OAKLAND UNIVERSITY™

School of Engineering and Computer Science

ECE 5721 – EMBEDDED SYSTEM DESIGN

CAN COMMUNICATION BETWEEN TWO TIVA BOARDS

PROJECT 2 REPORT

Professor:

Subramaniam Ganesan

Group Members:

Mina Babona (minayaqo@oakland.edu)

Lucas Costello (lcostello@oakland.edu)

Eduard Sufaj (esufaj@oakland.edu)

Nikola Sinishtaj (sinistovic@oakland.edu)

TABLE OF CONTENTS

ABSTRACT

In this project, two Tiva C Series microcontrollers are connected over a CAN (Controller Area Network) bus to examine communication. The MCP2551 transceivers are supplied by SparkFun's DEV-13262. One board reads temperature data from its onboard sensor, converts it to degrees Celsius, and sends the result via the CAN bus to the other board. After being received, the temperature data is transmitted through a serial UART connection to a computer for display on a serial monitor. This configuration demonstrates the practical implementation of CAN communication in data transmission and acquisition devices.

INTRODUCTION

In this project, we designed a system to transmit on-board temperature sensor data between two TM4C123G Tiva Launchpads. To accomplish this we used a CAN (Controller Area Network) bus and two MCP2551 transceivers. The way we implemented this was by having the ‘slave’ module send requests to the ‘master’ board. When the ‘master’ board gets a request, it reads the data from its on-board temperature sensor, converts it to degrees Celsius, and sends it to the ‘slave’ board. The ‘slave’ board receives the data and outputs it over UART to a computer so that it can be viewed on a serial monitor.

Systems like cars and industrial machines employ a communication protocol called Controller Area Network (CAN) to transfer data between several units. It uses two wires for reliable communication, even in loud environments. CAN allows devices, known as nodes, to share a single connection without the need for a central controller. Because each message has a unique ID, devices can determine which data to process. Because the protocol features error-checking procedures to ensure data accuracy, it is reliable for applications where data integrity is essential.

HARDWARE

The hardware used for this project is relatively straightforward. We used two TM4C123G Tiva C Series LaunchPad boards, each equipped with one of SparkFun's DEV-13262 CAN Bus Shields. The microcontrollers' CAN signals are converted into the correct differential signals by the MCP2551 chips supplied on the CAN Bus Shields. Even in noisy environments, this setup guarantees a secure and reliable connection. We connected the Vbus output from each board to the 5V power in of each bus shield to supply power to the MCP2551 chips. We also connected the grounds from each of the boards together to ensure correct signal referencing.

In order to reduce interference, we used a twisted pair of wires to connect the boards. At both ends of the twisted pair, we applied a 120 ohm resistor to further aid in reducing the interference.

IMPLEMENTATION

The implementation of this project involves setting up the CAN bus communication, configuring the microcontrollers, and integrating the temperature sensor and UART for data handling. Each board is programmed to fulfill its respective role as either the master or slave in the system.

Master Board:

The 'master' board is responsible for reading the temperature data from its on-board temperature sensor and then sending it over the can bus. The following steps outline our implementation for this:

1. **Temperature Sensor Configuration:** We initialize the ADC to read from the temperature sensor and convert that reading to degrees celsius.
2. **CAN Bus Setup:** CAN0 is configured to use the PB4 (RX) and PB5 (TX) pins from the Tiva Board.
3. **Data Transmission:** The temperature value is packed into a 4-byte format and then sent over the can bus.
4. **Error Handling:** Interrupts are enabled to detect and log and errors that appear during transmission.

Slave Board:

The 'slave' board is designed to receive data from the CAN bus, process it, and send it to a computer over serial UART. This board has similar steps for implementation, but does them a bit differently than the master board. The outline of our implementation for the slave is as follows:

1. **CAN Bus Configuration:** The slave's CAN0 interface is set up using the same pin configuration as the master board. We made the can bus listen for messages with a specific ID to ensure that only relevant data can be processed and displayed.
2. **Message Reception:** Incoming data is stored in a tCANMsgObject, and the temperature value is extracted from the payload. On this board we use the CAN interrupt to signal when a message is received.
3. **UART Communication:** With the extracted temperature value, we take it and send it through serial UART to be displayed on the computer. This lets us monitor the temperature readings in real time.
4. **Error Handling:** Like in the master board, this board also has error handling. Any message loss or reception issues are flagged and an error message is sent over the UART so we can handle it accordingly.

