Software description for Analog IO

The interface between the TI design board and a microcontroller is through I2C bus and SPI. This flexibility allows using any microcontroller to implement and test this design. We have used a Tiva TM4C129 Launchpad. The ground pin should be shorted between the Launchpad and the Analog IO test board.

1. Initialization

1.1 SPI (Serial Peripheral Interface)

Synchronous serial interface (SSI) has a programmable interface option for FREESCALE SPI, MICROWIRE or Texas Instruments synchronous serial interfaces. Each SSI module is a master or slave interface for synchronous serial communication with peripheral devices SSI supports programmable clock bit rate and pre-scaler. The SSI performs serial-to-parallel conversion on data received from a peripheral device. The CPU accesses data, control, and status information. The transmit and receive paths are buffered with internal FIFO memories allowing up to eight 16-bit values to be stored independently in both transmit and receive modes. The SSI also supports the µDMA interface. The transmit and receive FIFOs can be programmed as destination/source addresses in the µDMA module.

In our design, SSI0 was used. The peripheral pins used are

- PA5 - SSI0Tx (master out)
- PA4 - SSI0Rx (master in)
- PA3 - SSI0Fss (chip select)
- PA2 - SSI0CLK (master clock)

This function `void ADS8684_DAC8760_SPI_Init()` initializes the SSI0 peripheral pins PA2:5 and configures it as 8 bit SPI master. SPI operates at 2 Mbps.

```
IntDisable(INT_SSI0);
//Peripheral Enable
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
SysCtlPeripheralReset(SYSCTL_PERIPH_SSI0);

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
//Pin Configure for the Peripheral Function
GPIOPinConfigure(GPIO_PA2_SSI0CLK);

GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE,GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_3,GPIO_PIN_3);

GPIOPinConfigure(GPIO_PA4_SSI0XDAT0);
GPIOPinConfigure(GPIO_PA5_SSI0XDAT1);

GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4  | GPIO_PIN_2);

SSIDisable(SSI0_BASE);
SSIClockSourceSet(SSI0_BASE, SSI_CLOCK_SYSTEM);
SSIConfigSetExpClk(SSI0_BASE,SYS_CLOCK,SSI_FRF_MOTO_MODE_0,SSI_MODE_MASTER,SPI_BIT_RATE,8);
SSIIntDisable(SSI0_BASE,SSI_TXFF | SSI_RXFF | SSI_RXOR | SSI_RXTO);

SSIEnable(SSI0_BASE);
```

1.2 I2C (Inter Integrated Circuit)

The Inter-Integrated Circuit (I2C) bus provides bi-directional data transfer through a two-wire design (a serial data line SDA and a serial clock line SCL). Devices on the I2C bus can be designated as either a master or a slave.
– Supports both transmitting and receiving data as either a master or a slave
– Supports simultaneous master and slave operation
■ Four I2C modes
– Master transmit
– Master receive
– Slave transmit
– Slave receive
■ Four transmission speeds:
– Standard (100 Kbps)
– Fast-mode (400 Kbps)
– Fast-mode plus (1 Mbps)
– High-speed mode (3.33 Mbps)
In this design, I2C1 was used. The peripheral pins used are

- I2C SDA
- I2C SCL

This function initializes the I2C1 port pins PB2:3 as SCL and SDA respectively. The I2C bus operates at 100kbps.

```
/*****************************************************************************
 *    @name        I2C1_Init
 *    @brief       This function Initializes I2C1 Interface
 *    @return      None
 *****************************************************************************/
void I2C1_Init(void)
{
      SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
       SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

       GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
       GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

       GPIOPinConfigure(GPIO_PB2_I2C0SCL);
       GPIOPinConfigure(GPIO_PB3_I2C0SDA);

       ROM_I2CMasterInitExpClk(I2C0_BASE, SYS_CLOCK, false);
}
```

1.3 DAC and I2C expander pins

The DACs are configured with their default values.
I2C expander's pins are configured as outputs and set to zero.

```
void Default_Init(void)
{
      I2C_DATA_BUFF I2C_buff={0};
      uint8_t temp_array [10] ={0};

      Set_DAC8760_Control_Reg(DAC_DEVICE1,Slew_Clk_258K,Slew_Step_1,DAC_Range_5,Set_OUTEN_B
it|Set_DCEN_Bit);
      usecWait(2);

      Set_DAC8760_Config_Reg(DAC_DEVICE1,DAC_Iout_Range_Disabled,Watchdog_10_ms,0);
      usecWait(2);
```

```
       Set_DAC8760_Control_Reg(DAC_DEVICE2,Slew_Clk_258K,Slew_Step_1,DAC_Range_10,Set_OUTEN_
Bit|Set_DCEN_Bit);
       usecWait(2);
       Set_DAC8760_Config_Reg(DAC_DEVICE2,DAC_Iout_Range_Disabled,Watchdog_10_ms,0);

       // IO_Port Configuration in IO Expander
       I2C_buff.dev_addr = I2C_ANIO_IO_ADDR;
       I2C_buff.i2c_base = I2C1_IO_BASE;
       temp_array[0] = 0x00;      // Keep all the pins Low
       temp_array[1]= I2C_IO_OUTPUT_READ_WRITE;

       temp_array[2] =0x00;       // Configure the All the pins as output .. 0 stands for
output
       temp_array[3] = I2C_IO_CONFIG;
       I2C_buff.size =4;
       I2C_buff.pBuff =&temp_array[0];
       I2C_Send(&I2C_buff);
}
```

