Final Project

Vincent Crabtree

Justin Edwards

Jacob Shebesh

Youssef Souryal

Indiana University-Purdue University

ECE 56601 Real-Time Operating Systems and Applications

**Overview:**

This project is the combination of everything we have learned in the Real-Time Operating Systems course with Professor Dongsoo Kim. The project is a Home Sensor System that utilizes ESP32 microprocessor, particle sensor, gas sensor, temperature sensor, humidity sensor, photosensitive resistor, LCD, and a battery. All of these components are tightly assembled and soldered together on a dual sided prototyping board to ensure that the components and wires will not be disconnected unintentionally. The purpose of this device is to emulate the gathering of sensory data from within the home. The data collected is as follows: carbon monoxide levels, particulate levels, light levels, temperature, and humidity. Alarms will be raised if these levels are raised above a threshold, thus indicating some abnormality in the environment. The device will then attempt to correct this by triggering external devices that are emulated by LED’s.

**Implementation:**

The software used to program this project is ESP-IDF and FreeRTOS, there was no Arduino IDE utilized to program any of the components. The program architecture was designed with readability being top priority so all the sensors program in separate .C file with a specific header file. The code was neatly commented to explain the functionality and purpose of every section in the code. Also, semaphores were used to protect access of the shared buffer structure. This buffer needed protection because multiple tasks have a writer relationship with the structure. In order to reduce busy-waiting and improve utilization, functions such as vTaskDelay() and vTaskDelayUntil() were used for implementing periodic tasks and delays within the system.

Upon the connecting the USB to the ESP32, a SNTP task gets created and a wifi connection with a mobile hotspot is initiated to grab the real time from the network. The time is stored in a variable that get displayed on the terminal every five seconds and it is also displayed on the LCD after the sensor data. Also upon USB connection, the user is able to type “log on” and enter a password to access the system shell. In this serial environment, the user can use the command “set” to view and manipulate current environment variables. These variables are stored in non-volatile memory using the NVS library. The user can then use the command “log off” to close the serial display.

The MQTT communication is implemented using a third party mqtt library that allows the use of a secured connection via SSL (Secure sockets layer). The library is used to connect to the MQTT server using a data structure that contains the URI, port, username, password, and client ID. A callback function is also included in this data structure. This connection callback function is used to create a task that will publish our sensory information every five seconds. The messages to the MQTT server are in a JSON format. The fields included are a type and value. The type is specified as one of the five sensors (photo,gas,temperature,humidity,particle) and the value is a float read from a global variable populated by the manager task using the sensor queue. Once the MQTT initialization function is called the callback will be triggered creating the communications task.

The LCD used in this project uses I2C protocol to display the data. I2C uses only two wires, SCL (serial clock) and SDA (serial data), they are pulled up with a resistor to +Vdd. I2c communication transfers 8 bits at a time, bit 0 is used to read and write to the device. The LCD utilized is 2x16 bits LCD which allowed two sensors data to be displayed at the concurrently on two separate lines. In displaying the time, the day and date are displayed on the first line while the time is displayed on the second one.

Each sensor utilizes an API that was created in previous projects. The API provides open, read, control, and close functions for each sensor. In the implementation, all the sensors that are required for the project are opened and set to collect data. This data is the enqueued for later use by the manager task. The manager task utilizes the queue containing all of the sensory data, it will remove a data point from the que, calculate the exponentially weighted average, and then place this in a global variable to be accessed by other functions. This is wrapped in a semaphore so that only one task can access the queue at any given time. This prevents the sensory reads from writing to the queue while being accessed by the manager function. LED’s are activated based on the values of the data collected. When the manager task receives values from the sensor tasks, the manager will take the values and separate them into a global string array called the system signature. The values are placed into a string array that the LEDs will examine to turn off or on.

The LED’s are accessed and written to by using the same API wrapping functions described above. These LEDs are used to represent actuation equipment such as a furnace, air conditioner, humidifier and dehumidifier. The LEDs/actuation devices are controlled by another task called the command task. This task accesses the system signature to determine which equipment to activate. For example, if the temperature is high in the signature, the command task will activate the air conditioner actuator. If the humidity signature is high, the command task will activate the dehumidifier.

Every device on the system is accessed through use of the same device wrapper functions. These functions are dopen(), dread(), dwrite(), dctl() and dclose(). The dopen function takes a device name as an argument and then finds that device on the device table structure. This structure consists of function pointers to the device interface functions for that specific device. Once the open function is called, that device’s functions are moved to a file table, which is a list of all the function pointers for that device. Finally, the index of that device in the file table is returned from the dopen() function as a file descriptor. This file descriptor is passed to the other functions as an argument and allows those functions to locate the device’s index on the file table. For example, when dread() is called for a device, that device’s file descriptor is passed as an argument, and the read function at that index in the file table is called.

Each sensor is accessed using these wrapper functions in their own dedicated sensor tasks. These tasks call the dread() function for their respective sensor periodically, and push the received values onto a shared FIFO queue. The manager task then pops the values off the queue one-by-one to update the exponentially weighted moving average for each sensor value. The manager task then uses these values to determine the status of each sensor, and assigns a value of very low, low, normal, high or very high to that sensor in the system signature.

When plugged into a serial terminal, a command window displays the BLANK character to let user know it is active. The first thing the user needs to do is enter the password. After successfully checking the password with Argon2 discussed above. The user has the option to log off, set a variable in the non-volatile memory, or display all variables in memory. The mechanism for the set command is defined below.

The first thing to do for storing or reading values from NVS is initializing the default partition on the ESP-32. Esp\_err\_t = nvs\_flash\_init() will return ESP\_OK if there are no errors, if not there is an error catching mechanism that will be triggered when there is no NVS partition in the partition table, or when there are no empty pages in the NVS. The next step is to open a namespace, nvs\_open(const char \*name, nvs\_open\_mode open\_mode, nvs\_handle \*out\_handle), in the storage with a user defined handle. The user can dictate if it was read or write with the nvs\_open\_mode. After opening a data path, the user can either choose to read or write depending on what mode they opened with. The team chose to use nvs\_set\_str(nvs\_handle handle, const char \*key, const char \*value) to store data as the data being passed would be words for the password. When the program checks what password is in the memory, the team chose to use nvs\_get\_str(nvs\_handle handle, const char \*key, char \*out\_value, size\_t \*length ) for the same reasons above. The last thing to do is close the NVS handle with nvs\_close(nvs\_handle handle).

Task characteristics were collected for each of the sensor tasks. The three characteristics collected included execution time, slack time and utilization. Execution was captured for each task individually and is the time spent by the system to execute that task. Slack time is the difference between the period and the execution time for each task. Finally, utilization is the ratio of execution time to period for each task.