Temperature Sensor Data communication over CAN

Reading temperature and sending data over CAN

List of Authors (Alen Cehajic, Josh Kulwicki, Nishchay Kulkarni, Yash Gandham)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: acehajic@oakland.edu , Jkulwicki@oakland.edu , nkulkarni2@oakalnd.edu , ygandham@oakland.edu

***Abstract* — CAN Architecture is widely used in the automotive industry. The protocol is used to communicate between two Nexys A7 boards. The project will be useful in understanding an industry standard that is highly regarded in the automotive world.**

I. Introduction

The motivation for this project is to better understand the industry standard CAN protocol. CAN is a robust communication system which is used widely in the automotive industry. Getting a better understanding of this application will allow for new skills to be developed which are very highly desired in the automotive industry. In this project there will be a main focus on data communication in between two Nexys A7 50T boards. The data to be communicated will be received from the on-board temperature sensor on the Nexys board. The temperature will be read on board 1 and this reading will be then passed to the second board via CAN. When the temperature data is received by the second board, the temperature reading will be displayed on the 7 segment display on the second board. From this project understanding the various CAN frames and their individual responsibilities will be explored and understood alongside implementing the concept which is a very crucial part of the project.

II. Methodology

A. Design

The first design choice is to decide on what sensor should be used for as an input. Due to the preference of the Nexys Artrix-7 remaining stationary throughout the experiment, it is decided that the temperature sensor with the I2C protocol will be used. The CAN architecture will be used as a communication protocol between the temperature sensor board and the 7-segment display board.

Initially there was an approach plan created, firstly to remove the easy tasks and then tackle the harder task. This first task set was to read the temperature, this was done through the on board temperature sensor on the Nexys A7 50T. The board has an Analog Device ADT7420 temperature sensor, this sensor is capable of outputting 16-bit resolution with a typical accuracy better than 0.25 degrees.

B. Output Method

The next design problem is to decide on what form the output should be. LEDs were a choice but due to the complexity of binary numbers for the user the 7 segment display was chosen. This is due to numeric displays being far easier to read than binary and depending on the outside conditions of the board it may be a necessity.

III. Experimental Setup

The CAN protocol and the temperature sensor were experimented with separately. The temperature sensor was implemented to the input board and works successfully as a stand-alone unit. The CAN protocol has yet to be experimented.

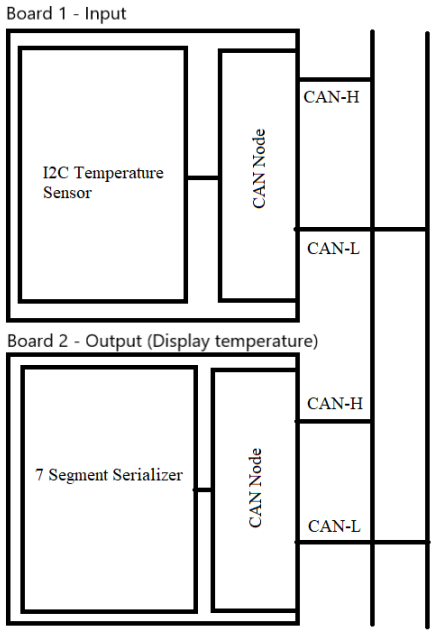

Figure 1 - Shows the initial stage of the CAN bus design that was proposed.

IV. CAN Implementation

The project was split into two main procedures. First one being the CAN communication protocol and the second being the 7 segment display module. The CAN communication protocol was implemented first. The procedure started out with understanding how CAN itself works, at the initial stage the understanding was that CAN would need 4 lines, power, ground, CAN signal high and CAN signal low. But this was a typical application of CAN, but in the scope of this project the goal was to only send data. The "bus system" for data communication was in effect not used. So taking a look at the goal again, it was realized that the base requirement was to send data to the second board. The second board was not going to pass data back. Which is why the set up was now changed to be power, ground and a CAN high. Since the board had power from the USB there was only a single wire connecting the two boards together and that is how CAN communication was set up. There was another procedure in the CAN module, which was better understanding how CAN signals work. The basics of a CAN signal are in the following image:

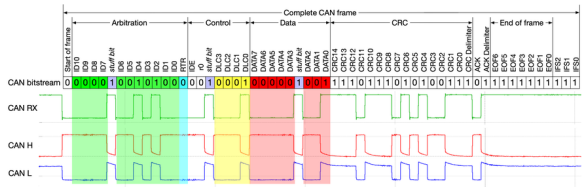

Figure 2 - CAN frames and the bits for the frames.