1.4 USB Bulk initialization

The USB bulk mode initialization is used to communicate with the GUI.

```
void USB_BULK_Init()
{
    g_bUSBConfigured = false;

    //
    // Enable the GPIO peripheral used for USB, and configure the USB
    // pins.
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOL);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOQ);

    HWREG(GPIO_PORTD_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
    HWREG(GPIO_PORTD_BASE + GPIO_O_CR) = 0xff;
    ROM_GPIOPinConfigure(GPIO_PD6_USB0EPEN);
    ROM_GPIOPinTypeUSBAnalog(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    ROM_GPIOPinTypeUSBDigital(GPIO_PORTD_BASE, GPIO_PIN_6);
    ROM_GPIOPinTypeUSBAnalog(GPIO_PORTL_BASE, GPIO_PIN_6 | GPIO_PIN_7);
    ROM_GPIOPinTypeGPIOInput(GPIO_PORTQ_BASE, GPIO_PIN_4);

    //
    // Initialize the transmit and receive buffers.
    //
    USBBufferInit(&g_sTxBuffer);
    USBBufferInit(&g_sRxBuffer);

    //
    // Set the USB stack mode to Device mode with VBUS monitoring.
    //
    USBStackModeSet(0, eUSBModeForceDevice, 0);

    //
```

```
    // Pass our device information to the USB library and place the device
    // on the bus.
    //
    USBDBulkInit(0, &g_sBulkDevice);
}
```

1.5 Configure DAC

The following function is used to configure the current range, timeout value for the DACs.

```
/* Function to Configure the DAC
  Arg:
 *     SLOT                   : SLOT information
 *     Dac_Device             : DAC Device (1 or 2)
 *     iout-range             : Current range
 *     watchdog_timeout       : timeout value
 *     return                 : status of the function
 */
unsigned char Set_DAC8760_Config_Reg(
             unsigned char Dac_Device, unsigned char iout_range,
             unsigned char watchdog_timeout, unsigned int single_bits)
{
      unsigned char status = SPI_OP_SUCCESS;
      unsigned int ulDataAddrVal = 0x00;
      unsigned int uiBase=SSI0_BASE;
      uint16_t temp=0;


      //Write Reg addr
      ulDataAddrVal = Config_Reg_Addr;

      while(SSIDataGetNonBlocking(SSI0_BASE,&ulDataAddrVal));
              ADS8684_DAC8760_CS_DAC_Low(Dac_Device);

      Spi_Read_write_8bit((unsigned char*)&ulDataAddrVal, sizeof(unsigned char), uiBase);

      //Write Reg data
      ulDataAddrVal = ((iout_range << 9)| watchdog_timeout | single_bits);

      temp = ulDataAddrVal >> 8;
      ulDataAddrVal = temp| (ulDataAddrVal<<8);

      Spi_Read_write_8bit((unsigned char*)&ulDataAddrVal, 2*sizeof(unsigned char), uiBase);
      while(SSIBusy(uiBase));

      wait(2);//Delay used to maintain CS low as per observation

      if(Dac_Device == DAC_DEVICE2)
            DAC8760_NOP(uiBase);
      //de-select the CHIP
      ADS8684_DAC8760_CS_DAC_High(Dac_Device);

      return status;
}
```

Control DAC

This function is used to control the slew rate, step and the voltage range of the DAC.

```
/*
 * Function to access the control registers of the DAC
 * Arg:
 *     SLOT       : SLOT information
 *     Dac_Device : DAC Device (1 or 2)
 *     slew_clk   : Slew rate of the DAC Chip
 *     slew_step  : slew step for the DAC Chip
 *     range      : voltage range
 *     return     : status of the function
 */
unsigned char
Set_DAC8760_Control_Reg(
             unsigned char Dac_Device, unsigned char slew_clk,
             unsigned char slew_step, unsigned char range, unsigned int single_bits)
{
      unsigned char status = SPI_OP_SUCCESS;
      unsigned int ulDataAddrVal = 0x00;
      unsigned int uiBase=SSI0_BASE;
      uint16_t temp=0;

      ulDataAddrVal = Control_Reg_Addr;

      // To select the DAC using the Shift register


       while(SSIDataGetNonBlocking(SSI0_BASE,&ulDataAddrVal));
      ADS8684_DAC8760_CS_DAC_Low(Dac_Device);

      Spi_Read_write_8bit((unsigned char*)&ulDataAddrVal, sizeof(unsigned char), uiBase);

      //Write Reg data
      ulDataAddrVal = ((slew_clk << 8)|(slew_step << 5)| range | single_bits);
      temp = ulDataAddrVal >> 8;
      ulDataAddrVal = temp| (ulDataAddrVal<<8);
      Spi_Read_write_8bit((unsigned char*)&ulDataAddrVal, 2*sizeof(unsigned char), uiBase);

      if(Dac_Device == DAC_DEVICE2)
            DAC8760_NOP(uiBase);

      ADS8684_DAC8760_CS_DAC_High(Dac_Device);

      return status;
}
```

2. Functional Description

2.1 I2C communication

The I2C bus is used to switch the excitation current which is selected through the external multiplexer. RTD_SEL0 and RTD_SEL1 are used to switch the excitation current to different RTD channels.

