Wireless Room Monitor with Base Station
Aaron Berns
https://github.com/anberns/pi

Overview
This independent project uses two Raspberry Pi 3's, basic sensors and a small OLED screen to monitor the activity in a given room or area. Both device implementations utilize the GPIO pins on the Raspberry Pi's, as well as the Unix file structure of the Raspbian OS to monitor and report area conditions. Both take advantage of concurrency by using multiple threads with the quad-core ARM processors on the Pi's. Socket programming and a client-server architecture are used for communication between the devices over an ad hoc network. Environmental readings are taken on the client device while the current temperature, sensor alerts and a sensor sensitivity settings menu are displayed on the screen of the server device, which also has four button inputs for menu navigation.

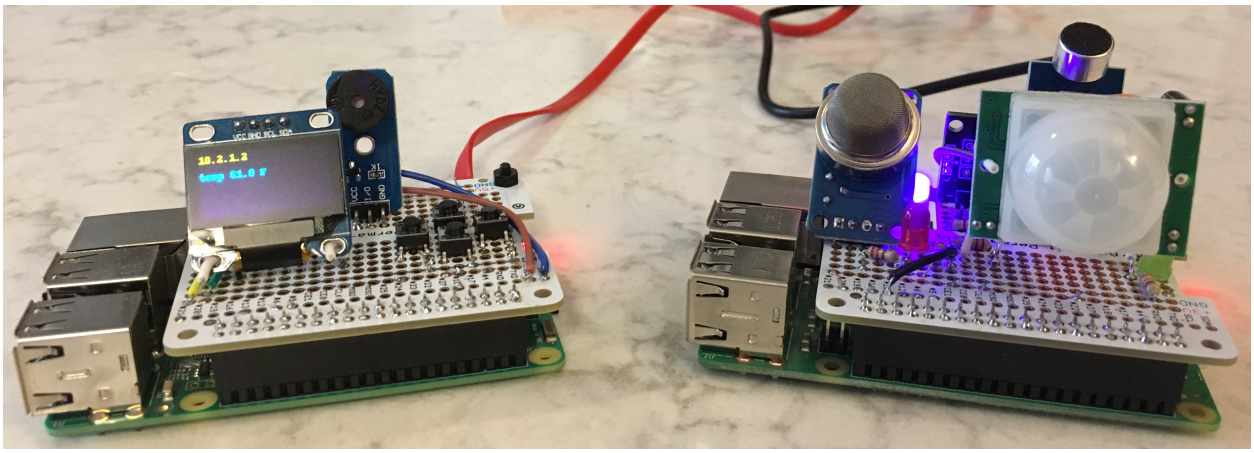

Fig 1. The server and client devices.

```
// attempt to connect to specified server and listening port
// exit if unable after 10 tries
int sockfd = -1;
int attempts = 0;
do {
    sockfd = estConnection(argv[1], argv[2]);
    if (sockfd < 0) {
        gpio_set_value(blue_led, 1);
        sleep(1);
        gpio_set_value(blue_led, 0);
        sleep(1);
        ++attempts;
    }
    else {
        gpio_set_value(blue_led, 1);
    }

} while (sockfd < 0 && attempts < 10);

if (sockfd < 0) {
    exit(1);
}
```

Fig 2. The client attempts to connect to the server.

The client device software is written in C and utilizes sockets (1) to communicate with the server device. When the client device is powered on, its software attempts to connect with the server device, allowing for disparities in power-up timing.

The client uses the OpenMP library to implement the concurrent monitoring of four different sensors: temperature, sound, motion and smoke. OpenMP sections are used to dedicate a thread from the thread team to polling each sensor. Triggered sensors acquire a mutex lock to

```
#pragma omp section
{
    sleep(1);
    while (!end) {
        pfd[0].fd = motion_fd;
        pfd[0].events = POLLPRI;
        lseek(motion_fd, 0, SEEK_SET);
        read(motion_fd, buf, sizeof(buf));
        rc = poll(pfd, nfds, POLL_TIMEOUT);

        if (pfd[0].revents & POLLPRI) {
            lseek(motion_fd, 0, SEEK_SET);
            read(motion_fd, buf, sizeof(buf));
            gpio_set_value(green_led, 1);
            omp_set_lock(&messageLock);
            if (sendEvent(sockfd, motion) != 0) {
                perror("sendEvent");
            }
            omp_unset_lock(&messageLock);
            sleep(1);
            gpio_set_value(green_led, 0);
        }
    }
}
```

Fig 3. The client's motion detection section.

