**Microprocessor Systems Lab 7**

Contoller Area Network (CAN)

Shane Conaboy

Justin Yost

ECSE-4790 Microprocessor Systems

December 7, 2015

**Introduction**

The Controller Area Network (CAN) is a serial bus communication protocol used as a standard for efficient and reliable communication between different nodes in industrial applications. All the nodes in the system share a common data bus and are assigned ID numbers, which also double as priority levels. Messages can be sent over the bus to the appropriate nodes by sending corresponding IDs, and nodes with lower priority will wait until the bus is clear to send other information. A famous application of this system is within an everyday automobile, where CAN is used to send data between the different parts within a car, some of which are shown in Figure 1.

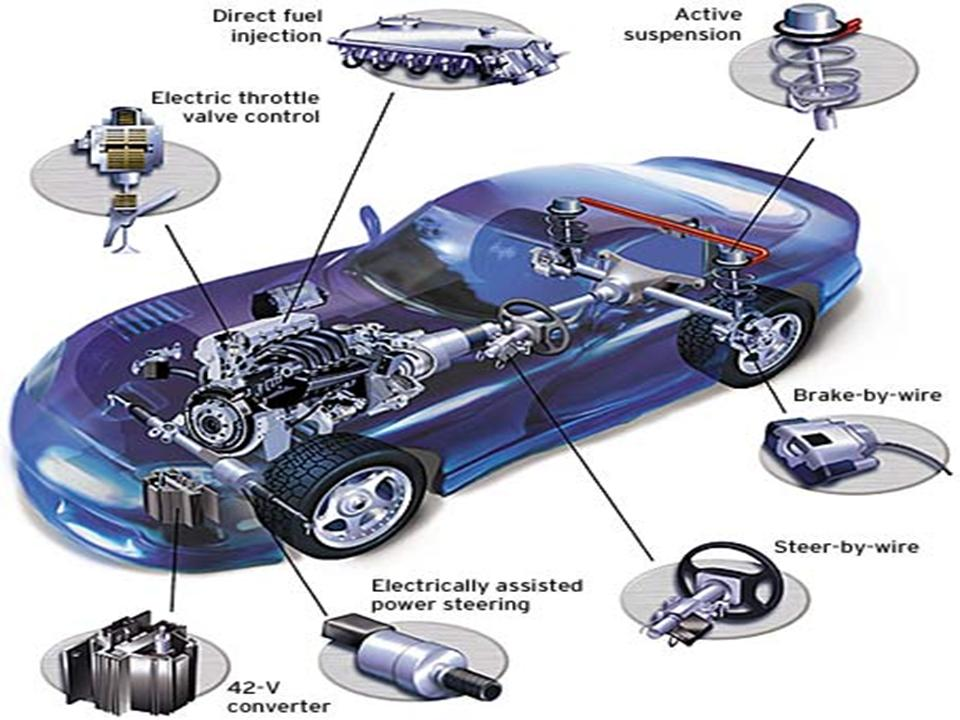


Figure 1: Different Nodes Within a Car's CAN System

The goal of this lab was to interface with an RC car set up to communicate using CAN and interact with the different peripherals, such as the turns signals, lights, horn, motor, and steering, much like how a driver would control an actual car. Once control of the RC car was established, three meters, current, temperature, and motor speed meters, were used to display information requested from the car. By completing this lab, a greater understanding of how CAN communication protocols work and the applications of CAN systems will be achieved.

**Procedure**

The first portion of the lab involved using LabVIEW to interact with the RC car using the CAN bus and make a virtual dashboard that could be used to send and display information. A basic LabVIEW VI, which allowed for sending messages with specified addresses and values, as well displaying the motor speed every several seconds, was already set up as a starting point. The appropriate buttons, lights, and meters were added to the dashboard to indicate the left and right turns signals, lights, horn, wheel direction, motor speed, motor current, and system temperature. In order to receive information from these nodes, the part of the LabVIEW VI that received messages from the bus had to be modified to look for each node address and output the received information to the appropriate dashboard items. Using the addresses and data formats shown in Tables 1 and 2, commands could be sent to the car using the transmit VI, and the information could be received and displayed by the final dashboard, shown in Figure 2.

