CS723 – ASSIGNMENT 2 REPORT

Connor Dunn & Lachlan Chan

Department of Electrical and Computer Engineering
University of Auckland, Auckland, New Zealand

1. Introduction

Cruise control is a common automotive feature that automatically maintains a vehicle's speed, reducing the driver's workload and improving comfort. Achieving this requires precise real-time control and reliable system responses under various operating conditions. In this project, a cruise controller was developed using Esterel, a synchronous programming language well-suited for real-time and embedded systems. The report outlines the functional requirements and explains how they are mapped into Esterel modules.

2. Specification

The cruise controller was described using a functional specification to show the general inputs/outputs, as well as to understand how different signals interact with each other. This was developed into a system context diagram, multiple finite state machines (FSMs), and data/control flow decomposition. In doing so, it clearly defines how the system should respond to specific combinations of inputs over time. The functional specifications were also crucial in developing the Esterel implementation, allowing parts to be divided into smaller modules that perform a particular task and make validation significantly easier.

2.1. Context Diagram and First Level Refinement

The general operation of the cruise controller is depicted in Figure 1. The context diagram provides a basic understanding of the types of inputs and outputs that go into the whole system. Pure signals *On*, *Off*, *Resume* or *Set* activate or modifies the cruise controller's behaviour while valued signals like *Speed*, *Accel* and *Brake* determine the cruise speed and throttle actions. There are 3 outputs: *CruiseSpeed* which is the target speed, *ThrottleCmd* that controls the acceleration of a vehicle and *CruiseState* to indicate what actions can be taken.

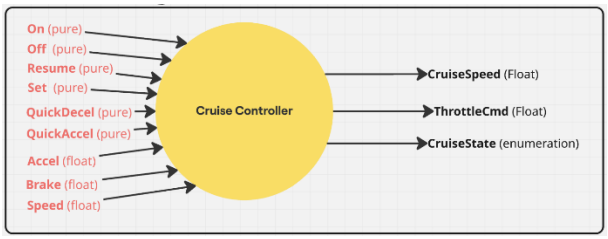


Figure 1: *Context Diagram of the Cruise Controller*

From the context diagram, this was then divided into three smaller modules: Cruise Controller, Speed Control and Throttle Control, as shown in Figure 2. The Cruise State Controller determines the operating mode of the system (On, Off, Standby or Disable) depending on the button inputs and pedal/speed conditions. Speed Control sets the cruise speed for the vehicle based on QuickAccel, QuickDecel or Set buttons. Throttle Control regulates the acceleration of the vehicle using either the accelerator pedal or a Proportional Integral (PI) feedback controller depending on what state the system is in.

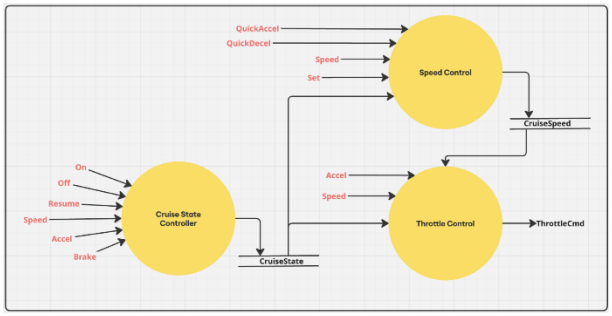


Figure 2: *First Level Refinement of the Cruise Controller*

2.2. Finite State Machine

The Cruise State Controller is modeled as an FSM as shown in Figure 3. Each state indicates the mode that the system is in which directly influences the actions the system performs from the Speed Control and Throttle Control. The Cruise State Controller will read the input buttons, speed and pedal conditions to determine which state the system will transition to. There also is a level of transition prioritization indicated by the red transitions. One example is that if an *OFF* signal is detected, regardless of other conditions, the whole cruise controller should be immediately turned off. This reflects the expected operation of a cruise control system in a real vehicle.

