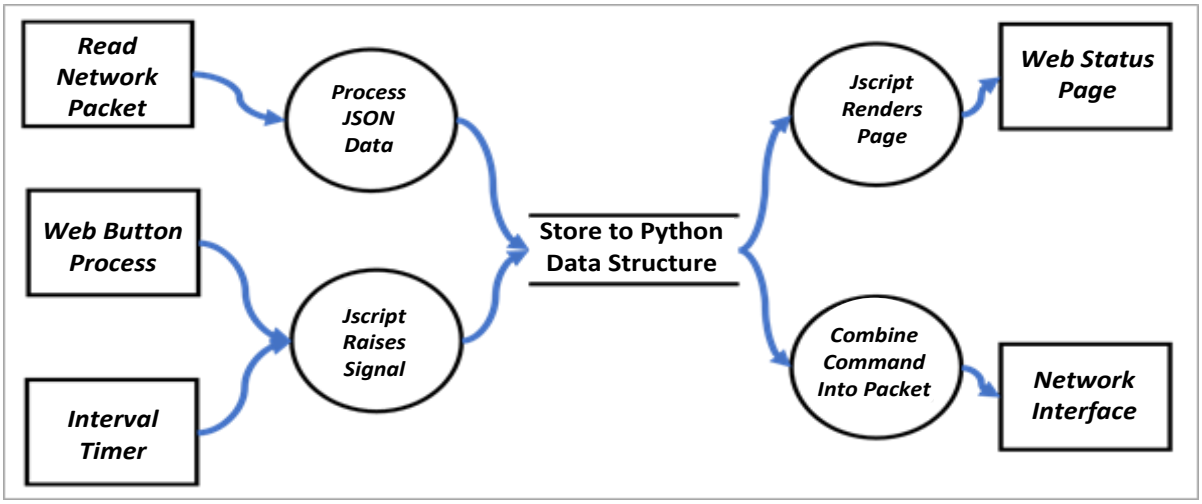

Fig. 18. Data flow graph of the web server subsystem.

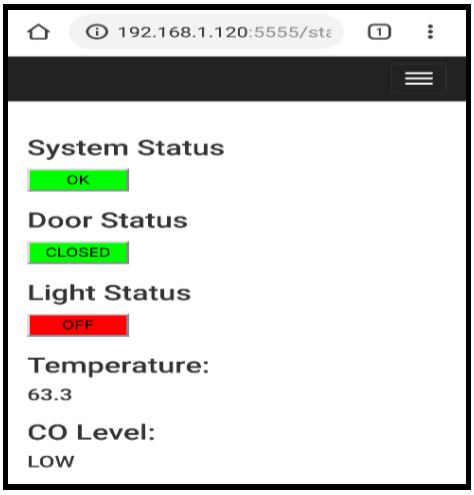

Fig. 19. Status page.

The settings page, shown in Fig. 20, allows the user to configure the target GarageRTC IP address. The user can configure the IP and Port as well and adjust the screen cycle time. This information is then written back to the database and used for sending/receiving to the target GarageRTC.

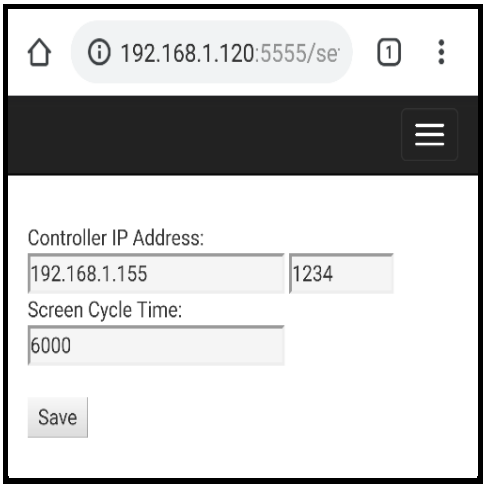

Fig. 20. Settings page.

5.5. Integration of Web and Embedded System

Finally, the web and embedded sections were brought together for combined testing. The team encountered minor issues as each team had made assumptions about how the other subsystem would operate. Some final integration modifications were made and the web interface was used to call the embedded functions on the bench. The use of the interface defined early in the design phase and simulators developed to support testing had enabled quick and successful integration between the two subsystems.

