CAN Temperature Monitor

CAN0 Internal Loopback on TM4C123GH6PM

ECE-4721: Embedded Systems

Peter Younan, Andrew Kashat, Yousif Fatohi

Oakland University

Department of Electrical and Computer Engineering

Winter 2026

Table of Contents

1. Abstract

This paper details the design of a CAN Temperature Monitor whereby the TM4C123GH6PM microcontroller in the Tiva C LaunchPad is programed to read the temperature value from the temperature sensor inside the chip using the ADC Channel. The data from the sensor is packaged and then transmitted as a standard CAN 2.0A message frame (ID=0x100) over the CAN0 interface at a speed of 500 kbps. Using the loopback test feature of the CAN0 hardware module, the microcontroller can transmit its message and also receive the same in order to perform both tests with just a single board without requiring another CAN node. The CAN transceiver MCP2551 helps the microcontroller communicate to the CAN bus, while the UART0 hardware module communicates with the computer serial port (PuTTY) via the ICDI USB debug port.

2. Introduction to CAN and CAN Shield

2.1 What is CAN?

Controller Area Network is a powerful serial protocol designed by Bosch in the 1980s specifically for vehicle use. CAN protocol provides a way for multiple microprocessors and other peripheral devices to exchange information among one another on a common bus without using a master host computer. In CAN communication system, any node can become a master since it is multi-master and message-oriented. Thus, a message does not identify a sender or receiver but its content that corresponds to a certain message ID.

The CAN protocol relies on differential mode transmission using two-wire bus (CAN_H and CAN_L). This ensures good resistance to noise, a crucial feature of the CAN protocol due to its usage in noisy environments like vehicles, industry machines, or medical equipment. CAN supports data rates up to 1 Mbps with various built-in fault detection features, including CRC check, bit stuffing, acknowledgment checking, and automatic frame transmission.

In standard CAN 2.0A protocol, every message type is uniquely identifiable by means of 11-bit identifier, providing for up to 2048 message types per bus. Maximum number of data items per CAN frame equals 8 bytes. Message priority is defined by message identifier number; lower values correspond to higher priorities. Arbitration is non-destructive and performed bitwise.

2.2 CAN on the TM4C123

The TM4C123GH6PM features two CAN controllers named CAN0 and CAN1. Each of these controllers has 32 message objects that can be individually configured as TX or RX channels, hardware acceptance filters based on ID masks, and can utilize both 11- and 29-bit identifiers. CAN0 is selected in this project to communicate using PE4 (CAN0RX) and PE5 (CAN0TX) pinout as alternate functions 8.

2.3 MCP2551 CAN Transceiver (CAN Shield)

The CAN module in the TM4C123 generates signals at the logical level. A transceiver should be added to enable communication with the actual CAN bus. This project employs the MCP2551 CAN transceiver to convert the single-ended TX and RX lines of the microcontroller into CAN_H and CAN_L lines in the differential two-wire bus. MCP2551 is capable of achieving up to 1 Mbps speed and conforms to ISO 11898 standard. On the CAN shield, the MCP2551 is wired with TXD to PE5 and RXD to PE4. There are 120-ohm resistors for the bus termination.

2.4 Loopback Mode

Loopback feature is available for the CAN controllers of the TM4C123 microcontroller. It can be activated using bit 4 (Lback) of the CANTEST register. In loopback mode, the TX output of the CAN controller is connected to its own RX input. Consequently, all messages sent out from the controller are also received by the controller in question, making it possible to perform TX/RX testing even without a secondary CAN module. Loopback does not turn off the transceiver; therefore, messages can be seen by external CAN nodes and CAN analyzers.

3. Hardware Detail

3.1 Components

Table 1. List of components used in the system.

Component	Description
TM4C123GH6PM LaunchPad	ARM Cortex-M4F MCU, operating at 80 MHz, 256 KB Flash, and 32 KB SRAM
MCP2551 CAN Transceiver	CAN transceiver that is compatible with ISO 11898 with a speed of 1 Mbps
CAN Shield / Breakout	PCB with MCP2551 with 120Ω terminating resistor and screw terminals for CAN_H/CAN_L
USB Cable (Micro-B)	Powering board and providing UART0 debugging through on-board ICDI
Jumper Wires	Connections between LaunchPad GPIO and CAN shield header

3.2 Pin Assignments

Table 2. Pin assignments between microcontroller and peripherals.