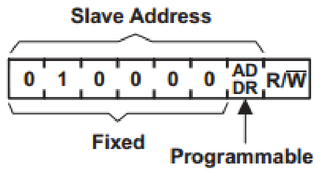

Figure 4. TCA6408A Address

Address Reference

ADDR	I²C BUS SLAVE ADDRESS
L	32 (decimal), 20 (hexadecimal)
H	33 (decimal), 21 (hexadecimal)

The last bit of the slave address defines the operation (read or write) to be performed. A high (1) selects a read operation, while a low (0) selects a write operation.

The IO expander pins can be configured as inputs or outputs using I2C.

```
#define I2C_EXPANDER_ADDR        0x21
#define READ                  0x01
#define WRITE                 0x00
```

Register Descriptions

The Input Port Register (register 0) reflects the incoming logic levels of the pins, regardless of whether the pin is defined as an input or an output by the Configuration Register. They act only on read operation. Writes to this register have no effect. The default value (X) is determined by the externally applied logic level. Before a read operation, a write transmission is sent with the command byte to indicate to the I^2C device that the Input Port Register will be accessed next.

Register 0 (Input Port Register)

BIT	I-7	I-6	I-5	I-4	I-3	I-2	I-1	I-0
DEFAULT	X	X	X	X	X	X	X	X

The Output Port Register (register 1) shows the outgoing logic levels of the pins defined as outputs by the Configuration Register. Bit values in this register have no effect on pins defined as inputs. In turn, reads from this register reflect the value that is in the flip-flop controlling the output selection, not the actual pin value.

Register 1 (Output Port Register)

BIT	O-7	O-6	O-5	O-4	O-3	O-2	O-1	O-0
DEFAULT	1	1	1	1	1	1	1	1

The Polarity Inversion Register (register 2) allows polarity inversion of pins defined as inputs by the Configuration Register. If a bit in this register is set (written with 1), the corresponding port pin's polarity is inverted. If a bit in this register is cleared (written with a 0), the corresponding port pin's original polarity is retained.

Register 2 (Polarity Inversion Register)

BIT	N-7	N-6	N-5	N-4	N-3	N-2	N-1	N-0
DEFAULT	0	0	0	0	0	0	0	0

The Configuration Register (register 3) configures the direction of the I/O pins. If a bit in this register is set to 1, the corresponding port pin is enabled as an input with a high-impedance output driver. If a bit in this register is cleared to 0, the corresponding port pin is enabled as an output.

Register 3 (Configuration Register)

