Application Note

Interfacing the CS5521/22/23/24/28 to the 80C51

TABLE OF CONTENTS

1. INTRODUCTION 1
2. ADC DIGITAL INTERFACE 1
3. SOFTWARE DESCRIPTION 1
3.1 Main Program Loop 2
3.2 Initialize 2
3.3 Transfer Data To/From ADC 3
3.4 Transfer Data To/From PC 3
3.5 Decoding PC Commands 5
3.6 Sample Index 6
3.7 Test Modes 6
4. MAXIMUM SCLK RATE 8
5. DEVELOPMENT TOOL DESCRIPTION 8
6. CONCLUSION 8
7. APPENDIX: 80C51 MICROCONTROLLER CODE 9

1. INTRODUCTION

This application note details the interface of Crystal Semiconductor's CS5521/22/23/24/28 Analog-to-Digital Converter (ADC) to an 80C51 microcontroller. It includes the complete code written for the CDB5521/22/23/24/28 Evaluation Board, which also interfaces the 80C51 to a PC. All algorithms discussed are included in Section 7. "Appendix: 80C51 Microcontroller Code" on page 9.

2. ADC DIGITAL INTERFACE

The CS5521/22/23/24/28 interfaces to the 80C51 through either a three-wire or a four-wire interface. Figure 1 depicts the interface between the two devices. This software was written to interface to Port 1 (P1) on the 80C51 with either type of interface. The ADC's serial port consists of four control lines: $\overline{CS}$, SCLK, SDI, and SDO.

$\overline{CS}$, Chip Select, is the control line which enables access to the serial port.

SCLK, Serial Clock, is the bit-clock which controls the shifting of data to or from the ADC's serial port.

SDI, Serial Data In, is the data signal used to transfer data from the 80C51 to the ADC.

SDO, Serial Data Out, is the data signal used to transfer output data from the ADC to the 80C51.

3. SOFTWARE DESCRIPTION

This note details all of the algorithms contained in the CDB5521/22/23/24/28 Evaluation Board software. The software is written for the 80C51 microcontroller on the evaluation board. The more important communication algorithms are written in

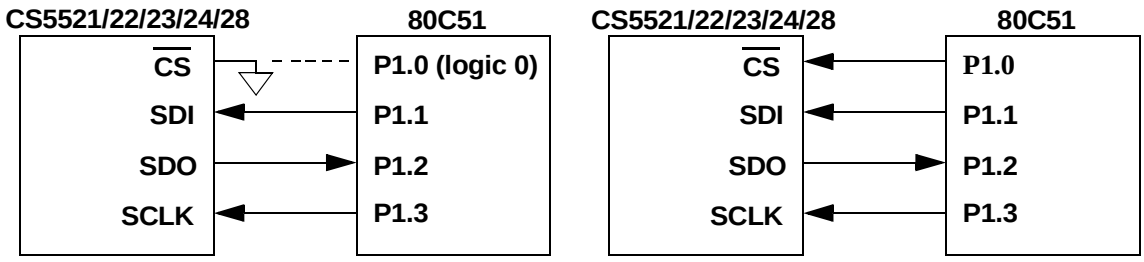

Figure 1. 3-Wire and 4-Wire Interfaces

CIRRUS LOGIC
P.O. Box 17847, Austin, Texas 78760
(512) 445 7222 FAX: (512) 445 7581
http://www.cirrus.com

Copyright © Cirrus Logic, Inc. 1999
(All Rights Reserved)

80C51 assembly language, providing a good set of tools for both C and assembly programmers to build their own designs upon. While reading this application note, please refer to Section 7. “Appendix: 80C51 Microcontroller Code” on page 9 for the code listing.

3.1 Main Program Loop

The *main* loop of the program is responsible for calling all other algorithms and controlling the program flow. Figure 2 depicts the data flow of the *main* loop. When the evaluation board is first powered up, or after a system reset, the microcontroller and ADC are set up by calling the *initialize* routine. The value present on the DIP switches is then checked. If this value is anything other than zero, the program goes into test mode. If the value is equal to zero, the program goes into a continuous loop, where it receives commands from the PC, decodes them, and performs the desired actions.

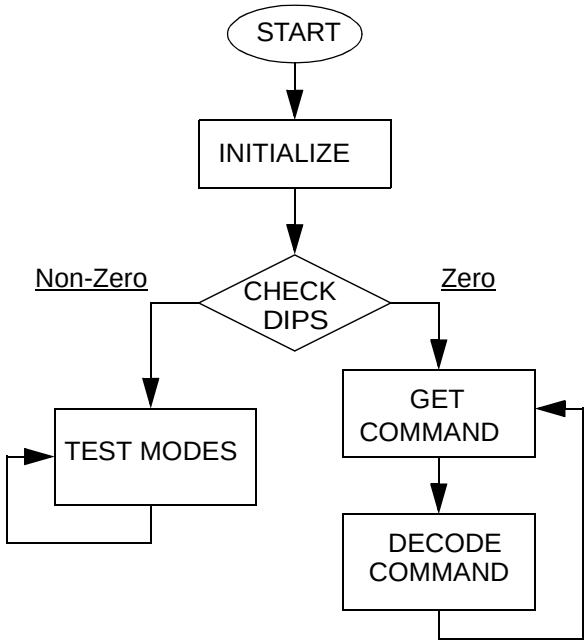

Figure 2. Software Flow Diagram

3.2 Initialize

Initialize is used to configure the microcontroller and the ADC to the proper settings that will allow the PC to control the ADC indirectly through the microcontroller. The 80C51’s port P1 is configured as depicted in Figure 1 (for more information on configuring ports refer to the 80C51 Data Sheet). Next, port P2 is written twice, once with all zeros and once with all ones, with a delay in between to control the LEDs attached to the port and inform the user that the board has been reset successfully. Port P3 is then set up to use the 80C51’s internal UART to interface to the PC at 9600 baud, no parity bit, eight data bits, and one stop bit. To allow time for the ADCs oscillator to start up, a delay state is entered (oscillator start-up time is typically 500ms). After this delay, the ADC is ready to accept data. However, it is a good idea to reset the ADC’s serial port before communicating with it. To reset the serial port on the ADC, SDI is asserted, and 255 SCLKs are provided. SDI is then cleared, and one final SCLK is provided (this is a slight overkill, as only 15 bytes of logic 1’s followed by a

Contacting Cirrus Logic Support
For a complete listing of Direct Sales, Distributor, and Sales Representative contacts, visit the Cirrus Logic web site at:
http://www.cirrus.com/corporate/contacts/

SPI is a trademark of Motorola.
MICROWIRE is a trademark of National Semiconductor.

Cirrus Logic, Inc. has made best efforts to ensure that the information contained in this document is accurate and reliable. However, the information is subject to change without notice and is provided “AS IS” without warranty of any kind (express or implied). No responsibility is assumed by Cirrus Logic, Inc. for the use of this information, nor for infringements of patents or other rights of third parties. This document is the property of Cirrus Logic, Inc. and implies no license under patents, copyrights, trademarks, or trade secrets. No part of this publication may be copied, reproduced, stored in a retrieval system, or transmitted, in any form or by any means (electronic, mechanical, photographic, or otherwise). Furthermore, no part of this publication may be used as a basis for manufacture or sale of any items without the prior written consent of Cirrus Logic, Inc. The names of products of Cirrus Logic, Inc. or other vendors and suppliers appearing in this document may be trademarks or service marks of their respective owners which may be registered in some jurisdictions. A list of Cirrus Logic, Inc. trademarks and service marks can be found at http://www.cirrus.com.

final byte with its LSB at logic 0 are needed to reset the serial port). This places the ADC in the command state, where it awaits a valid command. Finally, before returning to the *main* routine, the mode pin is queried to determine whether a three or four-wire interface is being used.

