**Thermal Control System Documentation**

**Introduction**

The thermal control system is designed to regulate the temperature inside a room by controlling the cooling power of two compressors. The system uses an Arduino board with Arduino FreeRTOS for task management and ArduinoJson library for JSON communication.

**Hardware Setup**

1. Arduino Board with FreeRTOS Support
2. Two Compressors (connected to pins 51 and 53)
3. Fan (controlled via PWM on pin 4)
4. Temperature Sensor (not included in the provided code)

**Global Variables and Constants**No table of contents entries found.

* **fanSpeedpin**: Analog pin to read the fan speed input.
* **compressorA** and **compressorB**: Digital pins connected to the two compressors.
* Various constants used for the thermal model and room properties.
* **outsideTemperature**: The outside temperature in degrees Celsius.
* **comp[2]**: An array to store the state of the two compressors (ON/OFF).
* **initState**: A flag to indicate the initialization state of the system.
* **insideTemperature**: The current temperature inside the room.
* **coolingPower**: The cooling power used to control the temperature.
* **desiredTemperature**: The desired temperature to be maintained inside the room.
* **switchTime**: The time interval for the compressor load balancing task.

**Task Handles and Queue Handle**

* **simulationTaskHandle**: Handle for the simulation task.
* **printTaskHandle**: Handle for the task responsible for printing JSON data.
* **receiveTaskHandle**: Handle for the task that receives commands via serial communication.
* **compressorTaskHandle**: Handle for the main compressor control task.
* **loadBalanceTaskHandle**: Handle for the load balancing task that switches compressors.
* **fanspeedTaskHandle**: Handle for the fan speed control task.
* **queueHandle**: Handle for the queue used for inter-task communication.

**Task Functions**

1. **simulationTask**: Calculates the heat transfer and updates the temperature inside the room periodically. It sends the temperature and cooling power data in JSON format to the print task.
2. **printTask**: Receives JSON data from the simulation task and prints it via the Serial interface.
3. **receiveTask**: Listens to the Serial interface for incoming commands (in JSON format) to control the system. It adjusts the desired temperature and starts/stops the compressor accordingly.
4. **compressorTask**: Manages the compressor state based on the desired temperature and inside temperature. It also initiates the load balancing task after the initial cooling phase.
5. **loadBalanceTask**: Periodically switches between the two compressors for load balancing purposes.
6. **fanspeedTask**: Reads the fan speed from an analog pin and controls the fan speed using PWM.

**Communication Protocol**

The system communicates via JSON messages received and sent through the Serial interface. The messages include temperature data, cooling power, compressor states, and acknowledgments.

**Initialization**

1. Serial communication is initialized with a baud rate of 9600 and a timeout of 1000ms.
2. The queue for inter-task communication is created.
3. All task functions are created and assigned to their respective handles.
4. The compressor and load balancing tasks are initially suspended.

**Main Loop (loop() function)**

The main loop is empty since the system runs on FreeRTOS tasks, and the loop should never be called.

**Task Scheduling and Execution**

1. In the **setup()** function, the tasks are created using the **xTaskCreate()** function, which takes the task function, a task name, stack size, and task priority as parameters. The task handles are stored in their respective variables.
2. The compressor and load balancing tasks are initially suspended using the **vTaskSuspend()** function.
3. The FreeRTOS scheduler is started with the **vTaskStartScheduler()** function, which begins executing the tasks.

**Simulation Task (simulationTask)**

1. The simulation task is responsible for calculating the heat transfer and updating the temperature inside the room.
2. It starts by retrieving the outside temperature using a temperature sensor (not included in the provided code). Alternatively, you can replace this with a fixed value or another temperature source.
3. Inside a loop, it calculates the heat transfer rate based on the temperature difference between the inside and outside, as well as other constants related to the thermal model.
4. It then updates the inside temperature using the heat transfer rate and a time step value.
5. The simulation task also calculates the cooling power needed to maintain the desired temperature inside the room.
6. It creates a JSON message containing the temperature and cooling power data and sends it to the print task using the **xQueueSend()** function.

