**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.

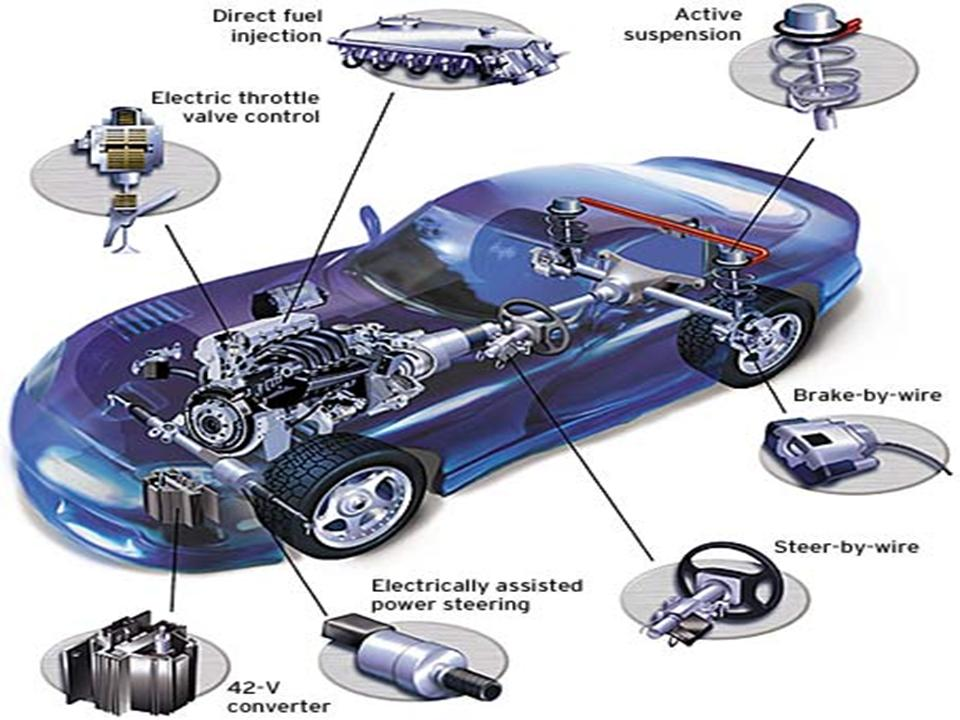


Figure : Different Nodes Within a Car's CAN System

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 : 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 : 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 |