Signal	Pin	Destination	Function
CAN0 TX	PE5	MCP2551 TXD	Alternate Function 8
CAN0 RX	PE4	MCP2551 RXD	Alternate Function 8
UART0 TX	PA1	ICDI USB → PC	Alternate Function 1
UART0 RX	PA0	ICDI USB → PC	Alternate Function 1

3.3 Circuit Description

TM4C123 LaunchPad communicates with the CAN transceiver shield MCP2551 using two GPIO pins; PE5 communicates to the transceiver's TXD pin, while PE4 gets its signal from the transceiver's RXD output. The transceiver then translates these signals to the differential CAN_H and CAN_L levels. The CAN shield also has a 120 ohm resistor as its bus termination resistor.

For UART0 communication, TXD and RXD are PA1 and PA0 respectively. These two are connected via ICDI circuitry to the USB port so that the same USB can be used for both programming and debugging purposes, and there is no need for an additional USB-to-UART converter. PuTTY on the PC is configured to connect to the Stellaris Virtual Serial Port using a baud rate of 9600.

As mentioned above, there is no need for additional hardware wiring because this temperature sensor is integrated in the TM4C123 die and is accessed from ADC0 by configuring sequencer configuration registers.

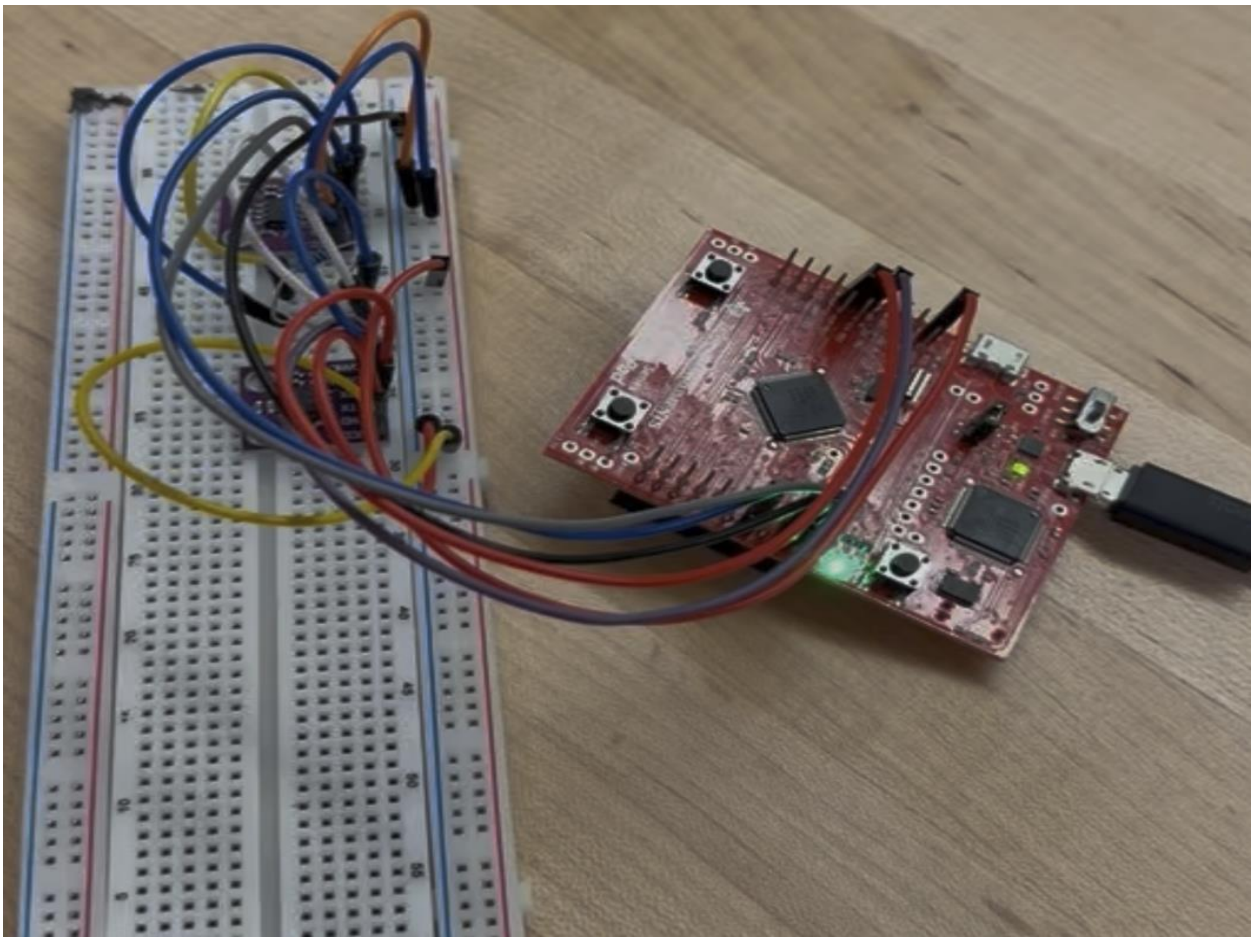

Figure 1. Hardware setup of the CAN temperature monitoring system.

