Modular Home Monitoring System

Robert Short, Robert Simon, Gary Leutheuser

Dept. of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — This paper details the design and implementation of the Modular Home Monitoring System. The Modular Home Monitoring System fills the need of a low cost, easy to use, and modular wireless sensor network for home monitoring. The MHMS provides Carbon Monoxide, smoke, and humidity detection as well as a live video feed of the users home for their peace of mind. Alerts are sent to the user when anomalies are detected and the user can log into the online web interface to view current sensor readings and the image feed. At any time, the user can choose to customize the setup of sensors in their home according to their need, up to any arbitrary number. This gives the user total control over the coverage of their home.**

I. Introduction

The market has a demand for smart home and home automation products, especially health and security monitoring systems. A familiar example of a product that fulfills this need is a home security system. Consisting of a base station and a number of linked sensors, these notify the homeowner (and possibly the local police) of any activity that might indicate a break-in. However, there are also other applications. Consider a home that is frequently left unoccupied for long periods of time, such as a vacation home. If, for example, a water pipe were to burst, the owner would need to know about it, preferably before significant damage were to occur. There are many similar scenarios (such as a fire or a power outage, to name a few) which the owner would like to know about without having to resort to constant personal supervision.

If an integrated, modular system for monitoring various sensors and generating notifications based on the gathered data were available, the task of taking care of a property would be greatly simplified. Perhaps more importantly, being able to remotely verify the safety of a structure would do wonders for the owner's peace of mind. Ideally, such a system would have configurable alerts, and a real-time monitoring function, so that the user could both receive alerts when an abnormal and possibly dangerous condition is encountered (such as a fire) or simply view sensor readouts to ensure that everything is working properly (by checking the temperature to ensure that the air conditioner is functioning, for example).

There are many systems on the market which try to fulfill this need. Unfortunately, most of them fall far short of the ideal. Commercially available security systems will monitor motion detectors or door sensors and alert the police when tripped. However, many do not alert the homeowner (giving him or her an opportunity to interrupt what may be a false alarm). Most are also tied to a particular service provider, and will not function at all unless a significant monthly fee is paid. Almost none have any support for environmental sensors (humidity sensors, smoke detectors, etc.). Dedicated environmental Sensor Pods exist, but these do not support security-related sensors. At present, the best option for anyone who desires both security and environmental monitoring is to maintain and install a separate system for each. This leads to a significant increase in both complexity and cost. This is where the Modular Home Monitoring System comes in. The MHMS offers users the ability to monitor the status of their house using any configuration of sensors connected to a base station/camera module all while being affordable and not requiring a monthly subscription.

II. System Architecture

The Modular Home Monitoring System uses a beacon architecture where all sensor pods are broadcasting packets following the iBeacon protocol to the Main Control Unit (MCU). The MCU will then push the data onto the Internet and to the Cloud Application. The Sensor Pods wirelessly broadcast to our Main Control Unit (a Raspberry Pi 2) via the Bluetooth Low Energy protocol. Functionality is built into the system to allow an arbitrary number of Sensor Pod broadcasts to be received by the main control unit simultaneously, however, our prototype will only include one of each of the three types of pods. The sensor pods send status messages and monitor the environment when powered on, sending sensor readouts at regular time intervals to the MCU. The MCU then sends that data over the internet through Wi-Fi (802.11) to the cloud application, which then checks if the sensor data readout meets any danger thresholds. If necessary, the application sends the user a SMS text message alert and displays the alert on the web user interface. The camera is connected to the Raspberry Pi via the dedicated camera ribbon cable using the Camera Serial Interface (CSI). The MCU starts streaming image data from the camera when the user logs into the user interface and elects to start the stream. More

details on the design of the sensors and MCU configuration will be discussed in the following sections.

The sensor pods will consist of two major components: a common “interface board”, handling functionality common to all sensors, as well as the “sensor board”, which houses the sensor itself and specific supporting circuitry. This architecture allows for rapid addition of new sensor types and for hardware reusability.