The image is a little small but there are 5 main parts in a CAN signal, first being the start frame which signals if there is information present or not. This can be represented by a 0 or 1. Second is the Arbitration frame which is a total of 12 bits and it includes the main procedure of filtering out on who exactly called the bus. In the scope of this project it is one of the least critical components because there is no "bus system" but also there is only a singular module which will be calling for the bus. Next is the most critical set of bytes, which is the Control frame. Here is where the data length will be received from. It's a total of 6 bites and it includes the identifier bit followed by the reserved bit and then followed by data length indication, in this case 4 bytes shows the total number of bytes of data. Next is the Data frame itself, which is where the data can be seen. It's a total of 8 bits of data.

Next is the CRC frame which is the error detection in the CAN signal. As of the date of writing this progress report the CRC has not been implemented, but the other frames prior to the CRC have been implemented and tested. The CRC, Cyclic redundancy check is an error detection code which is very commonly used in digital networks. The goal of a CRC in this CAN module is to make sure that the transmitted data is actually sent correctly and the data itself is validated. In the scope of this project the CRC might not be of the highest priority because only a singular bus is present and that bus is only carrying X bits of data. The room for error in this project is pretty small, but nonetheless the application of a CRC is a crucial component for understanding the CAN communication process. CRC involves a certain set of predefined bits that are used for generating a checksum, in the case of CRC-15-CAN, the checksum generated will be 15 bits and uses the hexadecimals 0xC599 to generate the checksum. The transmitter and receiver use the same modules, in theory when transmitted bits that carry the checksum field are processed by the receiver CRC module, the checksum generated by the receiver should be all zeros, indicating uncorrupted data.

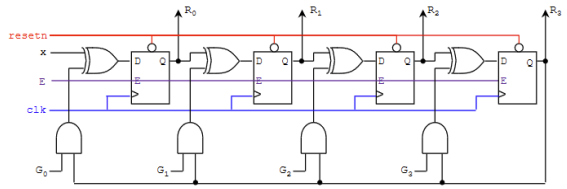

Figure 3 - Generic CRC diagram

The experimental setup shown in this paper was the correct implementation of CAN based on our initial studies, the only difference was as mentioned the wire count. Other than that the process was the same.

The second part was the LED set up and testing, this was a fairly straightforward task to accomplish because similar tasks were done in the labs. So the only part was setting the input data as the output on the LEDs. This portion was a little challenging but it was accomplished.

V. Bit Stuffing

The application of a bit stuffer was an essential component for transferring data over CAN to the second board. Bit stuffing is a technique used in digital communication to prevent errors in data transmission. It involves adding extra bits to the data being transmitted to ensure that a specific sequence

of bits does not occur in the data, which may be interpreted as a control signal by the receiver and lead to incorrect interpretation of the data. The basic idea of bit stuffing is to insert an additional "0" bit after every n consecutive "1" bits in the data being transmitted. For example in the application of this project n is 5. If the data being transmitted is "1111101111", bit stuffing would insert an additional "0" bit after the fifth "1", resulting in the stuffed data "1111100111". The receiver can then remove the stuffed bits before processing the data. Bit stuffing is commonly used in data communication protocols to prevent errors caused by the transmission of control characters in the data. The use of bit stuffing ensures that the transmitted data from the first board to the second board remains distinguishable from control frames, which leads to reducing the chances of data corruption and improving the reliability of data transmission.

In diagram X we can see the implementation of the bit stuffer which was used in this project.