Table 1: Car Controller Command Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Message ID** | **Data** |
| Headlights | 0x01 | 2 bytes: 0=Off, 1=On |
| Left Turn Signal | 0x02 | 2 bytes: 0=Off, 1=On |
| Right Turn Signal | 0x03 | 2 bytes: 0=Off, 1=On |
| Horn | 0x04 | 2 bytes: 0=Off, 1=On |
| Drive Motor | 0x05 | 2 bytes: 0 – 4095 |
| Steering Servo Motor | 0x06 | 2 bytes: 0 – 4095 but must be mapped to range: 850 (fully right) – 2150 (fully left) |

Table 2: Car Meter Monitor Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Message ID** | **Data** |
| Temperature | 0x07 | 2 bytes: ADC Reading 0-4095  (50°C => 910) |
| Rotational Speed (RPM) | 0x08 | 2 bytes: 0 - ~700 |
| Motor Current draw | 0x09 | 2 bytes: PIC ADC Reading 0 – 4095 |
| Left Turn Signal | 0x0A | 1 byte: 0=Off, 0xFF=On |
| Right Turn Signal | 0x0B | 1 byte: 0=Off, 0xFF=On |

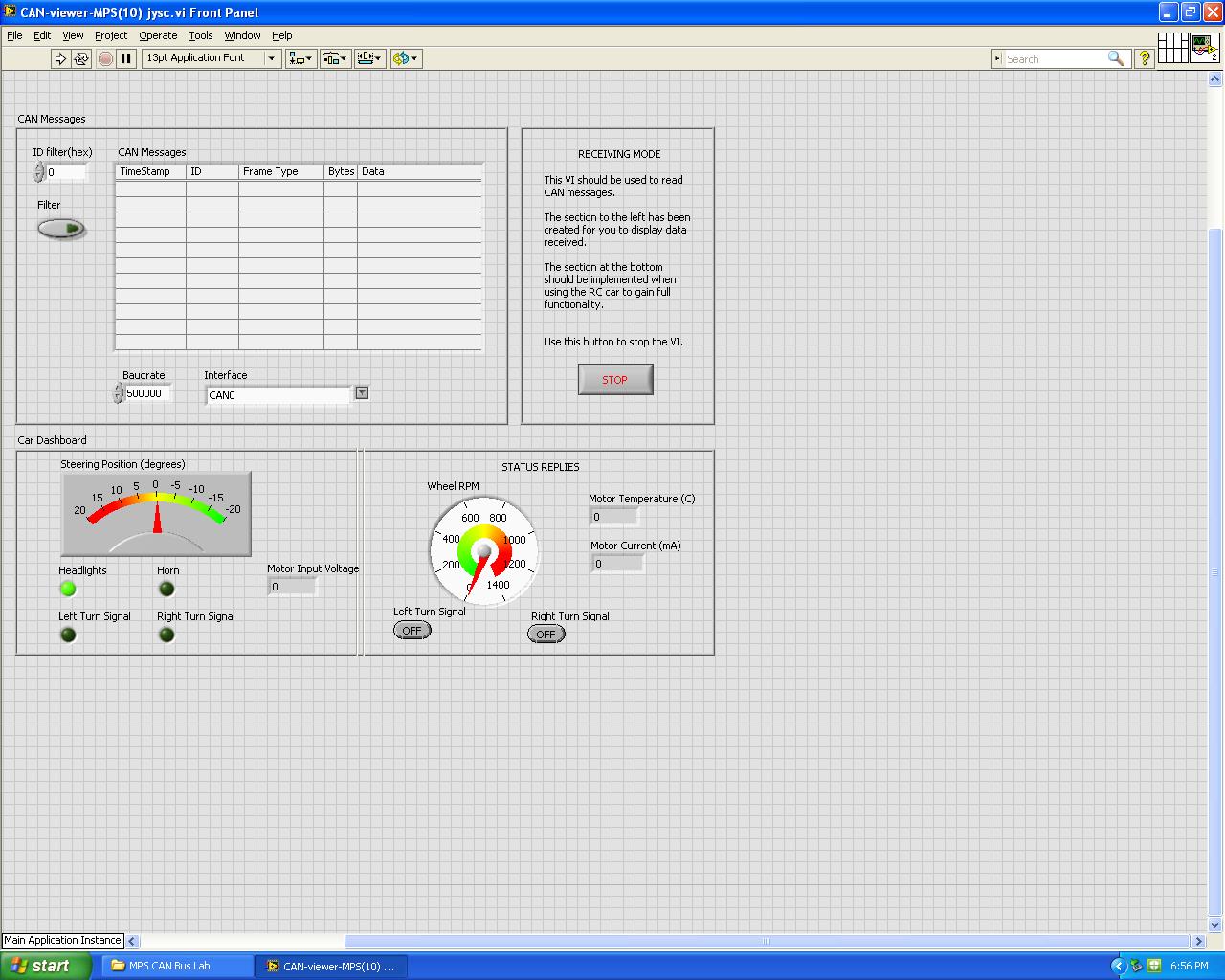


Figure 2: Final Dashboard for RC Car

Once the LabVIEW dashboard was working correctly, an understanding of how the CAN system works was established and code for the transmission of commands was developed. Using the headlights as an example, a switch was used to send the on or off signal to the 8051 microcontroller. The developed program would poll the appropriate input pin and send data to car over the CAN bus using functions included in the can.h library, which were as follows:

* *can\_get\_tx\_buf( ) -*  returned a reference to the transmit buffer for the bus
* *can\_set\_address\_std(can\_buffer, address\_ID)*  - sets the node address, *address\_ID*, for data stored in *can\_buffer*
* *can\_set\_buffer\_data(can\_buffer, data\_str, num\_bytes)* – filled *can\_buffer* with the data specified in *data\_str*, with *num\_bytes* being how many bytes of data
* *can\_send\_tx\_buf(can\_buffer)* – transmits the data in *can\_buffer* over the CAN bus

