Unit 4

Documentation for Implementing Cruise Control and Emergency Brake in AUTOSAR

4. Solution Approach

Step 1 - Design Phase, SystemDesk

The design phase in **SystemDesk** involves specifying Software Components (SWCs), adding necessary ports, defining internal behaviors, and generating the Runtime Environment (RTE). Here's a breakdown of the process:

---

5. Tasks

5.1 Task 1: Setting up the simulation environment

Task Description:

The first task is to set up the simulation environment, where the **VEOS simulator** will interact with your system. Initially, the pedal states (AcceleratorPedal and BrakePedal) will be passed through to control the **Accelerate** and **Brake** signals.

Steps for Implementation:

1. **Open SystemDesk**:
    - Open **SystemDesk** and create a new project or use the provided template project for this task.
2. **Check Pedal State Pass-through**:
    - In the **AdasComposition**, the initial configuration will pass the pedal states (`AcceleratorPedal` and `BrakePedal`) directly to the **Accelerate** and **Brake** signals.
    - This will allow you to manually control the acceleration and braking using the **W** and **Space** keys in the **2D simulator**.
3. **Run the VEOS Simulation**:
    - Connect **VEOS** to the **2D simulator**.
    - Press **W** for acceleration and **Space** for braking to verify the car responds as expected.
4. **Wire Custom Logic**:

- o Once the connection is confirmed to be working, you will remove the default pass-through connections and replace them with your custom logic for Cruise Control and Emergency Brake.

5.2 Task 2: Implementing Cruise Control

Task Description:

In this task, you will implement the **Cruise Control** (CC) system step-by-step. **Cruise Control** will control the speed of the vehicle based on the target speed. The user can **control acceleration**, **braking**, and adjust the **target speed**.

5.2.1 Sub-Task 1: Basic CC Logic with Constant Target Speed

1. **SystemDesk Configuration:**
   - o **SWC Creation**: Add a new **CruiseControlSWC** inside **AdasComposition**. This SWC will handle controlling the car's speed.
   - o **Ports**:
     - ▪ **CurrentSpeed**: To read the car's current speed from the simulator.
     - ▪ **AcceleratorPedal** and **BrakePedal**: To control acceleration and braking.
     - ▪ **TargetSpeed**: To set the target speed (e.g., 50 km/h).

   Example configuration of ports in **SystemDesk**:

   - o Create a **Sender-Receiver** interface for **AcceleratorPedal** and **BrakePedal** to control the accelerator and brake inputs from the simulator.
   - o Create **RPorts** for reading and writing data, such as **CurrentSpeed**, **TargetSpeed**, etc.
2. **Generate the RTE**:
   - o Once the SWC is created and ports are defined, generate the **RTE** (Runtime Environment) by creating a **V-ECU** (Virtual ECU).
   - o This will automatically generate **Rte_IRead** and **Rte_IWrite** methods that will be used in your C code to interact with the ports.
3. **Code Implementation in Visual Studio Code**:

   After generating the RTE, open the **Visual Studio Code** project and implement the logic for **CruiseControlSWC**.

   Code for CruiseControlSWC.c:

```
#include <Rte_CruiseControlSWC.h>
#include <Sab.h>
#include <Sab_Types.h>

void CruiseControlSWC_CruiseControlLogic()
```

```
{
    // Get the current speed and target speed
    float32 currentSpeed =
Rte_IRead_CruiseControlSWC_CurrentSpeed_Value();
    float32 targetSpeed = Rte_IRead_CruiseControlSWC_TargetSpeed_Value();

    // Basic CC logic to maintain a constant speed
    if (currentSpeed < targetSpeed) {
        Rte_IWrite_CruiseControlSWC_Accelerate_Value(1); // Accelerate
        Rte_IWrite_CruiseControlSWC_Brake_Value(0);     // No brake
    } else if (currentSpeed > targetSpeed) {
        Rte_IWrite_CruiseControlSWC_Accelerate_Value(0); // No
acceleration
        Rte_IWrite_CruiseControlSWC_Brake_Value(1);     // Brake
    } else {
        Rte_IWrite_CruiseControlSWC_Accelerate_Value(0); // No
acceleration
        Rte_IWrite_CruiseControlSWC_Brake_Value(0);     // No brake
    }
}
```

Verification:

- Run the **VEOS simulation** and ensure the car accelerates to 50 km/h and maintains this speed.