**Print Task (printTask)**

1. The print task receives the JSON messages from the all the tasks using the **xQueueReceive()** function.
2. It waits for a message to arrive in the queue and then retrieves it using the **xQueueReceive()** function.
3. The received JSON message is printed via the Serial interface using the **Serial.println()** function.

**Receive Task (receiveTask)**

1. The receive task listens to the Serial interface for incoming commands in JSON format.
2. It waits for a command to arrive using the **Serial.readStringUntil()** function.
3. Once a command is received, it parses the JSON message using the ArduinoJson library.
4. The command can include adjusting the desired temperature or starting/stopping the compressor.
5. If the command includes a desired temperature, it updates the **desiredTemperature** variable.
6. If the command includes a compressor state, it updates the **comp[2]** array to reflect the desired state of the compressors.
7. If the command includes a compressor start/stop action, it starts or stops the compressor by resuming or suspending the compressor task using the **vTaskResume()** and **vTaskSuspend()** functions.

**Compressor Task (compressorTask)**

1. The compressor task is responsible for controlling the compressors based on the desired temperature and the current temperature inside the room.
2. It starts by initializing the state of the compressors (**comp[2]**) and the **initState** flag.
3. Inside a loop, it checks the **initState** flag to determine if the system is in the initialization phase.
4. During initialization, it turns on the compressors and sets the cooling power to the maximum value.
5. After the initialization phase, it checks if the current temperature is above or below the desired temperature.
6. If the current temperature is higher than the desired temperature, it switches on whichever compressor’s turn has come as decided by the Load Balancing Task.
7. If the current temperature is lower than the desired temperature, it turns off whichever compressor was switched on previously.
8. The compressor task also updates the **comp[2]** array to reflect the current state of the compressors.

**Load Balancing Task (loadBalanceTask)**

1. The load balancing task periodically switches between the two compressors to distribute the load evenly.
2. It starts by suspending itself using the **vTaskSuspend()** function until it is initiated by the compressor task.
3. Once initiated, it switches the load between compressor A and compressor B by updating the **comp[2]** array.
4. After switching the load, it suspends itself again to wait for the next load balancing initiation from the compressor task.

**Fan Speed Task (fanSpeedTask)**

The fan speed task is responsible for reading an analog pin and adjusting the fan speed based on the analog voltage value received from a potentiometer.

It starts by initializing the analog pin for reading using the analogRead() function.

Inside a loop, it reads the analog voltage from the specified pin using the analogRead() function.

The analog voltage value is then mapped to a fan speed range using the map() function to convert it to a suitable fan speed value.

Finally, the fan speed is set accordingly using the appropriate mechanism or library for controlling the fan (e.g., PWM output).

The fanSpeedTask runs continuously, adjusting the fan speed based on the analog voltage input from the potentiometer.

**HMI Python Code Documentation**

This documentation provides a detailed overview of the HMI (Human-Machine Interface) Python code that acts as the interface for controlling a system with a fan and a potentiometer. The code uses the PySide6 library to create a graphical user interface (GUI) and communicates with the system through a serial connection. The purpose of this documentation is to explain the code structure, functionality, and usage instructions.

**Table of Contents**

1. Dependencies
2. Code Overview
   * **serial\_ports()**
   * **ackTimeout()**
   * **SerialThread**
   * **SerialsendThread**
   * **MainWindow**
3. GUI Components
4. Usage Instructions
5. Further Customization and Modification

**1. Dependencies**

The HMI Python code has the following dependencies:

* **serial**: A library for serial communication.
* **PySide6**: A Python binding for the Qt framework.
* **QtCharts**: A module for creating interactive charts in Qt applications.
* **numba**: A just-in-time compiler for accelerating numerical computations.

Make sure to install these dependencies before running the code.

**2. Code Overview**

The code consists of several classes, functions, and components that work together to establish a serial connection, read and send data, and update the GUI based on the received data. Let's take a closer look at each component:

**serial\_ports()**

This function is responsible for listing the available serial ports on the system. It detects the serial ports based on the platform (Windows, Linux, macOS) and returns a list of port names.