3.3 Transfer Data To/From ADC

Transferring data to and from the ADC is done through the *transfer_byte* and *receive_byte* functions. *Transfer_byte* takes one byte of input and sends it to the ADC one bit at a time (MSB first) by applying the bit information to P1.1 (SDI) and then pulsing P1.3 (SCLK). This is repeated eight times to transfer the entire byte of data to the ADC. *Receive_byte* works in just the opposite direction, obtaining byte information MSB-first on the P1.2 (SDO) line as it provides the clock on P1.3. The functions *write_to_register* and *read_register* make use of these byte transfer functions to send and receive internal register information in the proper order from the ADC. *Write_to_register* writes four bytes of information to the ADC using *transfer_byte*. The command byte is written first, followed by the high, middle, and low bytes of the 24-bit register word. *Read_register*, on the other hand, sends a command byte to the ADC using *transfer_byte*, and then obtains the 24-bit register word by calling *receive_byte* three times in succession. Due to the conversion time delay, reading conversion data requires a slightly different method than reading register information. The *read_conversion* algorithm is designed for this purpose. The configuration register is read and stored in memory using the *read_register* command. The LP, RC, and MC bits are then masked to zero, and the register is written using *write_to_register*. The command to start a conversion is then written using *transfer_byte*, and SDO is polled until the ADC indicates that the conversion is complete. Once a valid conversion has been obtained, eight zeros are sent to the ADC with *transfer_byte* to start the transfer of data, at which time *receive_byte* is called three times to obtain the data word (see Figure 3 for more detail on how the information is organized among the three bytes) Figure 6 shows the conversion data timing.

MSB **High-Byte**

D23	D22	D21	D20	D19	D18	D17	D16

Mid-Byte

D15	D14	D13	D12	D11	D10	D9	D8

Low-Byte

D7	D6	D5	D4	D3	D2	D1	D0

A) 24-bit Conversion Data Word (CS5522/24/28)

MSB **High-Byte**

D15	D14	D13	D12	D11	D10	D9	D8

Mid-Byte

D7	D6	D5	D4	D3	D2	D1	D0

Low-Byte

1	1	1	0	CI1	CI0	OD	OF

B) 16-bit Conversion Data Word (CS5521/23)

0 - always zero, 1 - always 1

CI1, CI0 - Channel Indicator Bits

OD - Oscillation Detect, OF - Overflow

Figure 3. Bit Representation/Storage in 80C51

3.4 Transfer Data To/From PC

Transferring data to and from a PC through the 80C51's serial port is accomplished with the routines *txser* and *rxser*. These two routines rely upon status bits in the 80C51's Serial Control Register (SCON, located at address 0x98 Hex). *Txser* transfers data to the serial port by first polling TI (Transfer Interrupt, bit 1 in SCON) until the 80C51's serial buffer is empty, and ready to receive information. Once TI is at a logic high level, the software clears the TI bit and moves the byte to be transferred into the 80C51's serial buffer, where it is transmitted to the PC LSB first. *Rxser* receives a byte in a similar fashion, by polling RI (Receive Interrupt, bit 0 in SCON) until there is valid data available in the 80C51's serial buffer. When RI is at a logic high level, the data is transferred out of the serial buffer to an immediate memory location.

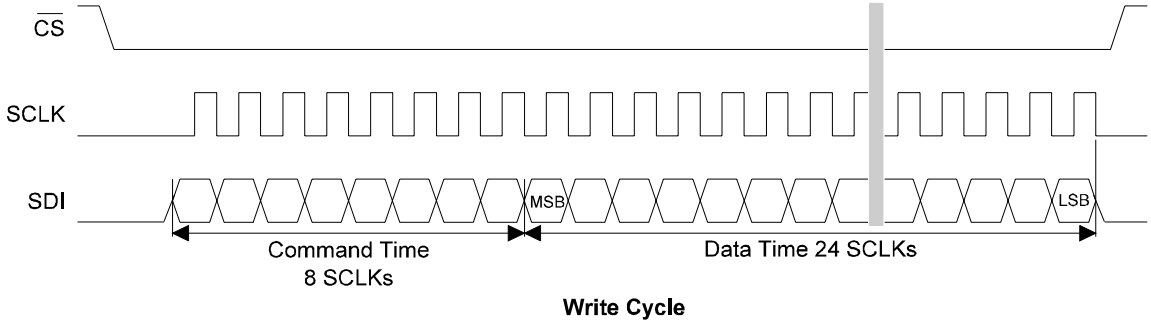

Figure 4. Write-Cycle Timing

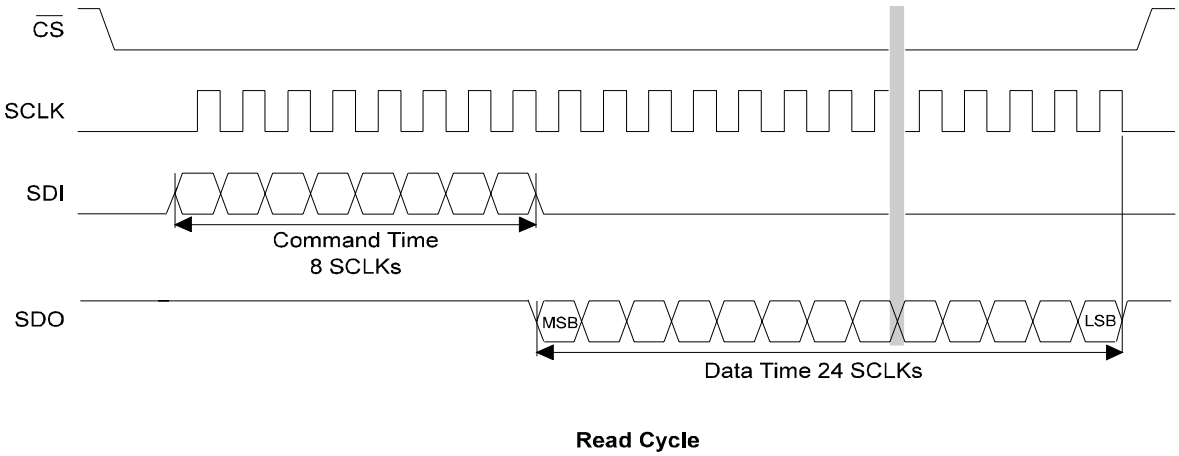

Figure 5. Read-Cycle Timing

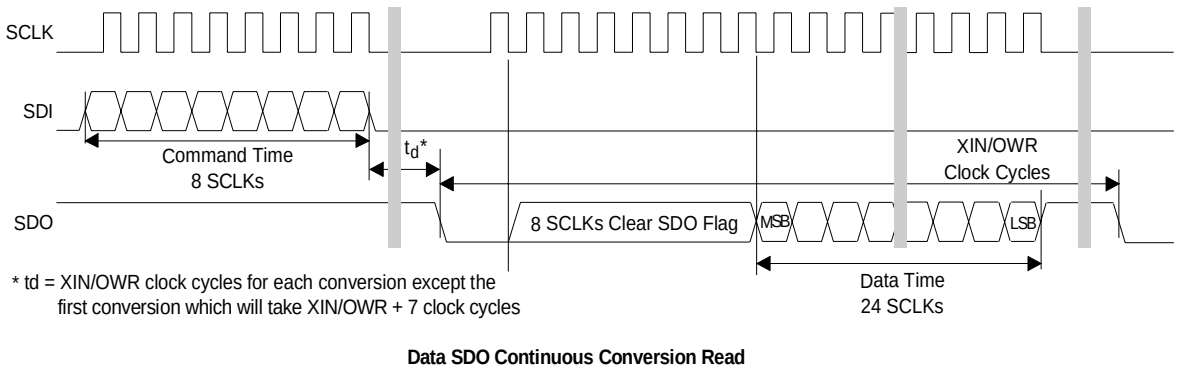

Figure 6. Conversion/Acquisition Cycle Timing

3.5 Decoding PC Commands

The *decode_command* routine is where most of the functionality of the program lies. It consists of many smaller routines to direct the flow of data to the proper locations. *Decode_command* accepts information from the PC, decides which tasks need to be done based on that information, and carries out those tasks accordingly. This is accomplished through the use of a very large switch statement, based on the input command byte, which tests for every possibility and performs the appropriate routines shown in Section 7. “Appendix: 80C51 Microcontroller Code” on page 9.