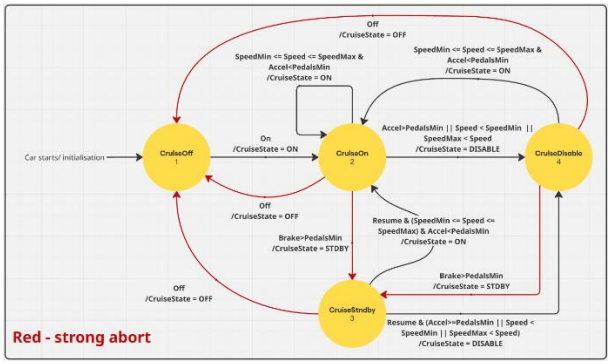


Figure 3: FSM of Cruise State Controller

2.3. Data Control Flow Decomposition of Speed Control

The Speed Control Module, as seen in Figure 4, is responsible for managing the cruise control speed target of the system. This module will only function while the cruise controller is not in the CruiseOff state, i.e. CruiseOn, CruiseStndby, or CruiseDisable. Under this condition, the cruise speed can be set, increased or decreased using the signals *Set*, *QuickAccel* and *QuickDecel* respectively. The current vehicle speed, *Speed*, as well as the thresholds, *SpeedMin* and *SpeedMax*, will determine the value the cruise speed will be set to.

- When *Set* is pressed, *CruiseSpeed* will be set to the current *Speed.*
- When QuickAccel is pressed, *CruiseSpeed* will be incremented by a set amount (*SpeedInc*).
- When QuickDecel is pressed, *CruiseSpeed* will be decremented by a set amount (*SpeedInc*).

If the current vehicle speed or the incremented/ decremented speeds are outside these thresholds, then the cruise speed will automatically be limited to those thresholds. This is represented in Figure 4.

By separating the speed control into one module, it provides a clear indication of how *CruiseSpeed* will be set over every tick.

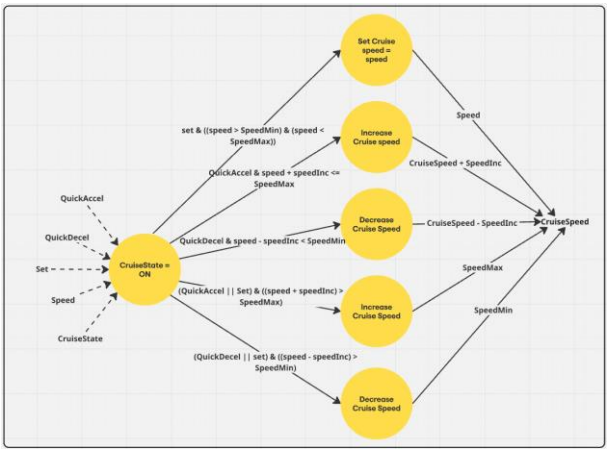


Figure 4: *Data Control Flow of Cruise Speed Control*

2.4. Data Control Flow Decomposition of Throttle Control

The Throttle Control Module, as seen in Figure 5, is responsible for managing the throttle of the system by reading the accelerator pedal, *Accel*, or a regulated output using a proportional integral (PI) controller provided in C (*RegulateThrottle()*). The resulting output, *ThrottleCmd*, adjusts the vehicles acceleration accordingly to follow a set cruise speed or not.

- When in states *CruiseOff, CruiseStndby, or CruiseDisable*, the *ThrottleCmd* will be given the same value as *Accel*
- When in state *CruiseOn*, the *ThrottleCmd* will calculate the error between the set *CruiseSpeed* and the current *Speed* and will apply the PI controller to calculate a throttle value
- To ensure the PI controlling is smooth, there is a throttle limit using the *SaturateThrottle()* C function.

When the system transitions to the *CruiseOn* state, the integral term of the PI controller will be reset to avoid any overshoot. If the throttle is at its limit, the integral term will be fixed to ensure a stable response over time.

Separating the throttle control from the entire system ensures that it will perform as expected without any interference from other inputs in the system.