5.6. Unit Testing and Code Refinement

Testing consisted primarily of unit testing during development. Unit tests were used to verify each component as it was incrementally added to the system. Eventually, as the system grew more complex and stable as features were added. Finally, the system could be tested as an assembly. Bench tests of the state machines and outputs were performed.

After the system was mostly integrated and functioning, a code baseline was established, and several rounds of performance tuning were performed. This consisted of moving variables out of the global space, reducing loop complexity, and minimizing memory usage. The remaining global variables were guarded by mutexes and local buffers established to ensure consistent operation. Several passes were made to remove dead code, simplify the math, and reduce primitive data types, such as changing integers to bytes. Rather than dealing programmatically with strings, static char arrays were constructed to allow indexing into static arrays using integer enumerations. Debug statements and the inaccessible code were removed.

6. TESTING

Functional testing consisted of walking through all the major functions on the device. First on the bench, then the unit was installed on a working garage door (see Fig. 21). Several minor software bugs were identified, and the system timings were adjusted. A video of the major features of the GarageRTC was produced and uploaded to YouTube [9].

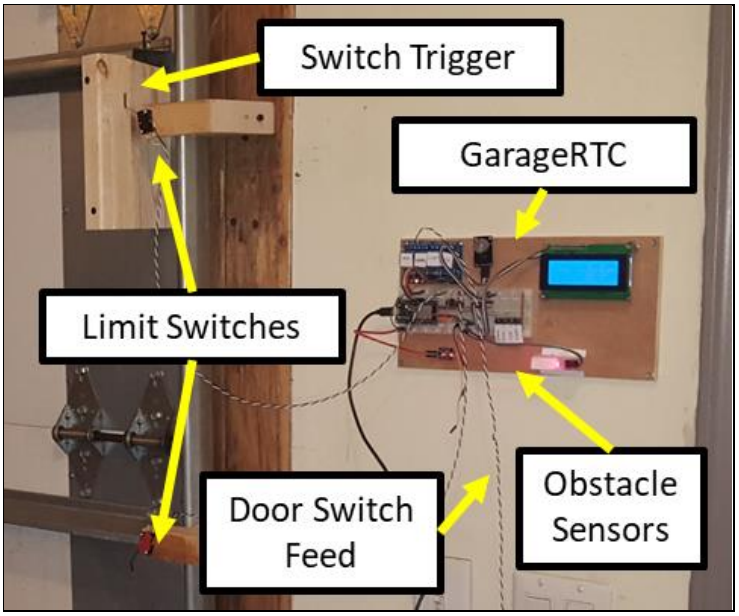

Fig. 21. Functional testing installation.

A model of the completed system with all submodules integrated into a single assembly was created, as shown in Fig. 22. This model demonstrates that the packaging could be substantially reduced, and a neater, more professional-looking package could be produced.

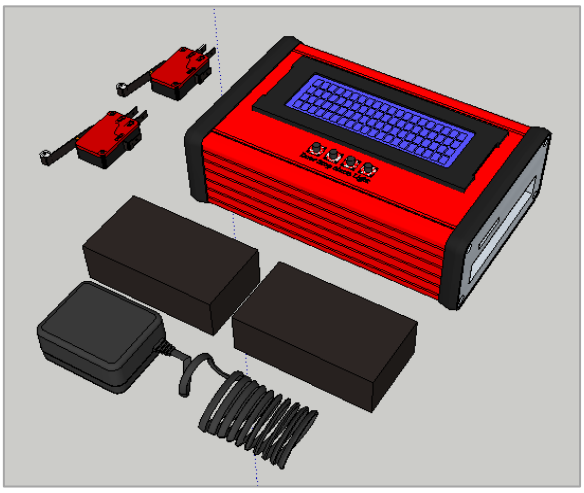

Fig. 22. Model of integrated RTC assembly.

6.1. Performance

Performance of the GarageRTC was measured in terms of utilization, memory usage, and measured response timing with respect to the original requirements.