Write Register - If the PC has sent a command to write to a specific register in the CS5521/22/23/24/28, three more bytes are received from the PC using *rxser* (24-bit data is always transferred via the UART in the order: low byte, middle byte, high byte). Then the information, including the command byte is sent to the converter using the *write_to_register* function. In contrast with the PC serial data, data transferred to or from the ADC is always in the order: high byte, middle byte, low byte.

Write Channel Setup Registers - In the case that the PC requests to write to the Channel Setup Registers (CSRs), the program receives another byte from the PC to find out how many registers to write to. The command word 0x05(HEX) is sent to the ADC to begin writing to the registers. The data is received from the PC using *rxser*, and sent directly to the ADC using *transfer_byte*. This process of receiving data from the PC and transferring it to the ADC is repeated until the requested number of bytes has been sent to the PC.

Read Registers - When the decoded command is asking to read the ADC’s internal registers, *read_register* is called, using the command sent from the PC. The information obtained by this function is then sent to the PC using *txser*.

Read Conversion Data FIFO - To read the conversion data FIFO, the sample size requested, the conversion channel, and the loop bit status are received from the PC, in that order. The conversion channel is then sent to the ADC to begin a conversion, and SDO is polled until it falls, indicating that the conversion is complete. At this time, a byte of zeros is sent to the ADC to initiate the data transfer, and the program loops through to receive each 24-bit conversion word and send it to the PC, for as many times as the sample size that has been requested. If the LP bit is set, then another sample set is received from the ADC, but not sent to the PC.

Read Channel Setup Registers - To read the CSRs, this algorithm first obtains the number of registers to read from the PC using *rxser*. It then transfers the 0x0D(HEX) command to the ADC, indicating that the CSRs are to be read. Next, the individual registers are read from the ADC and sent to the PC until the number of registers specified by the PC command has been met.

Normal Conversions - For a normal conversion using any Setup, the *read_conversion* subroutine is executed using the appropriate command word. The data is then sent directly to the PC using *txser*.

Self Calibration - To perform a self-offset or self-gain calibration using any Setup, the appropriate command word is sent to the ADC, and SDO is polled until the calibration is complete to avoid sending a new command when the ADC is busy.

System Calibration - System-offset and system-gain calibrations use the same procedure as self calibration, but areseparated in the code to set the two command sets apart from one another.

Variable Number of Normal Conversions - This routine is used when a set of samples is requested from the PC (as opposed to a single sample). The sample index is obtained from the PC, as well as the Setup to be used. The *decode_sample_index* routine is then called to find out exactly how many conversions to perform. The conversion channel

data is then transferred to the ADC, and SDO is polled until the conversion is ready. A byte of zeros is then transmitted to the ADC to begin the data transfer, and the data word is received. The data word is then sent to the PC, and the process is repeated until the sample size has been reached. Once enough samples have been collected and transferred to the PC, a command byte consisting of all ones is sent to the ADC to instruct it to stop converting data, and the final conversion word is received from the ADC and discarded.

Variable Number of Offset or Gain Calibrations - When more than one offset or gain calibration is requested, the software obtains a sample index, the type of calibration (self or system), and which Setup is being used to calibrate from the PC. *Decode_sample_index* is called to determine the actual number of calibrations to be performed, and then the calibration type is sent to the ADC. SDO is polled until the calibration is completed, and then the calibration register is sent. The calibrated register is returned by the ADC, and sent to the PC. This process continues until the number of calibrations requested have been performed.

Serial Port Initialization - This routine re-initializes the serial port on the ADC if it is requested. To accomplish this, the SDI pin is set to a logic high level, and SCLK is pulsed 255 times. SDI is then brought low and a single SCLK pulse is sent. This initialization routine is the same as what is done within the start-up *initialize* command, but is limited to the serial port on the ADC.

Reset Converter Via RS and RV Bits - This routine resets the ADC, and returns all of its internal registers to their initial states. The command 0x03 followed by 0x800000(HEX) is sent to the ADC, which sets the RS (Reset) bit and nothing else. Then the 0x0B(HEX) register is read, and the low byte is masked for the RV (Reset Valid) bit. If a valid reset has occurred, the RS bit is cleared. If no valid reset has occurred, the routine continues to cycle and poll the RV bit until the reset is successful.

Read Output Latch Pins - The ADC's output latch pins (A0 and A1) are connected to pins P1.4 and P1.5, respectively, on the 80C51. If the PC requests the status of these pins, they are read from the proper pins and transmitted to the PC.

Arbitrary Read - This routine is used to read any number of bytes from the ADC and return them to the PC. The number of bytes requested is obtained from the PC, and then bytes are successively read from the ADC and transmitted to the PC until that number has been reached.

Arbitrary Write - This routine asks for one byte from the PC, and transfers it to the ADC.

If none of these conditions are met, the software does nothing but return to the *main* routine.

3.6 Sample Index

When the PC requests an entire sample set, the microcontroller code assumes that it will be sending a sample index. This number, 0-10, is passed to the *decode_sample_index* routine to select one of the ten sample set size options. The sample set sizes are based on the FFT algorithms in the PC evaluation board software, which require a sample set size of 2^N samples, and the software's internal limitation of between 16 and 8192 samples. A single sample option is also included, and if none of these are selected, the sample size defaults to zero.

3.7 Test Modes

There are a seven self-test modes built in the controller software, contained in the routine *test_mode*. These tests allow the user to troubleshoot certain problems that may be occurring with the evaluation board and its software. The test modes are entered by setting the DIP switches to a value greater than zero and resetting the board. This instructs the *main* routine to skip over the normal operation loop and go straight into test mode. To exit any of these rou-

tines, the user must place the DIP switches back into the '000' position and reset the evaluation board. Each of the seven test modes is now described.

Mode 1, Loop Back Test - This test is used to check the 80C51's on-chip UART. To perform this test, the user must place HDR7 in loop back position as indicated on the evaluation board, and set the DIP switches to '001'. When this mode is entered, a byte (0xAA) is sent out to the UART and then a byte is received from the UART. A delay is inserted to allow time for the information to be transmitted and received, and then the received byte is compared to the transmitted information. If the received byte is equal to 0xAA(HEX), then the test was successful, and the code calls the *toggle_all_leds* routine, which continuously blinks all four LEDs. If the information does not match up, the test failed, and the code branches to the *toggle_two_leds* routine, which continuously blinks only the first two LEDs (RESET and COMM).

Mode 2, Read/Write to ADC - This mode tests the microcontroller's ability to read and write the internal registers of the ADC. To perform this test, the user sets the DIPs to '010' and resets the board. Upon entering this test mode, the code writes 0xAA0000(HEX) to both the offset and gain registers of Physical Channel 1 (PC1). The information is then read from these two registers, and the high byte of each is compared to the value 0xAA(HEX). If they are both the same, then the test passed, and *toggle_all_leds* is called. If either of the registers were not written properly, the test failed, and *toggle_two_leds* is called.

Mode 3, Continuously Acquire Single Conversion - In this mode, conversions are continuously obtained from the ADC. This test mode is initiated by setting the DIPs to '011'. The first two LEDs are lighted, and the code loops and continuously gathers conversions until the evaluation board is reset. By probing HDR6 on the evaluation board and using $\overline{CS}$ as a triggering pin, an oscilloscope or logic analyzer will display how the microcontroller reads conversion data in real-time.

Mode 4, Continuously Write Gain Register - In mode 4, the gain register of PC1 is written to continuously. The test mode is entered by setting the DIPs to '100'. The third LED (GAIN_CAL) is lit, and the code loops and keeps writing to the gain register of PC1 until the evaluation board is reset.By probing HDR6 on the evaluation board and using $\overline{CS}$ as a triggering pin, an oscilloscope or logic analyzer will display how the microcontroller writes information to the ADC's internal registers in real-time.

Mode 5, Continuously Read Gain Register - This mode continuously reads the gain register on PC1. Setting the DIPs to '101' selects this test mode. The odd LEDs (RESET and GAIN_CAL) are lit, and the code loops through to read from the gain register of PC1 until the evaluation board is reset. By probing HDR6 on the evaluation board and using $\overline{CS}$ as a triggering pin, an oscilloscope or logic analyzer will display how the microcontroller reads data from the ADC's internal registers in real-time.