4. Software Flowchart

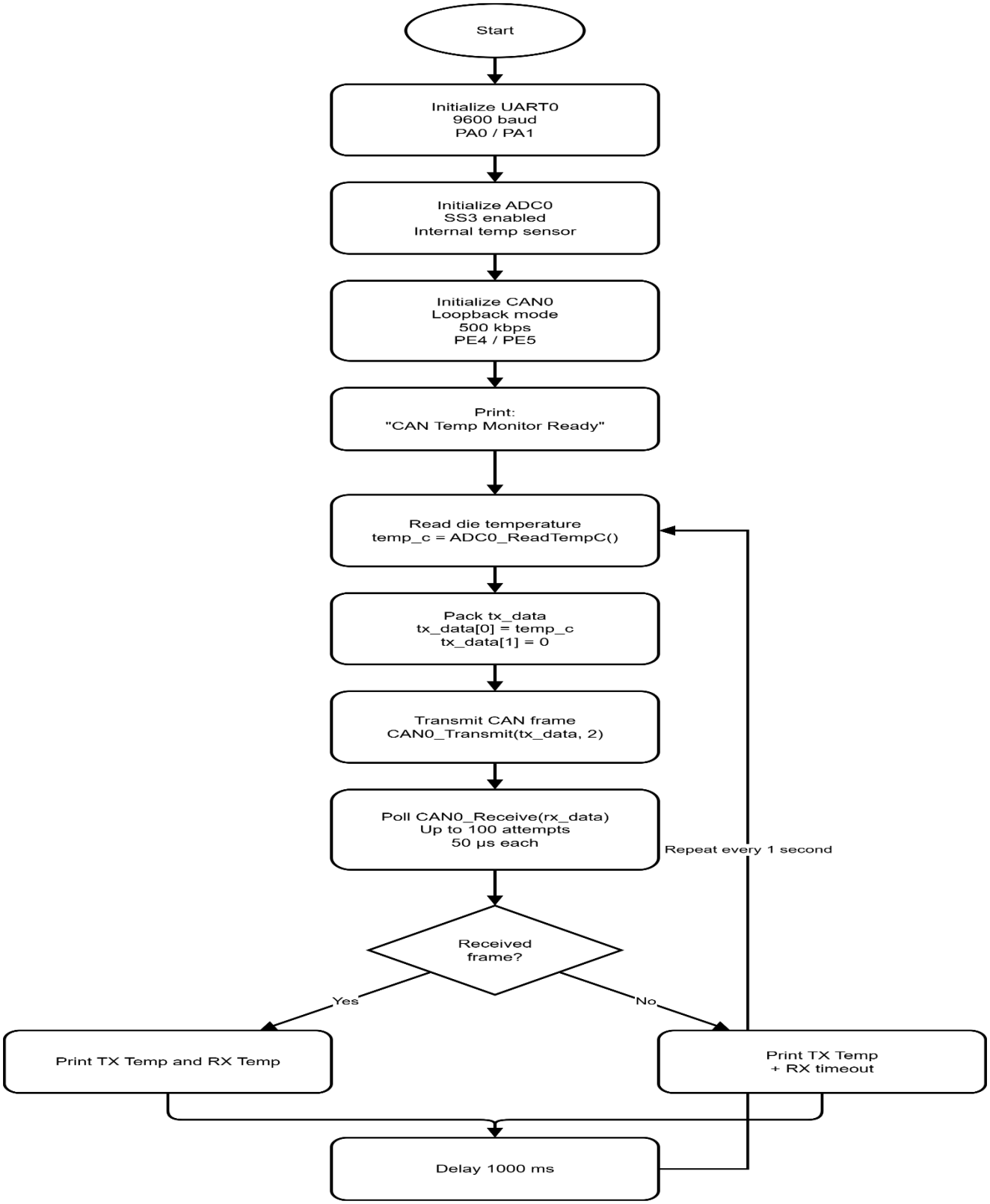

Figure 2. Software flowchart for system operation and CAN communication.

4.1 Initialization Details

- ADC0: Clock control, single-sample, internal temperature sensor selection (TS0), sequencer enable.
- CAN0: Clock control, PE4/PE5 AF8, init mode (INIT + CCE + TEST), loopback bit enable in CANTEST, bit timing configuration for 500 kbps (BRP = 1, TSEG1 = 11, TSEG2 = 4), end init mode, config. message object 1 (TX, ID 0x100, DLC = 2), message object 2 (RX, ID 0x100).
- UART0: Clock control, PA0/PA1 AF1, 9600 baud rate (IBRD = 104, FBRD = 11), 8N1 frame, TX/RX enable.