Continuing with the headlights example, the address would be set to 0x01, the data set to “\x00\x01” for On or “\x00\x00” for Off, and the number of bytes set to 2. The *can\_send\_tx\_buf* function would then be called to send the data to the RC car and turn the headlights on or off depending on the orientation of the switch. The turn signals and horn were toggled in a similar fashion by using a second switch and pushbutton respectively, while the steering servo and motor were controlled using potentiometer knobs. The voltage from the knobs would be read into the ADC on AIN0.0 and AIN0.1 for the motor and steering respectively. The motor ADC value was sent directly over the bus since the motor receives a value between 0 and 4095, while the steering value had to be normalized to the range of 850 – 2150 to prevent damage to the servo. The data was then sent over the bus using the functions specified above. The full transmit code can be found in Appendix A.

**Receive code procedure goes here (will write this once the code is uploaded to Git)**

* Explains what we did in the lab in detail. Mention what we modified in LabVIEW and how we modified it, the addressing scheme of each sensor and how we used that for both LabVIEW and the other parts of the lab.
* How we approached the transmitter and receiver code design, including pertinent details such as the ADC, DAC, PWM, etc.

**Challenges and Results**

For the LabVIEW part of this lab, the dashboard was set up successfully and data was sent over the CAN bus using the CAN-sender-MPS.vi panel. No large problems occurred during this portion of the lab, however this mainly because one of the group members was already exposed to LabVIEW in the past and the lab assignment had good detail on how to modify the given LabVIEW code. If neither member had LabVIEW exposure, this could have been less straightforward and more confusing.