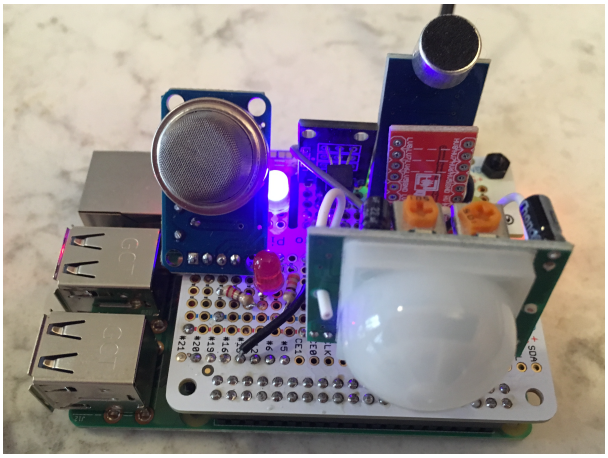

Fig 4. The client device sensors.

access the wireless connection to send a message to the server device. The polling method relies on waiting for GPIO interrupt signals (2), which is a blocking process. This prevents the polling while loop from consuming significant CPU resources.

While the sound, motion and smoke sensor threads are blocked until an interrupt is signaled, the temperature thread checks for a change in temperature every two seconds by reading the file (3) in which the sensor data is written by the OS. It then compares the current reading to the last reading, which is stored in a temporary file, and messages the server if the temperature has changed.

```
        while (!end) {
            fd = open(path, O_RDONLY);
            if (fd < 0) {
                perror("open temp file");
                break;
            }
            while ((numRead = read(fd, buf2, 256)) > 0) {
                strncpy(tmpData, strstr(buf2, "t=") +2, 5);
                temp = strtof(tmpData, NULL);
                temp = (temp / 1000) * (9/5) + 32;
                sprintf(message, "temp %.1f F", temp);
            }

            if (strcmp(message, oldMessage) != 0) {
                omp_set_lock(&messageLock);
                if (sendEvent(sockfd, message) != 0) {
                    perror("sendEvent");
                }
                omp_unset_lock(&messageLock);
                strcpy(oldMessage, message);
            }
            close(fd);
            sleep(2);
        }
```

Fig 5. The client checks for a temperature change. The path variable in the third line stores the path to the file in which the sensor data is written.

The client process ends when a button on the device is pressed. A section waits for this action to signal an interrupt, and when it occurs, the global flag variable, end, is updated. This causes all of the other monitoring threads to close up shop.

```
        #pragma omp section
        {
            while(!end) {
                pfd[0].fd = touch_fd;
                pfd[0].events = POLLPRI;
                lseek(touch_fd, 0, SEEK_SET);
                read(touch_fd, buf, sizeof(buf));
                rc = poll(pfd, nfds, POLL_TIMEOUT);

                if (pfd[0].revents & POLLPRI) {
                    end = 1;
                }
            }
        }
```

Fig 6. The OpenMP section that waits for a button press before initiating client shutdown.

The server device consists of a 128 x 64 pixel OLED display, a simple buzzer and four directional buttons. The software is written in Python and uses the thread and threading libraries to support multiple concurrent threads for handling sensor alerts, temperature updates and a settings menu as well as the RPi.GPIO library (4) for GPIO pin interaction.

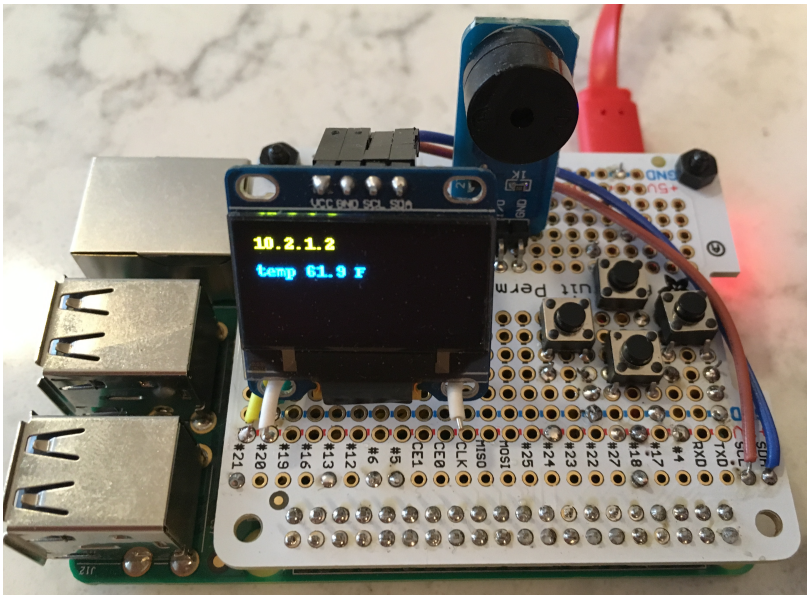

Fig 7. The server device.

The server uses the socket library to listen for, and connect to, a client device. When a client connects, the server attempts to find and load its settings file. This allows for user settings such as acceptable temperature range to persist between shutdowns. A new device is given default values.

```
try:
        serverSocket = socket(AF_INET, SOCK_STREAM)
        serverSocket.bind(('', int(sys.argv[1])))
        serverSocket.listen(1)
except IOError:
        print("Unable to create listening socket")
        exit(1)
```

```
connectionSocket, addr = serverSocket.accept()

# read from or create file for this device's settings
device_settings_path = "./data/" + addr[0] + "_settings"

try:
        if os.path.isfile(device_settings_path):
                infile = open(device_settings_path, "r")
                items = infile.readline().split()
                temp_low = int(items[0])
                temp_high = int(items[1])
                sound_level = int(items[2])
                motion_level = int(items[3])
                infile.close()
```

Figs 8 and 9. The server listening and attempting to load client settings..