6.1.1. Utilization

Using an oscilloscope connected to the debug pins, the worst-case timing (e_i) of each task was captured. The frequency of scheduling (F_{Meas}) was also captured and thus, the execution period (p_i) of each task is calculated. Using this information, the system utilization (U) was computed by summing the utilization factors (u_i) to be approximately 8.4%. This is considered extremely underutilized. However, while the ESP32 may be overpowered for the scheduled tasks, the integrated Wi-Fi is a big benefit. Table 6 shows utilization calculation.

Table 6. Utilization calculation.

Task	F_{Meas} [Hz]	e_i [ms]	p_i	$u_i = e_i/p_i$
TaskReadSensors	100	29.5 E-03	0.01	0.00295
TaskUpdateDisplay	2	39	0.5	0.078
TaskPriorityMachines	10	1.41 E-03	0.1	1.41E-05
TaskNetwork	2	2.015	0.5	0.00403
TaskWatchdog	Varies	1.05 E-03	0.67	1.58E-06
				U = 0.084

For memory utilization, the current application uses approximately 720 kB bytes or 54% of program memory. Without Networking and UDP libraries, the application consumed only approximately 15%. This means the network stack and UDP messaging consumes about 524 kB or almost 40% of program memory. Global variables use 38kB (11%) of dynamic memory leaving 290 kB for local variables which includes individual task stacks.

6.1.2. Timing

For measuring system timing, the oscilloscope was attached to the input line from the switch and the second channel was attached to the output signal. The average over 10 presses was measured on the door button, stop button, light button, and the obstacle sensor.

An example of capture between the pressing of the button and the reaction of the relay closing is shown in Fig. 23.

Fig. 23. Button press (Top) and door relay (Bottom).

The results were that switching occurred as quickly as 40 ms and never slower than 86.2 ms. The average response was around 58.6 ms. The worst-case switching was still far better than the strictest requirement of 150 ms. The timing for the LCD and network data was based on cycle time for the task using a debug pin toggling at the start and end of the task. The resulting timings compared to their respective requirements are shown in Table 7 in which P/F denotes pass/fail.

Table 7. Timing requirements and system measurement.

Parameter	Timing requirement	Measurement [ms]	P/F
Local display	Every 500 ms or better	500	P
Obstacle detection	Toggle opener within 150 ms from detection	< 86.2	P
Limit switch	Toggle opener within 150 ms from detection	< 86.2	P
Door movement	Toggle opener within 300 ms from press	< 86.2	P
Light	Latch light within 300 ms from press	< 86.2	P
Idle post to server	Post to the server every 5000 ms	500	P
Event push to server	Post to the server within 500 ms from event.	<= 500	P
Check server	Check remote messages every 1000 ms	500	P

7. ACCOMPLISHMENTS AND LIMITATIONS

7.1. Objectives Met

After reviewing the requirements as defined in the original project proposal, all objectives that were established at the outset of this project were met. Table 8 contains the completed requirements matrix.

Table 8. GarageRTC requirement compliance.

Requirement	Description	Met?
REQ-001	Manual interface switches	Yes
REQ-002	Local display	Yes
REQ-003	Light control	Yes
REQ-004	Sensing requirements	Yes
REQ-005	IoT connectivity	Yes
REQ-006	Web service	Yes
REQ-007	Timing requirements	Yes

7.2. Limitations

The system does have a couple of minor limitations. Firstly, the alarm push notifications to the server were combined into the same task as the regular network update task. This means that the best possible response time will be governed by the update time of TaskNetwork. Furthermore, there is a stacked delay between the reception of the UDP packet and publishing to the page. This seems to work well in practice, but for more responsive alerts, more tuning should be put into an asynchronous push from the embedded system and fine-tuning the cycle time of the web server.

Secondly, the CO detector was not able to be accurately calibrated. The sensitivity to moisture and temperature means that the calculation for the CO is a function of temperature and humidity. While temperature is available, the humidity was not. Temperature measurement was accomplished by the use of a glass bead thermistor on the regulated 3.3V DC rail. While this was good for gross measurement of temperature, if incorporated into the CO calculation, it could make the CO output unreliable. The gross calibration of the CO sensor seems to be mostly sufficient but occasionally will trigger CO false positives when the garage temperature drops below 10 °C. This could be resolved by sourcing a more robust sensor as a replacement.