This lab has required the most debugging of any of the labs so far. The car is a fairly large system with multiple subsystems that all need to be able to communicate with each other to work together. Additionally, the CAN code is already provided for use, so it is up to the users of this code to understand how it works to properly implement it for the lab’s application. To start off, the team had trouble getting the given example code to properly link and compile which was resolved by getting a fresh copy of the code from the professor. This may have been due to not properly downloading the original files.

It was unclear how to format the data to send when writing a speed to the CAN address for the motor speed. Ultimately this was because the confusing instructions were meant to be have been omitted from a deprecated version. The motor speed is represented with two bytes and the group needed to determine which order the bytes were transmitted since this was not a simple ‘on’ or ‘off’ signal.

The receiving code that was used for the display box was even more troublesome than the transmitting controller code. To start off, the receive code was having strange issues where the box would seemingly randomly cease working after 10 seconds to a minute. After some diagnostics, it was found that the code appeared to stop executing altogether. This was ultimately attributed to not properly flushing the receive buffer in code. It was first discovered that it was not proper to simply read the information transmitted from the transmitter box to then display. Instead, it was necessary to send a request to the car itself asking for the sensor status. To achieve this, the team needed to work out how the provided CAN code was meant to implemented in greater detail. When sending an RTR, the typical CAN transmit function is not meant to be used at all. This was made to work after some trial and error with using the given functions. Even with this working, the status information that was returned by the car to the receiver box would sometimes be incorrect. This also appeared to be random and occurred every few seconds where the status LED for a turn signal would be incorrect for just one iteration of the code. This may have been attributed to using long wires on the CAN bus which may also have not been properly terminated. Lastly, the team was unable to get the PWM indicator to work for the receiver. The cause was never determined, but the likely culprit is the hardware box itself may have poor wiring or a missing connection.

In past labs, the team rarely encountered debugging challenges that would take more than an hour to find a solution, and there were notably fewer altogether. This lab proved to be much more challenging to debug due to the complexity of the system and working with provided code where the functions were not documented. Overall, this lab was a huge success because the team was able to learn a lot about types of communications and further improved their debugging skills more so than would have been achieved for the alternative lab.

**Future Improvements**

In the future, this lab could be improved by putting the car controller hardware on the actual car itself. This would allow for a much cleaner workspace without as many wires hanging around in places where they could be unplugged. This would also allow the car to actually be driven to show off the features of the completed system and perhaps even demoed to other students as an example of what projects can be completed in this course. The documentation for the CAN code could be improved on so the users have an easier time determining what is actually happening with the CAN network making debugging simpler. The team would also put effort into looking into potential issues with the receiver box which may have led to the problems with sporadic false readings on the CAN bus and the temperature sensor not functioning properly. This is definitely one of the better and more involved labs that the team had the opportunity to work on this semester and the time commitment, learning outcomes, and overall complexity to this lab far exceeded the alternative lab option. In the future it could be possible to have this lab be used as a final project, though perhaps the teams could also write the CAN code to make the lab even more involved as an option.

**Conclusion**

* Quick summary of what was stated in the paper - what was learned in the lab, an overview of the problems faced, and what was or was not accomplished.

**Appendix A: CAN Bus Transmit Code**

// CAN transmit code for the RC car

#include <c8051f040.h>

#include "sysinit.h"

#include "uart0.h"

#include <stdio.h>

#include "can.h"

static char buf[40]; // for sprintf( )

void init\_xbar( ) {

char save = SFRPAGE;

SFRPAGE = CONFIG\_PAGE;

XBR0 = 0x04; // enable UART0

XBR1 = 0x00; // enable nothing

XBR2 = 0x40; // enable the crossbar

XBR3 = 0x80; // enable CAN

P0MDOUT &= ~0x02; // RX pin input

P0MDOUT |= 0x01; // TX

P3MDOUT &= 0xF0;