5. Full Code with Comments

The complete source code is provided below. Each function is self-contained and thoroughly commented to explain the register-level operations.

```
/* CAN Temperature Monitor - CAN0 Internal Loopback */
/* TM4C123GH6PM | ECE-4721 | Peter Younan, Andrew Kashat, Yousif
Fatohi | 2026  */

#include <stdio.h>
#include <stdint.h>
#include "TM4C123.h"
#include "inc/tm4c123gh6pm.h"

#define CAN_TEMP_ID  0x100U

/* --- Prototypes --- */
void    ADC0_Init(void);
int32_t ADC0_ReadTempC(void);
void    CAN0_Init(void);
void    CAN0_Transmit(uint8_t *data, uint8_t len);
uint8_t CAN0_Receive(uint8_t *data);
void    Uart0_Init(void);
void    uart0_send(uint8_t ch);
void    delay_us(int us);
void    delay_ms(int ms);

/* ---- MAIN ---- */
int main(void) {
    uint8_t tx_data[2], rx_data[2];
    int32_t temp_c;

    Uart0_Init();
    ADC0_Init();
```

```
    CAN0_Init();
    printf("CAN Temp Monitor Ready\r\n");

    while (1) {
        temp_c = ADC0_ReadTempC();          // 1. Read temp
        tx_data[0] = (uint8_t)temp_c;
        tx_data[1] = 0;
        CAN0_Transmit(tx_data, 2);          // 2. Transmit

        uint8_t received = 0;               // 3. Poll RX
        for (int t = 0; t < 100; t++) {
            if (CAN0_Receive(rx_data)) { received = 1; break; }
            delay_us(50);
        }

        if (received)                       // 4. Print
            printf("TX Temp=%d C | RX Temp=%d C\r\n",
                   (int)temp_c, (int8_t)rx_data[0]);
        else
            printf("TX Temp=%d C | RX timeout\r\n", (int)temp_c);

        delay_ms(1000);                     // 5. Wait 1s
    }
}

/* ---- ADC0: INTERNAL TEMPERATURE SENSOR ---- */
/* Formula: TEMP(C) = 147.5 - ((75 * 3.3 * raw) / 4096) */
void ADC0_Init(void) {
    SYSCTL->RCGCADC |= (1u << 0);           // Enable ADC0 clock
    while ((SYSCTL->PRADC & (1u << 0)) == 0);
    ADC0->ACTSS &= ~(1u << 3);              // Disable SS3
    ADC0->EMUX  &= ~(0xFu << 12);           // Software trigger
    ADC0->SSMUX3 = 0u;
    ADC0->SSCTL3 = (1u<<3)|(1u<<2)|(1u<<1); // TS0|IE0|END0
    ADC0->ACTSS |= (1u << 3);               // Enable SS3
}

int32_t ADC0_ReadTempC(void) {
    ADC0->PSSI |= (1u << 3);                // Trigger
    while ((ADC0->RIS & (1u << 3)) == 0);   // Wait
    uint32_t raw = ADC0->SSFIFO3 & 0xFFFu;  // 12-bit result
    ADC0->ISC |= (1u << 3);                 // Clear flag
    int32_t temp_x10 = 1475 - (int32_t)((75u*33u*raw)/4096u);
    return temp_x10 / 10;
}
```

```
/* ---- CAN0: LOOPBACK MODE ---- */
/* PE5=CAN0TX | PE4=CAN0RX | 500 kbps @ 16 MHz */
/* MsgObj 1 = TX | MsgObj 2 = RX */
void CAN0_Init(void) {
    SYSCTL->RCGCCAN  |= (1u << 0);          // CAN0 clock
    SYSCTL->RCGCGPIO |= (1u << 4);          // GPIOE clock
    while ((SYSCTL->PRGPIO & (1u << 4)) == 0);
    GPIOE->AFSEL |= (1u<<4)|(1u<<5);       // PE4/PE5 AF
    GPIOE->PCTL  |= (8u<<16)|(8u<<20);     // AF8 = CAN0
    GPIOE->DEN   |= (1u<<4)|(1u<<5);       // Digital enable
    while ((SYSCTL->PRCAN & (1u << 0)) == 0);
    CAN0->CTL |= (1u<<0)|(1u<<6)|(1u<<7);  // INIT|CCE|TEST
