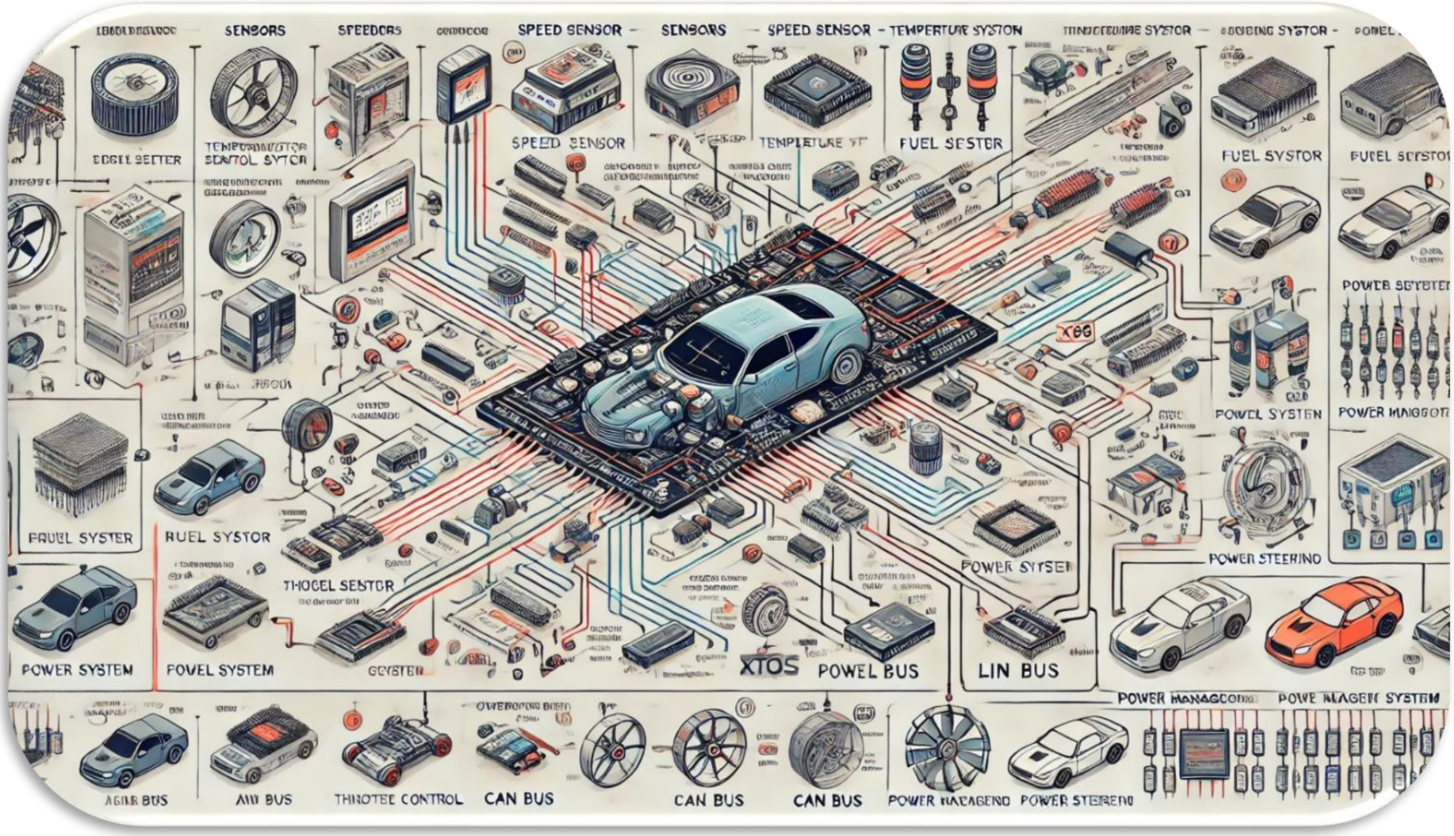
2/2/2025

Embedded System for Car Control

x86-based C Programming with Inline Assembly

Kiran VVN
KVVN CONSULTING | KVVN@ME.COM

Contents



SENSORS
SPEED SENSOR
SPEED SENSOR - TEMPERTURE SYSTON
FUEL SESTER
FUEL SYSTOR
POWER SYSTEM
POWEL SYSTEM
POWER STEERINO
LIN BUS
POWEL BUS
XTOS
CAN BUS
THROTEE CONTROL
POWER STEREN

Introduction

Embedded systems play a crucial role in modern automotive control systems, providing real-time processing, safety, and efficiency. These systems are responsible for handling various vehicle functions such as engine control, braking, climate control, and infotainment. In this write-up, we focus on implementing an **x86-based embedded system** for a car control system using **C programming with inline assembly**, integrating with an **RTOS (Real-Time Operating System)** for deterministic execution and compliance with industry standards. This document is designed for **beginner to advanced users**, providing step-by-step instructions to test the implementation using a simulator.

