Matthew Trembley

12/11/2022

Southern New Hampshire University

CS350

Final Project: Thermostat Documentation

The smart thermostat task scheduler is developed in a few steps. First it is important to declare important variables and data types such as:

|  |
| --- |
| char ready\_tasks  int global\_period  struct task\_entry  {  void (\*f)(); //function to call to perform the task  int elapsed\_time; //Amount of time since last triggered  int period; //Period of the task in ms  char triggered; //Whether or not the task was triggered  }; |

Each attribute to the struct task\_entry has its own purpose. A pointer function to dynamically allocate memory for each corresponding function. An integer of elapsed time to keep track of the time since the function or task was last triggered. The period of the task. Lastly, a char datatype “triggered” to determine whether or not the task had been set.

After this, we can then move forward and declare the specific forward declarations that will soon become our tasks within the task scheduler.

|  |
| --- |
| void setPointTask();  void tempCheckTask();  void displayOutput(); |

And to pair with “struct task\_entry” we can declare the different tasks within our data structure.

|  |
| --- |
| struct task\_entry tasks[NUMBER\_OF\_TASKS] =  {  {&setPointTask, 200, 200, FALSE},  {&tempCheckTask, 500, 500, FALSE},  {&displayOutput, 1000, 1000, FALSE}  }; |

Each element with the data structure is it’s own task, identified mainly by the function name as to what it will do, the elapsed time and period, and the trigger state. After this has been created, we are clear to create our different pointer functions which will act as each task for our task scheduler.

|  |
| --- |
| void setPointTask()  {  if (rightButtonPress == TRUE)  {  setpoint += 1; //increase the set point  rightButtonPress = FALSE; //reset button press back to zero  }  if (leftButtonPress == TRUE)  {  setpoint -= 1; //decrease the set point  leftButtonPress = FALSE;  }  } |

The first function to be discussed is the “setPointTask()”. This task detects whether a button has been pressed every 200ms, as identified in the struct parameter. If the button on the right is pressed, it will increase the setpoint temperature and reset the button callback back to false. The left button is similar, but instead of increasing the setpoint temperature, it will decrease.

|  |
| --- |
| void tempCheckTask()  {  temperature = readTemp();  if (temperature < setpoint)  {  GPIO\_write(CONFIG\_GPIO\_LED\_0, CONFIG\_GPIO\_LED\_ON);  heater = TRUE;  }  else  {  GPIO\_write(CONFIG\_GPIO\_LED\_0, CONFIG\_GPIO\_LED\_OFF);  heater = FALSE;  }  } |

This second function is to check the temperature every 500ms. This function also will operate the LED to inform the user whether the heater is on or not. It will start by reading the temperature and setting it to the variable temperature, declared elsewhere. It will introduce a conditional statement, where it says if the temperature is LESS than the setpoint, it will light the LED and set the heater to TRUE, or “on”. In all other cases, the LED will be OFF, and the heater will be set to FALSE or “off.

|  |
| --- |
| void displayOutput()  {  DISPLAY(snprintf(output, 64, "<%02d, %02d, %d, %04d>\n\r", temperature, setpoint, heater, seconds));  ++seconds;  } |

This last function, or task, will take place every second or 1000ms. All this function does it send output via UART to the server, in a format of temperature, user setpoint, heater state, and the time in seconds.

After initializing a timer, the timer will need a call back which is the control for the tasks and when they will be triggered.

|  |
| --- |
| void timerCallback()  {  //Set ready\_tasks to true if any task interval has expired  int i = 0;  //Iterate through each task  for (i = 0; i < NUMBER\_OF\_TASKS; i++)  {  //check if task's interval has expired  if (tasks[i].elapsed\_time >= tasks[i].period)  {  //Task timer is up, set flag and global flag  tasks[i].triggered = TRUE;  ready\_tasks = TRUE;  tasks[i].elapsed\_time = ZERO;  }  else  {  tasks[i].elapsed\_time += global\_period;  }  }  } |

This timer callback will iterate through each task via a for loop, and it will compare the elapsed time to the period of each task. If elapsed time is greater than or equal to the period, it will trigger the task, flag the global variable of ready\_tasks to true(which will be touched upon later), and reset elapsed time back to zero. If this doesn’t happen, it will increment elapsed time but the global period, declared above.

|  |
| --- |
| //Endless loop  while (TRUE)  {  //wait for task intervals(periods) to elapse  while (!ready\_tasks){}  //Process the tasks for which the interval has expired  int i = 0;  for (i = 0; i < NUMBER\_OF\_TASKS ; i++)  {  if (tasks[i].triggered == TRUE)  {  //perform tasks  tasks[i].f();  //reset trigger  tasks[i].triggered = FALSE;  }  }  //Reset everything (e.g. flags) and loop back to beginning  ready\_tasks = FALSE;  } |

Now last, the “while” loop which is set in the main function of the program. This will wait for “ready\_tasks” to be set false. It will iterate through the tasks to see the state of trigger(true or false). If a specific task is true, it will perform the task and then reset the task to false.

The following UML diagram represents the flow of the task scheduler.

Diagram

Description automatically generated