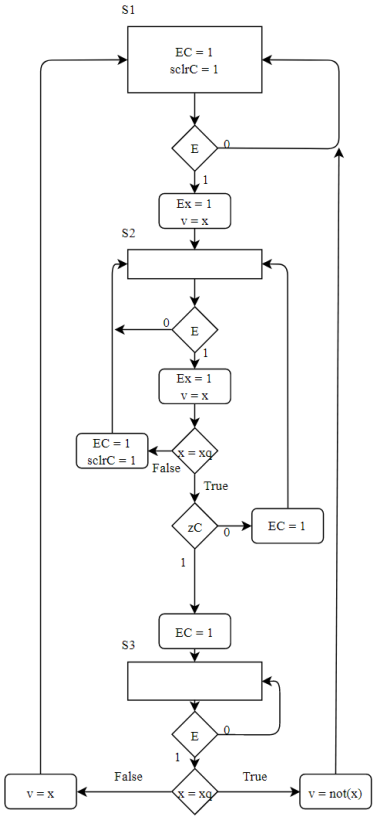

Figure 4 - Bit Stuffing Flow Chart

VI. Bit unstuffing

Since a bit stuffer is implemented, the other board would have to de-stuff the information that is being sent. Bit unstuffing is the process of removing extra bits that have been added to data during the bit stuffing process. The purpose of bit unstuffing is to recover the original data without the additional “0” bits. The process of bit unstuffing involves examining the incoming data for patterns that indicate stuffed bits. When the receiver detects the stuffed bit pattern, it removes the extra bit to recover the original data. To explain the process of bit unstuffing the example of bit stuffing will be referenced to. In the case of the bit stuffing technique that inserts an additional "0" bit after every five consecutive "1" bits, the receiver looks for every five consecutive "1" bits and removes the following "0" bit. The receiver in the project is able to correctly identify and remove the stuffed bits to interpret the data correctly. If bit unstuffing is not performed correctly, the data tends to have errors which has mostly led to the visual LEDs showing the wrong temperature.

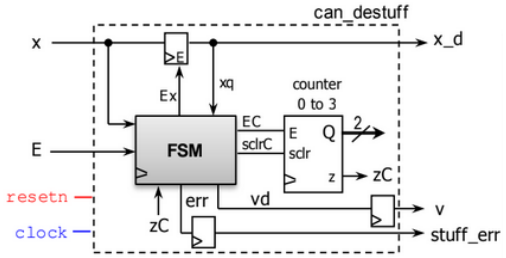

Figure 5 - Bit Un-stuffing Circuit Implementation

The block diagram in figure 5 displays how the bit un-stuffer was implemented in the project to read the data.

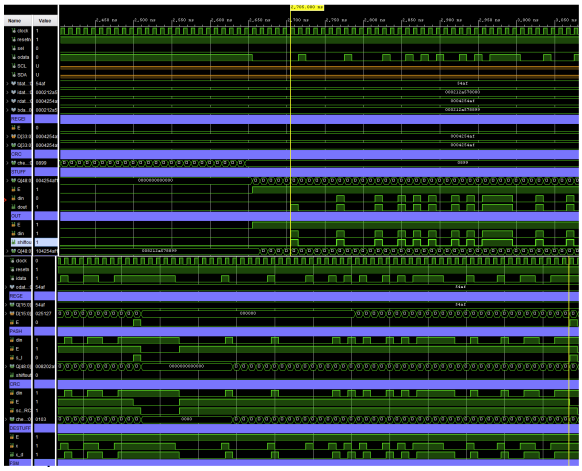

Figure 6 - Simulation of Transmitter and Receiver. Bit-destuffing error is presented.

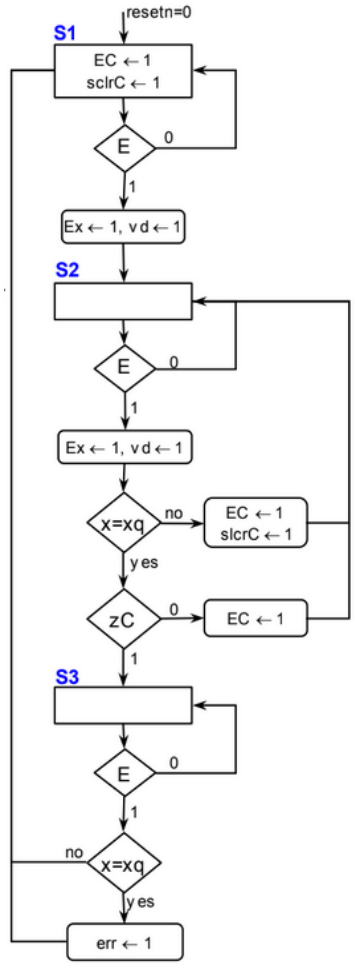

Figure 7 - Bit Un-stuffing Flow Chart

VII. FSM for CAN transmitter and receiver

This is the main core of this project, first the FSM for the CAN transmitter. This is what is responsible for controlling the transmission of the data frames over the CAN bus. The overview of the FSM working is as follows:

IDLE state: The transmitter is waiting for a new message to be received, until then it waits at this stage. When a new message is received, it moves to the arbitration state.

