Supervisory Control Program Implementation

(Linux/Ubuntu)

By: Kevin Hartwig

For: Professors M. Galle and A. Tehrani

Date: June 2, 2017

# introduction

This document describes the implementation of the Supervisory Control Program for the CAN Diagnostics Control (Elevator) project undertaken over the summer of 2017 by Kevin Hartwig, James Sonnenberg and Ovi Ofrim of the Electronic Systems Engineering program. The duration of the project is four months and will be graded as part of our 3rd Year, 2nd semester Engineering Project course. All project code is hosted at <https://github.com/kevin-hartwig/CAN-Node-Code>, and all code for the Supervisory Control Program can be found in a folder of the same name. The Supervisory Control Program can be cloned from this repository: reference the readme in the repository that explains how to get the code working on an Ubuntu system. The Supervisory Control Program is written entirely in C f Ubuntu 16.04

# program flow

. The flow of the main application loop is straightforward, consisting of two main sections:

1. A polled *read* on the CAN bus
2. A *step* through a state machine

The main application loop is entered after the initialization phase is completed. This phase consists of initialization of global parameters used to track elevator position, target position, door state, a three element *char* array[[1]](#footnote-1) used as a FIFO queue for floor requests, and floor request queue index variable. It also initializes variables used in the state machine to track current state and the current state transition condition (or “action”) and initializes the PCAN interface.

Upon successful initialization, the program enters the main application loop where it infinitely cycles between the two steps listed above. Interrupt-based reading on the CAN bus was not necessary in this implementation due to the large receive buffer present in the low level hardware. For this reason, we vied for polling for ease of implementation and simplicity.

# CAN bus read

The PCAN receive buffer is read once per iteration of the main application loop. This means that the receive buffer in the PCAN hardware is *not necessarily emptied every iteration.* However, because the main loop application cycles between only two main tasks (reading the PCAN Rx buffer and taking a step through the state machine), this is not an issue.

If a message is received from the PCAN Rx buffer, it is immediately parsed. During parsing, message ID (corresponding to the sender) and the message data are used to update global variables appropriately. For example, if a message is received with ID 0x201, a floor request is processed from Floor 2 (see message protocol for more information).

# State machine

A graphical representation of the state machine integrated in the Supervisory Control program can be seen in Figure 1. There are four states: *WAITING, MOVING, DOOR OPEN* and *DOOR CLOSED.* Transitions between states correspond to certain conditions, of which there are six: *Floor Arrival; Door is closed; Door is opened; Dequeue is invalid; Dequeue is valid;* and *Repeat.* The transitions in the block diagram are numbered to correspond with the state transition table in Section 4.1. In program initiation phase, the initial state is set to *WAITING*, and thus this is the state machine entry point.

The state machine was implemented using a function pointer/lookup table approach. The design principles are outlined below.

1. Functions were defined for each state. Each function executes a short procedure in which it checks global variables and returns an appropriate *condition.*
2. A lookup table was defined to relate transitions between states and the condition required for those transitions. This table corresponds directly with the state transition table described in Section 4.1
3. The condition returned from a state function is then passed along with the current state into a function which accesses the lookup table and determines the next state.

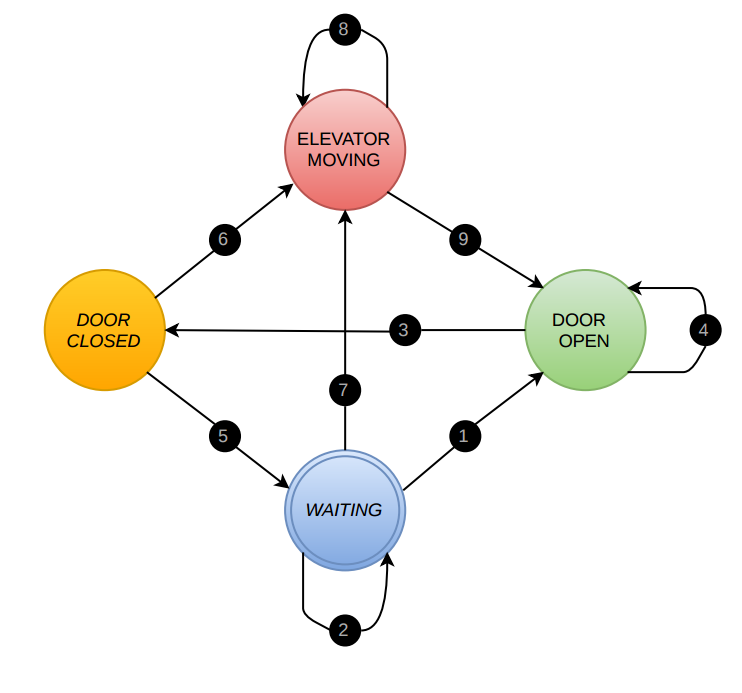


Figure : State Machine Block Diagram

# state transition table

Definition of all states and the conditions required for state transitions.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Present | Condition | Condition Source[[2]](#footnote-2) | Next |
| 1 | *Waiting* | *Door is opened* | *CC* | *Door Open* |
| 2 | *Waiting* | *Repeat* | *-* | *Waiting* |
| 3 | *Door Open* | *Door is closed* | *CC* | *Door Closed* |
| 4 | *Door Open* | *Repeat* | *-* | *Door Open* |
| 5 | *Door Closed* | *Dequeue Invalid* | *SC (Self)[[3]](#footnote-3)* | *Waiting* |
| 6 | *Door Closed* | *Dequeue Valid* | *SC (Self)* | *Moving* |
| 7 | *Waiting* | *Dequeue Valid* | *SC (Self)* | *Moving* |
| 8 | *Moving* | *Repeat* | *-* | *Moving* |
| 9 | *Moving* | *Floor Arrival* | *EC* | *Door Open* |

**CC:***Car Controller* | **SC:** *Supervisory Controller* |**EC:** *Elevator Controller*

1. Only three elements are required as there are only three floors and duplicate floor requests are not permitted. [↑](#footnote-ref-1)
2. Condition sources for all *Repeat* conditions are N/A as they indicate no change in state [↑](#footnote-ref-2)
3. These conditions are cited as *SC (Self)* as they depend on a successful dequeuing from the FIFO floor request queue. However, this queue is updated with requests received from all floors as well as the car controller. [↑](#footnote-ref-3)