SOURCE CODE

Code for the Master Board:

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

#include "hw_memmap.h"
#include "hw_types.h"
#include "gpio.h"
#include "pin_map.h"
#include "debug.h"
#include "sysctl.h"
#include "adc.h"
#include "uart.h"
#include "can.h"

#include "tm4c123gh6pm.h"
#include "1_tm4c123gh6pm.h"

// Flags
volatile bool requestReceived = false; // Flag to indicate a received request

// Message ID Definitions
#define SLAVE_REQUEST_ID 2
#define MASTER_RESPONSE_ID 1
#define REQUEST_PATTERN 0x01

// Function Prototypes
void InitCAN0(void);
void InitUART0(void);
void InitADC0(void);
void UART0_SendString(const char* str);
void Delay(uint32_t ms);
int ReadTemperature(void);
void SendTemperatureCAN(int temperature);

bool canLinked = false;

int main(void) {
    // Set system clock to 16 MHz
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);
```

```
    // Initialize peripherals
    InitUART0();
    UART0_SendString("\033[2J\033[H"); // Clear screen
    UART0_SendString("Initializing Temperature Acquisition & CAN Communication...\r\n");
    InitADC0();
    InitCAN0();
    UART0_SendString("The desired hardware has been initialized.\r\n");
    UART0_SendString("Waiting for Slave Requests...\r\n");

    while (1) {
        // Check if a request has been received
        if (requestReceived) {
            requestReceived = false; // Reset the flag

            // Read temperature from ADC
            int temperature_x100 = ReadTemperature();

            // Send the temperature via CAN
            SendTemperatureCAN(temperature_x100);

            UART0_SendString("Responded to Slave Request\r\n");
        }

        // Perform other tasks or sleep
    }
}

// Initialize UART0 for serial communication
void InitUART0(void) {
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); // Enable clock for GPIOA
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); // Enable clock for UART0

    GPIOPinConfigure(GPIO_PA0_U0RX); // Configure PA0 as UART0 RX
    GPIOPinConfigure(GPIO_PA1_U0TX); // Configure PA1 as UART0 TX
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1); // Enable UART pins


    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 9600,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
UART_CONFIG_PAR_NONE)); // Configure UART
}

// Send a string via UART0
void UART0_SendString(const char* str) {
    while (*str) {
        UARTCharPut(UART0_BASE, *str++); // Send each character
    }
}
```

```
// Initialize ADC0 for the temperature sensor
void InitADC0(void) {
   // Enable ADC0 peripheral and wait for it to stabilize
   SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
   while (!SysCtlPeripheralReady(SYSCTL_PERIPH_ADC0));

   // Configure sequencer 3 for single-sample temperature sensor reading
   ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
   ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_TS | ADC_CTL_IE | ADC_CTL_END);
   ADCSequenceEnable(ADC0_BASE, 3);

   // Clear any existing interrupts
   ADCIntClear(ADC0_BASE, 3);
}

// Read temperature from the internal temperature sensor
int ReadTemperature(void) {
   ADCProcessorTrigger(ADC0_BASE, 3);

   while (!ADCIntStatus(ADC0_BASE, 3, false)) {
      // Waiting
   }
   UART0_SendString("ADC Conversion Complete...\r\n");

   ADCIntClear(ADC0_BASE, 3); // Clear ADC interrupt

   uint32_t adcValue;
   ADCSequenceDataGet(ADC0_BASE, 3, &adcValue); // Retrieve ADC result

   UART0_SendString("ADC Value: ");
   UARTCharPut(UART0_BASE, (adcValue / 1000) % 10 + '0'); // Thousands place
   UARTCharPut(UART0_BASE, (adcValue / 100) % 10 + '0');  // Hundreds place
   UARTCharPut(UART0_BASE, (adcValue / 10) % 10 + '0');   // Tens place
   UARTCharPut(UART0_BASE, (adcValue % 10) + '0');        // Units place
   UART0_SendString("\r\n");

   // Perform integer math for temperature conversion (scaled by 100)
   UART0_SendString("Calculating Temperature...\r\n");
   float temperature_x100 = (147.5 - ((247.5 * adcValue) / 4096));
   int temperature_scaled = (int)(100 * temperature_x100);

   return temperature_scaled;
}

// Initialize CAN0 -- PB4 (RX) , PB5 (TX)
void InitCAN0(void) {
   // Enable CAN0 peripheral and GPIO port for CAN0
   SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
```