5.2.2 Sub-Task 2: Enable/Disable CC

1. **Enable/Disable Logic**:
    - Add a new port **CC_State** in the **CruiseControlSWC** to control whether the Cruise Control is enabled or disabled using the **C** key in the simulator.
2. **RTE Configuration**:
    - Modify the logic to check the state of **CC_State** (if **CC** is disabled, no acceleration or braking should occur automatically).
3. **Code Implementation in Visual Studio Code**:

    Code for CruiseControlSWC.c:

```
#include <Rte_CruiseControlSWC.h>
#include <Sab.h>
#include <Sab_Types.h>

void CruiseControlSWC_CruiseControlStateLogic()
{
    boolean ccState = Rte_IRead_CruiseControlSWC_CC_State_Value(); //
Read CC state

    if (ccState == 1) {
        // If CC is enabled, regulate the speed
        CruiseControlSWC_CruiseControlLogic();
```

```
    } else {
        // If CC is disabled, let the manual inputs take over
        Rte_IWrite_CruiseControlSWC_Accelerate_Value(0);
        Rte_IWrite_CruiseControlSWC_Brake_Value(0);
    }

    // Output the new CC state
    Rte_IWrite_CruiseControlSWC_CC_StateNew_Value(ccState);
}
```

Verification:

- Press the **C** key in the simulator to toggle the **CC_State**. Test that the car accelerates when **CC** is on and allows manual control when **CC** is off.

5.2.3 Sub-Task 3: Manual Acceleration/Braking

1. **Manual Control Logic**:
   - If **CC** is disabled, the car should directly respond to **AcceleratorPedal** and **BrakePedal** inputs.
   - If **CC** is enabled, pressing the accelerator should override the target speed.
2. **Code Implementation in Visual Studio Code**:

```
#include <Rte_CruiseControlSWC.h>

#include <Sab.h>

#include <Sab_Types.h>


void CruiseControlSWC_CruiseControl(void)

{

    // Get current speed and target speed

    float32 currentSpeed = Rte_IRead_CruiseControl_CurrentSpeed_Value();

    float32 targetSpeed = Rte_IRead_CruiseControl_TargetSpeed_Value();  // Target speed set to 50 km/h

    boolean ccState = Rte_IRead_CruiseControl_CC_State_Value();  // Read Cruise Control state

    boolean AcceleratorPedal = Rte_IRead_CruiseControl_AcceleratorPedal_Value();

    boolean brake = Rte_IRead_CruiseControl_BreakNew_Value();


    if (ccState)
```

```
{
    if (AcceleratorPedal)
        {
            Rte_IWrite_CruiseControl_Accelerate_Value(1);
            Rte_IWrite_CruiseControl_BrakePedal_Value(0);
        } else if(brake)
            {
            ccState = 0;
            }  else
                {
                if (currentSpeed < targetSpeed - 1.0f)  {
                        Rte_IWrite_CruiseControl_Accelerate_Value(1);
                        Rte_IWrite_CruiseControl_BrakePedal_Value(0);
                    } else if (currentSpeed > targetSpeed + 1.0f) {
                        Rte_IWrite_CruiseControl_Accelerate_Value(0);
                        Rte_IWrite_CruiseControl_BrakePedal_Value(1);
                    } else {
                        Rte_IWrite_CruiseControl_Accelerate_Value(1);
                        Rte_IWrite_CruiseControl_BrakePedal_Value(0);
                    }
        }       }
                    else {
                Rte_IWrite_CruiseControl_Accelerate_Value(AcceleratorPedal);
                Rte_IWrite_CruiseControl_BrakePedal_Value(brake);
                    }

  Rte_IWrite_CruiseControl_CC_State_New_Value(ccState);
}
```

Verification:

- o Use the **W** and **Space** keys to manually control the car, ensuring that **CC** is overridden when the accelerator is pressed.