    CAN0->TST |= (1u << 4);                 // Loopback enable
    CAN0->BIT  = 0x00003A01u;               // 500 kbps timing
    CAN0->BRPE = 0u;
    CAN0->CTL &= ~(1u << 0);                // Exit init
    while (CAN0->CTL & (1u << 0));

    /* Message Object 1 = TX */
    CAN0->IF1CMSK = 0xB3u;
    CAN0->IF1ARB2 = (1u<<15)|(1u<<13)|((CAN_TEMP_ID&0x7FFu)<<2);
    CAN0->IF1ARB1 = 0u;
    CAN0->IF1MCTL = (1u<<7)|2u;             // EOB=1, DLC=2
    CAN0->IF1DA1 = CAN0->IF1DA2 = 0u;
    CAN0->IF1DB1 = CAN0->IF1DB2 = 0u;
    CAN0->IF1CRQ = 1u;
    while (CAN0->IF1CRQ & (1u << 15));

    /* Message Object 2 = RX (accepts ID 0x100 only) */
    CAN0->IF2CMSK = 0xF3u;
    CAN0->IF2MSK2 = (0x7FFu<<2)|(1u<<14);
    CAN0->IF2MSK1 = 0u;
    CAN0->IF2ARB2 = (1u<<15)|((CAN_TEMP_ID&0x7FFu)<<2);
    CAN0->IF2ARB1 = 0u;
    CAN0->IF2MCTL = (1u<<12)|(1u<<7)|8u;        // UMASK|EOB|DLC=8
    CAN0->IF2CRQ = 2u;
    while (CAN0->IF2CRQ & (1u << 15));
}

void CAN0_Transmit(uint8_t *data, uint8_t len) {
    CAN0->IF1CMSK = 0x97u; // Set command mask
    CAN0->IF1MCTL = (1u<<8)|(1u<<7)|(len&0xFu);     // Set control and
length
    CAN0->IF1DA1 = (uint32_t)((data[1]<<8)|data[0]);     // Load first
2 bytes
```

```
    CAN0->IF1DA2 = CAN0->IF1DB1 = CAN0->IF1DB2 = 0u;     // Clear
other bytes
    CAN0->IF1CRQ = 1u; // Start transfer
    while (CAN0->IF1CRQ & (1u << 15));     // Wait until done
}

uint8_t CAN0_Receive(uint8_t *data) {
    CAN0->IF2CMSK = 0x3Fu; // Set command mask
    CAN0->IF2CRQ = 2u; // Read message object 2
    while (CAN0->IF2CRQ & (1u << 15)); // Wait until done
    if ((CAN0->IF2MCTL & (1u << 15)) == 0) return 0; // Return 0 if no
new data
    data[0] = (uint8_t)(CAN0->IF2DA1 & 0xFFu); // Read first byte
    data[1] = (uint8_t)((CAN0->IF2DA1>>8) & 0xFFu); // Read second
byte
    return 1; // Return 1 if data received
}

/* ---- UART0: 9600 baud, PA0/PA1 ---- */
void Uart0_Init(void) {
    SYSCTL->RCGCUART |= (1u << 0);       // Enable UART0 clock
    SYSCTL->RCGCGPIO |= (1u << 0);       // Enable GPIOA clock
    while ((SYSCTL->PRGPIO & (1u << 0)) == 0); // Wait for GPIOA ready
    GPIOA->AFSEL |= (1u<<0)|(1u<<1);    // Enable alternate function
    GPIOA->PCTL  |= (1u<<0)|(1u<<4);       // Set PA0/PA1 for UART
    GPIOA->DEN   |= (1u<<0)|(1u<<1);      // Enable digital I/O
    UART0->CTL &= (unsigned)(~((1u<<0)|(1u<<8)|(1u<<9)));
    UART0->IBRD = 104; UART0->FBRD = 11;    // Set baud rate
    UART0->LCRH |= (3u << 5);       // Set 8-bit word length
    UART0->CC = 0x05u;               // Set clock source
    UART0->CTL |= (1u<<0)|(1u<<8)|(1u<<9);
}

void uart0_send(uint8_t ch) {
    while (UART0->FR & (1u << 5));           // Wait if TX full
    UART0->DR = ch;                          // Send byte
}

int fputc(int ch, FILE *f) {
    (void)f;                        // Ignore file pointer
    uart0_send((uint8_t)ch);        // Send character
    return ch;                      // Return sent character
}

void delay_us(int us) { while(us--) __asm("nop"); }
void delay_ms(int ms) { for(int i=0;i<ms;i++) delay_us(1000); }
```