**ackTimeout()**

This function sets a global variable **timeout** to **True** when an acknowledgment timeout occurs. It is called when a timeout happens during the acknowledgment process.

**SerialThread**

This class is a subclass of **QThread** and handles the serial communication. It continuously reads data from the serial port and processes it. It emits signals based on the received data, such as temperature and time values or acknowledgment messages. The **run()** method runs in a separate thread and handles the serial communication loop.

**SerialsendThread**

This class is another subclass of **QThread** and is responsible for sending data to the serial port. It reads data from the **main.queue** list and sends it to the serial port. It waits for an acknowledgment (ACK) message from the system or a timeout before sending the next data. The **run()** method runs in a separate thread and handles the data sending loop.

**MainWindow**

The **MainWindow** class is the main backend class that interacts with the GUI and handles the serial connection, communication, and data processing. It contains slots that are connected to various GUI events, such as button clicks and value changes. It emits signals to update the GUI and provides methods for interacting with the system. Some of the important slots and methods in this class are:

* **getPorts()**: This slot is called after the application is loaded and lists the available serial ports. It emits a signal with the list of available serial ports, which can be accessed by the GUI.
* **sendBtn()**: This slot is called when the user clicks the Send button on the GUI. It performs the necessary actions based on the button press.
* **connectSerial()**: This slot is called when the user selects a serial port and baud rate and clicks the Connect button. It establishes a connection with the selected serial port and sets up the necessary variables and threads for communication.
* **disconnectSerial()**: This slot is called when the user clicks the Disconnect button. It terminates the serial communication thread and closes the serial port connection.
* **setTemp()**: This slot is called when the user sets the temperature value. It converts the value to a JSON string and adds it to the data queue for sending.
* **startController()**: This slot is called when the user clicks the Start button to start the controller.
* **stopController()**: This slot is called when the user clicks the Stop button to stop the controller.
* **setOuttemp()**: This slot is called when the user sets the output temperature value.
* **setSwitchtime()**: This slot is called when the user sets the switch time value.

The **MainWindow** class also contains several signals that are emitted to update the GUI with received data or notify certain events.

**3. GUI Components**

The GUI components are defined in the **main.qml** file. The QML file specifies the appearance and behavior of the user interface elements. The main components include:

* Labels: Displaying temperature and time values.
* Buttons: Connect, Disconnect, Send, Start, and Stop buttons.
* Sliders: For setting temperature, output temperature, and switch time values.
* Charts: Displaying temperature and time history.

The QML file interacts with the backend code by connecting to the slots and signals defined in the **MainWindow** class. The backend code updates the GUI based on the received data and user interactions.

**4. Usage Instructions**

To use the HMI application, follow these steps:

1. Install the required dependencies mentioned in the Dependencies section.
2. Connect the system with the fan and potentiometer to the computer using a serial connection.
3. Run the Python script, which initializes the GUI and starts the application.
4. The GUI window will appear, displaying the available serial ports in a dropdown list.
5. Select the desired serial port and baud rate from the dropdown list.
6. Click the "Connect" button to establish a connection with the system.
7. Once connected, you can interact with the system using the provided controls in the GUI. For example, you can set the desired temperature, output temperature, and switch time values using the sliders. You can also start and stop the controller by clicking the corresponding buttons.
8. Real-time temperature and time values will be displayed on the home page, and the temperature and time history will be visualized in the charts.
9. To disconnect from the system, click the "Disconnect" button.

**5. Further Customization and Modification**

The provided code can be customized and modified to fit specific requirements. Some possible enhancements or modifications could include:

* Adding additional GUI components or pages to control or monitor other aspects of the system.
* Implementing error handling and validation for user inputs.
* Extending the communication protocol to support additional commands or data.
* Improving the visualization of data by enhancing the charts or adding additional graphical elements.
* Implementing data logging or saving functionality to store temperature and time data to a file or database.
* Optimizing the code for performance or adding additional features using the **numba** library.