7.3. Lessons Learned

Some of the lessons learned in this project included building out simulators and well define interfaces prior to assigning tasks to developers. While the high-level interfaces defined during the early stages of this development project were helpful, if they were more rigidly defined it would have assured compatibility of the subsystems at integration.

The research team also quickly outgrew the Arduino IDE. The Arduino IDE contains several nice features, such as an integrated library manager and built-in examples, but it lacks common features like tab completion, click navigation, and Realtime debugging. Occasionally, tracking down documentation or finding specific functions within libraries were mostly hidden from the user. There are development toolchains for the ESP and a library from Espressif intended for the Eclipse IDE which is a much more feature rich development environment. Configuring this earlier on and using it throughout the entire development process would have made development easier especially as the complexity of the system increased. The Eclipse IDE also supports Python, JavaScript, and HTML providing a complete ecosystem for this project.

7.4. Possible Improvements

The project made several design decisions in the interest of simpler development. To ease network integration, the system employs UDP and no encryption. These are considered experimentation only configurations and would need to be resolved for a more production-ready product. The system also interfaces to the garage door lift motor through the operator button, this too could be improved. Finally, the system depends on the obstacle sensor being well aligned and can easily be spoofed by jumpering the signal wires. This section discusses potential improvements to solve these issues intended in a future revision of the GarageRTC project.

UDP simplifies the design by reducing the network configuration within the web app and embedded unit, but routing traffic to an area larger than the immediate subnet is impractical. The excessive packets will hinder network congestion and are not easily forwarded to another network if the server was not on the same subnet. To solve this, the system should be moved over to TCP based networking. This would enable security features such as transport layer security (TLS).

The lack of encryption certainly simplified the design by not having to create/manage keys or develop pairing processes for the server and embedded unit. Obviously, in this day and age, any IoT device is under heavy scrutiny from security researchers. With the current design, anyone on the network could easily replay a packet and get the garage door to move. The solution would involve adding application layer encryption between the embedded controller and server. For further protections, TLS could also be added by embedding certificates that are mutually authenticated.

Another limitation is that the GarageRTC interacts with the garage door lift motor through the opener button interface. Each garage door operates a little differently, such as having longer button pulses or automatic reversing after a down trigger. This makes the interaction between the GarageRTC and the opener not ideal. A potential solution would be to incorporate a direct-drive reversible H-Bridge controller into the project. That would give the controller better control over the door. The downside is there are significant safety impacts that need to be considered when taking direct control of the door lift mechanism.

Lastly, the laser transmitter and receiver used by the project could be improved by replacing it with a pair of conventional garage break beam eyes. These devices typically use a proprietary protocol but do incorporate anti-spoof circuitry, so the door controller could tell if they have been intentionally modified and are more tolerant of misalignment. Replacing the laser with libraries to interface with different conventional eyes would reduce system cost, improve safety, reliability, and performance.

8. COMPARISON OF COMMERCIAL OFFERINGS

Several commercial IoT garage devices are available to consumers. These devices are typically developed by major opener manufacturers and marketed as an upgrade to the existing opener. This section provides a short review of several of these offerings and their compatible hardware. Surprisingly, none of these technologies offered native integration with digital assistants such as Amazon Alexa. We provide a comparison between the GarageRTC features and the commercial offerings and discuss the benefits and drawbacks of these paired solutions.

8.1. Commercial IoT Garage Interfaces

The market was surveyed for commercial IoT garage interfaces that offer similar capabilities as to those implemented for the GarageRTC project. The two most prominent opener companies, Chamberlain and Genie, offered hardware designed to interface with their respective openers. There was also a couple of notable general-purpose add-on modules designed to interface with a variety of openers. The technologies surveyed were as follows:

- Chamberlain MYQ-G0301
- Aladdin Connect
- Garage Door Buddy
- GoControl GD00z-4

8.2. Comparison with GarageRTC Features

Next, the team collected documentation publicly available from the device's respective marking pages. From this, the research team composed a matrix considering connectivity, digital assistant integration, sensing and control, local interface, and openness of design. The compiled data is summarized in Table 9, where: 1 indicates that ESP32 contains the hardware but reference design does not implement; 2 is an indication that libraries exist that could integrate this capability [14]; 3 indicates that it can be integrated through webhook/REST API calls [15] and 4indicates that it requires purchase of additional hardware.