Validation State: Here the validation process involves comparing the message identifier with other messages. But in the scope of this project there is only one message to be received so no comparison actually takes place. So the message is always going to be the highest priority no matter what since there is only one message. The main checks here are the input data from the temperature sensor, CRC and the enable switches from the first stage.

Data State: Here is where the communication takes place, the start of frame (SOF) is sent, which is being checked in S3. In S4 is where the remainder of the frames are transmitted to the second board. This includes the data and the CRC. Next is the acknowledgement state.

Acknowledgement State: This is the state where the transmitting board waits for the acknowledgement signal to be received. If no acknowledgement is received it cycles through the Data state until the data is completely sent and acknowledgement is received. Due to the scope of the project focusing on 1-way serial communication, the Acknowledgement State is ignored.

The FSM transmitter flow chart can be seen in figure 8.

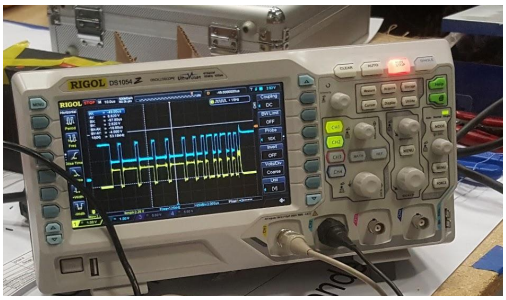

Image 1: Shows the CAN signal reading of transmission in blue and receiver in yellow on an oscilloscope.

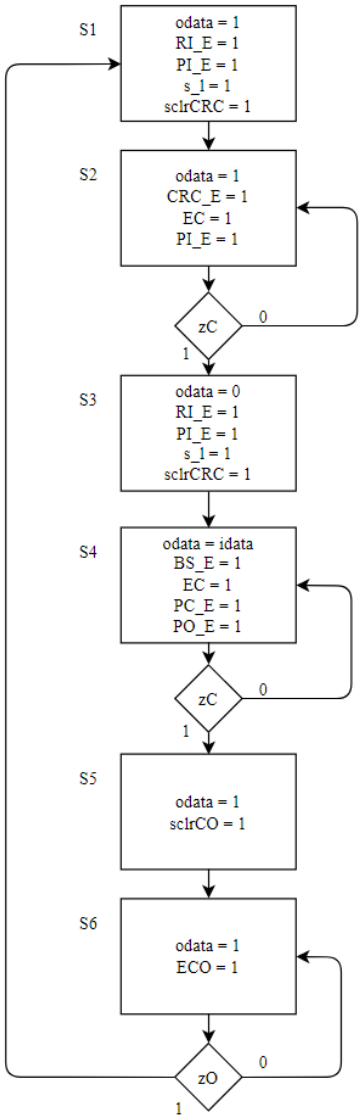

Figure 8 - CAN FSM Transmitter Flow Chart

FSM for CAN Reciever

The receiver has the following states:

The IDLE state checks if there is any data getting inputted, if there is none it will stay in state 1. The input it's specifically looking for is the CRC and EDS. Since those two components allow for data input determination. When that data is available the second stage is entered. The main checks are the CRC, EP and EP. S2 where all the data collection and data decomposition is done. The CAN signal is first sent to the bit un-stuffer, removing the "0" to then see the real reading of the temperature sensor on the board. Once the bit un-stuffer process has been completed the data is then sent to the LEDs to display the temperature. While in S2 the CRC module is enabled, the CRC in this case processes the incoming data to check if it has been corrupted during transmission. If the checksum generated is not all zeros, the transmission is corrupted. The CAN receiver flow diagram is shown below in figure 7.