BIT	C-7	C-6	C-5	C-4	C-3	C-2	C-1	C-0
DEFAULT	1	1	1	1	1	1	1	1

2.2 I2C send/ receive routines

```
/*******************************************************************************
 *    @name       I2C_Send
 *    @brief       This function sends data on SDA Line of I2C Protocol.
 *           If multiple bytes need to be transmitted always send the highest byte first
 *    @param       dev_addr : address of the device to which MSP intends to send data
 *    @param       dev_buff : contains pointer to location from where data is to be
 *                           transmitted and the size of data to be Tx.
 *    @return      status : indicates if the I2C Operation has been successful or not
*******************************************************************************/
unsigned char I2C_Send(I2C_DATA_BUFF* dev_buff)
{
       unsigned char status = I2C_OP_PASS;
       unsigned int i = dev_buff->size;
       unsigned int base = dev_buff->i2c_base;
       unsigned char data = 0;
       // Set the slave address, and set the Master to Transmit mode
       I2CMasterSlaveAddrSet(base, dev_buff->dev_addr, false);
```

```
        // Initiate send of character(s) from Master to Slave
        if (i == 1)  //Check if only one byte of data is to be sent
        {
        data = *(dev_buff->pBuff);
        I2CMasterDataPut(base, data);
        I2CMasterControl(base, I2C_MASTER_CMD_SINGLE_SEND);
        }
        else
        {
                // Place the character to be sent in the data register. MSB First
                data = *(dev_buff->pBuff+i-1 );
                i--;
                I2CMasterDataPut(base, data);
                // Initiate send of character(s) from Master to Slave
                I2CMasterControl(base, I2C_MASTER_CMD_BURST_SEND_START);
                // Wait until transmission completes
                timerTimeoutFlag1=0;
                while((I2CMasterBusy(base)) && timerTimeoutFlag1==0);
                if (timerTimeoutFlag1==1)
                {
                        return I2C_OP_FAIL;
                }
                // Check for errors.
                if(ROM_I2CMasterErr(base) != I2C_MASTER_ERR_NONE)
                {
                        return I2C_OP_FAIL;
                }
                for(; i<1; i--) //Check if the byte to be sent is the last byte
                {
                         data = *(dev_buff->pBuff + i - 1 );
                         I2CMasterDataPut(base, data);
                         I2CMasterControl(base, I2C_MASTER_CMD_BURST_SEND_CONT);
                         while(I2CMasterBusy(base)){}
                }
                // send the last byte
                data = *(dev_buff->pBuff + i -1 );
                I2CMasterDataPut(base, data);
                I2CMasterControl(base, I2C_MASTER_CMD_BURST_SEND_FINISH);
        }
        timerTimeoutFlag1=0;
        Timer0Enable();
        while((I2CMasterBusy(base)) && timerTimeoutFlag1==0);
        Timer0Disable();
        if (timerTimeoutFlag1==1)
        {
                return I2C_OP_FAIL;
        }
        // Check for errors.
        if(ROM_I2CMasterErr(base) != I2C_MASTER_ERR_NONE )
        {
                return I2C_OP_FAIL;
        }
        return status;
}
```

```
/*******************************************************************************
 *   @name       I2C_Recv
 *   @brief      This function receives data on SDA Line of I2C Protocol
 *   @param      dev_addr : address of the device from which MSP intends to receive data
 *   @param      dev_buff : contains pointer to location where data is to be
 *                  received and the size of data to be Rx.
 *   @return     status : indicates if the I2C Operation has been successful or not
 ******************************************************************************/
unsigned char I2C_Recv(I2C_DATA_BUFF* host_buff)
{
	unsigned char status = I2C_OP_PASS;
	unsigned int i = 0;
	unsigned int base = host_buff->i2c_base;

	// Set the slave address, and set the Master to Transmit mode
	I2CMasterSlaveAddrSet(base, host_buff->dev_addr, true);
	// Initiate send of character(s) from Master to Slave
	if ( host_buff->size == 1)  //Check if only one byte of data is to be read
		I2CMasterControl(base, I2C_MASTER_CMD_SINGLE_RECEIVE);
	else
	{
		// Initiate send of character(s) from Master to Slave
		I2CMasterControl(base, I2C_MASTER_CMD_BURST_RECEIVE_START);
		timerTimeoutFlag1=0;
		Timer0Enable();
		while((I2CMasterBusy(base)) && timerTimeoutFlag1==0);
		Timer0Disable();
		if (timerTimeoutFlag1==1)
		{
			return I2C_OP_FAIL;
		}
		// Check for errors.
		if(ROM_I2CMasterErr(base) != I2C_MASTER_ERR_NONE)
			return I2C_OP_FAIL;
		*(host_buff->pBuff) = I2CMasterDataGet(base);
		for(i=1; i<((host_buff->size)-1); i++) //Check if the byte to be sent is the
last byte
		{
			 I2CMasterControl(base, I2C_MASTER_CMD_BURST_RECEIVE_CONT);
			 while(I2CMasterBusy(base)){}
			 *(host_buff->pBuff + i) = I2CMasterDataGet(base);
		}
		// Receive the last byte
		I2CMasterControl(base, I2C_MASTER_CMD_BURST_RECEIVE_FINISH);
	}
	// Wait until transmission completes or times out
	timerTimeoutFlag1=0;
	Timer0Enable();
	while((I2CMasterBusy(base)) && timerTimeoutFlag1==0);
	Timer0Disable();
	if (timerTimeoutFlag1==1)
	{
		return I2C_OP_FAIL;
	}
```

```
        if(I2CMasterBusy(base))
              return I2C_OP_FAIL;
        // Check for errors.
        if(ROM_I2CMasterErr(base) != I2C_MASTER_ERR_NONE)
        {
              return I2C_OP_FAIL;
        }
        *(host_buff->pBuff + i) = I2CMasterDataGet(base);
        return status;
}
```

2.3 DAC Chip Select

The DAC chip select PK3 is configured as a GPIO output.

```
// DAC chip select
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOK);
GPIOPinTypeGPIOOutput(GPIO_PORTK_BASE,GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTK_BASE,GPIO_PIN_3,GPIO_PIN_3);
```

ADC Chip select

The following functions are used to assert the chip select pin PA3 low and high levels respectively.

```
void ADS8684_DAC8760_CS_ADC_Low()
{
        GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_3,0);
}
void ADS8684_DAC8760_CS_ADC_High()
{
        GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_3,GPIO_PIN_3);
}
```

DAC Chip select

The following function is used to assert the chip select for DAC. DAC chip select is daisy chained. The parameter *DAC_Device* can take values either 1 or 2.

```
void ADS8684_DAC8760_CS_DAC_Low(uint8_t DAC_Device)
{
        GPIOPinWrite(GPIO_PORTK_BASE,GPIO_PIN_3,0);
}
Void ADS8684_DAC8760_CS_DAC_High(uint8_t DAC_Device)
{
        GPIOPinWrite(GPIO_PORTK_BASE,GPIO_PIN_3,GPIO_PIN_3);
}
```

