Co2 Monitoring and controlling system for Greenhouse

Group : 5

Siddhartha Lama (2112923)

Subash KC (2012190)

Sulav Thapa (2012220)

Anatolii Rubanenko (1901923)

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Abstract

Author:	Siddhartha Lama, Sulav Thapa, Subash K:C, Anatolii Rubanenko
Title:	Co2 Monitoring and controlling system for Greenhouse
Number of Pages:	13 pages
Date:	22 December 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Smart IoT System
Supervisor:	Keijo Länsikunnas

The project aims to create an Automated system for controlling the Co2 level in the Greenhouse. The readings of Various environmental factors that affect the growth of the crops such as temperature, humidity, and Co2 level are obtained from sensors using Modbus connections. The microcontroller used in this project is LPC 1549. The readings are then forwarded to the local user interface where they can be displayed. The Co2 set point is controlled using the rotary encoder and its button.

The Solenoid valve opens, and closes based on the Co2 Setpoint set by the user. Once the Valve is open the Co2 gas is injected until the Co2 setpoint is achieved. The readings of Co2, Temperature, Humidity, and valve state along with the Co2 set point are also sent to the Cloud-based user interface via MQTT.

Table of Content

1. User Manual 4
   - 1.1 Local User Interface (LCD) 4
   - 1.2 Rotary Encoder 4
   - 1.3 Thing Speak Cloud-based User Interface 5
2. Program Documentation 7
   - 2.1 Flowchart 7
   - 2.2 Implementation Principle 8
   - 2.3 Detail Description of the Program 9
   - 2.3.1 sensor_task Function 9
   - 2.3.2 lcd_task Function 11
   - 2.3.3 prvMQTTtask Function 12

1. User Manual:

1.1 Local user interface (LCD)

FIG1: Local user Interface (LCD) for the Co2 monitoring and Control System.

There are 5 parameters in the LCD screen:

C → Co2 reading from the Co2 Sensor

T → Temperature Reading from the Temperature Humidity sensor

rh → Relative Humidity Reading from the Temperature Humidity sensor

co2setPT → Co2 level Set Point: This is the set point that can be set as per need.

V → It represents the Status of the Solenoid Valve: V=1 (Valve open) and V=0 (Valve closed)

1.2 Rotary Encoder

Fig 2: Rotary Encoder

The Rotary encoder is used to change and set the value of the Co2 level as required by the user.

- Step1: Press the button on the tip of shaft once first to enter the Co2 level changing mode
- Step 2: Rotate the Shaft clockwise to increase the Co2 set point.
- Step 3: Rotate the Shat anti-clockwise to decrease the Co2 set point.
- Step 4: Press the button on the tip of Shat once to set the Co2 Level.
- Step 5: Once the Desired Setpoint (co2setPT) is achieved the Valve closes itself (i.e., V= 0)