Mode 6, PC to Microcontroller RS-232 Communication Link Test - This test is performed in conjunction with the PC to verify that the RS-232 link is fully operational on both ends. Setting the DIPs to '110' selects for this test mode. When HDR7 is in the Normal position, the controller receives three bytes of information from the PC, and then echoes them back in the same order that they were received and calls *toggle_all_leds*. When HDR7 is in the Loopback position, the three bytes are automatically looped back to the PC through the RS-232 circuitry. The PC software indicates whether this test passes or fails, based upon the bytes it receives back from the board.

Mode 7, Toggle LEDs - This mode is used to test the functionality of the LEDs on the evaluation

board, and is entered by setting the DIPs to '111'. Immediately upon entering mode 7, the *toggle_all_leds* routine is called to blink the LEDs repeatedly until the board is reset.

4. MAXIMUM SCLK RATE

A machine cycle in the 80C51 consists 12 oscillator periods or 1µs if the microcontroller's oscillator frequency is 12 MHz. Since the CS5521/22/23/24/28's maximum SCLK rate is 2MHz, additional no operation (NOP) delays may be necessary to reduce the transfer rate if the microcontroller system requires higher rate oscillators.

5. DEVELOPMENT TOOL DESCRIPTION

The code in this application note was developed using a software development package from Franklin Software, Inc. The code consists of intermixed C and assembler algorithms which are used by the CDB5521/22/23/24/28, a customer evaluation board designed to help customers evaluate the performance of the CS5521/22/23/24/28 devices.

6. CONCLUSION

This application note describes code that can be used to interface an 80C51 to both the CS5521/22/23/24/28 ADC and a PC. It is divided into two main sections: hardware and software. The hardware interface illustrates both a three-wire and a four-wire interface between the 80C51 and the ADC. The three-wire is *SPI*™ and *MICROWIRE*™ compatible. The software, developed with development tools from Franklin Software, Inc., is used in conjunction with the PC evaluation software and the CDB5521/22/23/24/28 evaluation board to provide a system by which the CS5521/22/23/24/28's performance can be evaluated. There are four main routines which interface the CS5521/22/23/24/28 to the 80C51, and provide the communication between the 80C51 and the PC, e.g. *transfer_byte*, *receive_byte*, *txser*, and *rxser*. These four routines are written in 80C51 assembly language, and are easily imported into application-specific designs by both C and assembly programmers.

All of the software described in this note is included Section 7. "Appendix: 80C51 Microcontroller Code" on page 9.