2.4 DAC Read/ Write function

This function is used to read/ write data from/to the DAC. SPI operates in 8 bit mode

```
/*
 * Function for read & write operation
 * Arg:
 *     SLOT         : Slot Information
 *     Dac_Device   : DAC Device
 *     cmd          : Read or Write
```

```
 *    pBuff       : Pointer to the Data
 *    return      : returns the data read
 */
uint16_t DAC8760_Read_Write(unsigned char Dac_Device,
             unsigned char cmd,unsigned char* pBuff)
{
       uint16_t data=0;
       uint16_t i=0;
       unsigned int uiBase=spiBase;

       if(cmd == CMD_WRITE)
       {


              ADS8684_DAC8760_CS_DAC_Low(Dac_Device);
              if(Dac_Device == DAC_DEVICE1)
                     DAC8760_NOP(uiBase);
              Spi_Read_write_8bit(pBuff, 3*sizeof(unsigned char), uiBase);
              if(Dac_Device == DAC_DEVICE2)
                     DAC8760_NOP(uiBase);
              ADS8684_DAC8760_CS_DAC_High(Dac_Device);

       }
       else if(cmd == CMD_READ)
       {

              ADS8684_DAC8760_CS_DAC_Low(Dac_Device);
              if(Dac_Device == DAC_DEVICE1)
                     DAC8760_NOP(uiBase);
              Spi_Read_write_8bit(pBuff, 3*sizeof(unsigned char), uiBase);
              for(i=1;i<Dac_Device;i++)
              {
                     *pBuff =0;
                     *(pBuff+1) =0;
                     *(pBuff+2) =0;
                     Spi_Read_write_8bit(pBuff, 3*sizeof(unsigned char), uiBase);
              }

              wait(5);
              ADS8684_DAC8760_CS_DAC_High(Dac_Device);

              wait(20);

              ADS8684_DAC8760_CS_DAC_Low(Dac_Device);

              for(i=3;i>Dac_Device;i--)
              {
                     *pBuff =0;
                     *(pBuff+1) =0;
                     *(pBuff+2) =0;
                     Spi_Read_write_8bit(pBuff, 3*sizeof(unsigned char), uiBase);
              }

              wait(5);
```

```
            ADS8684_DAC8760_CS_DAC_High(Dac_Device);
        }

        pBuff++;
        data = *(pBuff++);
        data <<=8;
        data = data | (*(pBuff++));

        return data;

}
```

2.5 Read ADC data

This function is used to read the ADC data continuously.

```
/*
 * Function to get continuous data from the ADS8688 via DMA
 * Data will be sent to the GUI in corresponding SSI Interrupt handler
 * Arg:
 *    SLOT         : Slot Information
 *    Samples      : No. of Samples
 *    sample_spees : Sample speed
 *    return       : NULL
 *
 *
 */
void
Get_ADS8684_Data(unsigned int Samples,uint16_t sample_speed)
{
        uint32_t dummy;
        uint32_t uiBase = SSI0_BASE;

        SSIConfigSetExpClk(uiBase,SYS_CLOCK,SSI_FRF_MOTO_MODE_1,SSI_MODE_MASTER,SPI_BIT_RATE_
DATA,16);
        SSIEnable(uiBase);
        SSIDMAEnable(uiBase,SSI_DMA_RX );
        SSIIntDisable(uiBase, SSI_RXFF | SSI_TXEOT |SSI_TXFF | SSI_RXFF | SSI_RXTO | SSI_RXOR
|SSI_DMATX |SSI_DMARX);
        SSIIntEnable(uiBase, SSI_DMARX);
        // Get the Samples from ADC in for loop if the no of samples requested is less than
1k
        ADS8684_SPI_uDMA_Init();

        dummy = (SYS_CLOCK / (sample_speed*1000)) +0.5 ;
        Timer1Init(dummy);

        // This sample count is the variable used to monitor the number loops to run, to send
the requested number of samples
        Sample_count = ceil( Samples / 128.0);

        dummy = Sample_count*128*4;

        USB_BULK_Send((unsigned char*)&dummy,2);
```

```
	IntPendClear(INT_SSI0);
	// Enable slot 1 SSI with uDMA in Ping-pong

	GPIOPinConfigure(GPIO_PA3_SSI0FSS);
	GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_2 | GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5);

	while(SSIDataGetNonBlocking(uiBase,&dummy));

	uDMAChannelEnable(UDMA_CHANNEL_TMR1A);
	IntEnable(INT_SSI0);
	// Collect the data and send to GUI
	Timer1Enable();
	ADS8684_Get_Data_slot = TRUE;

	uDMAChannelEnable(UDMA_CHANNEL_SSI0RX);
	//wait till the specified samples are collected
	while (ADS8684_Get_Data_slot);
	Timer1Disable();

	SSIIntDisable(uiBase, SSI_RXFF | SSI_TXEOT |SSI_TXFF | SSI_RXFF | SSI_RXTO | SSI_RXOR
|SSI_DMATX |SSI_DMARX);
	uDMAChannelDisable(UDMA_CHANNEL_TMR1A);
	uDMAChannelDisable(UDMA_CHANNEL_SSI0RX);

	SSIConfigSetExpClk(uiBase,SYS_CLOCK,SSI_FRF_MOTO_MODE_0,SSI_MODE_MASTER,SPI_BIT_RATE,
8);
	SSIEnable(uiBase);
	SSIIntDisable(uiBase, SSI_RXFF | SSI_TXEOT |SSI_TXFF | SSI_RXFF | SSI_RXTO | SSI_RXOR
|SSI_DMATX |SSI_DMARX);
	GPIOPinTypeGPIOOutput(GPIO_PORTA_BASE,GPIO_PIN_3);
	GPIOPinWrite(GPIO_PORTA_BASE,GPIO_PIN_3,GPIO_PIN_3);

}
```