1.3 Thing Speak Cloud-based User Interface

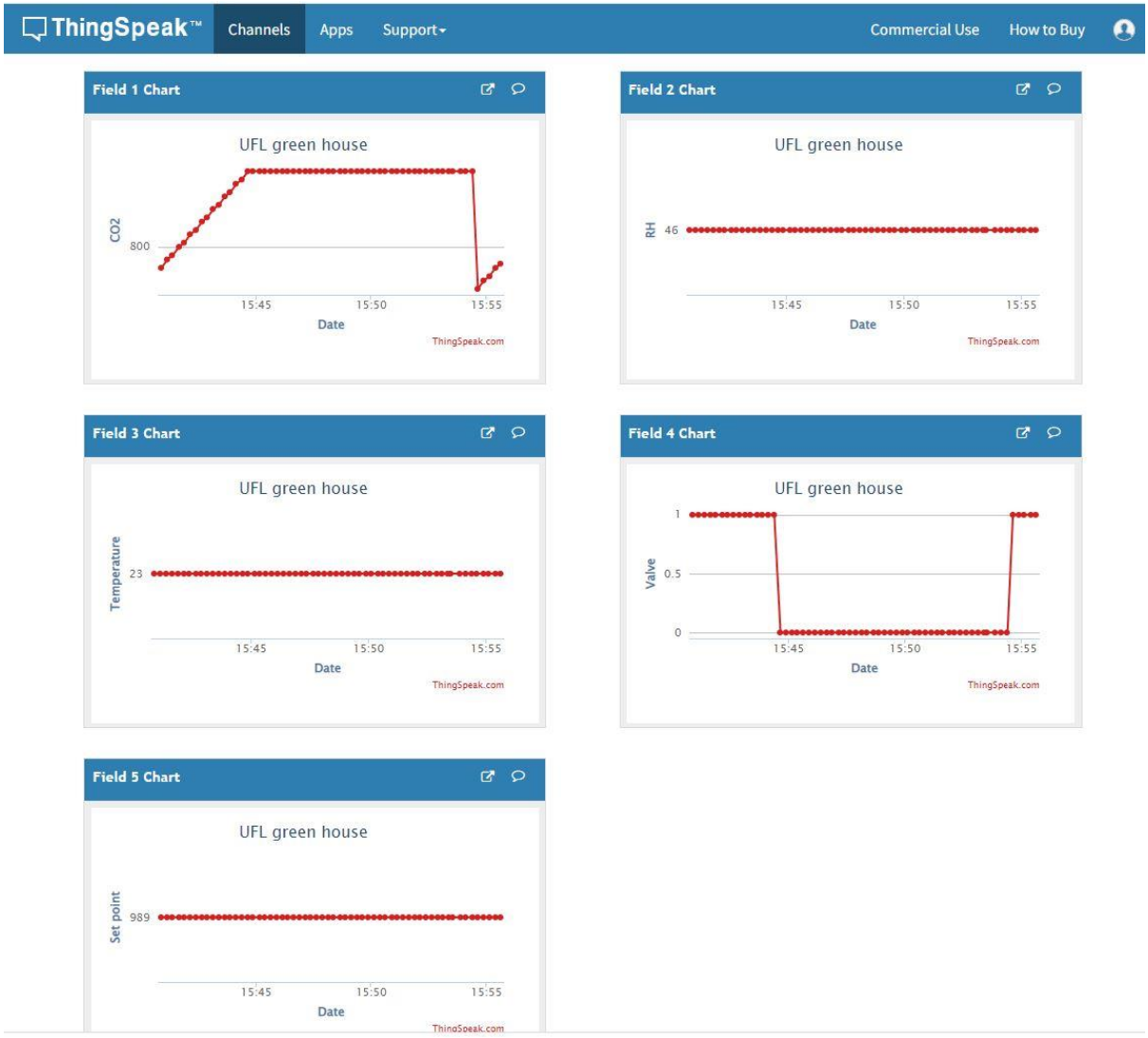


FIG3: Thing Speak Cloud-based User interface

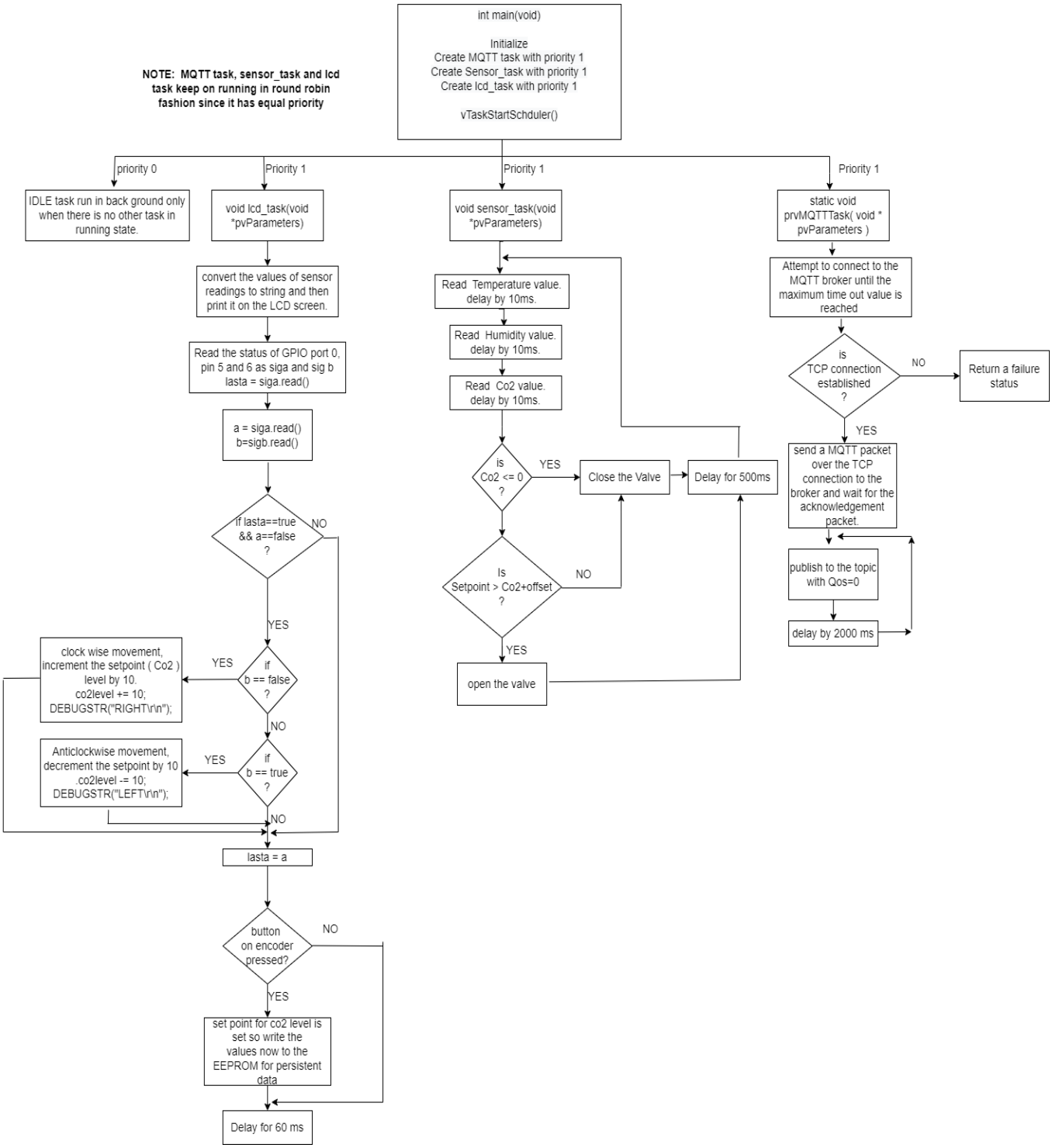
- The Readings from Sensor and the Valve state can also be viewed in the Thing speak channel
- Go to the link https://thingspeak.com/channels/1955513 to view it.

Thing Speak Cloud-based UI description:

- Filed1 = Graphical view of Co2 reading from the sensor over the period.
- Field2 = Graphical view of Relative Humidity Readings from the sensor over the period.
- Field3 = Graphical view of Temperature readings from the sensor over the period.
- Field4 = Solenoid Valve state over the period. 1 represents open and 0 represents closed.
- Field5 = The set points for Co2 level over the period.

2. Program Documentation

2.1 Flow Chart



2.2 Implementation Principles

The idea behind building this project was to use Free RTOS with LPC 1549 microcontroller to achieve our goal of CO2 level monitoring and controlling system. The advantages of using FREE RTOS are:

Real-time operating systems generally have the following characteristics:

- **Small size:** Going hand in hand with simplicity is FreeRTOS's **small size**. Many tiny microprocessors have limited RAM and limited flash storage. FreeRTOS is tiny, which can lower our cost of goods considerably.
- **Ubiquitous** Next: FreeRTOS is **ubiquitous**. It runs on pretty much any small, modern MCU. That's good when you're bringing a product to market because it gives you a degree of hardware isolation. Building our code on top of FreeRTOS means it's a little bit more portable, and this in turn means you're a little bit less locked into whichever microprocessor vendor you have selected.
- **Open source:** As the "Free" in the name suggests, FreeRTOS is fully **open source**. There's some fear, uncertainty and doubt (FUD)
- **Small footprint.** Compared to general OSes, real-time operating systems are lightweight.
- **High performance.** RTOSes are typically fast and responsive.
- **Determinism.** Repeating inputs end in the same output.
- **Safety and security.** Safety-critical and security standards are typically the highest priority, as RTOSes are frequently used in critical systems.
- **Priority-based scheduling.** Tasks that are assigned a high priority are executed first followed by lower-priority jobs.
- **Timing information.** RTOSes are responsible for timing and providing Application Programming Interfaces.