```
    // Wait until peripherals are ready
    while (!SysCtlPeripheralReady(SYSCTL_PERIPH_CAN0));
    while (!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOB));

    // Configure GPIO pins for CAN0 RX and TX
    GPIOPinConfigure(GPIO_PB4_CAN0RX);
    GPIOPinConfigure(GPIO_PB5_CAN0TX);
    GPIOPinTypeCAN(GPIO_PORTB_BASE, GPIO_PIN_4 | GPIO_PIN_5);

    // Initialize CAN0 module
    CANInit(CAN0_BASE);

    // Set CAN0 bit rate to 500 kbps (must match CAN1)
    CANBitRateSet(CAN0_BASE, SysCtlClockGet(), 500000);

    // Enable CAN0
    CANEnable(CAN0_BASE);

    // Set CAN linked status to true
    canLinked = true;
}

// CAN0 Interrupt Handler to wait for slave to communicate
void CAN0IntHandler(void) {
    uint32_t ui32Status;
    tCANMsgObject sCANMessage;
    uint8_t pui8MsgData[2];

    // Get interrupt status
    ui32Status = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE);

    // Check if the interrupt is caused by a message object
    if (ui32Status == 1) {
        // Clear the interrupt
        CANIntClear(CAN0_BASE, 1);

        // Prepare to receive the message
        sCANMessage.ui32MsgID = SLAVE_REQUEST_ID; // Using defined constant
        sCANMessage.ui32MsgIDMask = 0xFFFFFFFF; // Exact match for this ID
        sCANMessage.ui32Flags = MSG_OBJ_RX_INT_ENABLE | MSG_OBJ_USE_ID_FILTER;
        sCANMessage.ui32MsgLen = sizeof(pui8MsgData);
        sCANMessage.pui8MsgData = pui8MsgData;

        // Get the message from the CAN controller
        CANMessageGet(CAN0_BASE, 1, &sCANMessage, true);

        // Check if the message matches a "request" pattern
        if (pui8MsgData[0] == REQUEST_PATTERN) {
            requestReceived = true; // Set flag to process in the main loop
        }
    } else {
        // Clear any other CAN interrupt
        CANIntClear(CAN0_BASE, ui32Status);
    }
}

// Send Temperature Value through CAN
void SendTemperatureCAN(int temperature) {
    tCANMsgObject sCANMessage;
    uint8_t pui8MsgData[2];

    // Set the data to be sent (temperature, split into 2 bytes)
    pui8MsgData[0] = (temperature >> 8) & 0xFF;  // High byte
    pui8MsgData[1] = temperature & 0xFF;         // Low byte

    sCANMessage.ui32MsgID = MASTER_RESPONSE_ID;
    sCANMessage.ui32MsgIDMask = 0;
    sCANMessage.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
    sCANMessage.ui32MsgLen = sizeof(pui8MsgData);
    sCANMessage.pui8MsgData = pui8MsgData;

    // Send the CAN message
    CANMessageSet(CAN0_BASE, 1, &sCANMessage, MSG_OBJ_TYPE_TX);
}

// Delay function
void Delay(uint32_t ms) {
    SysCtlDelay((SysCtlClockGet() / 3000) * ms); // Delay in milliseconds
}
```

Code for Slave Board:

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

#include "hw_memmap.h"
#include "hw_types.h"
#include "gpio.h"
#include "pin_map.h"
#include "debug.h"
#include "sysctl.h"
```