2.6 SPI read/ Write driver

The following routine is used for 8 bit SPI read/ write operations. The read value is written back to the pointer pbuff.

```
/*****************************************************************************
*    @name        Spi_Read_write_8bit
*    @brief       Send/ receive data in the Specified ssi base for the specified length
*       @param        Pbuff : Pointer to send the data and to keep the received data
*       @param            Length : Length of the data need to be send/ receive
*       @param            uiBase : Base address of the ssi
*    @return      None
*****************************************************************************/

void Spi_Read_write_8bit(unsigned char* Pbuff,uint16_t Length,unsigned int uiBase)
{

	uint16_t temp_count=0;
	uint32_t temp;
	while(SSIDataGetNonBlocking(uiBase,&temp));
	for(temp_count=0;temp_count<Length;temp_count++)
	{
```

```
            SSIDataPut(uiBase,*(Pbuff));
            SSIDataGet(uiBase,&temp);
            *(Pbuff++) = temp;
        }
}
```

3. GUI communication

This function receives command from GUI using USB and passes the control to the message decoder.

```
/*****************************************************************************
 *    @name   ListenToGuiCommands
 *    @brief        Listens To Gui Commands over RS485(UART)
 *    @param    None
 *    @return   None
 *****************************************************************************/
void ListenToGuiCommands()
{
        unsigned char cmd_status = CMD_NOT_RECEIVED;
        while(1)
        {
                /* Wait for Command from GUI */

                cmd_status = MsgParser();
                if(cmd_status != CMD_NOT_RECEIVED)
                {
                        MsgDecoder();
                }

        }
}
```

3.1 Message Parser

Decodes the message into relevant fields

```
/*****************************************************************************
 *    @name         MsgParser
 *    @brief        Decomposes the received packet into relevant fields

 *    @param        None
 *    @return       None
 *****************************************************************************/
unsigned char MsgParser(void)
{
        unsigned int i = 0,j=0;
        uint16_t crc = 0;
        unsigned char timeout_flag = 0;
        if(!g_rx.pWrite)
        {/* if Write pointer is at Zero location i.e. no new command has been received */
                return CMD_NOT_RECEIVED;
        }

        /* Start Packet Parsing i.e. decompose it into required fields */
        g_cmd.hdr1 = g_rx.buff[j];

        if(g_cmd.hdr1 != CMD_HEADER_1)
        {
```

```
            //Invalid Command Received or Data Corrupted on UART Bus.
            // Alert User to debug and return
            g_rx.pWrite = 0; //reinitialize the pointer for next command
            return CMD_NOT_RECEIVED;
      }

      timerTimeoutFlag1=0;
      Timer0Enable();
      while(g_rx.pWrite <= 2 && timerTimeoutFlag1==0)
      {
      }
      Timer0Disable();
      if (timerTimeoutFlag1==1)
      {
            timeout_flag = 1;
      }

      if(timeout_flag == 1)
      {
            g_rx.pWrite = 0; //reinitialize the pointer for next command
            return CMD_NOT_RECEIVED;
      }

      j += sizeof(g_cmd.hdr1);
      g_cmd.hdr2 = g_rx.buff[j];
      if(g_cmd.hdr2 != CMD_HEADER_2)
      {
            //Invalid Command Received or Data Corrupted on UART Bus.
            // Alert User to debug and return
            g_rx.pWrite = 0; //reinitialize the pointer for next command
            return CMD_NOT_RECEIVED;
      }
      /* atleast 4 bytes required to proceed.
       * 4 = 1 byte for mpicAddr, 1 byte for Command, 1byte of Subcommand and 1 byte for
Payload length*/
      j += sizeof(g_cmd.hdr2);
      i = sizeof(g_cmd.hdr1) + sizeof(g_cmd.hdr2) + sizeof(g_cmd.mpicAddr)
+sizeof(g_cmd.slot)+ sizeof(g_cmd.cmd)+ sizeof(g_cmd.subCmd) + sizeof(g_cmd.len);

      timerTimeoutFlag1=0;
      Timer0Enable();
      while(g_rx.pWrite <= 2 && timerTimeoutFlag1==0)
      {
      }
      Timer0Disable();
      if (timerTimeoutFlag1==1)
      {
            timeout_flag = 1;
      }

      if(timeout_flag == 1)
      {
            g_rx.pWrite = 0; //reinitialize the pointer for next command
            return CMD_NOT_RECEIVED;
```

```
	}

	g_cmd.mpicAddr = g_rx.buff[j];
	if(g_cmd.mpicAddr != GetMPICAddr())
	{
		g_rx.pWrite = 0; //reinitialize the pointer for next command
		return CMD_NOT_RECEIVED;
	}

	j += sizeof(g_cmd.mpicAddr);
	g_cmd.slot = g_rx.buff[j];
	j += sizeof(g_cmd.slot);
	g_cmd.cmd = g_rx.buff[j];
	j += sizeof(g_cmd.cmd);
	g_cmd.subCmd = g_rx.buff[j];
	j += sizeof(g_cmd.subCmd);
	g_cmd.len = g_rx.buff[j];
	g_cmd.len = (g_cmd.len<<8) | g_rx.buff[j+1];
	j += sizeof(g_cmd.len);
	i += (g_cmd.len + sizeof(g_cmd.crc));//now i represents total packet size

	timerTimeoutFlag1=0;
	Timer0Enable();
	while( g_rx.pWrite <= g_cmd.len && timerTimeoutFlag1==0)
	{
	}
	Timer0Disable();
	if (timerTimeoutFlag1==1)
	{
		timeout_flag = 1;
	}

	if(timeout_flag == 1)
	{
		g_rx.pWrite = 0; //reinitialize the pointer for next command
		return CMD_NOT_RECEIVED;
	}

	g_cmd.pBuff = &g_rx.buff[j];
	j += g_cmd.len;
	g_cmd.crc = g_rx.buff[j];
	g_cmd.crc <<= 8;//shift it to upper byte
	g_cmd.crc |= g_rx.buff[j+1];

	//Checksum – 2's complement of Sum of Instrument ID, Functionality ID, Payload Length
and Payload  Data
	crc = EvalCRC( &g_rx.buff[sizeof(g_cmd.hdr1) + sizeof(g_cmd.hdr2) ],
			i-(sizeof(g_cmd.crc) + sizeof(g_cmd.hdr1) + sizeof(g_cmd.hdr2)) );

	if(crc != g_cmd.crc)
	{
		// Invalid Packet Received or Data Corrupted on UART Bus.
		// Alert User to debug and return
```

```
            g_rx.pWrite = 0; //reinitialize the pointer for next command
            g_cmd.cmd = CMD_ERROR;
            g_cmd.subCmd = CMD_INVALID_CRC;
            return CMD_RECEIVED_ERRONEOUS;
        }

        g_rx.pWrite = 0; //reinitialize the pointer for next command
        /* End of Packet Parsing*/
        return CMD_RECEIVED_VALID;

}
```