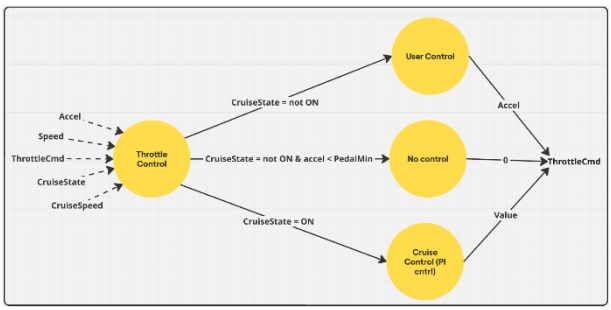


Figure 5: Data Control Flow of Cruise Throttle Control

Through splitting key functions of the cruise controller into their own modules, it is easier to ensure deterministic behaviour for each component and hence the entire system. It also allows each module to be run concurrently, synchronized using broadcast communication and a single master clock using tick delays.

3. Design in Esterel

3.1. Modules

The Esterel implementation of the cruise controller reflects the same specifications in Section 2. There is one main module (*CruiseController*) that consists of 3 submodules (*CruiseFSM*, Cruise*SpeedControl*, *ThrottleControl*) that run concurrently.

- *CruiseController:* This module is the top-level that runs the other submodules concurrently. It also receives all the inputs and links the output from *MainFSM* to the other two submodules to use as inputs.
- *MainFSM*: Executes FSM that determines the state of the controller. It takes the required inputs to determine its state (*On, Off Resume, Accel, Brake, Speed*) and will output *CruiseState*. The control logic is displayed in Figure 3.
- *CruiseSpeedControl*: This module will output the *CruiseSpeed* that the vehicle should be aiming for while the cruise controller is on. This depends on the inputs *Set, QuickAccel* and *QuickDecel* as well as the current *Speed*.
- *ThrottleControl*: This module is responsible for modulating the throttle (*ThrottleCmd*) using a PI algorithm while the cruise controller is on or matches the input accelerator pedal while the cruise controller is off.

3.2. Use of Data and Interfaces

Each module contains the data type declarations, functions, inputs and output interface signals. For our design, we have implemented the data and interfaces for each module in the following Figures:

```
%% Main Module
module cruiseControl:

input on, off, resume, quickAccel, quickDecel, set;
input speed: float, accel: float, brake: float;

output throttleCmd: float;
output cruiseSpeed: float;
output cruiseState: integer;
%// 1 - CruiseOff
%// 2 - CruiseOn
%// 3 - CruiseStndby
%// 4 - CruiseDisable

run MainFSM [
    signal on/on;
    signal off/off;
    signal resume/resume;
    signal speed/speed;
    signal accel/accel;
    signal brake/brake;

    signal cruiseState/cruiseState
]
||
run CruiseSpeedControl [
    signal quickAccel/sQuickAccel;
    signal quickDecel/sQuickDecel;
    signal set/sSet;
    signal cruiseState/sCruiseState;
    signal speed/sSpeed;

    signal cruiseSpeed/sCruiseSpeed
]
||
run ThrottleControl [
    signal accel/accel;
    signal speed/speed;
    signal cruiseSpeed/tcruiseSpeed;
    signal cruiseState/tcruiseState;

    signal throttleCmd/throttleCmd
]

end module
```

Figure 6: CruiseController Top-Level Module

Figure 6 is the Top-level module that encapsulates the entire system and clearly shows the 3 concurrent submodules. It should be noted that the output of the *MainFSM* is used as an input to *CruiseSpeedControl* and *ThrottleControl* via broadcast communication. This has also caused the naming of different signals to ensure that the correct signal from one module is linked properly to another. An example of this is using *CruiseState* from the output of *MainFSM* that feeds into *CruiseSpeedControl* and *ThrottleControl*. While in their submodule, they have been given different names (*sCruiseState* and *tcruiseState*) to ensure that the right signals are joined. Since the overall output state of the system is only determined by *MainFSM*, it can be directly linked to the output of the top-level module *CruiseController*.

```
%% MainFSM Module
module MainFSM:

input on, off, resume;
input speed: float, accel: float, brake: float;

output cruiseState: integer;

constant PedalsMin = 3.00f : float;
constant SpeedMin = 30.00f : float;
constant SpeedMax = 150.00f: float;
```