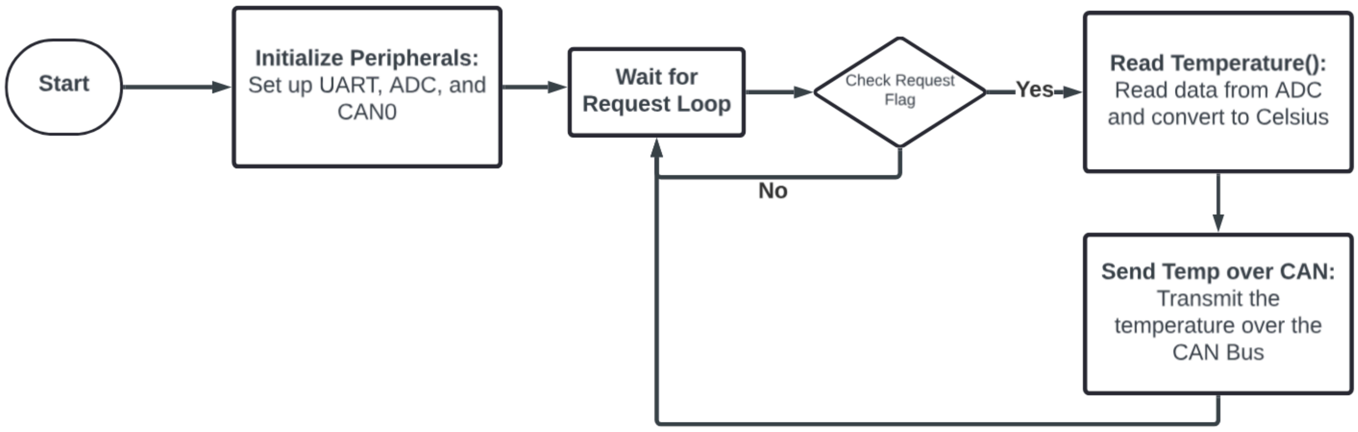
```
#include "uart.h"
#include "can.h"
#include "interrupt.h"

#include "tm4c123gh6pm.h"
#include "1_tm4c123gh6pm.h"

// Flags
volatile bool dataReceived = false; // Flag to indicate data received from master

// Function Prototypes
void InitCAN0(void);
void InitUART0(void);
void UART0_SendString(const char* str);
void SendRequestToMaster(void);
void CAN0IntHandler(void);

int main(void) {
  // Set system clock to 16 MHz
  SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
SYSCTL_XTAL_16MHZ);

  // Initialize peripherals
  InitUART0();
  InitCAN0();

  UART0_SendString("\033[2J\033[H"); // Clear screen & Reset Cursor
  UART0_SendString("CAN Slave Initialized. Sending Request to Master...\r\n");

  while (1) {
    // Send a request to the master
    SendRequestToMaster();
    UART0_SendString("Request sent to Master. Waiting for response...\r\n");

    // Wait for data to be received
    while (!dataReceived) {
      // Idle loop until data is received
    }

    dataReceived = false; // Reset the flag
  }
}

// Initialize UART0 for serial communication
void InitUART0(void) {
  SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); // Enable clock for GPIOA
  SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); // Enable clock for UART0

  GPIOPinConfigure(GPIO_PA0_U0RX); // Configure PA0 as UART0 RX
  GPIOPinConfigure(GPIO_PA1_U0TX); // Configure PA1 as UART0 TX
```

```
   GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1); // Enable UART pins

   UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 9600,
                  (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
UART_CONFIG_PAR_NONE)); // Configure UART
}

// Send a string via UART0
void UART0_SendString(const char* str) {
   while (*str) {
      UARTCharPut(UART0_BASE, *str++); // Send each character
   }
}

// Initialize CAN0 -- PB4 (RX) , PB5 (TX)
void InitCAN0(void) {
   // Enable CAN0 peripheral and GPIO port for CAN0
   SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
   SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

   // Wait until peripherals are ready
   while (!SysCtlPeripheralReady(SYSCTL_PERIPH_CAN0));
   while (!SysCtlPeripheralReady(SYSCTL_PERIPH_GPIOB));

   // Configure GPIO pins for CAN0 RX and TX
   GPIOPinConfigure(GPIO_PB4_CAN0RX);
   GPIOPinConfigure(GPIO_PB5_CAN0TX);
   GPIOPinTypeCAN(GPIO_PORTB_BASE, GPIO_PIN_4 | GPIO_PIN_5);

   // Initialize CAN0 module
   CANInit(CAN0_BASE);

   // Set CAN0 bit rate to 500 kbps (must match CAN1)
   CANBitRateSet(CAN0_BASE, SysCtlClockGet(), 500000);

   // Enable CAN0 interrupt
   CANIntEnable(CAN0_BASE, CAN_INT_MASTER);
   IntEnable(INT_CAN0);

   // Register CAN0 interrupt handler
   CANIntRegister(CAN0_BASE, CAN0IntHandler);

   // Enable CAN0
   CANEnable(CAN0_BASE);
}

// Send a request to the master
void SendRequestToMaster(void) {
   tCANMsgObject sCANMessage;
   uint8_t pui8MsgData[1] = {0x01}; // Request pattern
```