6. Application: Remote Environmental Monitoring

CAN-based temperature monitoring also has many practical applications socially and environmentally. One such useful application would be remote environmental monitoring for wildfire detection systems.

6.1 Wildfire Early Detection Networks

In areas prone to wildfires, sensors scattered through an extensive network of temperature monitors could measure temperatures around the region and feed this information into CAN buses. In addition to just temperature information, humidity levels, wind speeds, and particulate content could also be monitored. Information on the temperature will be read out by each node of the network and published through the CAN bus.

CAN is best suited for this application owing to its immunity from external noise through its differential signal technique, multi-master system allowing any node to transmit independently, and its built-in error-checking feature that ensures accuracy of the information regardless of the environment. Differential signals can cover bus lengths of hundreds of meters at slower data transmission rates.

6.2 Social Impact

Alerts about increasing temperature levels within forested areas are essential to create sufficient time for rescue and evacuation purposes. The west coast of the USA is one area where fire season has occurred in California, Oregon, and Washington during the past few years. A network of CAN-based sensors can be used to create alerts if the threshold value of ambient temperature is reached in a certain region.

6.3 Environmental Impact

In addition to fire detection, such a CAN-based monitoring network is helpful in analyzing climatic data as well. It is useful for detecting thermal pollution around industrial areas and collecting data on temperature fluctuations for various ecological purposes. The energy consumption of CAN-based sensors is fairly low as they can operate in low-power mode, which allows using solar-powered sensors.