7. APPENDIX: 80C51 MICROCONTROLLER CODE

```
/*CS5521/22/23/24/28 Runtime Constants and Prototypes*/
/*PROTOTYPES*/
void initialize(void);
void decode_command(char command);
void decode_sample_index(void);
void TXSER(char);
char RXSER(void);
char receive_byte(void);
void transfer_byte(char);
void write_to_register(char command,char low,char mid, char high);
void read_register(char command);
void read_conversion(char command);
void Delay(void);
void test_mode(void);
void toggle_all_leds(void);
void toggle_two_leds(void);

/*  BYTE Register equates for the register ports */
sfr P0      =       0x80;
sfr P1      =       0x90;
sfr P2      =       0xA0;
sfr P3      =       0xB0;
sfr PSW     =       0xD0;
sfr ACC     =       0xE0;
sfr B       =       0xF0;
sfr SP      =       0x81;
sfr DPL     =       0x82;
sfr DPH     =       0x83;
sfr PCON    =       0x87;
sfr TCON    =       0x88;
sfr TMOD    =       0x89;
sfr TL0     =       0x8A;
sfr TL1     =       0x8B;
sfr TH0     =       0x8C;
sfr TH1     =       0x8D;
sfr IE      =       0xA8;
sfr IP      =       0xB8;
sfr SCON    =       0x98;
sfr SBUF    =       0x99;

/*PORT 1 equates*/
sbit    CS              =       0x90        ; /* Chip Select */
sbit    SDI             =       0x91        ; /* Serial Data In */
sbit    SDO             =       0x92        ; /* Serial Data Out */
```

```
sbit   SCLK          =        0x93       ; /* Serial Clock */
sbit   A1            =        0x94       ; /* Latch Input */
sbit   A0            =        0x95       ; /* Latch Input */

sbit   RESET         =        0xA3;
sbit   COMM          =        0xA2;
sbit   GAIN_CAL      =        0xA1;
sbit   OFFSET_CAL    =        0xA0;

sbit   TEST_BIT1     =        0xB2;
sbit   TEST_BIT2     =        0xB3;
sbit   TEST_BIT3     =        0xB4;

sbit   MODE          =        0x80;
sbit   CTRL3         =        0x83;
sbit   CTRL2         =        0x84;
sbit   CTRL1         =        0x85;

/*  BIT Register  */
/*  PSW   */
sbit CY     =        0xD7;
sbit AC     =        0xD6;
sbit F0     =        0xD5;
sbit RS1    =        0xD4;
sbit RS0    =        0xD3;
sbit OV     =        0xD2;
sbit P      =        0xD0;

/*  TCON  */
sbit TF1    =        0x8F;
sbit TR1    =        0x8E;
sbit TF0    =        0x8D;
sbit TR0    =        0x8C;
sbit IE1    =        0x8B;
sbit IT1    =        0x8A;
sbit IE0    =        0x89;
sbit IT0    =        0x88;

/*  IE   */
sbit EA     =        0xAF;
sbit ES     =        0xAC;
sbit ET1    =        0xAB;
sbit EX1    =        0xAA;
sbit ET0    =        0xA9;
sbit EX0    =        0xA8;

/*  IP   */
sbit PS     =        0xBC;
sbit PT1    =        0xBB;
```

```
sbit PX1    =           0xBA;
sbit PT0    =           0xB9;
sbit PX0    =           0xB8;

/*  P3  */
sbit RD     =           0xB7;
sbit WR     =           0xB6;
sbit T1     =           0xB5;
sbit T0     =           0xB4;
sbit INT1   =           0xB3;
sbit INT0   =           0xB2;
sbit TXD    =           0xB1;
sbit RXD    =           0xB0;

/*  SCON  */
sbit SM0    =           0x9F;
sbit SM1    =           0x9E;
sbit SM2    =           0x9D;
sbit REN    =           0x9C;
sbit TB8    =           0x9B;
sbit RB8    =           0x9A;
sbit TI     =           0x99;
sbit RI     =           0x98;

/*****************************************************************/
/* CDB5521/22/23/24/28 Engineering Microcontroller code Version 1.00    */
/* Purpose:  Interfaces the CCS5521/22/23/24/28 to a PC                 */
/* Comments:                                                            */
/*****************************************************************/
#pragma nointvector             /* Disable Interrupt Vectors */
#pragma debug                   /* Include Debug Information with .obj file */
#pragma code                    /* Include Assemble Code with .lst file */
#pragma small                   /* Use a Small Memory Model*/
#pragma RB(0)                   /* Use Register Bank Zero */
#pragma OPTIMIZE(3)             /* Use Highest Level of Optimizing */
#include "cs5522.h"
/* Global Variable */
char        command,            /* One Byte Variables */
            ext_byte,
            high_byte,
            mid_byte,
            low_byte,
            temp,
            temp1;
int         sample_index,       /* Two Bytes Long */
            sample_size,
            i,
            mode;               /* 0 = three wire 1 = 4 wire */
```

```
/*************************************************************/
/* Routine     - Main                                        */
/* Input       - none                                        */
/* Output      - none                                        */
/* Description - This routine is the main algorithm used to call */
/*       all subroutines.                                    */
/*************************************************************/
main() {
        initialize();

        /*Check Dip Switches to see which Mode to Enter*/
        temp  = TEST_BIT3;
        temp  = temp<<1;
        temp  = temp |TEST_BIT2;
        temp  = temp<<1;
        temp  = temp |TEST_BIT1;

        if(temp ==0){
                while(1) {
                        COMM        = 0x01;             /*Turn off LED*/
                        command     = RXSER();          /*Get Command from PC*/
                        COMM        = 0x00;             /*Turn on LED*/
                        decode_command(command);        /*Decode Command*/
                }
        }
        else
                test_mode();                             /*Enter Test Modes*/
}

/*************************Subroutines*************************/
/*************************************************************/
/* Routine     - Initialize                                  */
/* Input       - none                                        */
/* Output      - none                                        */
/* Description - This routine is used to setup the microcontroller */
/*       and reset the converter.                            */
/*************************************************************/
void initialize()
{       data int j;     /* Local counter variable */

        P1   = 0xF4;            /*SCLK - output, therefore low*/
                                /*SDI  - output, therefore low*/
                                /*SDO  - input, therefore high*/
                                /*CSb  - ouput, therefore low */

        P2      = 0x00;         /*Toggle all LED's, Start Up Sequence indicator*/
        Delay();
        P2      = 0xFF;
```

```
        P3     = 0xFF;          /*Use the alternative function of Port 3*/

        EA     = 0x00;          /*Disable all interrupts*/
        SCON   = 0x72;          /*8 bit UART*/
                                /*Set TI to enable TXSER UART function*/
                                /*Transmission Rate is 9600,N,8,1*/

        TMOD   = 0x20;          /*Use TIMER1, 8-bit auto-reload */
        TH1    = 0xFD;          /*Initialize Auto-reload to FD for 11.059MHz external clock*/
        PCON   = 0x00;          /*when set we run at 19200 Baud, 9600 when clear*/
        TCON   = 0X40;          /*Turn internal UART timer on*/

        for(j=0;j<2047;j++){            /*Delay 2048 SCLK cycles, ... */
                SCLK = 0x01;            /*to allow oscillator to start*/
                SCLK = 0x00;
        }

        /*Reset Serial Port*/
        SDI = 1;                        /*SET SDI PIN*/
        for(j=0;j<254;j++) {
                SCLK = 0x01;            /*SET SCLK*/
                SCLK = 0x00;            /*CLR SCLK*/
        }
        SDI = 0;                        /*RESET SDI PIN*/
        SCLK = 0x01;                    /*SET SCLK*/
        SCLK = 0x00;                    /*CLR SCLK*/

        mode = P0 & 0x01;               /*Test Mode Pin*/
                                        /*if Mode = 0, three wire mode ...*/
                                        /*if Mode = 1, four wire mode. ...*/
                                        /*  NOTE: To change mode board must be Reset.*/
}

/*************************************************************/
/* Routine     - write_to_register                           */
/* Input       - command, low, mid, high                     */
/* Output      - none                                        */
/* Description - This routine is used to write to any internal */
/*      register.                                            */
/*************************************************************/
void write_to_register(char command,char low,char mid,char high){
        if(mode == 1)   P1   = 0xF4;    /* Clear CSb */
        transfer_byte(command);
        transfer_byte(high);
        transfer_byte(mid);
        transfer_byte(low);
        if(mode == 1)   P1   = 0xF5;    /* Set CSb */
}
```

```
/*************************************************************/
/* Routine     - read_register                               */
/* Input       - command                                     */
/* Output      - none                                        */
/* Description - This routine is used to write to any internal */
/*      register.                                            */
/*************************************************************/
void read_register(char command){
        if(mode == 1)   P1   = 0xF4;        /*Clear CSb*/
        transfer_byte(command);             /*Send Command*/
        high_byte       = receive_byte();   /*Receive Bytes*/
        mid_byte        = receive_byte();
        low_byte        = receive_byte();
        if(mode == 1)   P1   = 0xF5;        /*Set CSb*/
}

/*************************************************************/
/* Read a Single Conversion                                  */
/*************************************************************/
void read_conversion(char command){
        read_register(0x0B);                /* Read Configuration Register */
        high_byte = high_byte&0xF8;         /* Mask Loop, RC, MC bits to 0 */
        write_to_register(0x03,low_byte, mid_byte, high_byte);

        /*Start Conversion*/
        if(mode == 1)   P1   = 0xF4;        /* Clear CSb */
        transfer_byte(command);             /* Initiate Single conversion */
        do { /* Nothing*/} while (SDO !=0); /* Wait for Conversion to Complete */
        transfer_byte(0x00);                /* Send Command */
        high_byte       = receive_byte();   /* Receive Bytes */
        mid_byte        = receive_byte();
