Aaron J Walls

Professor

CS 499

November 11, 2021

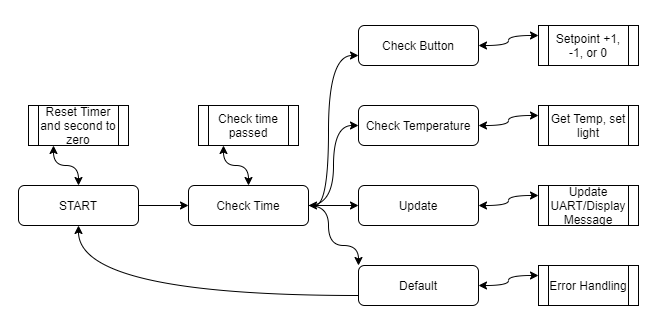
Milestone 2: Software Design and Engineering

The GPIOinterrupt.c is a thermostat application written in C for the TI SimpleLink Wi-Fi CC3220s launch pad designed by Texas Instruments. It is a thermostat designed to display the current temperature, a specific set point, and turn on a red light if the set point is higher than the current temperature. The set point can be manipulated by the user when they press two buttons: one for adjusting the temperature up 1, and one for adjusting the temperature down 1. This program was designed to showcase a state machine. It emphasizes the separation between actions and transitions.

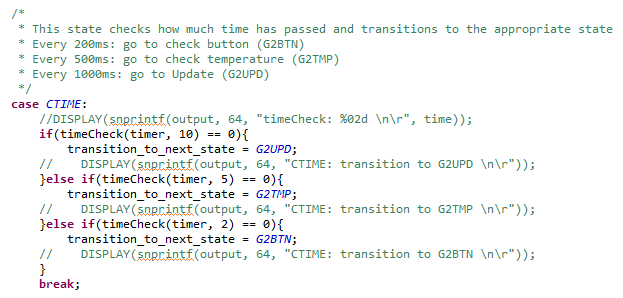
The original state machine had a cyclic flow, where the states happened one after another in the same order unless something unusual occurred, where it would restart. This diagram depicts how the

Diagram

Description automatically generatedflow of state machine. It has five defined states, start, check button, check temperature, update, and default. The start state sets the values of the variables to zero. The check button state checks to see if the user pressed any buttons and modifies the set point accordingly. The check temperature phase gets input from the temperature sensor and turns on a LED if the temperature is less than the set point. Update displays the temperature, set point, LED light status and total elapsed time. The default state catches any unhandled events and transitions back to start, effectively resetting the system. This first version of the state machine works well and was considerably robust. Its simple logic made it easy to code, understand, and work. It avoided race-conditions where the application would rely on an interrupt before continuing or state collisions where multiple states should be happening at once by having a set order for each state. The disadvantages this approach has is the machine isn’t checking for user input and temperature sensor are only checked once every 200ms and 500ms respectively.

 The update to the application fixes this by adding an intermediate state. Now instead of happening in a fixed cycle, the state machine uses a check time state to transition to the check button, check temperature, and update states. The diagram describes the changes, as the logic now includes a branching path. From check time, the state machine can go to check button, and back to check time. Then it can go to any of the other states and back again. Using this strategy, any of the states can at specific intervals. However, it requires additional logic to solve how to handle multiple state triggers. The state machine used two enums to clearly define how the state transitioned. The TRANSITION\_GATE enum, was updated to contain a go to value for each state as seen below:

In this way, it is possible to control what transition needs to happen from Check time. Prior to this, the enum only contained two values which determined whether enough time has passed to advance to the next state in the line. From here, check time can set appropriate flag in *transition\_to\_next\_state* using an if-else statement. The transition block can check *transition\_to\_next\_state* value to change to the correct state using an if-else statement. Since Check button, check temperature, update, and default all transition back to check time, there transitions are all identical (only their actions are different). Below are the actions of Check Time (CTIME).



Below are the transitions of Check Time:

Text, letter

Description automatically generated

This change wasn’t enough to completely solve the problem. As is, the transition to Update would happen between 10 or 12 times every 1/10th of a seconds, because how *fast* the application could execute the code. This means we’d have to control what Update was doing during the tiny space of time between the first time Update was executed and when Timer would be incremented.

Graphical user interface, text, application

Description automatically generated

…

Background pattern

Description automatically generated with low confidence

So, to prevent update from printing the values more than once per second, a lastUpdate variable was declared. This variable is set to one when the timer is incremented, which means update can only happen once every 1/10th of a second. The while loop used to be empty, which means after the logic has happened, the program would wait the remainder of the 1/10th second before starting again. In this adaptation, I couldn’t utilize that control mechanism. If I attempted to, the check temperature state would be skipped in favor for the update state. This also means the loop is always doing something, which means extra logic had to be included (like the lastUpdate control variable) to prevent update from being spammed. Included in the new while loop, (the one above) timer is reset to 0 after 100 seconds to prevent numeric overflow and any associated security flaws that may present.

State Machines are used commonly as a software design solution. This project not only demonstrates state machines, but interrupts, timers, and bare metal development of IoT devices. When the project is in front of you completed, the solution seems straight forward, however discovering the solution is a different story. Application development can be difficult because there is no definitive way to solve a problem. There are a million different solutions to this project, and I have shown two. It is up to the developer to decide what solution is the best. This rendition of the state machine prioritizes checking the states based on time elapsed using modulo, but because the loop runs faster than the timer is incremented it created a lot of unforeseen issues to manage. This discovery process of development is the most enjoyable but also can be the most frustrating if the solution is too elusive.