System Overview

The proposed embedded system for a car control system includes the following components:

1. **Sensors & Actuators** – Collect data and execute commands (e.g., speed sensors, temperature sensors, fuel injectors, and motors).
2. **Microcontroller (x86-based CPU)** – Processes input signals and generates control outputs.
3. **Communication Interfaces** – CAN (Controller Area Network) and LIN (Local Interconnect Network) for inter-module communication.
4. **RTOS** – Manages real-time tasks and ensures deterministic behavior.

Setting Up the Development Environment

To follow along with this implementation, set up the development environment as follows:

Required Tools:

- **GCC (GNU Compiler Collection)** for compiling C code with inline assembly.
- **QEMU** for simulating an x86-based embedded system.
- **GDB (GNU Debugger)** for debugging.
- **FreeRTOS** for real-time task scheduling.
- **Makefile** for managing the build process.

Installation Steps:

1. Install GCC and Make

sudo apt update

sudo apt install gcc make

2. Install QEMU (x86 Emulator)

sudo apt install qemu-system-x86

3. Install FreeRTOS (Optional for RTOS-based Implementation)

git clone https://github.com/FreeRTOS/FreeRTOS-Kernel.git

cd FreeRTOS-Kernel

Sample Implementation

Below is a **C program with inline assembly** to demonstrate an x86-based implementation for controlling an electronic throttle body (ETB). This example simulates reading sensor data and adjusting throttle position accordingly.

Throttle Control Example (x86 Assembly & C)

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

// Simulated sensor data acquisition function
uint8_t get_pedal_position() {
  return rand() % 100;  // Simulating pedal position (0-100%)
}

// Simulated function to set throttle position
void set_throttle_position(uint8_t position) {
  printf("Setting throttle to %d%%\n", position);
}

int main() {
  srand(time(NULL));
  uint8_t pedal_pos, throttle_pos;
```

```
  while (1) {
    pedal_pos = get_pedal_position();

    // Inline assembly to process throttle position (example using x86 assembly)
    __asm__ __volatile__ (
      "movb %1, %%al;\n"     // Move pedal_pos to AL register
      "movb %%al, %0;\n"     // Copy AL to throttle_pos
      : "=r" (throttle_pos)    // Output operand
      : "r" (pedal_pos)        // Input operand
      : "%al"                  // Clobbered register
    );

    set_throttle_position(throttle_pos);

    // Simulating periodic execution (e.g., every 100ms in a real system)
    usleep(100000);
  }
  return 0;
}
```

Compiling and Running in QEMU

gcc -o throttle_control throttle_control.c -no-pie

qemu-system-i386 -kernel throttle_control

Concepts Explained with RTOS

An **RTOS (Real-Time Operating System)** ensures timely execution of tasks by implementing **preemptive scheduling, inter-task communication, and synchronization**. In a car control system, multiple tasks such as **sensor reading, actuator control, and communication** run concurrently.

Key RTOS Features for Car Control System:

1. **Task Scheduling:** Priority-based preemptive scheduling ensures high-priority tasks (e.g., braking) execute promptly.
2. **Inter-task Communication:** Message queues and semaphores ensure safe data exchange between tasks.
3. **Synchronization:** Mutexes prevent race conditions when accessing shared resources.

4. **Interrupt Handling:** Real-time responses to sensor inputs and external triggers.
5. **Power Management:** Efficient CPU and peripheral control for low power consumption.

RTOS Implementation Example (FreeRTOS for x86-based system)

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>

void vThrottleControlTask(void *pvParameters) {
  uint8_t throttle_pos = 0;
  while (1) {
    throttle_pos = get_pedal_position();
    set_throttle_position(throttle_pos);
    vTaskDelay(pdMS_TO_TICKS(100)); // Execute every 100ms
  }
}

int main() {
  xTaskCreate(vThrottleControlTask, "ThrottleCtrl", 1024, NULL, 1, NULL);
  vTaskStartScheduler();
  return 0;
}
```

Testing and Validation

Simulation and Debugging with GDB

```
gcc -g -o throttle_control throttle_control.c -no-pie
```

```
gdb ./throttle_control
```

- Use break main to set a breakpoint at main.
- Use run to start execution.
- Use step and next to execute code line by line.