3.2 Message decoder

Checks whether command is valid and segregates the command based on command type, and calls respective functions to handle the command and responds with a reply.

```
/******************************************************************************
 *    @name        MsgDecoder
 *    @brief       Decodes the command and invokes the corresponding Function

 *    @param       None
 *    @return      None
 ******************************************************************************/
void MsgDecoder(void)
{
        uint16_t Sample_rate=0, No_of_Samples;
        uint16_t temp=0;
        uint32_t dummy;
        uint16_t data;
        unsigned char temp_array[20]={0};
        I2C_DATA_BUFF I2C_buff={0};
        uint8_t status=0;

        // Check whether the command is valid or not
        if(g_cmd.cmd == CMD_ERROR)
        {
                g_cmd.cmd = CMD_ERROR;
                g_cmd.subCmd = 0x00;
                temp = CMD_ERROR;
                GuiMsg( CMD_ERROR_RESPONSE_SIZE,(unsigned char*) &temp);
                return;
        }


        switch (g_cmd.cmd)
        {
        case CMD_SYS_RESET:
                // System Reset
                SysCtlReset();
                break;
        case ANIO_CARD: //0xC0


                switch(g_cmd.subCmd)
```

An IMPORTANT NOTICE at the end of this TI reference design addresses authorized use, intellectual property matters and other important disclaimers and information.

```
            {
/*****************************************DAC8760*****************************************
***********************************************/
        //FOR device 1
            case DAC_1_READ://0xD1
                  data=DAC8760_Read_Write(DAC_DEVICE1,CMD_READ,g_cmd.pBuff);
                  GuiMsg(sizeof(data),(unsigned char*)&data);// to send the read data to
the GUI
                  break;

            case DAC_1_WRITE://0xD2
            case DAC_1_SET_DATA://0xD3
                  data=DAC8760_Read_Write(DAC_DEVICE1,CMD_WRITE,g_cmd.pBuff);
                  GuiMsg(sizeof(data),(unsigned char*)&data);//send the written data back
to the GUI
                  break;

//FOR device 2
            case  DAC_2_READ://0xD9
                  data=DAC8760_Read_Write(DAC_DEVICE2,CMD_READ,g_cmd.pBuff);
                  GuiMsg(sizeof(data),(unsigned char*)&data);
                  break;

            case  DAC_2_WRITE://0xDA
            case  DAC_2_SET_DATA://0xDB
                  data=DAC8760_Read_Write(DAC_DEVICE2,CMD_WRITE,g_cmd.pBuff);
                  GuiMsg(sizeof(data),(unsigned char*)&data);
                  break;
/*******************************************************************ADS8688
********************************************************************/
            case ADC_READ://0xA1
            case ADC_WRITE://0xA2

                  HWREG(spiBase + SSI_O_CR0) = (HWREG(spiBase + SSI_O_CR0) & 0xFFFFFF7F) |
0x80;

                  while(SSIDataGetNonBlocking(spiBase,&dummy));

                  ADS8684_DAC8760_CS_ADC_Low();

                  Spi_Read_write_8bit(g_cmd.pBuff,g_cmd.len,spiBase);

                  ADS8684_DAC8760_CS_ADC_High();

                  HWREG(spiBase + SSI_O_CR0) = (HWREG(spiBase + SSI_O_CR0) & 0xFFFFFF7F);

                  GuiMsg(4,g_cmd.pBuff);
                  break;

            case ADC_GET_DATA://0xA3
                  No_of_Samples =  ((*(g_cmd.pBuff++)) << 8) | (*(g_cmd.pBuff++));

                  Sample_rate =  ((*(g_cmd.pBuff++)) << 8) | (*(g_cmd.pBuff++));
                  Get_ADS8684_Data(No_of_Samples,Sample_rate);
```

```
                break;

            case CMD_READ_IO://0xA5

                I2C_buff.i2c_base = I2C1_IO_BASE;
                I2C_buff.dev_addr = I2C_ANIO_IO_ADDR;

            // Keep all the pins Low
                temp_array[0]= I2C_IO_OUTPUT_READ_WRITE;

                I2C_buff.size =1;
                I2C_buff.pBuff =&temp_array[0];
                status = I2C_Send(&I2C_buff);

                temp_array[0]= I2C_IO_OUTPUT_READ_WRITE;

                I2C_buff.size =1;
                I2C_buff.pBuff =&temp_array[0];
                status = I2C_Recv(&I2C_buff);


                if(status)
                      Send_Nack();
                else
                      GuiMsg(2,temp_array);

            case CMD_WRITE_IO:

                I2C_buff.i2c_base = I2C1_IO_BASE;
                I2C_buff.dev_addr = I2C_ANIO_IO_ADDR;