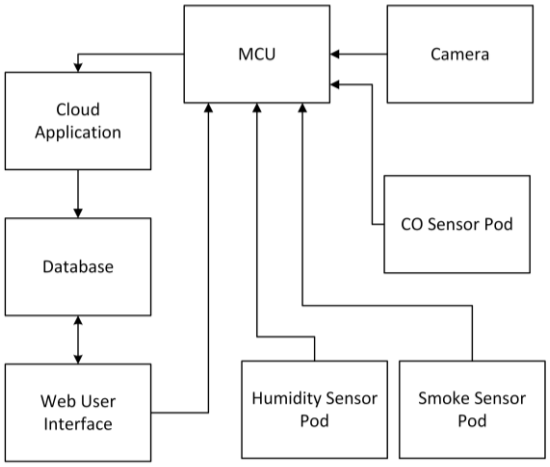

Fig. 1. System block diagram which shows the three unique sensor pods and their high level functions, as well as the camera and major functional cloud blocks.

III. Design and Implementation

A. Hardware Overview

The MHMS hardware is composed chiefly of two items: the interface board, and the sensor board, whose combination will be simply referred to hereafter as the “Sensor Pod”. This configuration is shown in Fig. 2. The interface board is unchanged between all Sensor Pods, with the only exception being the firmware programmed into its SoC (the RFduino). The sensor boards vary between Sensor Pods, with the only shared characteristic being the physical interface they utilize to connect to the interface board. The electrical signals of the interface are not the same between sensor boards.

B. Interface Board

The interface board has three functions:

1) The interface board provides power to the Sensor Pod. This is accomplished via two linear voltage regulators, which output 3.3 V and 5 V. Linear regulators are used because of the low amount of current drawn by the Sensor Pod – introducing cost and complexity in order to achieve high efficiency does not have as much benefit when the system’s power dissipation is so low.

The main source of power is an off-the-shelf, 9 V transformer that plugs into a standard North American home electrical outlet. In the event that this source should be cut off, a battery backup is also provided, with circuitry that only allows the battery’s use when the main source is lost.

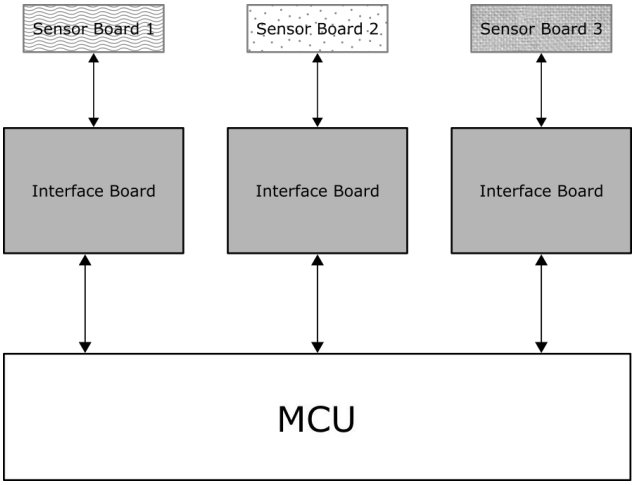

Fig. 2. Major hardware blocks, including the unique sensor boards, the common interface boards, and the central MCU. Communication between interface board and sensor board is accomplished by wired connections, and communication between the interface boards and the MCU is wireless.

2) The interface board provides a physical connection to the sensor board. Off-the-shelf, the RFduino module supports “shields” via a specific layout of female pin sockets. This layout was incorporated into our board for compatibility with this additional hardware. The most significant advantage of providing this compatibility is that it allows for the RFduino programming module to be connected for in-system programming for the interface boards.

3) The interface board broadcasts sensor data in accordance with iBeacon. The RFduino SoC integrates a 32-bit ARM Cortex M0 CPU with an embedded 2.4 GHz transceiver, providing a single IC that allows for communication of sensor data to the MCU.

The interface board was implemented as a two-layer printed circuit board and is approximately 2 inches by 4 inches in size. Through-hole components were selected where possible to simplify the assembly process.

C. Sensor Boards

The sensor boards uniquely accommodate each individual sensor. The three sensors supported by MHMS

at present are carbon monoxide (CO), humidity, and smoke (which, at a higher level, serves to detect smoke-producing fires).