Figure 7: MainFSM Submodule Interface

```
%% CruiseSpeedControl Module
module CruiseSpeedControl:

input sQuickAccel, sQuickDecel, sSet;
input sCruiseState: integer;
input sSpeed: float;

output sCruiseSpeed: float;

constant SpeedMin = 30.00f : float;
constant SpeedMax = 150.00f: float;
constant SpeedInc = 10.00f  : float;
```

Figure 8: CruiseSpeedControl Submodule Interface

```
%% ThrottleControl Module
module ThrottleControl:

input accel: float, speed: float, tcruiseSpeed: float;
input tcruiseState: integer;

output throttleCmd: float;

function regulateThrottle(integer, float, float): float;
```

Figure 9: ThrottleControl Submodule Interface

3.3. Causality and Handling

During the development of the MainFSM, we encountered an issue relating to constructive causality. The design, as per Figure 10, determined states using control signals, (BrakeGood, SpeedGood, and AccelGood), which are computed in parallel to the MainFSM. Every state in MainFSM was guarded by these signals and logically always had a state to settle into. While this worked well in simulation and provided immediate state decisions, Esterel's validator reported a causality error because it could not guarantee a valid transition for every possible input combination. To resolve this, we removed a guard from the CruiseDisable state, making it so the FSM always has a state to land in, which satisfied formal validation. This change, however, introduced a minor side effect: transitions that would have moved directly from CruiseStndby to CruiseOn now pass through the CruiseDisable state first, resulting in an extra tick delay before reaching the intended state.

```
signal brakeGood, accelGood, speedGood in

    %// signal setting
    loop
        if (?brake < PedalsMin) then emit brakeGood; end if;
        if (?accel < PedalsMin) then emit accelGood; end if;
        if (?speed >= SpeedMin) and (?speed <= SpeedMax) then emit speedGood; end if;
        pause;
    end loop;

||
%// FSM
loop
    %// state 1 - CruiseOff
    abort
        sustain cruiseState(1)
    when on;

    abort
        loop

            %// state 2 - CruiseOn
            present [speedGood and accelGood and brakeGood] then
                abort
                    sustain cruiseState(2)
                when
                    case [not brakeGood] %// to STNDBY
                    case [not accelGood or not speedGood] %// to DISABLE
                end abort;
            end present;
            %// ----------------------

            %// state 3 - CruiseStndby
            present [not brakeGood] then
                abort
                    sustain cruiseState(3)
                when
                    case [resume and accelGood and speedGood] %// to On
                    case [resume and (not accelGood or not speedGood)] %// to DISABLE
                end abort;
            end present;
            %// ----------------------

            %// state 4 - CruiseDisable
            present [not speedGood or not accelGood] then
                abort
                    sustain cruiseState(4)
                when
                    case [not brakeGood] %// to STNDBY
                    case [accelGood and speedGood] %// to ON
                end abort;
            end present;
            %// ----------------------

        end loop;
    when off;
end loop;
end signal;
```

Figure 10: FSM with Causality Issue

4. Testing

Formal verification was performed on the Esterel code which initially yielded causality errors; however, after rectification via refactoring, the program passed with flying colours. Rigorous manual testing was performed for all state transitions and boundary cases, using provided test vectors and additional edge cases (e.g., speed at limits, pedals at thresholds). Output was compared with expected results. Testing results can be found in the Appendix below.

Acknowledgements

The authors would like to thank the supervisor Dr. Avinash Malik for teaching the basics of the Esterel language. The authors would like to thank Dr Zoran Salcic for providing a fundamental background in real-time embedded systems. Thank you to Dr Partha Roop for providing an example Esterel report. Also, thanks go to the TAs for providing excellent assistance.