                temp_array[0] = *(g_cmd.pBuff);                                    //
Keep all the pins Low
                temp_array[1]= I2C_IO_OUTPUT_READ_WRITE;

                I2C_buff.size =2;
                I2C_buff.pBuff =&temp_array[0];
                status = I2C_Send(&I2C_buff);
                if(status)
                      Send_Nack();
                else
                      Send_Ack();
                break;
            default:
                Sub_Command_Invalid();
                break;

                }//Switch case end
            break;//Comb_card end case

        case CMD_HEALTH_CHECK: // 0x50 :
            if(g_cmd.subCmd ==0x00)
            {
                //Health Check
```

An IMPORTANT NOTICE at the end of this TI reference design addresses authorized use, intellectual property matters and other important disclaimers and information.

```
                static unsigned char test[5] = {0};
                test[0] = 0x12;
                test[1] = 0x34;
                test[2] = 0x56;
                test[3] = 0x78;
                test[4] = 0x90;
                GuiMsg(5,test);
            }
            else
                Sub_Command_Invalid();
            break;

        default:
            Command_Invalid();
            break;

        }

}
```

IMPORTANT NOTICE FOR TI REFERENCE DESIGNS

Texas Instruments Incorporated ("TI") reference designs are solely intended to assist designers (“Buyers”) who are developing systems that incorporate TI semiconductor products (also referred to herein as “components”). Buyer understands and agrees that Buyer remains responsible for using its independent analysis, evaluation and judgment in designing Buyer’s systems and products.

TI reference designs have been created using standard laboratory conditions and engineering practices. **TI has not conducted any testing other than that specifically described in the published documentation for a particular reference design.** TI may make corrections, enhancements, improvements and other changes to its reference designs.

Buyers are authorized to use TI reference designs with the TI component(s) identified in each particular reference design and to modify the reference design in the development of their end products. HOWEVER, NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY THIRD PARTY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT, IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI REFERENCE DESIGNS ARE PROVIDED "AS IS". TI MAKES NO WARRANTIES OR REPRESENTATIONS WITH REGARD TO THE REFERENCE DESIGNS OR USE OF THE REFERENCE DESIGNS, EXPRESS, IMPLIED OR STATUTORY, INCLUDING ACCURACY OR COMPLETENESS. TI DISCLAIMS ANY WARRANTY OF TITLE AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT, QUIET POSSESSION, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS WITH REGARD TO TI REFERENCE DESIGNS OR USE THEREOF. TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY BUYERS AGAINST ANY THIRD PARTY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON A COMBINATION OF COMPONENTS PROVIDED IN A TI REFERENCE DESIGN. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, SPECIAL, INCIDENTAL, CONSEQUENTIAL OR INDIRECT DAMAGES, HOWEVER CAUSED, ON ANY THEORY OF LIABILITY AND WHETHER OR NOT TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, ARISING IN ANY WAY OUT OF TI REFERENCE DESIGNS OR BUYER’S USE OF TI REFERENCE DESIGNS.

TI reserves the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques for TI components are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers’ products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers’ products and applications, Buyers should provide adequate design and operating safeguards.

Reproduction of significant portions of TI information in TI data books, data sheets or reference designs is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards that anticipate dangerous failures, monitor failures and their consequences, lessen the likelihood of dangerous failures and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in Buyer’s safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed an agreement specifically governing such use.

Only those TI components that TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components that have ***not*** been so designated is solely at Buyer's risk, and Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265