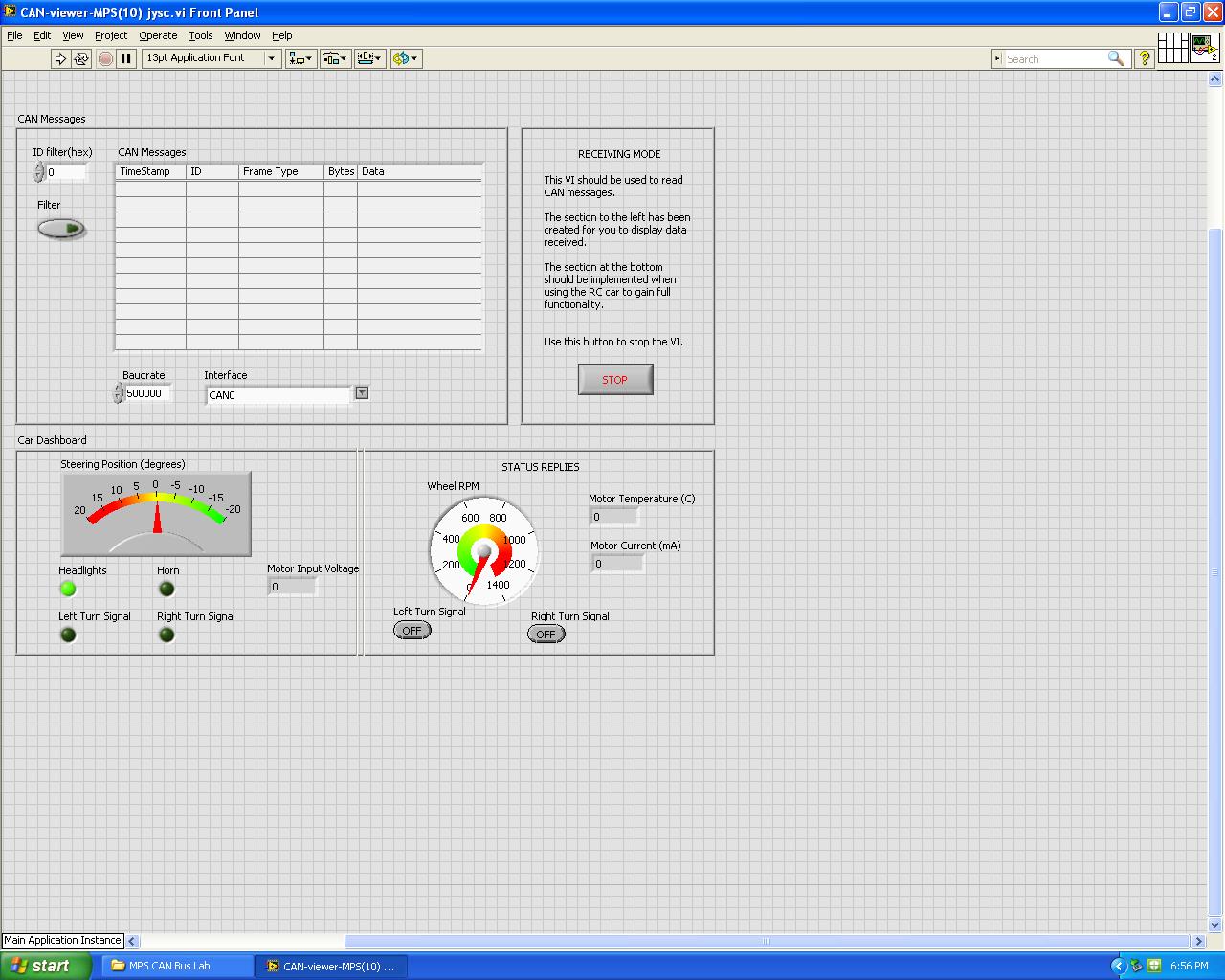


Figure : Final Dashboard for RC Car

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.

Developing the CAN receive code was similar to the transmit code, but had a few small changes and the additional task of listening to the bus. To acquire data from the nodes on the car, the sensors had to be sent a signal to request the data. The *can\_send\_rtr(can\_buffer)* function was used to poll the specified node address and the *can\_get\_rx\_msg()* function acquired the data being sent back over the CAN bus. To continuously poll all the sensors, a counter was set up so that on each increment of the counter, a different address was sent over the bus. Then, depending on from which address the RX message came, the data would be processed according to the table in Appendix C and be output to the physical dashboard box. The turn signals would light up corresponding left and right LEDs, and the motor current, miles per hour (MPH), and temperature would be output to analog meters. The motor speed and current were output through the two DACs on the 8051, while the temperature meter took in a PWM signal that could be generated using the Programmable Counter Array (PCA). The full receive code can be found in Appendix B.

**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.

The rest of the lab 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 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. Once the correct byte order was determined, the motor speed could be successfully controlled using the potentiometer knob.

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 control box would cease working after ten 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 a solution, the team needed to analyze the given CAN code in greater detail.

After looking at some example code, it was discovered that 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, specifically for the turn signals. The errors appeared to be random and occurred unpredictably, causing the LED status to be incorrect for a single iteration of code, which resulting in a rapidly flashing LED, rather than a solidly-lit one, when either of the turn signals were active. The observed errors may have been attributed to using long wires on the CAN bus which also may not have been properly terminated. However, despite these errors, the turn signal LEDs, motor speed, and motor current indicators were successfully implemented.

Lastly, the team was unable to get the temperature indicator to work for the receiver due to the inability to output a PWM signal. The cause of this was never determined, but the likely culprit is the hardware box itself may have poor wiring or a missing connection. Instead, the temperature readings were displayed on the Prcomm Plus terminal and verified to be correct.

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 well. 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 keypad 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, which would make debugging simpler. Given more time, 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 lab 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**

This lab provided a great introduction and in-depth exercise of how CAN communication is implemented and what applications it can be used for. Using the RC car allowed for a number of different nodes to interact with and control, giving the lab a greater appeal and making it more realistic to how an actual CAN system would be implemented. To successfully control the car and display pertinent information, such as speed, temperature, etc., the CAN protocol and accompanying C functions had to be well understood. Debugging was also a large part of this lab due to the many different problems that could and did arise. Working with the RC car felt like a much larger project as compared to previous labs due to the implementation of many different aspects of the 8051, the ADC, DAC, PWM, and UART for example, as well as learning a new communication protocol from scratch and using it correctly. The time commitment and amount of work seemed to be much greater than the alternative lab, the keypad and LCD display, but the skills that were developed were worth the extra effort.