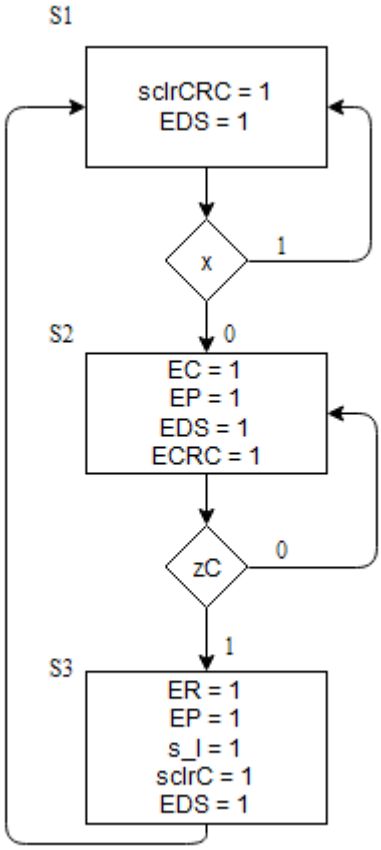

Figure 9 - Receiver FSM

VIII. Implementation - Testing stage

The image below shows the ideal design before the implementation of a bit-stuffer. While the process of creating the real design was all on 1 board with 1 finite state machine for CRC testing purposes, the final design involves splicing the CRC testing design, inserting a start and stop bit before transmission and making 2 finite state machines for each board instead of 1 big finite state machine.

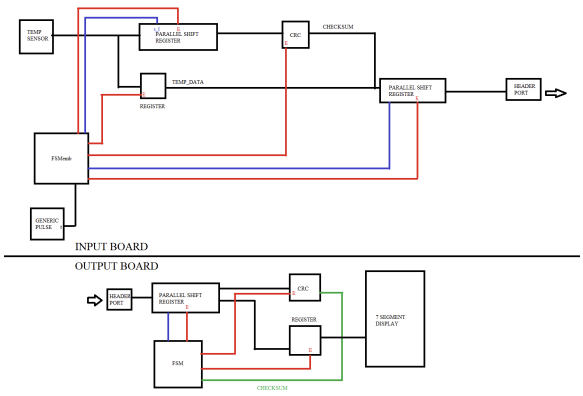

Figure 10 - Original Design

X. *Results and Conclusions*

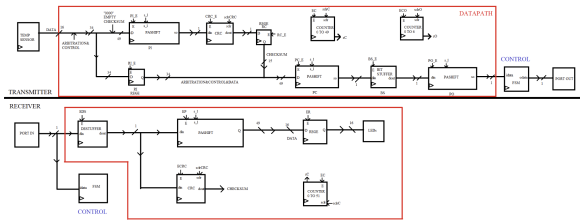

Figure 11 - Final Design

Transmission of data to the board through a single wire is successful, but due to the faulty design of the bit-stuffer and the inefficiency of the transmitter design the data sent is slightly corrupted. During testbench the CRC is successful standalone and the bitstuffer is successful standalone; when both the transmitter and receiver are combined into one testbench file the data is uncorrupted. When the transmitter and receiver are separated then the issues are shown. The simulation of the transmitter shows the data is uncorrupted yet it reveals the inefficiency of the design as one transmission requires ~55 clock cycles before the next set of data is processed instead of the CAN standard of 11 bit between transmissions. Using the data generated by the transmitter simulation, the receiver simulation processes the input as expected. The issue arises from the checksum generated by the receiver, tracing back shows the de-stuffer is de-stuffing the wrong bits but not producing an error flag. Possibly due to the data used does not require stuffing in the transmission board the error is unnoticeable until a real demonstration is taken place where the stuffing errors are apparent. The current transmission design only focuses on one data set at time, for efficiency the finite state machine should be redesigned to have the CRC and bit-stuffer work in parallel with different data sets, this should reduce the transmission time by approximately half. The bit-stuffer also should be redesigned as while it stuffs bit as intended, it does not cooperate with the de-stuffer.

Image 2: Final output

XI. *References*

[1] J. A. Cook, J. S. Freudenberg, “Controller Area Network (CAN),” University of Michigan, 13-Oct-2008. [online]. Available: https://www.eecs.umich.edu/courses/eecs461/doc/CAN_notes.pdf [Accessed: 22-Mar-2023].

[2] S. Corrigan, “Introduction to the Controller Area Network (CAN) Application Report Introduction to the Controller Area Network (CAN),” 2002 [online]. Available: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1681973692045 [Accessed: 20-Mar-2023].

[3] D. Llamocca, “CAN Bus + Automotive Ethernet”, Oakland University, 5-Oct-2021.

[4] Wikipedia Contributors, “CAN bus,” *Wikipedia*, Apr. 16, 2019. [online]. Available: https://en.wikipedia.org/wiki/CAN_bus [Accessed: 22-March-2023]