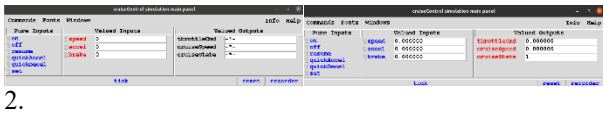
Appendix

Vector.in and Vector.out

On	Off	Resume	Set	QuickAccel	QuickDecel	Accel	Brake	Speed	CruiseSpeed	ThrottleCmd	CruiseState
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0	0	0	0	0	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0	0	0	0	0	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	51.234	0	0	0	51.234001	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	51.234	0	2.0937	0	51.234001	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	51.234	0	4.1862	0	51.234001	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	51.234	0	6.2772	0	51.234001	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	51.234	0	8.3661	0	51.234001	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	10.453	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	13.051	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	15.644	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	18.233	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	20.816	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	23.393	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	25.962	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	28.524	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	31.078	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	89.687	0	33.623	0	89.686996	1
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0	0	36.158	0	0	1
TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	0	0	36.049	36.049	0	2
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	0	0	35.94	36.049	0.938827	2

Our Results (skipping 6, 7 and, 10 through 16)

1.



2.

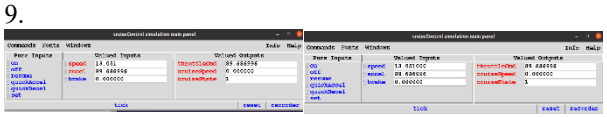
3.

4.

5.

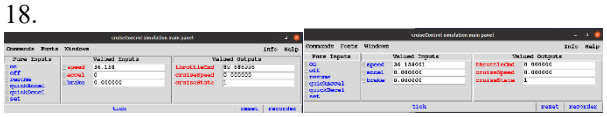
8.

9.

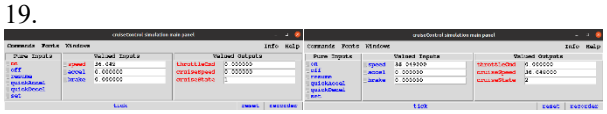


17.

18.



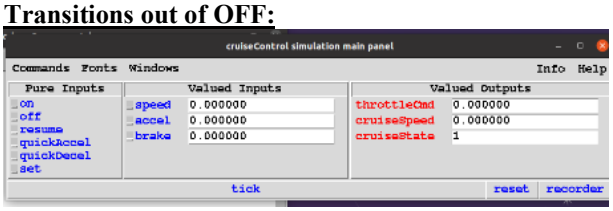
19.



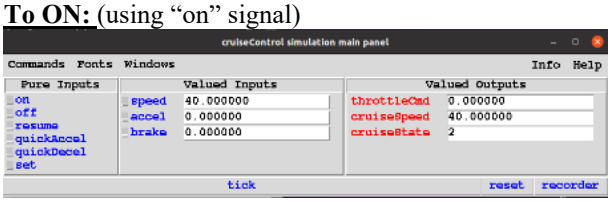
20.

Testing Transitions:

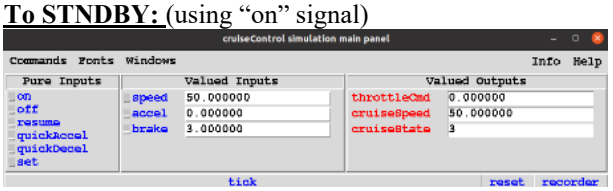
Transitions out of OFF:



To ON: (using "on" signal)

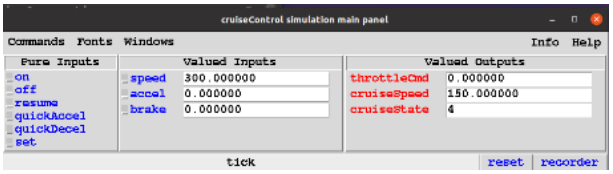


To STNDBY: (using "on" signal)

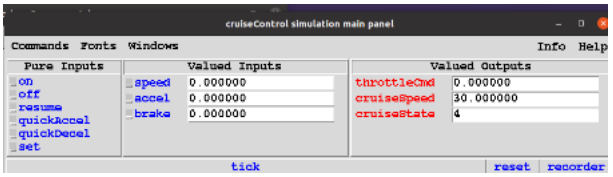


To DISABLE: (using "on" signal)

