CAN Bus Controller

ECE 4710 Final Project

List of Authors (John Brooks, Evan Manser, Emad Eissa)
Electrical and Computer Engineering Department
School of Engineering and Computer Science
Oakland University, Rochester, MI
e-mails: johnbrooks@oakland.edu, emanser@oakland.edu, meeissa@oakland.edu

***Abstract*— The purpose of this project is to design and implement the bare minimum receive only functionality for a CAN bus controller. The controller will be designed in VHDL and provide data from a specific CAN frame on the CAN bus.**

I. INTRODUCTION

The project will cover designing a CAN bus controller from scratch in VHDL. The CAN bus controller is used by microcontrollers to communicate with other microcontrollers. It is most often used in the automotive industry for networking local embedded controllers called ECUs (electronic control units). The microcontrollers in these ECUs interface to the CAN controller which uses a CAN transceiver to produce the electrical signals that are transmitted on the CAN bus.

The reason our group is choosing this project is because the CAN bus network technology is highly relevant to our local automotive industry. By building a CAN controller our group will learn the intimate details of the technology.

This project will require the construction of several finite state machines to process in the serial data be processed through a series of stages somewhat like a pipeline. Our project also will require interfacing with real CAN hardware including the CAN transceiver and another working CAN node.

II. METHODOLOGY

There are several stages that the serial data from the transceiver needs to go through in order for the contained data bytes to be processed by the controller.

A. Serial Data Processor (SDP)

To read the serial data from the CAN bus the controller must synchronize itself with the bus. There is no clock signal in CAN, so the rising or falling edge of each new data bit synchronizes all the nodes on the network. To make this work a state machine generates a bit sampling clock for the 500 Kbps data on the bus.

In the absence of any rising edges the SDP uses a 500 KHz clock signal to sample the incoming bits. When a rising edge is detected, that 500 KHz pulse generator will get reset so that it stays synchronized with the data.

The serial data processor was successfully implemented. It works correctly in simulation and correctly samples the bits on the real hardware.

B. Destuffer

Because the CAN bus uses the data line as the clock, it is sometimes necessary to synchronize the nodes on the bus when a long series of dominant or recessive bits are transmitted without any rising or falling edges that would trigger the resynchronization. To fix this, extra bits are added to the serial bit stream called “stuff bits”. The bits are not part of the data and must be thrown out. They simply serve to synchronize the bus.

To account for these stuff bits, the controller will need a stage which removes the stuff bits from the data stream and produces the true datastream. The destuffer will also keep track of the number of bits that it has processed and logically separate the serial bit stream into registers which contain the CAN data.

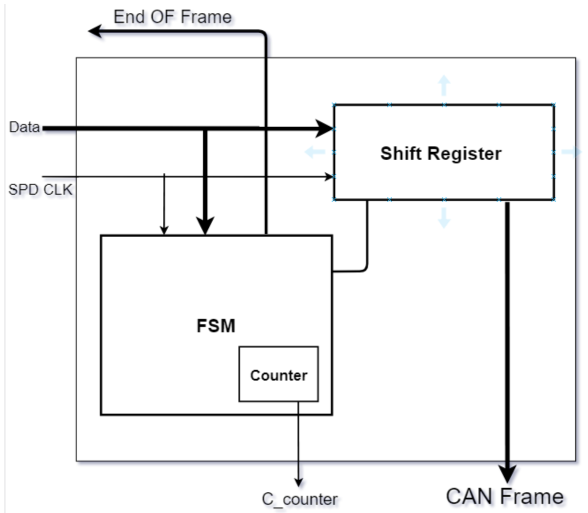

Figure 1. Destuffer Schematic

The destuffer operates through a shift register shifting left allowing for the next bit to come in. While the shift register is receiving data, the FSM will be tracking the bit stream to insure there are no more than five excessive bits from the same polarity. For example, then the destuffer gets data as 111111 then 11111 will appear in the shift register.

The destuffer designed consisting of a shift (left) register and embedded counter in FSM (state machine component). the shift register task to save the data coming from SDP to the register through shift in port. The FSM controls the enables of both the shift register and the counter. The state machine will keep tracking the data and skip or ignore the sixth bit after five sequential dominant or recessive bits. The FSM will set the enable low for the shift register to avoid grabbing the stuffed bit. Thus, the FSM will skip counting the stuffed bit and only count the actual data. In addition, the destuffer will send an (EOF) End of Frame signal to the SDP after receiving the CAN Frame. The counters inside the FSM are accessible to other components so they can track the actual number of bits inside the register at any time. the CRC component mainly will depend on the destuffer counter getting to the 83-bit value which will signal to the CRC component that it must begin its calculation.