Table 9. Comparison of commercial devices.

Feature		MYQ-G301 [10]	Connect [11]	Door buddy [12]	GoControl [13]	GarageRTC
Connectivity	WiFi	✔	✔	✔		✔
	Bluetooth					1
	ZWave				✔	
Digital assistant	Alexa					2
	Google					3
	Siri					3
Sense/Control	Aux control	4				✔
	Amb. temp					✔
	Amb. CO					✔
	Door position	✔	✔	✔	✔	✔
Local control	LCD display					✔
	Door	✔	✔	✔	✔	✔
	Aux/Light					✔
Open source	Web App					✔
	FOSS					✔
	Published API					✔
	HW schematic					✔
	Universal			✔	✔	✔

8.3. How GarageRTC Differs from Current Solutions

The GarageRTC is comparable to commercial offerings in all categories and stands out in terms of design openness. The open source nature of the GarageRTC makes it considerably more adaptable and extensible when compared to the commercial offerings.

It is important to note that security was omitted from the table. The team found that while the manufacturers advertised various "security assurances", they did not publish details on how these were achieved. The GarageRTC does not currently employ any sort of link encryption and depends completely on the security of the wireless access point. The team expected the security category to be the location where the commercial offerings would outshine the project. However, since the details of their security mechanisms could not be determined, it is difficult to say if the commercial systems are offering more than security through obscurity. The research work done by Margulies et al. indicates verifying levels of security effectiveness [16, 17]. For these reasons, the security offerings cannot be effectively compared without a more thorough evaluation. A proposal for security enhancement on the GarageRTC project can be found in section 7.4 of this paper.

The commercial offerings did have the advantage in that they tend to offer applications with credentials. All surveyed solutions offered Android and IOS native apps for their remote interface. Frequently, these applications are merely wrappers around a web interface. This simplifies development and can help with security if the site is properly protected. The location of the IoT service could not be conclusively determined from the surveyed documentation, but it is reasonable to assume that it is most likely a closed source cloud-hosted service. If the support for the cloud service is suspended, then these devices would likely fail to function. Since the GarageRTC hosts a generic web page, it's reasonable to say that the GarageRTC offers equivalent performance.

8.4. Benefits and Drawbacks of the Integrated Solution

The primary benefit of the commercially available devices is the corporate-backed development and support. This translates to more refined software and user-friendly interfaces as they have superior developer resources as compared to the GarageRTC research team. They have the capacity to fully implement and test their solutions and maintain professional assistance hotlines. Furthermore, greater availability of mass production of their assemblies' results with prices typically ranging from 50$ to 100$. However, if the popularity of GarageRTC caught on, the resulting user community could quickly make up this gap. Further development could produce layouts to meet off the shelf housings or 3D printed enclosures.

The major drawbacks of the commercial units are their closed source nature. Vulnerabilities have been identified in the garage door solutions and countless other issues may exist [18]. These units have not undergone any sort of security community analysis and publish very little documentation on security controls available. The GarageRTC does not currently employ any security as it was not the focus of this effort, but could easily integrate any flavor of security controls the developer prefers. The ESP32 supports hardware acceleration of cryptographic processes, making encryption/decryption on the microcontroller fast and lightweight. Since the source is open, the security community could

then review the implementation, identify potential flaws and make recommendations for remediation.

The closed source communication protocols also mean that the lifetime of these products is directly tied to the supplier's willingness to continue support. If the companies go out of business or cease support in favor of a newer generation part, customers will be out of luck. The GarageRTC project offers the ability for anyone to stand up or share their own web application servers and therefore can indefinitely support hardware built to meet the GarageRTC protocols.

9. CONCLUSIONS AND CLOSING REMARKS

The GarageRTC met most of the performance and design requirements as identified in the concept and requirements phase. The system was more than capable of proving responsive interactivity with a residential garage door system.

The high-level design outlined an organized approach to system development during the integration phase. The test plan outlined and iterative updates to the user's manual kept development on track and features aligned with the original intent. The ESP based hardware and Arduino IDE performed well when coupled with the FreeRTOS for the embedded system. Flask, JavaScript, and SocketIO simplified the development of the web interface and packet engine interface to the embedded system.