D. Carbon Monoxide Sensor Board

The CO sensor board features an MQ-7 gas sensor. This component has a resistive element that is sensitive to CO in the air it resides in. A simple measure of the resistance of the element is sufficient to "read" the sensor. In order to prevent contaminated readings through absorption of other gases onto the resistive element, it must be periodically heated by providing voltage to the coil, allowing it to heat via current, which "cleans" the element of the other gases, and allows confidence that only CO levels are being measured.

In order to accommodate this relatively high-power heating current, a high-side MOSFET driver controlled by the RFduino is used. A typical heating and measurement cycle consists of 60 seconds of heating (by way of supplying 5 V to the resistive element) and 90 seconds of cooling (by way of providing a lower 1.4 V to the resistive element while allowing the conductivity to settle).

E. Humidity Sensor Board

The humidity sensor board houses a Parallax HS1101 humidity sensitive capacitor. The capacitance of this device changes with the amount of humidity in the air that it is in, in a conveniently very nearly linear fashion. The mathematical representation of the response, per the device datasheet, is shown in (1), where c is the capacitance in picofarads, RH is the percent relative humidity of the air the capacitor is in, and c_{55} is the capacitance at 55% relative humidity.

$$c = c_{55}\,(1.2510^{-7} \cdot RH^3 - 1.3610^{-5} \cdot RH^2 + 2.1910^{-3} \cdot RH + 9.010^{-1}) \quad (1)$$

The datasheet further suggests circuit consisting of a 555 timer and discrete components to measure the capacitance of the sensor. A block diagram of this approach is shown in

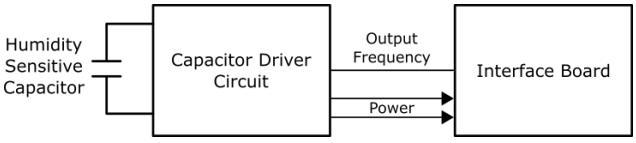

Fig. 3. A block diagram of the humidity sensor board functionality. Only three signals are required, two for power and one for output frequency.

The theory of operation is as follows: the timer continuously charges and discharges the sensor capacitor, while a discrete output toggles every time the voltage across the capacitor passes a certain threshold. The amount of time the capacitor takes to pass this threshold will depend on the RC time constant of the circuit, where R is a fixed value and C will vary with the RH. In this way, a discrete signal with a frequency proportional to the RH is output from the sensor board for conversion to a RH percentage.

F. Smoke Sensor Board

The smoke sensor board utilizes a photoelectric smoke sensor, one of the two typical types of sensors used in commercial smoke alarms [1]. The smoke detection itself is achieved via an IR LED and photodiode. The LED emits IR light into a chamber. The chamber is designed such that light cannot normally reach a photodiode placed at the opposite end of the chamber. However, once smoke has filled the chamber, the light reflects off of it and is able to reach the photodiode and thus indicate that smoke is present.

The RFduino cannot provide any real drive current out of its GPIO, and thus, a driver is needed for the LED. A small signal MOSFET is used on the smoke sensor board, controlled by the RFduino, to provide LED drive capabilities of 50 mA. This driver schematic is shown in Fig. 4.