C. CRC

As part of the CAN bus protocol there is a Cyclic Redundancy Check or CRC checksum used for error detection on the preceding bits. Therefore, as part of the stages which process the CAN frame data a CRC component will calculate the CRC per the CAN specification, and, if the CRC matches the transmitted CRC from the transmitting node, then a signal coming from the CRC to the SDP will signal that the CRC is a match so that it can properly ACK the CAN frame. Now, the CRC calculates the checksum of the first eighty-three unstuffed bits of the data frame produced from the destuffer component. In this instance, the CRC is calculated within about one hundred main clock cycles finishing well before the eighty-fourth bit of the frame is generated from the destuffer. Once the CRC component receives the data the calculation can begin. For the CAN controller to work, the CRC component must be designed to execute the correct generator polynomial. The generator-polynomial is vital to the CRC calculation as it determines the accuracy of the checksum. As per CAN specification, the polynomial used in this case is $x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + 1$. This polynomial is XORed with the 83-bit input. In hardware, this operation is carried out through a series of registers and XOR gates strategically placed before the registers in the order of the polynomial. Essentially, these XOR operations perform the division of the input by the polynomial. The output of this module, or checksum, is the modulo-2 of this division. Physically speaking the 83-bit input was concatenated with fifteen bits of zero value. These are utilized to act as placeholders for the final 15-bit output. The 15-bit output is taken from the inputs of the fifteen registers that cycled through the ninety-eight bits.

The result is then to be compared with the generated CRC from the transceiver to ensure that the message is valid. This can be done in hardware by a comparator which would send the ACK back to the serial data processor as shown in Fig. 1. Furthermore, this ACK would enable the Data Extractor component.

The CRC functions perfectly when simulated behaviorally by itself and with the Serial Data Processor and destuffer. It executes the calculation when specified and performs the operation in a timely manner. The comparison of the generated CRC from the Serial Data Processor and the calculated checksum from the CRC component was not implemented.

D. Data Extractor

Once the CRC has been validated and the CAN frame data received the Data Extractor will compare the arbitration ID of the CAN frame to the set filter and if it is a match, it will set a register which stores the data byte of interest. The extractor will then display the byte of data on a seven-segment display. In hardware the Data Extractor is a simple thirty-two bit register which passes through the first thirty-two bits of the 64-bit data field in the CAN frame. To enable the register requires two stages. First, the eleven-bit Arbitration ID would have to be ANDed with a hardwired value to generate a single bit indicating that the Arbitration ID matches. Second, this bit would have to be ANDed with the ACK bit from the calculated CRC matching the frame generated CRC.

The output of the Data Extractor register is directly connected to the component controlling the seven-segment display. This component is a simple serializer. Specifically, the one that was designed in Lab 3 of this course is compatible completely with this CAN controller.

The project member that was assigned this component was unable to continue the course preventing this component from being designed. However, the serializer was implemented and could generate the first thirty-two bits of the data field. Therefore, apart from confirming the CRC, the output simulated as anticipated.

E. Transmitting Node

In order to have data to process we connected a transmitting node which is already a finished product. This transmitting node gives us the ability to send CAN frames to our microcontroller and change the data byte value.

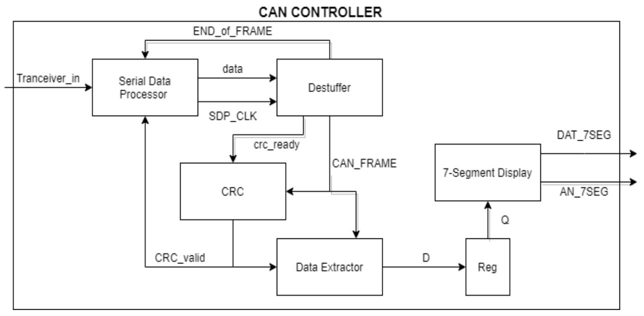

Figure 2. CAN Controller Circuit Diagram

III. RESULTS AND DISCUSSION

The CAN controller was completely successful in behavioral simulation. Getting the design to work on the real hardware was a slightly different matter which involved using the LEDs as debugging tools and a healthy amount of trial and error.

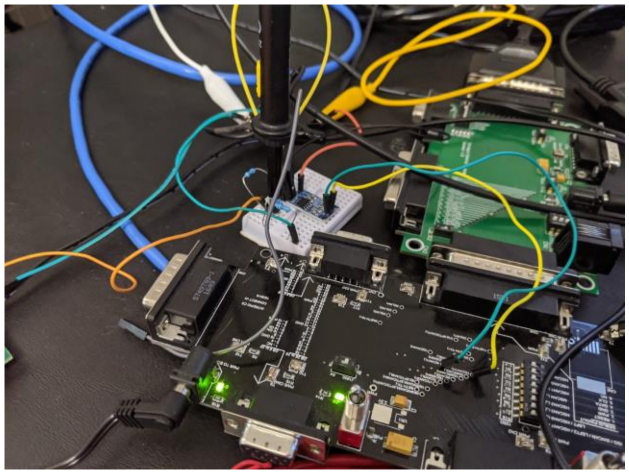

Figure 3

Part of the difficulties in getting the design to work on the hardware had to do with the current shelter-in-place order due to the COVID-19 pandemic. The team member with the CAN transceivers and CAN transmit hardware also does not have a soldering iron, because it is locked in the robotics laboratory at Oakland University. Because of that, the hardware was hastily assembled with jumper wires and a breadboard. The connections are depicted in Figure 4.