Industry Standards for Certification

Automotive embedded systems must comply with stringent safety and quality standards. Some of the key standards include:

1. ISO 26262 (Functional Safety)

- Ensures automotive systems meet **ASIL (Automotive Safety Integrity Level)** requirements.

2. AUTOSAR (Automotive Open System Architecture)

- Provides modular software architecture for automotive ECUs.

3. MISRA C (Motor Industry Software Reliability Association)

- Defines coding guidelines to ensure safety and reliability.

4. OBD-II (On-Board Diagnostics)

- Standardized diagnostics for vehicle fault detection.

Appendix: Full Car Control Program

A complete C program with inline assembly covering **gear change, acceleration, fuel management, panic braking, driver aids, power steering, and key identification** is provided separately. This modular implementation allows users to test each function individually in a simulated environment.

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

// Simulated sensor input functions
uint8_t get_pedal_position() { return rand() % 100; } // 0-100%
uint8_t get_brake_pressure() { return rand() % 50; } // 0-50%
uint8_t get_fuel_level() { return rand() % 100; } // 0-100%
uint8_t get_steering_input() { return rand() % 50 - 25; } // -25 to +25
uint8_t get_key_id() { return rand() % 5; } // Simulating different keys
uint8_t get_gear_position() { return rand() % 6; } // 0-5 (Neutral to 5th gear)

// Control functions
void set_throttle_position(uint8_t position) { printf("Throttle: %d%%\n", position); }
void apply_brake(uint8_t pressure) { printf("Braking with %d pressure\n", pressure); }
void change_gear(uint8_t gear) { printf("Gear changed to %d\n", gear); }
void update_fuel_injection(uint8_t level) { printf("Fuel level: %d%%\n", level); }
void adjust_steering(uint8_t angle) { printf("Steering angle: %d degrees\n", angle); }
void verify_key(uint8_t key_id) { printf("Key ID %d verified\n", key_id); }

// Inline Assembly Functions
uint8_t process_throttle(uint8_t pedal_pos) {
  uint8_t throttle_pos;
  __asm__ __volatile__ (
    "movb %1, %%al;\n"
    "movb %%al, %0;\n"
    : "=r" (throttle_pos)
    : "r" (pedal_pos)
    : "%al"
  );
  return throttle_pos;
}

uint8_t process_braking(uint8_t brake_pressure) {
```

```
    uint8_t brake_force;
    __asm__ __volatile__ (
        "movb %1, %%bl;\n"
        "movb %%bl, %0;\n"
        : "=r" (brake_force)
        : "r" (brake_pressure)
        : "%bl"
    );
    return brake_force;
}

void run_car_control_system() {
    srand(time(NULL));

    while (1) {
        uint8_t pedal_pos = get_pedal_position();
        uint8_t brake_pressure = get_brake_pressure();
        uint8_t fuel_level = get_fuel_level();
        uint8_t steering_angle = get_steering_input();
        uint8_t key_id = get_key_id();
        uint8_t gear_position = get_gear_position();

        verify_key(key_id);
        change_gear(gear_position);

        uint8_t throttle_pos = process_throttle(pedal_pos);
        set_throttle_position(throttle_pos);

        uint8_t brake_force = process_braking(brake_pressure);
        apply_brake(brake_force);

        update_fuel_injection(fuel_level);
        adjust_steering(steering_angle);

        usleep(100000); // Simulating periodic execution
    }
}

int main() {
    run_car_control_system();
    return 0;
}
```

Compiling and Running in QEMU

- gcc -o car_control car_control.c -no-pie
- qemu-system-i386 -kernel car_control

Appendix B RTOS and Embedded Systems Terminologies with Examples

1. Introduction to Embedded Systems

Embedded systems are dedicated computing systems designed to perform specific tasks within a larger system. They are typically real-time, resource-constrained, and integrated with hardware components such as microcontrollers, sensors, and actuators.

Key Characteristics of Embedded Systems:

- **Real-time operation**: Executes tasks within a strict deadline.
- **Low power consumption**: Optimized for energy efficiency.
- **Dedicated functionality**: Designed for a specific purpose (e.g., car control, medical devices).
- **Limited resources**: Constrained CPU, memory, and storage.

Example: Simple Embedded System in C