7. Public Safety, Health, Welfare, and Broader Factors

7.1 Public Safety

In contrast, CAN bus is an ISO-standardized protocol (ISO 11898) that is explicitly designed for use cases where safety is paramount. By virtue of its inherent error detection (CRC, bit monitoring, acknowledgment checks), followed by automatic retransmissions, CAN will never propagate any corrupted data. As such, its usage in the automotive or industrial fields ensures public safety because sensors providing engine management, brakes, or industrial process control must be reliable.

7.2 Public Health

Temperature monitoring through CAN bus finds application in hospital medical devices and pharmaceutical cold chain transportation, as well as in HVAC (heating, ventilation, and air conditioning). The ability to monitor accurately the temperature guarantees that vaccines are stored properly, operating rooms maintain appropriate climate conditions, and patient monitoring devices function correctly.

7.3 Environmental Factors

Using an internal temperature sensor makes it possible to completely eliminate hardware sensors. The loop back test reduces the number of PCBs required during the development stage, thereby lowering the amount of consumed resources during manufacturing.

7.4 Cultural and Global Factors

CAN is a widely implemented protocol with a massive ecosystem of development tools, transceivers, protocol analyzers, and other components available throughout the world. Designing CAN communication into projects ensures compatibility with equipment produced in any part of the globe. At the same time, the educational value of the presented project is also global in nature.

8. Potential Improvements and Additional Sensors

The following are additional changes that can enhance the effectiveness of the current design:

- External temperature sensor (e.g., TMP36/DS18B20): Since the internal die temperature varies based on chip heating, it would be best to use an external analog/digital temperature sensor.
- Humidity sensor (e.g., DHT22/BME280): Relative humidity measurement is useful for both monitoring and HVAC purposes.
- CAN bus implementation: Using two TM4C123 boards and the CAN shield (or attaching a secondary Arduino board) will showcase multi-node messaging using CAN bus, ID filtering, and bus arbitration techniques.
- LCD screen (e.g., 16×2 character LCD/SPI OLED): The temperature will be shown locally at the board level eliminating the need for a terminal program to view temperature readings.
- SD Card log recording: An SD Card connected through SPI will store temperature values for a period of time.

9. Theoretical Feature: External BME280 Sensor Node

This feature describes designing another theoretical CAN node with BME280 environmental sensor (temperature, humidity, barometric pressure) connected to a second TM4C123 LaunchPad.

9.1 Design Description

BME280 is attached to the second TM4C123 through I2C interface (PB2 = SCL, PB3 = SDA). It receives data for three parameters, packs them in one CAN frame with identifier 0x200 (6 bytes per parameter, integer value), and transmits on CAN bus (500 kbps). Original node still transmits die temperature using ID 0x100.

9.2 Power Consumption Analysis

Table 3. Estimated power consumption of system components.

Component	Current Draw	Notes
TM4C123 (active)	~25 mA @ 3.3V	Running CAN + I2C + ADC
BME280 (measuring)	~0.7 mA @ 3.3V	During forced measurement
MCP2551 transceiver	~75 mA @ 5V	Dominant state, worst case
Total (node 2)	~100.7 mA	Peak during TX

The current design consumes around 100 mA (TM4C123 + MCP2551). The addition of another node using BME280 will increase this value by about 100.7 mA peak consumption, doubling the power consumption of the system to around 200.7 mA if both nodes are in operation simultaneously.

9.3 Cost Analysis

Table 4. Cost breakdown for additional hardware components.

Item	Estimated Cost
TM4C123 Tiva C LaunchPad	$12.99
BME280 breakout module	$3.50 – $8.00
MCP2551 CAN transceiver module	$2.00 – $5.00
Jumper wires, headers	$1.00
Total additional cost	$19.49 – $26.99

10. Conclusion and Future Developments

10.1 Conclusion

The present work successfully demonstrates the CAN bus usage on the TM4C123GH6PM MCU in developing a simple temperature monitoring system sending/receiving CAN frames on a single board via the CAN loopback mode. Three main embedded peripherals have been used in the system — ADC for sensors, CAN for networking, and UART for the human interface – within one unified firmware structure.

