Lab 7 Report

EE 4341

Spring 2016

Cole Nielsen niels538@umn.edu

Abstract

An example code implementing a CAN bus in loopback mode was modified to work with the PIC32 USB Starter Kit 2 board. The code was modified in order to transmit data pertaining to button presses on the PIC32 board over the CAN bus. Then, via loopback the transmitted values were received and display on the LEDs corresponding to each switch on the PIC board to show which had been pressed.

1 Introduction

This lab focused on using the CAN (Controller Area Network) bus with the PIC32 USB Starter kit to transmit and receive data pertaining to button presses on the board in loopback mode. CAN is a robust, industry standard automotive communications protocol which is designed to work without a host computer. CAN works sharing a differential signal bus between several participating devices via arbitration, and data is sent between devices in a message format. Loopback is an operational mode of a transceiver were transmitted data flows back into the transciever and is then received. This is a useful mode for verifying proper operation of a transceiver. In this lab an example code was used to set up the CAN bus to work in loopback mode, using the PIC32's CAN transceiver peripheral to perform the transceiving. This code was then modified by adding code to check for button presses, encode the button presses into a data packet to be transmitted, and then decode the received data in order to light up the LEDs in accordance to button press transmitted data.

2 Implementation

Implementation for this lab was performed using a given help file, which implemented the CAN loopback feature. Using this file, the button pressing and corresponding LED displaying of button presses was implemented

The Provided Code

The provided code will be discussed in brief to give an idea of how it works without looking to deeply at the code as it is primarily register initialization. Essentially, the provided code turns on the CAN module, sets fifo buffer sizes and enables them, sets the message ID and then puts the module into loopback mode. The code in the main function works essentially by data to be transmitted in a transmit buffer, telling the CAN module to transmit, and then waiting until data is looped back. That data is then available in another buffer.

Implemented Button Press/LED Display Code

In order to implement the button press sensing and corresponding LED displaying of received data, the input switches, onnected to RD6, RD7 and RD13 were set to inputs, and then were pulled high using the internal pullup resistors of the PIC32. The board's LEDs, connected to RD0, RD1 and RD2 were also set to be outputs. This is all shown in the following black.

```
    TRISDbits.TRISD0=0;
    TRISDbits.TRISD1=0;
    TRISDbits.TRISD2=0;
    TRISDbits.TRISD6 = 1;
    TRISDbits.TRISD7 = 1;
    TRISDbits.TRISD13 = 1;
    CNPUEbits.CNPUE6 = 1;
    CNPUEbits.CNPUE7 = 1;
    CNPUEbits.CNPUE13 = 1;
    PORTDbits.RD0 = 0;
    PORTDbits.RD1 = 0;
    PORTDbits.RD2 = 0;
```

The main while loop of the code implemented the aforementioned functionality is in the following code listing. It is in a infinite while loop so it perpetually checks over the buttons, transmits/receives data and displays data. The first three if/else statements are used to check for button presses and encode them onto to_send, the data word that is transmitted over the CAN bus. Each button is checked by checking the level of each pin connected to a button. The buttons are active low (pulled up high, goes low with button press). The values of the button presses are encoded onto to_send with the following scheme: each of the first three bits of to_send represents a button, and if the button is pressed the corresponding bit if toggled using XOR. The code that follows after is the example code, which gets the address of the TX buffer, and then writes necessary data for each of the CAN message fields into the Tx bus. Transmit is then enabled, and the microcontroller waits until data is looped back, which verified by polling the CAN reveive interrupt bit. Then, the three PORTD assign statements write the three data bits of the CAN message data containing the button press data to be outputed to the LEDs so the received data can be observed. The last several statments are used to increment the fifo buffer, and then to implement a debounce. The switches on the PIC32 board were very erratic, so a simple time delay using a for loop was used in order to wait out bouncing, and then three while statements were used following to wait until each of the buttons were released before rechecking the button press states and retransmitting (in order to avoid toggling the LEDs many times for one button press).

```
  while(1) {
//                                      transmit


     if(!PORTDbits.RD6) //encode the three switch values to first 3 bits of
         to_send
        to_send ^= 1; //toggle bit0 if switch0 pressed
     else if(!PORTDbits.RD7)
         to_send ^= 2; //toggle bit1 if switch1 pressed
     else if(!PORTDbits.RD13)
         to_send ^= 4; //toggle bit2 if switch2 pressed
```

```
      addr = PA_TO_KVA1(C1FIFOUA1);   // get FIFO 1 (the TX fifo) current message
         address
      addr[0] = MY_SID;               // only the sid must be set for this example
      addr[1] = sizeof(to_send);      // only DLC field must be set, we indicate
         4 bytes
      addr[2] = to_send;              // 4 bytes of actual data
      C1FIFOCON1SET = 0x2000;         // setting UINC bit tells fifo to increment
         pointer
      C1FIFOCON1bits.TXREQ = 1;       // request that data from the queue be sent
//                                          receive

     while(!C1FIFOINT0bits.RXNEMPTYIF); // wait to receive data
     addr = PA_TO_KVA1(C1FIFOUA0);   // get the VA of current pointer to the RX
        FIFO

     PORTDbits.RD0 = (addr[2] & 1);
     PORTDbits.RD1 = ((addr[2] & 2)>>1);
     PORTDbits.RD2 = ((addr[2] & 4)>>2);

     C1FIFOCON0SET = 0x2000;         // setting UINC bit tells fifo to increment

     int j;
     for(j = 0; j<20000; j++); //delay to improve debouncing (switches sensitive)

     while(!PORTDbits.RD6); //wait until button release
     while(!PORTDbits.RD7);
     while(!PORTDbits.RD13);

     int j;
     for(j = 0; j<20000; j++); //delay to improve debouncing (switches sensitive)
  }
```

3 Summary

The outcome of this lab was a functioning code for transceiving data using the PIC32's CAN peripheral module. This sucessful operation was verified using a loopback and the boards buttons and LEDS. A significant problem faced in the implementation of this labs code was due to the sensitivity of the switches on the board. The PIC has an extemely weak pull up resistor used to hold the switch at an inactive high state, where any slight perturbation (touching the metal housing of the switch, or a trace) seemed to cause the value read by the microcontroller to change erratically. To solve this problem a debouncing scheme using time delays had to be used to reject button bounce/erratic signals essentially by waiting for the switch to reach a steady state.