The transceiver is wired to a 5V power supply for power and the output of the transceiver is run through a voltage divider (2/3rds) to get the input voltage below 3 volts so that it is safe for the Nexys A7 to read.

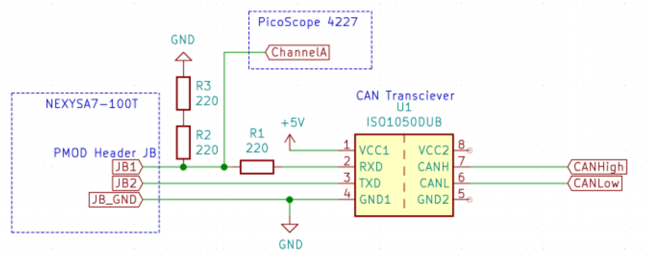

Figure 4. Electrical Schematic

Generating the CAN signals that we are reading is an Intrepid Control Systems' neoVI FIRE2 and Vehicle Spy 3 software. We can control the CAN bus message transmission rate and databytes with the software. This device is pictured in Fig. 5.

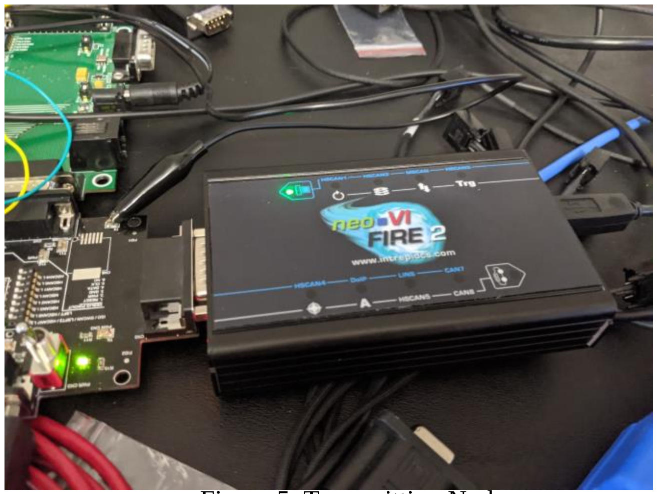

Figure 5. Transmitting Node

In order to make sure that the transceiver is functioning correctly, a PC based oscilloscope was used as shown in Fig. 6. The oscilloscope used was the PicoScope 4227, used with the PicoScope6 software.

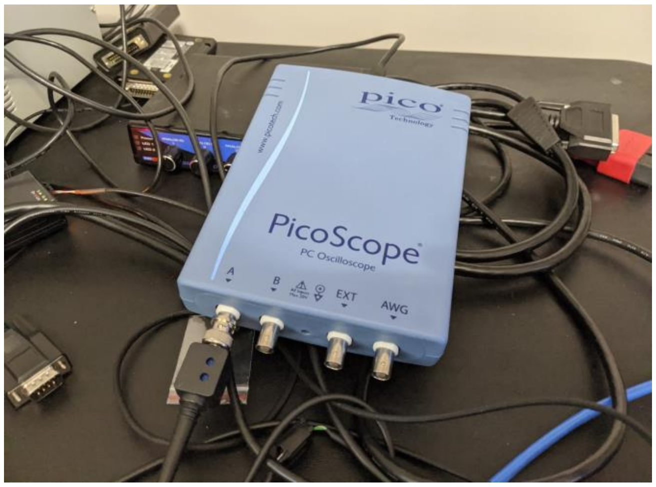

Figure 6. PicoScope

To verify that the CAN controller is working correctly a test frame was transmitted from the Vehicle Spy 3 software. The first byte cycled a single bit through each bit position and the remaining bytes spelled 0xCOFFEE as shown in Fig. 7

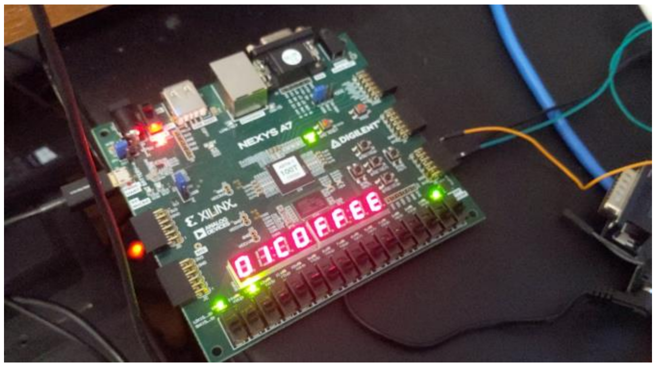
Figure 6. Seven-Segment Output Display

The hardware implemented design is able to decode the transmitted CAN frame on the seven-segment display. To test the CRC in simulation we fed a simulated CAN bus frame into the top level which performed all operations of the design (including CRC). We then compared the generated CRC value against the known good value that we saw decoded on the PicoScope. The behavioral simulation showed the correct CRC being calculated; however, it did not produce the correct value when run on the real FPGA. Unfortunately, we ran out of time to debug these errors before the end date of the project. So, the current design does not perform the check for CRC during our demonstration video. The remaining components were working correctly, as can be seen in the demonstration.