The server displays the device name, current temperature and any sensor alerts on a generic OLED display by utilizing the Adafruit_Python_SSD1306, python-imaging and python-smbus libraries (5). Threading locks are used to control access to the display, buzzer and global variables.

```
def smokeEvent():
        global smoke_event

        #first smoke message sounds alarm until second received
        led_lock.acquire()
        draw.rectangle((0,ztop4,width,zbottom4), outline=0, fill=0)
        draw.text((x, ztop4), "SMOKE EMERGENCY", font=font, fill=255)
        disp.image(image)
        disp.display()
        led_lock.release()
```

Fig 10. Accessing and printing to the display.

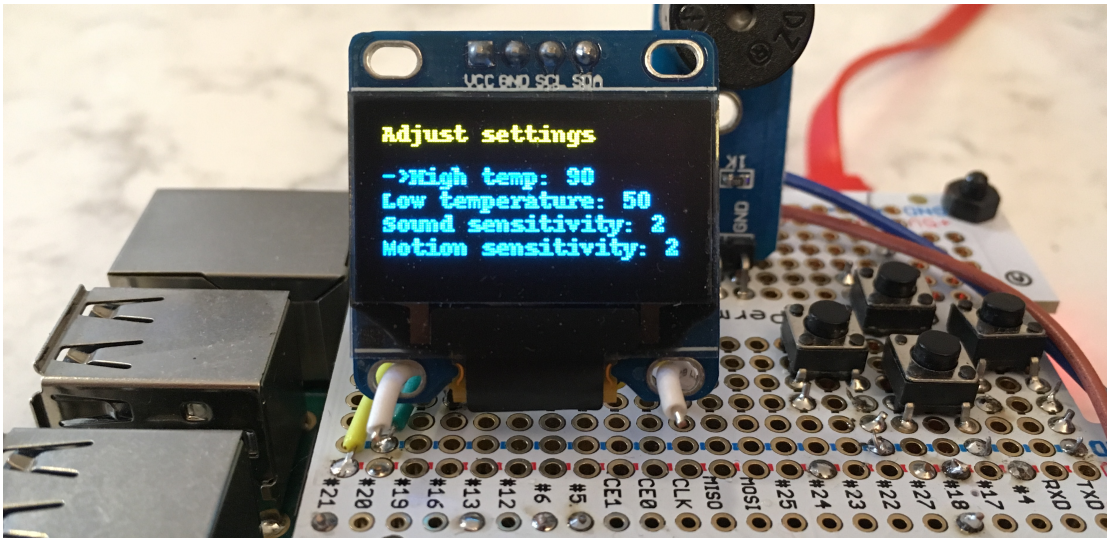

Fig 11. The settings menu and navigation buttons.

Threads dedicated to handling temperature range monitoring and the settings menu run along side the main process while new threads are created when messages are received from the client.

```
# launch settings menu thread
try:
        thread.start_new_thread(runMenu, ())
except IOError:
        print("Menu thread failed")

# launch temp monitoring thread
try:
        thread.start_new_thread(monitorTemp, ())
except IOError:
        print("Temperature monitoring thread failed")
```

```
while(1):

        #wait for message from client
        message = connectionSocket.recv(1024).decode()

        # sensor concurrency
        if 'temp' in message:
                try:
                        thread.start_new_thread(updateTemp, (message,))
                except IOError:
                        print("Temperature update thread failed")

        elif message == "smoke":
```

Figs 12 and 13. Some threads run throughout the process, others are spawned to handle client messages.

A settings menu allows for the acceptable high and low temperature to be changed as well as the device's sensitivity incoming sensor alert messages. Sensor messages can activate a short buzzer alert each time they are received to set a low threshold for activity monitoring, can be monitored for frequency so that only sustained activity triggers an alert or can be ignored completely with settings of 2, 1 and 0 respectively. The menu is accessed and navigated using the four directional buttons.

Both the client and server are designed to run independent of any other peripherals. When powered up, an additional line added to the .bashrc file of each device calls a bash script (6). The script runs the appropriate program and shutdowns the Raspberry Pi after it exits. The devices connect over an ad-hoc network (7) and can run on battery power, allowing them to be used anywhere.

Sources

1. https://beej.us/guide/bgnet/
2. https://developer.ridgerun.com/wiki/index.php/How_to_use_GPIO_signals , https://developer.ridgerun.com/wiki/index.php?title=Gpio-int-test.c
3. http://bradsrpi.blogspot.com
4. https://pypi.org/project/RPi.GPIO/
5. https://learn.adafruit.com/ssd1306-oled-displays-with-raspberry-pi-and-beaglebone-black/usage
6. https://www.dexterindustries.com/howto/run-a-program-on-your-raspberry-pi-at-startup/
7. http://www.jumpnowtek.com/rpi/Raspberry-Pi3-AdHoc-Wifi-Network.html