P3MDIN |= 0x0F;

// write "0xdead" to the watchdog register, disabling the watchdog

WDTCN = 0xde;

WDTCN = 0xad;

// ADC initialization

SFRPAGE = ADC0\_PAGE;

AMX0CF = 0x00;

AMX0SL = 0x00;

ADC0CF = 0x40;

ADC0CN = 0x00;

REF0CN = 0x03;

AD0EN = 1;

SFRPAGE = save;

}

void boot\_system( ) {

char SFRPAGE\_SAVE = SFRPAGE; // Save Current SFR page

SFRPAGE = CONFIG\_PAGE;

init\_sysclk( );

init\_xbar();

uart0\_init( );

can\_init( );

printf("Hello World!\r\n");

EA = 1; /\* enable interrupts now that everything is initialized \*/

SFRPAGE = SFRPAGE\_SAVE;

}

void main( ) {

unsigned char ADval[2];

unsigned int result;

CAN\_BUFFER canbuf;

boot\_system( );

while (1) {

canbuf = can\_get\_tx\_buf( ); // acquire the can tx buffer

// Headlighs

can\_set\_address\_std(canbuf,0x01);

if(P3 & 0x01 > 0) // poll the headlight switch

can\_set\_buffer\_data(canbuf, "\x00\x01", 2); //send On

else

can\_set\_buffer\_data(canbuf, "\x00\x00", 2); // send Off

can\_send\_tx\_buf(canbuf);

// Right Turn Signal

can\_set\_address\_std(canbuf,0x03);

if(P3 & 0x02 > 0)

can\_set\_buffer\_data(canbuf, "\x00\x01", 2); // On

else

can\_set\_buffer\_data(canbuf, "\x00\x00", 2); // Off

can\_send\_tx\_buf(canbuf);

// Light Turn Signal

can\_set\_address\_std(canbuf,0x02);

if(P3 & 0x04 > 0)

can\_set\_buffer\_data(canbuf, "\x00\x01", 2);

else

can\_set\_buffer\_data(canbuf, "\x00\x00", 2);

can\_send\_tx\_buf(canbuf);

// Horn

can\_set\_address\_std(canbuf,0x04);

if(P3 & 0x08 > 0)

can\_set\_buffer\_data(canbuf, "\x00\x01", 2);

else

can\_set\_buffer\_data(canbuf, "\x00\x00", 2);

can\_send\_tx\_buf(canbuf);

// Motor

SFRPAGE = ADC0\_PAGE;

AMX0SL = 0x00;

AD0INT = 0;

AD0BUSY = 1; // Initate AD conversion on AIN0.0 pin

while(AD0INT == 0);

ADval[1] = ADC0L;

ADval[0] = ADC0H;

SFRPAGE = 0x00;

// Send motor data over CAN bus

can\_set\_address\_std(canbuf,0x05);

can\_set\_buffer\_data(canbuf, ADval, 2);

can\_send\_tx\_buf(canbuf);

// Direction control

SFRPAGE = ADC0\_PAGE;

AMX0SL = 0x01; // switch to AIN0.1 pin for ADC

AD0INT = 0;

AD0BUSY = 1; // Initiate AD conversino on AIN0.1 pin

while(AD0INT == 0);

ADval[0] = ADC0L;

ADval[1] = ADC0H;

SFRPAGE = 0x00;

// Normalize the vlue to a range of 850 - 2150 to protect servo components

result = ((unsigned int)ADval[0] + (ADval[1]<<8))\*13/41+850;

ADval[1] = result;

ADval[0] = result>>8;

// Output steering data to CAN bus

can\_set\_address\_std(canbuf,0x06);

can\_set\_buffer\_data(canbuf, ADval, 2);

can\_send\_tx\_buf(canbuf);

}

}

**Appendix B: Code**