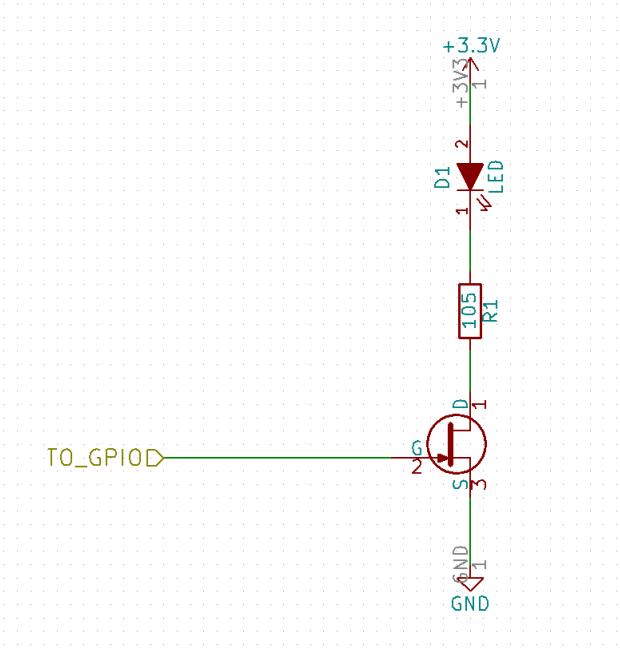

Fig. 4. IR LED MOSFET Driver, controlled by a RFduino GPIO, to avoid source current limitations of RFduino.

The current generated by the reflected light striking the photodiode is relatively small, and consequently generates a relatively small (less than 100 mV) voltage across the photodiode. This voltage is amplified by an op-amp and

before being presented to the RFduino for measurement and transmission.

G. Firmware Overview

The interface board firmware targets the RFduino, and its primary goal is to read sensor data, and transmit it to the MCU via Bluetooth.

The RFduino conveniently supports the Arduino libraries, and allows for rapid prototyping (as the Arduino enables very high level software), while still allowing minute details and efficiency to be gleaned through register-level access to the target device. In this way, the software aims to exist at the breaking point of the efficiency curve, in the sense that it is efficient to develop as well as being efficient when run on the hardware.

For all sensor boards, the RFduino libraries provide the necessary functions to update and broadcast major and minor fields per the iBeacon protocol.

H. Firmware – CO Sensor

The CO sensor firmware is responsible for controlling the periodic driving of the CO resistive element and reading the voltage across it (which is proportional to its resistance).

The timed coil driving is accomplished via the Arduino function delay( ). This function implements a software delay, which is suitable because there is no processing required while managing the coil, so there are no other statements to block. Driving the coil through the MOSFET is straightforward for high voltage, as the 5 V required are already present on the power rail provided to the sensor board. The lower voltage, however, is driven through PWM, and the functionality is readily provided by Arduino per the analogWrite( ) function. This function requires extra care, however, as the transition from the 3.3 V domain of the RFduino must be mapped to the 5 V domain of the MOSFET and CO sensor. This is accomplished via a custom pwmVoltage( ) function. Also, analogRead( ) provides a front-end to the onboard ADCs which are used to read in the heater voltage.

An overview of the CO firmware functionality is shown in Fig. 5.

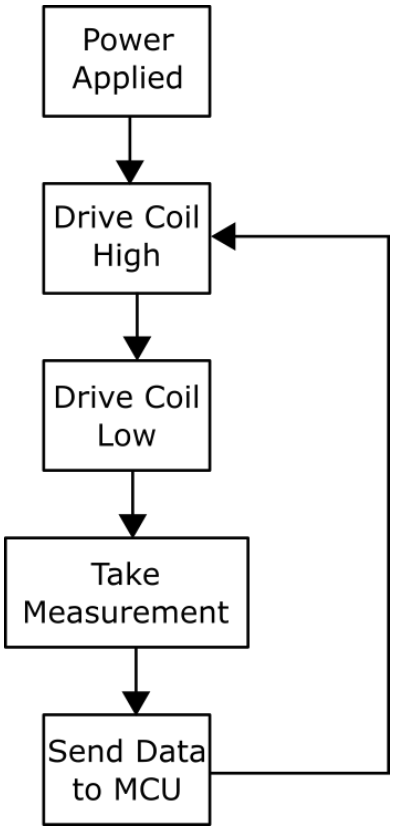

Fig. 5. CO sensor firmware flowchart, illustrating the cyclical nature of heating and cooling the coil.

I. Firmware – Humidity Sensor

The humidity sensor firmware is only responsible for interpreting the frequency of the signal originating from the humidity sensor board. In order to accurately determine the frequency of the signal, the timer modules of the RFduino SoC are used. The theory of operation, shown in Fig. 6 is as follows: an interrupt is triggered by the humidity sensor output, which starts the timer. Upon receiving the next rising edge of the humidity sensor output, the timer is stopped, read, and cleared to prepare for the next sample. In this way, as the timer frequency is known, and the number of timer pulses accrued during the humidity pulse is known, the unknown frequency can be determined by simple division.