CAN0 loopback mode turned out to be an effective engineering approach which eliminated the pin allocation conflict of CAN1 and UART0, provided full TX/RX checking without another CAN node, and allowed for keeping the MCP2551 CAN transceiver alive in the bus for possible external CAN observation. It was shown in practice that proper CAN functionality testing and software development can be carried out on limited hardware, including just one LaunchPad and one transceiver.

10.2 Future Developments

- Multi-board CAN network: Expansion to multiple boards with individual message IDs to showcase actual bus arbitration, prioritized messaging, and distributed sensor networks.
- CAN interrupt-based reception: Replacement of poll-based message reception by interrupt-based handling using CAN0 interrupts for higher CPU efficiency and faster processing.
- FreeRTOS implementation: Operation of CAN communication protocol within FreeRTOS, with separate tasks assigned for sensor readings, CAN transmission, CAN reception, and display operations allowing for preemptive multi-tasking and improved timing guarantees.
- OBD-II diagnostic interface: Implementation of standard OBD-II PIDs (Parameter IDs) on CAN bus communication to provide practical automotive CAN communication experience.
- Wireless connectivity: Addition of a wireless module (ESP32) to establish a CAN to cloud communication gateway for remote access to CAN bus communication data from web-based or mobile dashboards.
- Extended CAN (CAN 2.0B): Upgrading to an extended CAN 2.0B format to provide a larger identifier space and enable compatibility with industrial/jtruck J1939 CAN standards.
- Lowered power consumption: Introduction of sleep modes within TM4C123 microcontroller during idle periods and wakeup messages over CAN bus to minimize power consumption during operation.

11. Challenges Faced

11.1 Design Challenges

One of the biggest challenges in the design was the problem of pin conflicts between CAN1 and UART0. Since both devices make use of PA0/PA1, it was necessary to understand in great depth how TM4C123 alternate function mapping works and select CAN0 on Port E with loopback as the appropriate choice.

11.2 Implementation Challenges

In calculating the bit timing settings for the CAN protocol, it was important to understand the meaning of the packed data in CANBIT register, containing TSEG1, TSEG2, SJW, and BRP bits. Incorrect calculation results in a silent failure, where the transmission occurs, but is not received (loopback included) and there is no trace of any error except for using a CAN analyzer. The correct value, 0x00003A01, was calculated based on the bit timing formula: system clock of 16 MHz, BRP = 1 (gives us 8 MHz CAN clock), 16 time quanta per bit (1 sync+11 TSEG1+4 TSEG2)=500 kbps.

One difficulty in using the CAN message objects involved the IF1/IF2 registers. It is impossible to directly access these registers, as you have to issue a series of commands to transfer data into and out of the actual message object via interface registers. Learning the CMSK (command mask), ARB (arbitration), MCTL (message control), and CRQ (command request) registers was a must.

11.3 Testing Challenges

Testing whether the CAN message was properly received when using the loopback method necessitated including a timeout function in case the receive failed, thus hanging the loop indefinitely. Using a 100-iteration delay with 50 µs intervals proved effective enough to give the loop at least 5 ms to check for an error.

12. References

[1] Texas Instruments, “TM4C123GH6PM Microcontroller Data Sheet,” TI Document SPMS376E, 2014.

[2] Texas Instruments, “Tiva C Series TM4C123G LaunchPad Evaluation Board User’s Guide,” TI Document SPMU296, 2013.

[3] Microchip Technology, “MCP2551 High-Speed CAN Transceiver Data Sheet,” Microchip Document DS21667, 2010.

[4] Robert Bosch GmbH, “CAN Specification Version 2.0,” 1991.

[5] ISO 11898-1:2015, “Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signaling,” International Organization for Standardization.

[6] J. W. Valvano, “Embedded Systems: Real-Time Interfacing to ARM Cortex-M Microcontrollers,” 6th Edition, 2023.

[7] ARM Limited, “Cortex-M4 Technical Reference Manual,” ARM Document DDI0439D.