2.3 Detail Description of The Program :

Three tasks MQTT-Task, Sensor task, and LCD task were created in total as shown below with equal priority and stack size was assigned higher than configMINIMAL_STAK_SIZE to avoid overhead issues. No parameters have been passed for any of the tasks. A Null pointer has been passed as a task handle for sensor_task and lcd_task. Whereas for MQTT-TASK no task handle has been passed.

```
xTaskCreate( prvMQTTTask,"MQTT-TASK",
configMINIMAL_STACK_SIZE+1024,NULL,tskIDLE_PRIORITY +1UL,NULL );

xTaskCreate(sensor_task, "sensor_task",
configMINIMAL_STACK_SIZE * 8, NULL, (tskIDLE_PRIORITY + 1UL), (TaskHandle_t*) NULL);

xTaskCreate(lcd_task, "lcd_task",
configMINIMAL_STACK_SIZE * 8, NULL, (tskIDLE_PRIORITY + 1UL), (TaskHandle_t*) NULL);


vTaskStartScheduler();
```

When each task is created, a specified stack size is allocated to each task in the heap memory along with the TCB. TCB keeps vital information about the task such as the priority, location, and such.

When vTaskstartscheduler() is executed, it looks for the task with the highest priority to execute but, since here all the task has equal priority so all three tasks will be executed in turn in a Round-robin fashion. The function associated with each task is described in detail in the following section

2.3.1 sensor_task Function

→The Modbus master is created for each sensor and the Modbus register is for more specific things such as checking if the sensor is ready and the actual sensor values. The addresses used in the constructor are used to communicate with the master and register over Modbus. In conclusion, we created a connection to the sensor over Modbus.

```
ModbusMaster humiditytemperature(241);
humiditytemperature.begin(9600);
ModbusRegister humidity(&humiditytemperature, 256, true);
ModbusRegister temperature(&humiditytemperature, 257, true);

ModbusMaster co2sensor(240);
co2sensor.begin(9600);

ModbusRegister co2val(&co2sensor, 257, true);

ModbusRegister co2status(&co2sensor, 0x800, true);
ModbusRegister humiditytemperaturestatus(&humiditytemperature, 0x200, true);
```

Also, we set a baud rate of 9600 for each sensor as shown in the snippet.

→Digital input/output pin number 27 of port 0 was set to write the status of the Solenoid Valve so that the valve can be controlled.

```
DigitalIoPin Valve(0, 27, DigitalIoPin::output);
```

→After reading the sensor values from the sensor the value obtained was multiplied and divided by 10 as necessary to scale to the units that we wanted to display on the User interface as shown below. And a Delay of 10ms was used after each obtained reading of the sensors to keep the readings flow smoothly from three different sensors.

```
if(humiditytemperaturestatus.read())
{
    vTaskDelay(10);
    temperaturevalue = temperature.read() / 10;
    vTaskDelay(10);
    humidityvalue = humidity.read() / 10;
}


vTaskDelay(10);
if(co2status.read() == 0)
    {
        vTaskDelay(10);
        co2value = co2val.read() * 10;
    }
```

→The valve was open and closed based on the logic that

if the reading from the sensor (co2value) is less than or equal to zero then the valve is left closed.

if the setpoint (Co2level) is greater than the reading of Co2 (Co2value) plus the offset then the valve is opened so that the Co2 gas can be injected until the setpoint is reached. Once the setpoint is achieved the valve is closed.

After that, the delay of 500ms was implemented as shown in the code snippet below. The offset value was set to 10 in our case.

```
            if (co2value <= 0)
            {
                Valve.write(false);
                Valvevalue = false;
            }
            else if(co2level > (co2value + offset))
            {
                Valve.write(true);
                Valvevalue = true;

            }
            else
            {
                Valve.write(false);
                Valvevalue = false;

            }


                vTaskDelay(500);
```

2.3.2 Lcd_task Function

→ setting up the LCD pins for displaying the values. `lcd.begin(16, 2);` indicates 16 columns and 2 rows are being used in the LCD

```
// create lcd
DigitalIoPin pinrs(0, 29, DigitalIoPin::output);
DigitalIoPin pinen(0, 9, DigitalIoPin::output);
DigitalIoPin pind4(0, 10, DigitalIoPin::output);
DigitalIoPin pind5(0, 16, DigitalIoPin::output);
DigitalIoPin pind6(1, 3, DigitalIoPin::output);
DigitalIoPin pind7(0, 0, DigitalIoPin::output);
LiquidCrystal lcd(&pinrs, &pinen, &pind4, &pind5, &pind6, &pind7);
lcd.begin(16, 2);

DigitalIoPin button(1, 8, DigitalIoPin::pullup,true);
DigitalIoPin siga(0, 5, DigitalIoPin::pullup);
DigitalIoPin sigb(0, 6, DigitalIoPin::pullup);

bool set_co2 = false;
co2level = atoi(co2.c_str());
bool lasta = siga.read();
```