        low_byte        = receive_byte();
        if(mode == 1)   P1   = 0xF5;        /* Set CSb */
}

/*************************************************************/
/* Routine     - decode_command                              */
/* Input       - command                                     */
/* Output      - none                                        */
/* Description - This routine is used to transfer the command byte */
/*      from the 80C51 to the converter.                     */
/*************************************************************/
void decode_command(char command)
{  data int j,size;
        switch (command) {
                /***********************/
                /* Write to Register   */
                /***********************/
                case 0x01:      /* Offset Register Physical Channel 1 */
```

```
case 0x11:    /* Offset Register Physical Channel 2 */
case 0x21:    /* Offset Register Physical Channel 3 */
case 0x31:    /* Offset Register Physical Channel 4 */
case 0x41:    /* Offset Register Physical Channel 5 */
case 0x51:    /* Offset Register Physical Channel 6 */
case 0x61:    /* Offset Register Physical Channel 7 */
case 0x71:    /* Offset Register Physical Channel 8 */

case 0x02:    /* Gain Register Physical Channel 1 */
case 0x12:    /* Gain Register Physical Channel 2 */
case 0x22:    /* Gain Register Physical Channel 3 */
case 0x32:    /* Gain Register Physical Channel 4 */
case 0x42:    /* Gain Register Physical Channel 5 */
case 0x52:    /* Gain Register Physical Channel 6 */
case 0x62:    /* Gain Register Physical Channel 7 */
case 0x72:    /* Gain Register Physical Channel 8 */

case 0x03:    /* Configuration Register */

            low_byte   = RXSER(); /*Receive data low byte first*/
            mid_byte   = RXSER();
            high_byte  = RXSER();
            COMM = 0x00;          /* Turn on LED */
            Delay();
            COMM = 0x01;          /* Turn off LED */
            write_to_register(command,low_byte,mid_byte,high_byte);
            break;

/*****************************/
/* Write Channel-Setup Registers   */
/*****************************/
case 0x05:
            COMM   = 0x00;                  /* Turn on LED*/
            sample_size   = RXSER();        /* How many Registers? */
            if(mode == 1)  P1  = 0xF4;      /* Clear CSb */
            transfer_byte(command);         /* Send command to A/D */
            for (j=0; j<sample_size; j++) {
                        low_byte   = RXSER();       /* Receive Data */
                        mid_byte   = RXSER();
                        high_byte  = RXSER();
                        transfer_byte(high_byte);   /* Transfer Data to A/D */
                        transfer_byte(mid_byte);
                        transfer_byte(low_byte);
            } /* END for loop */
            if(mode == 1)  P1  = 0xF5;      /* Set CSb */
            COMM   = 0x01;                  /* Turn off LED*/
            break;
```

```
/***********************/
/* Read from Register      */
/***********************/
case 0x09:     /* Offset Register Physical Channel 1 */
case 0x19:     /* Offset Register Physical Channel 2 */
case 0x29:     /* Offset Register Physical Channel 3 */
case 0x39:     /* Offset Register Physical Channel 4 */
case 0x49:     /* Offset Register Physical Channel 5 */
case 0x59:     /* Offset Register Physical Channel 6 */
case 0x69:     /* Offset Register Physical Channel 7 */
case 0x79:     /* Offset Register Physical Channel 8 */

case 0x0A:     /* Gain Register Physical Channel 1 */
case 0x1A:     /* Gain Register Physical Channel 2 */
case 0x2A:     /* Gain Register Physical Channel 3 */
case 0x3A:     /* Gain Register Physical Channel 4 */
case 0x4A:     /* Gain Register Physical Channel 5 */
case 0x5A:     /* Gain Register Physical Channel 6 */
case 0x6A:     /* Gain Register Physical Channel 7 */
case 0x7A:     /* Gain Register Physical Channel 8 */

case 0x0B:     /* Configuration Register */

            read_register(command);              /* Read register's content */
            TXSER(low_byte);                     /* Transfer bytes to PC*/
            TXSER(mid_byte);
            TXSER(high_byte);
            COMM    = 0x00;                      /* Turn on LED*/
            Delay();
            COMM    = 0x01;                      /* Turn off LED*/
            break;

/*****************************/
/* Read Conversion Data FIFO      */
/*****************************/
case 0x0C:
            COMM = 0x00;                         /* Turn on LED */
            sample_size  = RXSER();              /* How many Conversions? */
            temp       = RXSER();                /* What conversion channel? */
            temp1            = RXSER();          /* Is LP bit set? */
            /*Initiate Continuous Conversion*/
            if(mode == 1)   P1   = 0xF4;         /* Clear CSb */
            transfer_byte(temp);                 /* Initiate Single conversion */
            do { /* Nothing*/} while (SDO !=0);  /* Wait for SDO to fall */
            transfer_byte(0x00);                 /* Send all zeros */

            for (j=0; j<sample_size; j++) {
                        high_byte      = receive_byte();/* Receive Bytes */
                        mid_byte       = receive_byte();
```

```
                    low_byte      = receive_byte();

                    TXSER(low_byte);          /* Return bytes to PC */
                    TXSER(mid_byte);
                    TXSER(high_byte);
          }

          if (temp1 == 1 ){
                    do { /* Nothing*/} while (SDO !=0);/* Wait for SDO to fall */
                    transfer_byte(0xFF);          /* Send all zeros */
                    for (j=0; j<sample_size; j++) {
                      high_byte     = receive_byte();/* Receive last conversion */
                      mid_byte      = receive_byte();
                      low_byte      = receive_byte();
                    } /* END for */
          } /* END if */

          if(mode == 1)   P1   = 0xF5;              /* Set CSb */
          COMM = 0x01;                              /* Turn off LED */
          break;

/*****************************/
/* Read Channel-Setup Registers    */
/*****************************/
case 0x0D:
          COMM    = 0x00;                           /* Turn on LED*/
          sample_size    = RXSER();                 /* How many Registers? */
          if(mode == 1)   P1   = 0xF4;              /* Clear CSb */
          transfer_byte(command);                   /* Send command to A/D */
          for (j=0; j<sample_size; j++) {
                    high_byte   = receive_byte();/* Receive Bytes */
                    mid_byte    = receive_byte();
                    low_byte    = receive_byte();
                    TXSER(low_byte);          /* Transfer data to PC */
                    TXSER(mid_byte);
                    TXSER(high_byte);
          } /* END for loop */
          if(mode == 1)   P1   = 0xF5;              /* Set CSb */
          COMM    = 0x01;                           /* Turn off LED*/
          break;


/***********************/
/* Normal Conversions     */
/***********************/
case 0x80:          /* Normal Conversion on Setup 1 */
case 0x88:          /* Normal Conversion on Setup 2 */
case 0x90:          /* Normal Conversion on Setup 3 */
case 0x98:          /* Normal Conversion on Setup 4 */
```

```
case 0xA0:              /* Normal Conversion on Setup 5 */
case 0xA8:              /* Normal Conversion on Setup 6 */
case 0xB0:              /* Normal Conversion on Setup 7 */
case 0xB8:              /* Normal Conversion on Setup 8 */
case 0xC0:              /* Normal Conversion on Setup 9 */
case 0xC8:              /* Normal Conversion on Setup 10 */
case 0xD0:              /* Normal Conversion on Setup 11 */
case 0xD8:              /* Normal Conversion on Setup 12 */
case 0xE0:              /* Normal Conversion on Setup 13 */
case 0xE8:              /* Normal Conversion on Setup 14 */
case 0xF0:              /* Normal Conversion on Setup 15 */
case 0xF8:              /* Normal Conversion on Setup 16 */
        read_conversion(command);
        TXSER(low_byte);      /* Return bytes to PC */
        TXSER(mid_byte);
        TXSER(high_byte);
        COMM    = 0x00;       /* Turn on LED */
        Delay();
        COMM    = 0x01;       /* Turn off LED*/
        break;

/*********************/
/* Self Calibration */
/*********************/
case 0x81:              /* Self-Offset Cal on Setup 1 */
case 0x89:              /* Self-Offset Cal on Setup 2 */
case 0x91:              /* Self-Offset Cal on Setup 3 */
case 0x99:              /* Self-Offset Cal on Setup 4 */
case 0xA1:              /* Self-Offset Cal on Setup 5 */
case 0xA9:              /* Self-Offset Cal on Setup 6 */
case 0xB1:              /* Self-Offset Cal on Setup 7 */
case 0xB9:              /* Self-Offset Cal on Setup 8 */
case 0xC1:              /* Self-Offset Cal on Setup 9 */
case 0xC9:              /* Self-Offset Cal on Setup 10 */
case 0xD1:              /* Self-Offset Cal on Setup 11 */
case 0xD9:              /* Self-Offset Cal on Setup 12 */
case 0xE1:              /* Self-Offset Cal on Setup 13 */
case 0xE9:              /* Self-Offset Cal on Setup 14 */
case 0xF1:              /* Self-Offset Cal on Setup 15 */
case 0xF9:              /* Self-Offset Cal on Setup 16 */

case 0x82:              /* Self-Gain Cal on Setup 1 */
case 0x8A:              /* Self-Gain Cal on Setup 2 */
case 0x92:              /* Self-Gain Cal on Setup 3 */
case 0x9A:              /* Self-Gain Cal on Setup 4 */
case 0xA2:              /* Self-Gain Cal on Setup 5 */
case 0xAA:              /* Self-Gain Cal on Setup 6 */
case 0xB2:              /* Self-Gain Cal on Setup 7 */
case 0xBA:              /* Self-Gain Cal on Setup 8 */
```

```
case 0xC2:              /* Self-Gain Cal on Setup 9 */
case 0xCA:              /* Self-Gain Cal on Setup 10 */
case 0xD2:              /* Self-Gain Cal on Setup 11 */
case 0xDA:              /* Self-Gain Cal on Setup 12 */
case 0xE2:              /* Self-Gain Cal on Setup 13 */
case 0xEA:              /* Self-Gain Cal on Setup 14 */
case 0xF2:              /* Self-Gain Cal on Setup 15 */
case 0xFA:              /* Self-Gain Cal on Setup 16 */
        COMM    = 0x00;                         /* Turn on LED */
        if(mode == 1)   P1   = 0xF4;            /*Clear CSb*/
        transfer_byte(command);