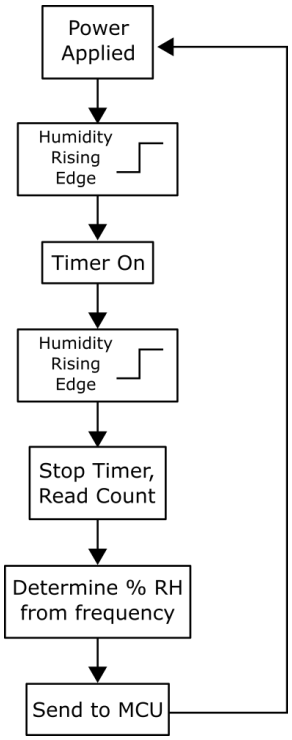

Fig. 6. Humidity sensor firmware flowchart. The firmware is interrupt driven and capitalizes on the SoC system clock being significantly faster than the output of the humidity sensor circuit.

J. Firmware – Smoke Sensor

The smoke sensor firmware's functionality is to provide a driving signal for the IR LED, and to read the amplified output of the receiving photodiode.

In order to save power, the LED is not continuously turned on. Instead, the LED are pulsed at a very low duty cycle during sampling periods, which are spread out on the order of seconds, as shown in Fig. 7.

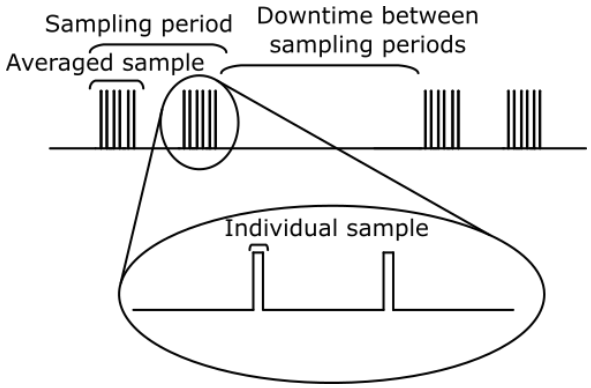

Fig. 7. Sampling technique used in smoke sensor firmware in order to minimize power usage. The number of samples used to average one sample and the number of averaged samples in a period are chosen according to experimental results.

K. Software - Main Control Unit

The MCU acts as a middle man for moving data between the sensors and the cloud application and it continuously monitors the iBeacon broadcasts of each Sensor Pod within range. On top of that, the MCU is responsible for capturing video frames from the camera and sending it to the web user interface at least once per second when desired by the user. The MCU software includes individual processes to take care of each of these tasks.

The process that controls communication consists primarily of two functions. First, it scans the area around it for broadcasting beacons. It is possible that the user will introduce a new sensor pod to the network at any given moment, so the software is able to quickly pick up the addition of new beacons on the Bluetooth network. Every beacon scan cycle, the MCU extracts the Address, UUID, Major and Minor fields from every beacon it sees. Since it is not enough to do just one scan then send, several scans are performed before data is sent to the Cloud to prevent a non-continuous transmitting beacon from being ignored.

When the beacon scan cycle is complete, the software packs the data into a message for the cloud service and sends it via a client API. The cloud application is then responsible for interpreting and displaying the data to the user, which includes the status of the sensors, alerts, and potentially sensor readouts.

The communication handling software must have access to Bluetooth communication functions natively on the Raspberry Pi. It was decided that Python would be used to write this software. Python is recommended by many users of Raspberry Pi because it is a lightweight, multi-paradigm programming language that is well supported by Raspbian, the OS that is recommended for use on Raspberry Pi. By using Python, libraries like Python-Bluez were able to be incorporated which allow for simple interfacing to the Bluetooth Dongle connected to the Pi. In addition, pre-made python libraries from IBM, ibmiotf allows for interfacing with the IBM Internet of Things Foundation which is how the MCU sends the sensor's data to the cloud application and how the image streaming software sends pictures to the MQTT Client on the web user interface. Using Python-Bluez ibmiotf and other supporting libraries, the MCU is able to perform its function in the system.