→ Setting up the digitaliopin pin number “8 of port 1” for the button of the encoder and “pin 5 of port 0” for siga and “pin number 6 of port 0” for sigb.

```
DigitalIoPin button(1, 8, DigitalIoPin::pullup,true);
DigitalIoPin siga(0, 5, DigitalIoPin::pullup);
DigitalIoPin sigb(0, 6, DigitalIoPin::pullup);
```

→ converting the values of sensor data, valve status, and setpoint values to strings and displaying it on the LCD by setting the cursor point of the LCD accordingly.

```
std::string co2string = std::to_string(co2value),
temperaturestring = std::to_string(temperaturevalue),
humiditystring = std::to_string(humidityvalue),
co2levelstring = std::to_string(co2level),
Valvestring = std::to_string(Valvevalue);


lcd.setCursor(0, 0);
lcd.print("C="+co2string + " " +"t=" + temperaturestring + " " +"rh="+ humiditystring + "            ");

lcd.setCursor(0, 1);
lcd.print("co2setPT=" + co2levelstring + " " +"V=" + Valvestring + "              ");
```

→ Implementing the logic of the rotary encoder: At the falling edge of Signal A if the value of signal B is 0 then it is clockwise but if signal B is 1 at the moment, then it is counter clockwise. It is based on 2 bits Gray codes logic, Where only one bit changes at a time.

If the button is pressed once first and if the movement is clockwise, we increase the value of setpoint by 10 at each clockwise movement. And if the movement is counter clockwise the co2 setpoint is decreased by 10 for each counterclockwise movement. Its shown in the code snippet below.

```
if(set_co2)
  {
        bool a = siga.read();
        bool b = sigb.read();


      if(lasta == true && a == false)
      {
              if(b == false)
          {
              co2level += 10;
              DEBUGSTR("RIGHT\r\n");
          }

              else if(b == true)
              {
                  co2level -= 10;
                  DEBUGSTR("LEFT\r\n");
              }

      }
      lasta=a;
  }
```

2.3.3 prvMQTTtask Function.

→ Attempts to connect to the MQTT broker until the maximum number of configured attempts are reached or the time-out values expire. If the TCP connection is not established even after the maximum configured attempts are reached then it returns a failure status.

```
xNetworkStatus = prvConnectToServerWithBackoffRetries( &xNetworkContext );
configASSERT( xNetworkStatus == PLAINTEXT_TRANSPORT_SUCCESS );
```

→ once the TCP connection is established the following code sends the MQTT packets over the established connection and waits for the acknowledgment.

```
LogInfo( ( "Creating an MQTT connection to %s.", democonfigMQTT_BROKER_ENDPOINT ) );
prvCreateMQTTConnectionWithBroker( &xMQTTContext, &xNetworkContext );
```

→ publishing the message to the topic with a QoS value of 0. A delay of 2000 ms has been used so that the value gets updated in the ThingSpeak user interface every 2 seconds.

```
for( ;; )
{

    sprintf(buff, "field1=%d&field2=%d&field3=%d&field4=%d&field5=%d", co2value, humidityvalue, temperaturevalue, Valvevalue, co2level);
    prvMQTTPublishToTopic( buff, &xMQTTContext );
    vTaskDelay(2000);
}
```

Besides the LCD local user interface, The data is also published to the Thingspeak cloud user interface Via MQTT. The credential such as used for Thingspeak are obtained from the Thingspeak website which is shown below.

```
#define CHANNEL_ID "1955513"
#define SECRET_MQTT_USERNAME "DCAmDzgFFhoKKy8kCBw3NQA"
#define SECRET_MQTT_CLIENT_ID "DCAmDzgFFhoKKy8kCBw3NQA"
#define SECRET_MQTT_PASSWORD "qKxqzEzD+xMf2LMg0SU24WYk"
/**
```

The TCP port used for the MQTT connection here is 1883 and the topic we are publishing to is

“channels/195513/publish”. In Our case the values are updated in the Cloud interface at every 2 seconds. The MQTT broker endpoint In this case would be `mqtt3.thingspeak.com`