**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: CAN Bus Receive Code**

// CAN receive 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 = 0x0C; // enable UART0

XBR1 = 0x00; // enable nothing

XBR2 = 0x40; // enable the crossbar

XBR3 = 0x80; // enable CAN

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

P0MDOUT |= 0x01; // TX

P0MDOUT |= 0x34;

P3MDOUT &= 0xF0;

P3MDIN |= 0x0F;

// Disable watchdog timer

WDTCN = 0xde;

WDTCN = 0xad;

// ADC initialization

SFRPAGE = ADC0\_PAGE;

AMX0CF = 0x00;

AMX0SL = 0x00;

ADC0CF = 0x40;

ADC0CN = 0x00;

REF0CN = 0x03;

AD0EN = 1;

// DAC initialization

SFRPAGE = DAC0\_PAGE;

DAC0CN = 0x80;

SFRPAGE = DAC1\_PAGE;

DAC1CN = 0x80;

// PCA initialization to generate PWM

SFRPAGE = 0x00;

PCA0MD = 0x88;

PCA0CPM0 = 0x42;

PCA0CPL0 = 0x00;

PCA0CPH0 = 0x00;

PCA0CN = 0x40;

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 counter = 0;

unsigned int result1;

unsigned int result2;

CAN\_BUFFER canbuf;

boot\_system( );

while (1) {

canbuf = can\_get\_tx\_buf(); // get the CAN tx buffer reference

SFRPAGE = CAN0\_PAGE;

switch(counter)

{

case 100: // don't send too many requests over the BUS, so delay by an arbitrary number

can\_set\_address\_std(canbuf,0x07); // Temperature

can\_send\_rtr(canbuf);

break;

case 400:

can\_set\_address\_std(canbuf,0x09); // Motor current

can\_send\_rtr(canbuf);

break;

case 700:

can\_set\_address\_std(canbuf,0x0A); // Left turn signal

can\_send\_rtr(canbuf);

break;

case 900:

can\_set\_address\_std(canbuf,0x0B); // Right turn signal

can\_send\_rtr(canbuf);

break;

case 1000:

counter = 0;

break;

default:

//counter = 0;

}

counter++;

if (canbuf = can\_get\_rx\_msg( )) {

switch(can\_get\_address(canbuf))

{

case 0x0007: // Temp sensors sends 2 bytes

result1 = can\_get\_data\_byte(canbuf, 0);

result2 = can\_get\_data\_byte(canbuf, 1);

PCA0CPL0 = 0x88; // Getting an output PWM signal was ultimately unsuccessful

PCA0CPH0 = 0x88; // This was an attempt to get an arbitrary output to the analog meter

//PCA0CPL0 = (unsigned int)(((float)(result1<<8)+result2)\*-51/182 + 255);

// We were able to display the temperature data on the Procomm Plus terminal

printf("%d %d %d\r\n", result1, result2, (unsigned int)(((float)(result1<<8)+result2)\*-51/182 + 255));

break;

case 0x0008: // RPM sensors sends 2 bytes

result1 = can\_get\_data\_byte(canbuf, 0);

result2 = can\_get\_data\_byte(canbuf, 1);

printf("%d %d\r\n", result1, result2);

SFRPAGE = DAC1\_PAGE;

DAC1L = result2\*3.35; // Output to DAC1

DAC1H = result1\*3.35;

SFRPAGE = 0x00;

break;

case 0x0009: // Motor current sensors sends 2 bytes

result1 = can\_get\_data\_byte(canbuf, 0);

result2 = can\_get\_data\_byte(canbuf, 1);

SFRPAGE = DAC0\_PAGE;

if(result1 == 255) // If the motor was off, the current would indicate 255

{

DAC0L = 0x00;

DAC0H = 0x00;

}

else // Otherwise, output the actual sensor data

{

DAC0L = result2>>0;

DAC0H = result1>>0;

}

SFRPAGE = 0x00;

break;

case 0x000A: // Left turn signal

if(can\_get\_data\_byte(canbuf, 0) == 0xff) // On

{

P0 &= 0xEF;

}

else // Off

{

P0 |= 0x10;

}

break;

case 0x000B: // Right turn signal

if(can\_get\_data\_byte(canbuf, 0) == 0xff) // On

{

P0 &= 0xDF;

}

else // Off

{

P0 |= 0x20;

}

break;

default:

printf("");

}

can\_free\_rx\_buf(canbuf); // Free the RX buffer

}

}

**Appendix C: RC Car CAN Arbitration IDs, Functions, and Value Ranges**