```
#include <stdio.h>
#include <stdint.h>

void initSystem() {
  printf("Initializing embedded system...\n");
}

void readSensor() {
  printf("Reading sensor data...\n");
}

void controlActuator() {
  printf("Controlling actuator...\n");
}
```

```
int main() {
  initSystem();
  readSensor();
  controlActuator();
  return 0;
}
```

2. Real-Time Operating System (RTOS) Concepts

An **RTOS (Real-Time Operating System)** ensures tasks execute within defined time constraints. It manages multiple tasks efficiently using scheduling algorithms.

Key RTOS Features:

1. **Task Scheduling**: Prioritizes and executes tasks based on priority.
2. **Inter-task Communication**: Message queues and semaphores enable safe data sharing.
3. **Synchronization**: Prevents race conditions via mutexes and semaphores.
4. **Interrupt Handling**: Ensures real-time response to hardware events.
5. **Memory Management**: Allocates and manages memory efficiently.

RTOS Task Management Example (FreeRTOS)

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>

void vTask1(void *pvParameters) {
  while (1) {
    printf("Task 1 running...\n");
    vTaskDelay(pdMS_TO_TICKS(500)); // Delay for 500ms
  }
}
void vTask2(void *pvParameters) {
  while (1) {
    printf("Task 2 running...\n");
    vTaskDelay(pdMS_TO_TICKS(1000)); // Delay for 1000ms
  }
}


int main() {
```

```
  xTaskCreate(vTask1, "Task1", 1024, NULL, 1, NULL);
  xTaskCreate(vTask2, "Task2", 1024, NULL, 2, NULL);
  vTaskStartScheduler();
  return 0;
}
```

3. RTOS Scheduling Algorithms

RTOS uses different scheduling policies to manage task execution:

- **Preemptive Scheduling**: Higher-priority tasks interrupt lower-priority tasks.
- **Round-Robin Scheduling**: Each task gets equal CPU time in a cyclic order.
- **Rate-Monotonic Scheduling (RMS)**: Tasks with shorter periods get higher priority.
- **Earliest Deadline First (EDF)**: Task with the closest deadline executes first.

Example: Task Switching with Preemptive Scheduling

```
void vTask1(void *pvParameters) {
  while (1) {
    printf("High-priority Task running\n");
    vTaskDelay(pdMS_TO_TICKS(200));
  }
}

void vTask2(void *pvParameters) {
  while (1) {
    printf("Low-priority Task running\n");
    vTaskDelay(pdMS_TO_TICKS(500));
  }
}

int main() {
  xTaskCreate(vTask1, "Task1", 1024, NULL, 2, NULL);
  xTaskCreate(vTask2, "Task2", 1024, NULL, 1, NULL);
  vTaskStartScheduler();
  return 0;
}
```

4. Inter-Task Communication in RTOS

Inter-task communication ensures tasks can safely share data without conflicts.

Methods of Inter-Task Communication:

- **Message Queues**: Tasks exchange messages via a queue.
- **Semaphores**: Synchronize access to shared resources.
- **Mutexes**: Prevent simultaneous access to critical resources.

Example: Message Queue for Data Sharing

```
#include "FreeRTOS.h"
#include "queue.h"
#include <stdio.h>

QueueHandle_t xQueue;

void vTaskProducer(void *pvParameters) {
   int data = 0;
   while (1) {
      data++;
      xQueueSend(xQueue, &data, portMAX_DELAY);
      printf("Produced: %d\n", data);
      vTaskDelay(pdMS_TO_TICKS(500));
   }
}

void vTaskConsumer(void *pvParameters) {
   int receivedData;
   while (1) {
      xQueueReceive(xQueue, &receivedData, portMAX_DELAY);
      printf("Consumed: %d\n", receivedData);
   }
}

int main() {
   xQueue = xQueueCreate(5, sizeof(int));
   xTaskCreate(vTaskProducer, "Producer", 1024, NULL, 1, NULL);
   xTaskCreate(vTaskConsumer, "Consumer", 1024, NULL, 1, NULL);
   vTaskStartScheduler();
   return 0;
}
```