        do { /* Nothing*/} while (SDO !=0);     /* Wait for cal to Complete */
        if(mode == 1)   P1   = 0xF5;            /*Set CSb*/
        COMM    = 0x01;                         /* Turn off LED */
        break;

/***********************/
/* System Calibration      */
/***********************/
case 0x85:              /* System-Offset Cal on Setup 1 */
case 0x8D:              /* System-Offset Cal on Setup 2 */
case 0x95:              /* System-Offset Cal on Setup 3 */
case 0x9D:              /* System-Offset Cal on Setup 4 */
case 0xA5:              /* System-Offset Cal on Setup 5 */
case 0xAD:              /* System-Offset Cal on Setup 6 */
case 0xB5:              /* System-Offset Cal on Setup 7 */
case 0xBD:              /* System-Offset Cal on Setup 8 */
case 0xC5:              /* System-Offset Cal on Setup 9 */
case 0xCD:              /* System-Offset Cal on Setup 10 */
case 0xD5:              /* System-Offset Cal on Setup 11 */
case 0xDD:              /* System-Offset Cal on Setup 12 */
case 0xE5:              /* System-Offset Cal on Setup 13 */
case 0xED:              /* System-Offset Cal on Setup 14 */
case 0xF5:              /* System-Offset Cal on Setup 15 */
case 0xFD:              /* System-Offset Cal on Setup 16 */

case 0x86:              /* System-Gain Cal on Setup 1 */
case 0x8E:              /* System-Gain Cal on Setup 2 */
case 0x96:              /* System-Gain Cal on Setup 3 */
case 0x9E:              /* System-Gain Cal on Setup 4 */
case 0xA6:              /* System-Gain Cal on Setup 5 */
case 0xAE:              /* System-Gain Cal on Setup 6 */
case 0xB6:              /* System-Gain Cal on Setup 7 */
case 0xBE:              /* System-Gain Cal on Setup 8 */
case 0xC6:              /* System-Gain Cal on Setup 9 */
case 0xCE:              /* System-Gain Cal on Setup 10 */
case 0xD6:              /* System-Gain Cal on Setup 11 */
case 0xDE:              /* System-Gain Cal on Setup 12 */
case 0xE6:              /* System-Gain Cal on Setup 13 */
```

```
case 0xEE:              /* System-Gain Cal on Setup 14 */
case 0xF6:              /* System-Gain Cal on Setup 15 */
case 0xFE:              /* System-Gain Cal on Setup 16 */
        COMM    = 0x00;                     /* Turn on LED */
        if(mode == 1)   P1   = 0xF4;        /*Clear CSb*/
        transfer_byte(command);
        do {/* Nothing*/} while (SDO !=0); /* Wait for cal to Complete */
        if(mode == 1)   P1   = 0xF5;        /*Set CSb*/
        COMM    = 0x01;                     /* Turn off LED */
        break;

/*******************************/
/* Variable # of Normal Conversions */
/*******************************/
case 0x1F:
        COMM = 0x00;                        /* Turn on LED */
        sample_index    = RXSER();          /* How many Conversions? */
        temp            = RXSER();          /* What conversion channel? */
        decode_sample_index();

        /*Initiate Continuous Conversion*/
        if(mode == 1)   P1   = 0xF4;        /* Clear CSb */
        transfer_byte(temp);                /* Initiate Single conversion */
        for (j=0; j<sample_size; j++) {
                do { /* Nothing*/} while (SDO !=0); /* Wait for SDO to fall */
                transfer_byte(0x00);        /* Send all zeros */
                high_byte       = receive_byte();/* Receive Bytes */
                mid_byte        = receive_byte();
                low_byte        = receive_byte();

                TXSER(low_byte);            /* Return bytes to PC */
                TXSER(mid_byte);
                TXSER(high_byte);
        }
        transfer_byte(0xFF);                /* Send all zeros */
        high_byte       = receive_byte();   /* Receive last conversion */
        mid_byte        = receive_byte();
        low_byte        = receive_byte();
        if(mode == 1)   P1   = 0xF5;        /* Set CSb */
        COMM = 0x01;                        /* Turn off LED */
        break;

/************************************/
/* Variable # of Offset or Gain Calibrations */
/************************************/
case 0x2F:
        COMM = 0x00;                        /* Turn on LED */
        sample_index    = RXSER();          /* How many Cals? */
        temp            = RXSER();          /* What type of Cal? */
```

```
                temp1       = RXSER();              /* Which PC Cal. register is influenced? */
                decode_sample_index();
                if(mode == 1)   P1   = 0xF4;        /* Clear CSb */
                for (j=0; j<sample_size; j++) {
                        transfer_byte(temp);        /* Perform Calibration */
                        do {/* Nothing*/} while (SDO !=0); /* Wait for cal. to Complete */

                        transfer_byte(temp1);       /*Send Command*/
                        high_byte    = receive_byte();/*Receive Bytes*/
                        mid_byte     = receive_byte();
                        low_byte     = receive_byte();
                        TXSER(low_byte);            /* Transfer Caled Register back */
                        TXSER(mid_byte);
                        TXSER(high_byte);
                } /* END for loop */
                if(mode == 1)   P1   = 0xF5;        /* Set CSb */
                COMM = 0x01;                        /* Turn off LED */
                break;

/***********************/
/* Serial Port Initialization */
/***********************/
case 0x3F:
                COMM = 0x00;                        /* Turn LED on */
                if(mode == 1)   P1   = 0xF4;        /* Clear CSb */
                SDI = 1;                            /* SET SDI PIN */
                /* Write 255 bits of 1's to SDI and then 1 zero */
                for(j=0;j<254;j++) {
                        SCLK = 0x01;                /* SET SCLK */
                        SCLK = 0x00;                /* CLR SCLK */
                }
                SDI = 0;                            /* RESET SDI PIN */
                SCLK = 0x01;                        /* SET SCLK */
                SCLK = 0x00;                        /* CLR SCLK */
                if(mode == 1)   P1   = 0xF5;        /* Set CSb */
                COMM = 0x01;                        /* Turn LED off */
                break;

/*******************************/
/* Reset Converter via RS & RV bits */
/*******************************/
case 0x4F:
                write_to_register(0x03,0x80,0x00,0x00);  /* Set RS bit */

                /* Test for Valid Reset */
                do {
                        read_register(0x0B);
                        temp = low_byte&0x40;       /* Mask RV bit to 1 */
                        if(temp == 0x40)
```

```
                        write_to_register(0x03,0x00,0x00,0x00); /* Clear RS Bit */
            } while (temp != 0x40);
            break;

/***********************/
/* Read Output Latch Pins */
/***********************/
case 0x5F:
            temp  = A1;
            temp  = temp<<1;
            temp  = temp |A0;
            TXSER(temp);
            COMM = 0x00;    /* Turn on LED */
            Delay();
            COMM = 0x01;    /* Turn off LED */
            break;

/**********************/
/* Arbitrary Read          */
/**********************/
case 0xEF:
            temp= RXSER();  /* How many byte to return to PC*/
            for(j=0;j<temp;j++){
                        if(mode == 1)   P1   = 0x04;
                        temp1 = receive_byte();
                        if(mode == 1)   P1   = 0x05;

                        TXSER(temp1);
                        COMM = 0x00;    /* Turn LED on */
                        Delay();
                        COMM = 0x01;    /* Turn LED off */
            } /* END for loop */
            break;

/**********************/
/* Arbitrary Write         */
/**********************/
case 0xFF:
            temp = RXSER();         /* How many bytes to write to A/D */
            COMM = 0x00;            /* Turn on LED */
            Delay();
            COMM = 0x01;            /* Turn off LED */

            if(mode == 1)   P1   = 0x04;
            transfer_byte(temp);
            if(mode == 1)   P1   = 0x05;
            break;
```

```
                    /*********************************/
                    /* No Cases were met, let port time out */
                    /*********************************/
                    default:
                            break;
            }
}/* END decode_command*/

/*****************************************************************/
/* Routine     - decode_sample_index                             */
/* Input       - sample_index                                    */
/* Output      - none                                            */
/* Description - This routine is used to decode the meaning of the */
/*      variable sample_index.                                   */
/*****************************************************************/
void decode_sample_index()
{
            switch (sample_index) {
                    case 0:
                            sample_size = 16;
                            break;
                    case 1:
                            sample_size = 32;
                            break;
                    case 2:
                            sample_size = 64;
                            break;
                    case 3:
                            sample_size = 128;
                            break;
                    case 4:
                            sample_size = 256;
                            break;
                    case 5:
                            sample_size = 512;
                            break;
                    case 6:
                            sample_size = 1024;
                            break;
                    case 7:
                            sample_size = 2048;
                            break;
                    case 8:
                            sample_size = 4096;
                            break;
                    case 9:
                            sample_size = 8192;
                            break;
                    case 10:
```

```
                        sample_size = 1;
                        break;
                default:
                        sample_size = 0;
                        break;
        }
} /* END decode_sample_index */

/*************************************************************/
/* Routine     - test_mode                                   */