Python code development was done directly on the Raspberry Pi using the Integrated Development Environment (IDLE). IDLE was chosen because it is extremely simple and lightweight with no frills that many IDE's now a days have. Even though it is lightweight it still comes with some useful features for novice Python programmers such as an integrated debugger and auto-completion. Programming on the Raspberry Pi directly

would be an inconvenience without being able to use a VNC connection which is why the group used TightVNC on a desktop computer on the same network as the Pi to connect graphically to develop.

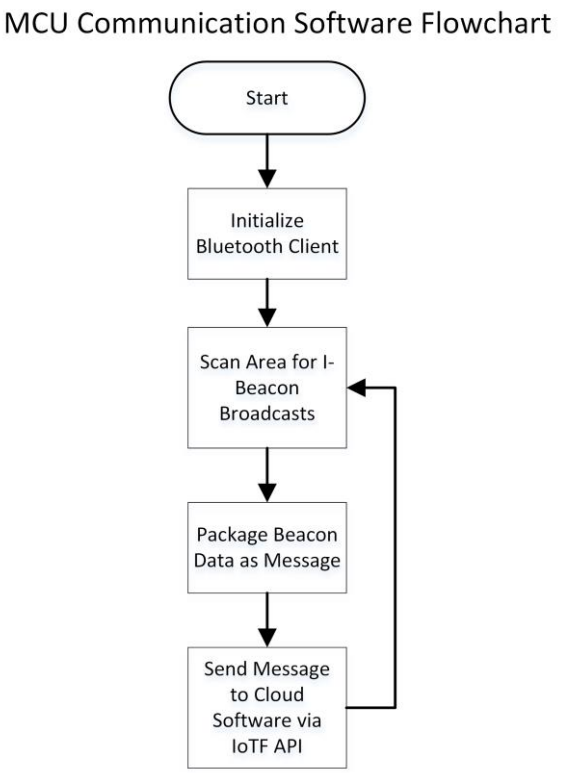

Fig. 8. MCU Communication Software Flowchart

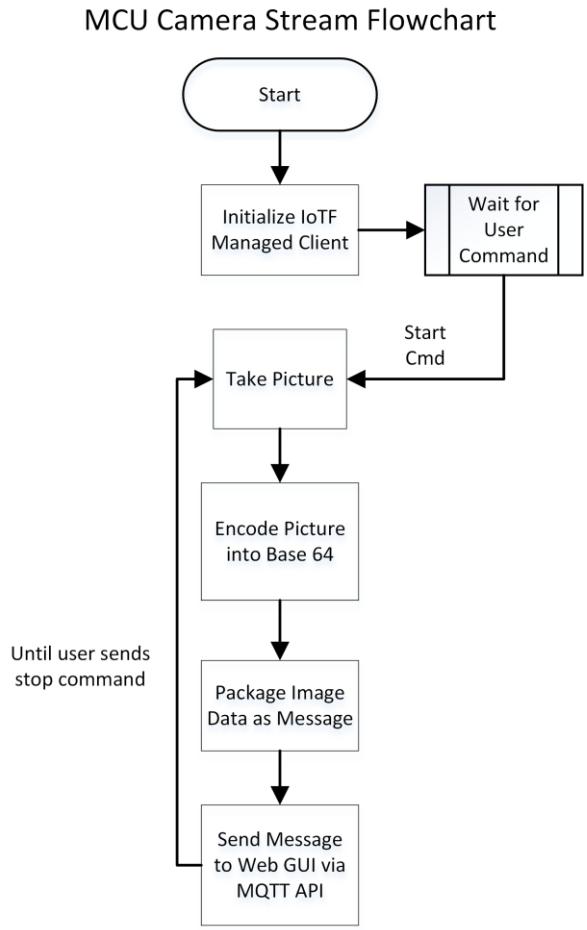

Fig. 9. Camera Stream Software Flowchart

To stream the series of still images from the Raspberry Pi's camera to the web user interface, the MCU uses a custom built software package to take, encode, send and decode pictures. The software package has the capability of transmitting an image stream to the Cloud at a specified frame rate and resolution, however because of limitations on data usage, care was taken to ensure the image stream is not always transmitting. The stream is only be active when the user is logged into the user interface and clicks a button to start the steam. That button sends a signal to the MCU to start capturing and sending images. The images taken on the MCU are encoded to Base64 in order to be split into packets and sent to the cloud server via MQTT [2]. From the cloud server, a MQTT client on the web page receives the packets, decodes the image and displays it on the web page.