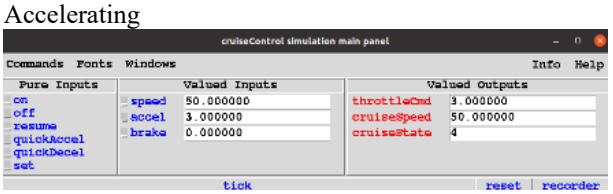
Too fast



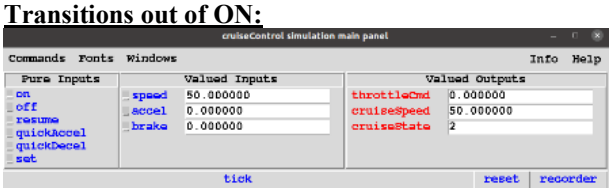
Too slow



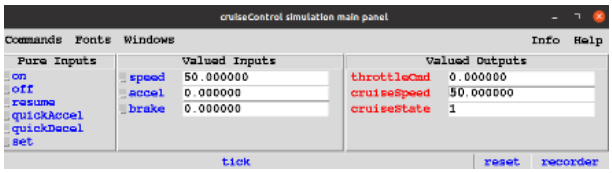
Accelerating



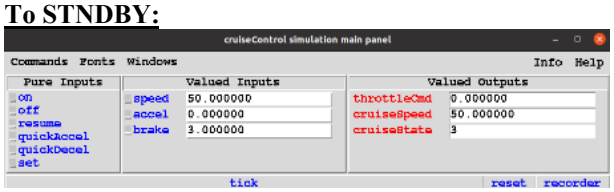
Transitions out of ON:



To OFF: (using "off" signal)

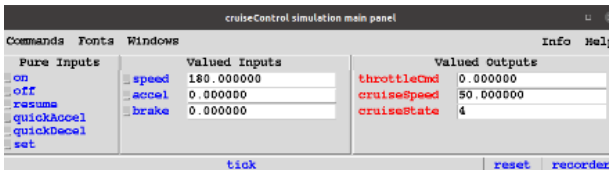


To STNDBY:

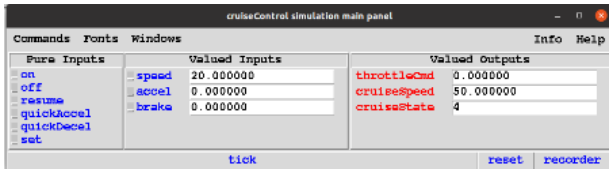


To DISABLE:

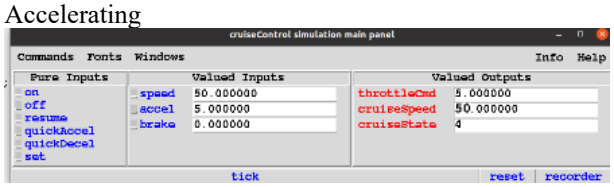
Too fast



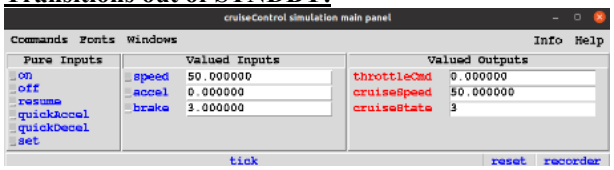
Too slow



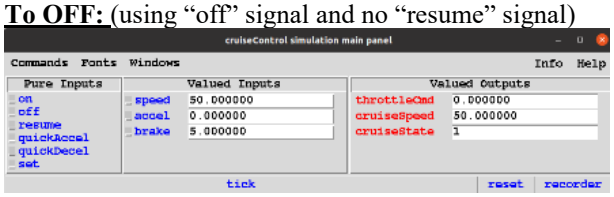
Accelerating



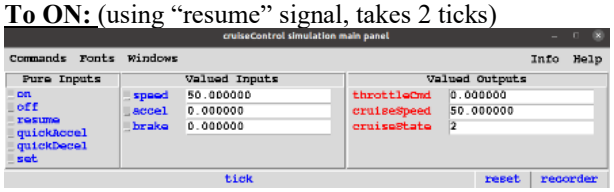
Transitions out of STNDBY:



To OFF: (using "off" signal and no "resume" signal)