/* Input       - none                                        */
/* Output      - none                                        */
/* Description - This routine is used to perform special test*/
/*      functions to aid in debugging the CDB hardware.      */
/*************************************************************/
void test_mode(){
        switch (temp) {
                /* Test Mode 1: For this test, user must place HDR7 into
                        loop back position.  A byte of data is sent out UART
                        of 80C51 and received at same time.     It the byte
                        that is returned is the same all LEDs toggle.
                        Otherwise, only half the LEDs toggle. */
                case 0x01:
                        RESET = 0x00;                   /* Set LED Indicator */
                        TXSER(0xAA);
                        low_byte = RXSER();             /* low byte first*/
                        Delay();
                        if(low_byte == 0xAA)
                                toggle_all_leds();      /* Test Passed */
                                else
                                  toggle_two_leds();    /* Test Failed */
                        break;

                /* Test Mode 2:  This test mode writes to the offset and gain
                        registers of PC1. It then reads the registers.
                        If the correct data is read back all the LED's toggle
                        otherwise only half of them toggle to indicate an error. */
                case 0x02:
                        COMM = 0x01;
                        write_to_register(0x01,0x00,0x00,0xAA); /* Write Offset PC1 */
                        write_to_register(0x02,0x00,0x00,0xAA); /* Write Gain PC1 */
                        read_register(0x09);            /* Read Offset PC1 */
                        if(high_byte ==0xAA){           /* Was data sent? */
                                read_register(0x0A);    /* Read Gain PC1 */
                                if(high_byte==0xAA)
                                  toggle_all_leds();
                                else
                                  toggle_two_leds();    /* Test Failed */
                        } /* END if */
```

```
            else
                        toggle_two_leds();          /* Test Failed */
            break;

/* Test Mode 3: This test lights the first two LED and then
            repeatively iniates a conversion and reads it from the serial port. */
case 0x03:
            RESET       = 0x00;
            COMM        = 0x00;
            while(1)
                        read_conversion(0x80);
            break;

/* Test Mode 4:  This test lights the third LED's. Then it
            continuously write to the Gain Registers of PC1. */
case 0x04:
            GAIN_CAL    = 0x00;

            while(1)
                        write_to_register(0x02,0xAA,0xAA,0xAA);
            break;

/* Test Mode 5:  This test lights the odd LED's. Then it
            continuously reads the gain register of PC1. */
case 0x05:
            RESET       = 0x00;
            GAIN_CAL    = 0x00;

            while(1)
                        read_register(0x0A);  /* Read Gain Register of PC1 */
            break;

/* Test Mode 6: This test mode tests the RS232 link.
            it receives three bytes from the PC and echoes them
            back.  This PC determines if the test passed. The last
            three LED's are lit to indicate this test mode. */
case 0x06:
            COMM        = 0x00;
            GAIN_CAL    = 0x00;

            low_byte    = RXSER();  /* Receive low byte from PC */
            mid_byte    = RXSER();
            high_byte   = RXSER();

            TXSER(low_byte);      /* Transfer bytes back to PC */
            TXSER(mid_byte);
            TXSER(high_byte);
            toggle_all_leds();
            break;
```

```
                    /* Test Mode 7: This test mode tests the operation of
                    the LEDS's by turning all of them on and off. */
                    case 0x07:
                            toggle_all_leds();
                            break;
            } /* END switch */
} /* End test_mode */

/*****************************************************************/
/* Routine     - toggle_all_leds                                 */
/* Input       - none                                            */
/* Output      - none                                            */
/* Description - This routine is used toggle all four LED        */
/*****************************************************************/
void toggle_all_leds(){
            while(1){
                    P2   = 0x00;            /*Toggle all LED's for Start Up Sequence indicator test*/
                    Delay();
                    P2   = 0xFF;
                    Delay();
            }
}

/*****************************************************************/
/* Routine     - toggle_two_leds                                 */
/* Input       - none                                            */
/* Output      - none                                            */
/* Description - This routine is used toggle two LEDS            */
/*****************************************************************/
void toggle_two_leds(){
            while(1){
                    RESET       = 0x00;
                    COMM         = 0x00;

                    Delay();

                    RESET       = 0x01;
                    COMM         = 0x01;
                    Delay();
            }
}
```

```
/*****************************************************************/
/* Routine     - Delay                                           */
/* Input       - none                                            */
/* Output      - none                                            */
/* Description - This routine is used as a LED delay routine     */
/*****************************************************************/
void Delay(void) {
        data int j;
        for (j=0; j<10000; j++)
                for (j=0; j<10000; j++);
}

;*****************************************************************
;* Routine     - RECEIVE_BYTE
;* Input       - none
;* Output      - Byte received is placed in R7
;* Description - This subroutine receives 1 byte from converter
;*****************************************************************

; The function prototype is:   char RECEIVE_BYTE(void);

$DEBUG
USING   0                       ; Use register bank 0
TCOD    SEGMENT CODE            ; Define ROUT as a segment of code

PUBLIC  RECEIVE_BYTE            ; Make subroutine global

RSEG    TCOD                    ; Make code relocatable
RECEIVE_BYTE:
        MOV     R1,#08          ; Set count to 8 to receive byte

LOOP:                           ; Receive the byte
        MOV     C,P1.2          ; Move bit to carry
        RLC     A               ; Rotate A in preparation for next bit
        SETB    P1.3            ; Set SCLK
        CLR     P1.3            ; Clear SCLK
        DJNZ    R1,LOOP         ; Decrement byte, repeat loop if not zero

        MOV     R7,A            ; Byte to be return is placed in R7
        RET                     ; Exit subroutine
END
```

```
;*****************************************************************
;* Routine     - transfer_byte
;* Input       - Byte to be transmitted is placed in Accumulator
;* Output      - None
;* Description - This subroutine sends 1 byte to converter
;*****************************************************************

;The function prototype is:   void TRANSFER_BYTE(char);

$DEBUG
USING   0                       ; Use register bank 0
TCOD    SEGMENT CODE            ; Make TCOD a segment of code

PUBLIC  _TRANSFER_BYTE  ; Make subroutine global

RSEG    TCOD                    ; Make code relocatable
_TRANSFER_BYTE:
            MOV     A, R7       ; Move byte to be transmitted to ACC
            MOV     R1,#08      ; Set count to 8 to transmit byte
            CLR     P1.3        ; Clear SCLK

loop:                           ; Send Byte
            RLC     A           ; Rotate Accumulator, send MSB 1st
            MOV     P1.1,C      ; Transmit MSB first through C bit
            SETB    P1.3        ; Set SCLK
            CLR     P1.3        ; Clear SCLK
            DJNZ    R1,loop     ; Decrement byte, repeat loop if not zero
            SETB    P1.1        ; Reset SDI to one when not transmitting
            RET                 ; Exit subroutine
END
```

```
;*****************************************************************
;* Routine     - RXSER
;* Input       - none
;* Output      - Byte received is placed in R7
;* Description - This subroutine receives 1 byte from converter
;*   via UART.  It uses the RS-232 serial protocol to transmit
;*   one byte from a PC/UART system to the 8051.
;*****************************************************************

; The function prototype is:   char RXSER(void);

$DEBUG
USING   0                       ; Use register bank 0
TCOD    SEGMENT CODE            ; Define ROUT as a segment of code

PUBLIC  RXSER                   ; Make subroutine global

RSEG    TCOD                    ; Make code relocatable

RXSER:
            JNB         SCON.0,$ ; Poll RI
            MOV         R7,SBUF  ; Place received byte in R7
            CLR         SCON.0   ; Reset RI bit
            RET
END
```

```
;*********************************************************************
;* Routine     - TXSER
;* Input       - Byte to be transmitted is placed in R7
;* Output      - None
;* Description - This subroutine transfers 1 byte from converter
;*   via UART.  It uses the RS-232 serial protocol to transmit
;*   one byte from a 80C51 to the PC/UART system. To
;*   function properly, the programmer must first initialize the
;*   TI bit in the SCON register to 0X01.
;*********************************************************************

;The function prototype is:   void TXSER(char);

$DEBUG
USING   0                       ; Use register bank 0
TCOD    SEGMENT CODE            ; Define TCOD as a segment of code

PUBLIC  _TXSER                  ; Make subroutine global

RSEG    TCOD                    ; Make code relocatable
_TXSER:  JNB        SCON.1,$ ; Poll TI
         CLR        SCON.1   ; Reset TI
         MOV        SBUF, R7 ; Move byte to output register
         RET                 ; Exit subroutine
END
```

• Notes •

SMART
Analog
™