The camera streaming software was written in Python using the IBM Internet of Things (IoT) Foundation (ibmiotf) library. It has a special feature where it only sends the image stream when the user presses the button on the web page to do so. In order to implement this functionality for the prototype, we used the client.commandcallback function which allowed us to specify a function to execute whenever a command was sent to the MCU. That function was then responsible for setting a flag that told the main thread to start or stop the image stream.

L. Software - Cloud Application and Database

The cloud application runs on a cloud PaaS provider that allows for hosting a web page, IoT connectivity, and database management. IBM Bluemix is the PaaS that the MHMS uses for these services. The bulk of the cloud application takes the form of a Node-RED flow that

performs logic functions and routing on messages received from the Internet of Things Foundation connection with the MCU.

The IBM Bluemix platform provides many "Services" to aid in the development of any project or application. We have implemented some of these provided services into our project prototype to ensure all the requirements of the cloud application are met. First of all, the MHMS uses the IBM IoTF to connect the MCU to the cloud and allow simple communication using a REST API. Once we registered the MCU with the IoTF website, we created an application with Node-RED. Node-RED allows users to "wire" together hardware devices on the IoT with APIs and other online services. We used Node-RED as our main control for our cloud application. The Node-RED flow has the responsibility of decoding messages, placing data in the database, and sending alerts.

The MCU sends all sensor data to the cloud application without doing any interpretation of data. The application decodes the message to determine the sensor type and content. Once it has this information, it is stored in a non-relational database called Sensor Status. If a sensor's readings are higher than a certain threshold, the message is sent to a Sensor Alerts database and an alert is displayed on the web user interface's recent alerts panel. Using a simple database check, the system checks if this is an alert that has not been sent to the user already, if not, a text message is sent to the user via a web SMS provider. The SMS provider used by the MHMS is Twilio due to its low cost and ease of use.

The Twilio third party service was integrated into the application to allow for SMS messages to be sent to the user when a hazardous situation is detected. Twilio SMS allows users to programmatically send, receive, and track messages worldwide. Once a user (or organization in this case) makes an account, a phone number is assigned that can be used in API calls within all types of software. The Twilio REST API is served over HTTPS with a base URL of https://api.twilio.com/2010-04-01. Sending a SMS text message is made very simple with the use of the Twilio Node-Red node which took care of the API specifics for us. Because we do not want users being bombarded with text messages, a check is done to detect if this particular alert has been sent to the user already within the past 10 minutes.

We have decided to use a non-relational database as our data store because of the small variety of data we actually have to store and the fact that non-relational DB's are easier to learn how to use. Our database only stores information about alerts and status messages from the sensor pods and such it will not be very complex. Since none of the MHMS team members were experienced in database management, it was decided early on that the cloud application would use the Cloudant NoSQL data base service to provide easy to use access to a JSON data layer in which alerts/status messages can be saved and user data can be stored for login authentication. Cloudant NoSQL DB is a NoSQL database as a service (DBaaS). It is built to handle a wide variety of data types like JSON, full text and geospatial. It is advertised to be an operational data store optimized to handle concurrent reads and writes, and provide high availability of data durability.

M. Software - Web User Interface

Any good home monitoring or automation system must have a way to either interact with or display data to the user. This is the point of systems like these, thus the User Interface is a very important component of the overall system. For the MHMS, it was debated whether the interface should be a program that runs on the user's computer, or a web interface that can be loaded on any web browser. Our group decided on using a web based user interface which is accessible and fully featured even when the user is not within their home network.