Detailed design produced detailed state machines prior to implementation and helped align runtime tasks with desired functional behavior. The OLED display was identified early to not meet desired performance and the 20x4 character LCD was substituted with minimal re-work. The network simulators being developed allowed the development timelines of the embedded system and the web to be decoupled. Integration - of the web server and the embedded portion of the project - was eased by the use of the simulators and the early established network command interface.

Unit testing verified each component's functionality independently allowing the research team to revalidate hardware periodically. This also ensured that both team members were aligned when implementing the hardware. Extensive bench testing ensured that only minor issues were encountered when installing the completed system in the intended environment.

The system was functionally benchmarked against other commercial offerings. The flexibility of the GarageRTC platform enables it to outperform its commercial counterparts. Although the GarageRTC did not implement security within its design, the other systems did not offer details on their security and no firm conclusions could be drawn.

This paper set out to present the features, design, and implementation of a reference architecture built on an ESP32 microcontroller and FreeRTOS software. The materials created during this effort have been posted publicly and freely available. The researchers believe this proof of concept demonstrates that open source IoT products are viable and can offer similar performance to a commercial product but without the burden of developing and maintaining dedicated closed source cloud services.

REFERENCES

[1] A. Zeichick, "Throwing our IoT investment in the trash thanks to NetGear," *Network World,* 2016. < https://www.networkworld.com/article/3093439/throwing-our-iot-investment-in-the-trash-thanks- to-netgear.html>

[2] J. Perlow, "When your IoT goes dark: why every device must be open source and multicloud," *ZDNet,* 2019. <https://www.zdnet.com/article/when-your-iot-goes-dark-why-every-device-must-be-open-source-and-multicloud/ >

[3] S. Levin, "Squeezed out: widely mocked startup Juicero is shutting down," *The Guardian*, 2017. <https://www.theguardian.com/technology/2017/sep/01/juicero-silicon-valley- shutting-down>

[4] R. Suehle, "Q&A. What is the secret of Red Hat's success?," *Technology Innovation Management Review,* 2012. < https://timreview.ca/article/513>

[5] R. Barry, "Mastering the FreeRTOS real time kernal," *Real Time Engineers Ltd*, 2016. <https:// freertos.org /Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf >

[6] H. Gochkov, I. Grokhtkov, "Arduino core for ESP32 WiFi chip," 2020. <https://github.com/espressif/arduino-esp32>

[7] A. Ronacher, "Flask a microframework for Python," *The Pallet Project*, 2020. <https://www.palletsprojects.com/p/flask/>

[8] G. Rauch, "Socket.IO," *Automattic*, 2019. <Available: https://socket.io/>

[9] D. Zajac, J. Harmer, "GarageRTC video demonstration," 2020. <https://www.youtube.com/watch?v=IwsxoqProoo>

[10] Chamberlain, "MYQ smart garage hub," *Chamberlain*, 2020. <https://www.chamberlain.com/myq-smart-garage-hub/p/MYQ-G0301-D>

[11] Genie, "Aladdin connect," *Genie*, 2020. <https://www.geniecompany.com/ aladdinconnect/default.aspx>

[12] GarageDoorBuddy, "GarageDoorBuddy - remotely control your garage doors from anywhere," *Garage Door Buddy*, 2020. <http://garagedoorbuddy.com/>

[13] GoCOntrol, "GD00Z-4: z-wave garage door opener remote controller accessory," *GoControl*, 2020. < http://www.gocontrol.com/detail.php?productId=4>

[14] X. Pérez, "FauxmoESP - Amazon Alexa support for ESP8266 and ESP32 devices," 2019.

[15] X. Pérez, "Using google assistant to control your ESP8266 devices," *Tinkerman*, 2020. <https://tinkerman.cat/using-google-assistant-control-your-esp8266-devices/>

[16] J. Margulies, "Garage door openers: an internet of things case study," *IEEE Security & Privacy,* 2015.

[17] C. Goudie, R. Weidner, "Home hackers: digital invaders a threat to your house," *ABC 7 News, Chicago IL,* 2015.

[18] J. Pawar, S. Amirthaganesh, S. ArunKumar, K. Satiesh, "Real time energy measurement using smart meter," *Online International Conference on Green Engineering and Technologies*, 2016.