5.2.4 Sub-Task 4: Manual Definition of Target Speed

1. **Manual Speed Adjustment**:
   - o Add a **TargetSpeedButtonState** port to increase or decrease the target speed using the **Arrow Up** and **Arrow Down** keys in the simulator.
2. **Code Implementation in Visual Studio Code**:

```
#include <Rte_TargetSpeedControl.h>

#include <Sab.h>

#include <Sab_Types.h>


static float32 targetSpeed = 10.0;

static uint8 prevBtnState = 0;  // Variable to track the previous state of the button


void TargetSpeedControl_Target(void)

{

  uint8 btn = Rte_IRead_Target_ButtonState_Value();


  // Check if the button is pressed and was previously not pressed (button press event)

  if (btn == 1 && prevBtnState == 0 && targetSpeed <= 125.0f) {

    targetSpeed += 5.0f;

    Sab_SubmitInfo("TargetSpeed = %.1f", targetSpeed);

  }

  // Check if the button is pressed and was previously not pressed (button press event)

  else if (btn == 2 && prevBtnState == 0 && targetSpeed >= 15.0f) {

    targetSpeed -= 5.0f;

  }
```

```
  // Update the previous button state for the next cycle
  prevBtnState = btn;

  Rte_IWrite_Target_TargetSpeed_Value(targetSpeed);
}
```

Verification:

- Press **Arrow Up** and **Arrow Down** in the simulator to adjust the **TargetSpeed** and verify that the car accelerates/decelerates accordingly.

5.3 Task 3: Implementing Emergency Brake

Steps for Implementation:

1. **Emergency Brake Logic**:
   - Implement the logic for emergency braking based on the **distance** to a leading vehicle.
   - Use the formula for braking distance:

     dbrake=current speed22×decelerationd_{brake} = \frac{{\text{{current speed}}^2}}{{2 \times \text{{deceleration}}}}

   - Trigger the emergency brake if the car is too close to an obstacle.
2. **Code Implementation in Visual Studio Code**:

   Code for EmergencyBrakeSWC.c:

```
#include <Rte_Emergency_Brake.h>
void Emergency_Brake_Emergency()
{
  float32 speed = Rte_IRead_Emergency_CurrentSpeed_Value(); // km/h
  float32 distance = Rte_IRead_Emergency_DistanLeading_Value(); // meters
  boolean ccState = Rte_IRead_Emergency_CurrentSpeed_Value();

  float32 speed_mps = speed / 3.6f;
```

```
  float32 d_brake = (speed_mps * speed_mps) / (2.0f * 14.0f); // deceleration = 14 m/s^2

  if (distance < d_brake + 3.0f) {
      Rte_IWrite_Emergency_CC_State_Value(1);
      Rte_IWrite_Emergency_CC_State_New_Value(0); // deactivate CC
  } else {
      Rte_IWrite_Emergency_CC_State_Value(0);
      Rte_IWrite_Emergency_CC_State_New_Value(ccState); // preserve state
  }
}
```

Verification:

- Test the **Emergency Brake** functionality by spawning a leading vehicle and verifying that the car slows down when the distance is too short.

Conclusion

Each task is designed to build incrementally from basic functionality (Cruise Control with a fixed target speed) to more advanced features (manual override, target speed adjustment, and emergency braking). Following this step-by-step process ensures that each sub-task functions as expected before moving to the next. The solution uses **SystemDesk** for configuration, **Visual Studio Code** for implementation, and **VEOS** for testing the functionality in the simulator.