The user interface of the MHMS is a web page with access to information on the status of your connected Sensor Pods, recent alerts, and a stream from the MCU Camera. The sensor pod status section of the page shows if the pods are active and sending periodic messages or if they are reporting some issue that needs attention. A client side script checks the database for new alerts and display the alerts as well as a timestamp in the recent alerts panel. The image stream from the MCU camera is displayed on the right hand side of the user interface to provide users the ability to check on their home visually even when they are not there. Two buttons above the stream allow the user to start and stop the image stream, however if the user leaves the page some other way the stream will automatically stop to conserve resources. A Status indicator informs the user if the MCU is online as well.

A good user interface must also be responsive and have good aesthetics, which is why the MHMS user interface has a modern style and scalable design that uses pre-existing styling libraries. The user interface pulls data about sensors and alerts from a database every 5 seconds using a script. Simple database calls are used within the script to retrieve any updated information to be displayed. Because it is bad practice to have to reload the entire page when information needs to be updated, the User Interface is designed in such a way that only the parts that need to be changed will be updated.

The prototype for the user interface is accessible and hosted through IBM Bluemix. Bluemix applications have a HTML front end and come with a standard URL of myappname.mybluemix.net. This was sufficient for

building a prototype of the Modular Home Monitoring System as it is well integrated into the other Bluemix services that we used and required no extra work in setting up like an external web host would.

The user interface is written in HTML/JavaScript. The styling on the UI is done using Cascading Style Sheets (CSS). Because the user will be checking this software often, we tried to give the website a modern look to it. However, because this user interface prototype is mainly focused on functionality, the priority in development for the UI was to get a basic, functional version done first, then add any styling later.

To view the image stream from the MCU on the webpage, a PAHO MQTT client was embedded into the website through JavaScript. The client sends commands to the MCU to start or stop the video stream, receives the image files as Base64 data chunks and recompiles it to show to the user. An IBM Recipe was followed to create this functionality and pass image data through IBM IoTF.

Development of the website was done using Notepad++ for the HTML, CSS, JavaScript and PHP code. Files were then deployed to Bluemix using the cloud foundry command line interface which was downloaded directly from the Bluemix console. GitHub was also used to manage different versions of the build and keep track of changes.

IV. Conclusion

The ability for homeowners to remotely monitor the health and security of their home is a rising priority in today's day and age. The Modular Home Monitoring System meets this need by providing an integrated, low cost and modular system for monitoring CO, smoke and humidity in the home, as well as providing access to an image stream.

This product provides users peace of mind that if a dangerous event occurs in their home they will be first to know and will be able to react quickly. Overall the MHMS prototyping project has been a huge success. The lessons learned in time management, research and prototyping have be incredibly valuable to us and will be even more so as we move forward into our respective careers.

Acknowledgement

The authors wish to acknowledge the assistance and support of Dr. Richie throughout this course.

The authors would also like to acknowledge the faculty committee that is reviewing this project: Dr. Mingjie Lin, Dr. Ronald F. DeMara, and Dr. Kalpathy B. Sundaram.

References

[1] *Ionization vs. Photoelectric*. (2014, February 26). (National Fire Protection Association) Retrieved November 19, 2015, from http://www.nfpa.org/safety-information/for-consumers/fire-and-safety-equipment/smoke-alarms/ionization-vs-photoelectric

[2] Nguyen, D. (n.d.). Sending and Receiving Pictures From a Raspberry Pi via MQTT. Retrieved 10 1, 2015, from https://developer.ibm.com/recipes/tutorials/sending-and-receiving-pictures-from-a-raspberry-pi-via-mqtt/

Gary Leutheuser is currently a senior at the University of Central Florida. He plans to graduate with a Bachelor's of Science in Electrical Engineering in December 2015. He plans to pursue a master's degree in Computer Engineering at the University of Central Florida.

Robert Short is currently a senior at the University of Central Florida. He plans to graduate with a Bachelor's of Science in Electrical Engineering in December 2015. He plans to pursue a Master's Degree in Optics and Photonics.

Robert Simon is currently a senior at the University of Central Florida. He plans to graduate with a Bachelor's of Science in Computer Engineering in December 2015. He is currently working at Lockheed Martin MFC as a College Work Experience Participant and has accepted a full time position as a software engineer.