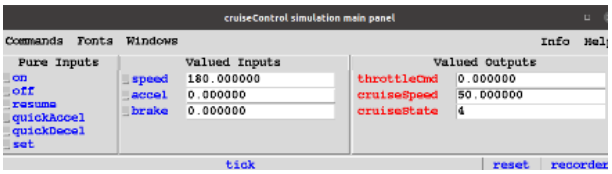


To ON: (using "resume" signal, takes 2 ticks)

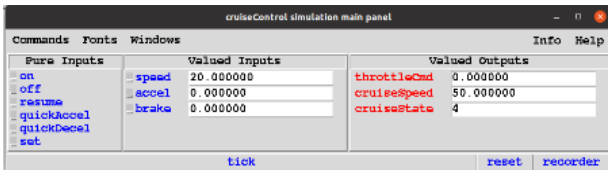


To DISABLE: (using "resume" signal)

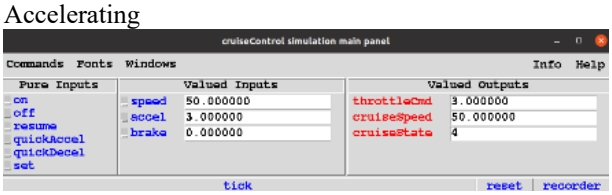
Too fast



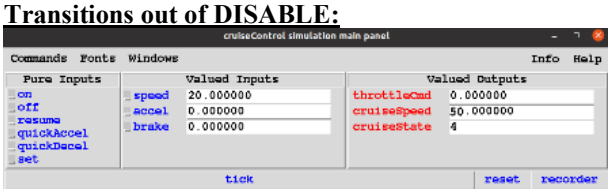
Too slow



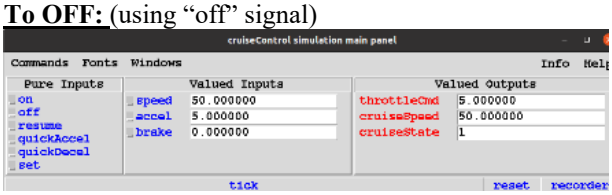
Accelerating



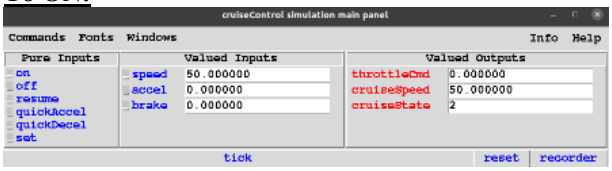
Transitions out of DISABLE:



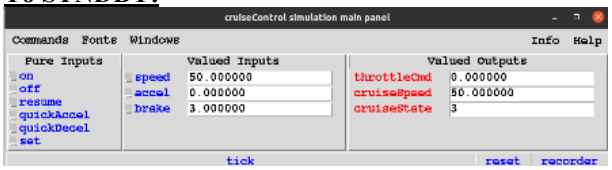
To OFF: (using "off" signal)



To ON:

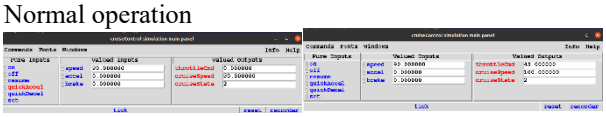


To STNDBY:



Testing QuickAccel:

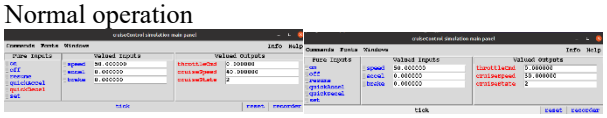
Normal operation



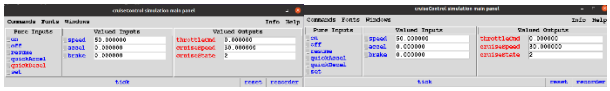
Too fast

Testing QuickDecel:

Normal operation

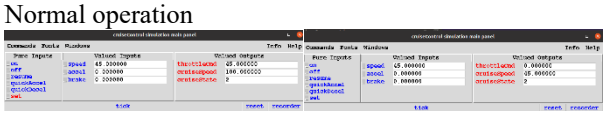


Too slow



Testing Set:

Normal operation



Too fast

Too slow