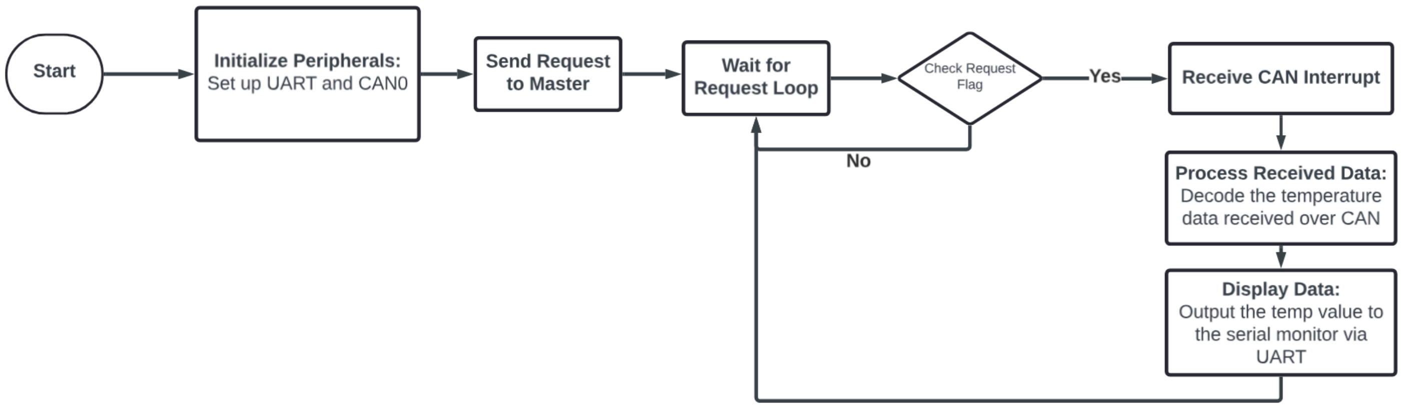
```
    sCANMessage.ui32MsgID = 2;                    // Message ID for request
    sCANMessage.ui32MsgIDMask = 0;
    sCANMessage.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
    sCANMessage.ui32MsgLen = sizeof(pui8MsgData);
    sCANMessage.pui8MsgData = pui8MsgData;

    // Send the CAN message
    CANMessageSet(CAN0_BASE, 1, &sCANMessage, MSG_OBJ_TYPE_TX);
}

// CAN0 Interrupt Handler to receive data from master
void CAN0IntHandler(void) {
    uint32_t ui32Status;
    tCANMsgObject sCANMessage;
    uint8_t pui8MsgData[2];

    // Get interrupt status
    ui32Status = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE);

    // Check if the interrupt is caused by a message object
    if (ui32Status == 1) {
        // Clear the interrupt
        CANIntClear(CAN0_BASE, 1);

        // Prepare to receive the message
        sCANMessage.ui32MsgID = 1; // Message ID for master response
        sCANMessage.ui32MsgIDMask = 0;
        sCANMessage.ui32Flags = MSG_OBJ_RX_INT_ENABLE | MSG_OBJ_USE_ID_FILTER;
        sCANMessage.ui32MsgLen = sizeof(pui8MsgData);
        sCANMessage.pui8MsgData = pui8MsgData;

        // Get the message from the CAN controller
        CANMessageGet(CAN0_BASE, 1, &sCANMessage, true);

        // Extract the temperature value (scaled by 100)
        int temperature_x100 = (pui8MsgData[0] << 8) | pui8MsgData[1];

        // Display temperature value on UART in XX.XX format
        UART0_SendString("Received Temperature: ");

        // Handle negative temperatures, if possible
        if (temperature_x100 < 0) {
            UARTCharPut(UART0_BASE, '-');
            temperature_x100 = -temperature_x100; // Make it positive for further processing
        }

        int integer_part = temperature_x100 / 100; // Integer part (e.g., 25 for 25.12)
        int fractional_part = temperature_x100 % 100; // Fractional part (e.g., 12 for 25.12)
        // Send integer part (always two digits)
        UARTCharPut(UART0_BASE, (integer_part / 10) + '0'); // Tens place
        UARTCharPut(UART0_BASE, (integer_part % 10) + '0'); // Units place

        // Send decimal point
        UARTCharPut(UART0_BASE, '.');

        // Send fractional part (always two digits)
        UARTCharPut(UART0_BASE, (fractional_part / 10) + '0'); // Tenths place
        UARTCharPut(UART0_BASE, (fractional_part % 10) + '0'); // Hundredths place

        // Send newline
        UART0_SendString(" C\r\n");

        dataReceived = true; // Set the flag to indicate data received
    } else {
        // Clear any other CAN interrupt
        CANIntClear(CAN0_BASE, ui32Status);
    }
}
```

FLOWCHART

Master Board Flowchart:

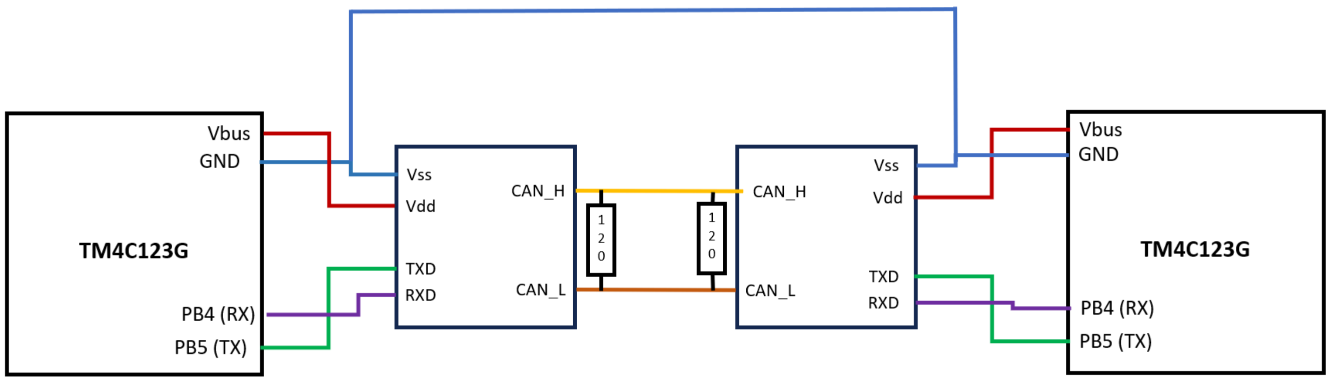

Start
Initialize Peripherals: Set up UART, ADC, and CAN0
Wait for Request Loop
Check Request Flag
Yes
No
Read Temperature(): Read data from ADC and convert to Celsius
Send Temp over CAN: Transmit the temperature over the CAN Bus

Slave Board Flowchart:


Start
Initialize Peripherals: Set up UART and CAN0
Send Request to Master
Wait for Request Loop
Check Request Flag
Yes
No
Receive CAN Interrupt
Process Received Data: Decode the temperature data received over CAN
Display Data: Output the temp value to the serial monitor via UART

HARDWARE IMAGES

Circuit Diagram:



CHALLENGES

Due to lacking or incompatible libraries, one of the major obstacles encountered during this project was the inability to deploy the system on the TM4C123G LaunchPad boards. Although the code and theoretical design were finished, the development environment was unable to incorporate the necessary library dependencies for the TM4C123G microcontroller and CAN communication.

This challenge highlighted the importance of ensuring library availability and compatibility before beginning a hardware-dependent project. As a result, the project remained in a simulation and design phase, unable to proceed to hardware testing and deployment. Future work will focus on securing the appropriate development tools and libraries to enable successful implementation.

CONCLUSION

This project demonstrated how to build and create a system that uses CAN communication to transfer temperature data between two TM4C123G LaunchPad boards. Temperature data was read by the master board, converted to degrees Celsius, and then sent to the slave board using the CAN bus. After receiving the data, the slave board used UART to output it to a computer for real-time monitoring. This configuration demonstrated how CAN and UART communication protocols can be combined in embedded systems to provide dependable and effective data transport.

Although the theoretical design and code development were finished, the lack of appropriate libraries made it difficult to implement on the real hardware. This restriction made it impossible to test and validate the system's hardware. Nevertheless, the project demonstrated how important library compatibility is for hardware-dependent programs and offered a strong basis for further development after the required resources are accessible.

REFERENCES

[1] Texas Instruments. (Year). Tiva™ TM4C123GH6PM Microcontroller. [Online]. Available: [TM4C123GH6PM]

[2] Microchip Technology. (Year). MCP2551 High-Speed CAN Transceiver. [Online]. Available: [MCP2551]