Software Description

MCU selection and peripherals Pin details

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 TM4C123 GXL Launchpad for design and testing. The ground pin should be shorted between the Launchpad and the RTD test board.

I2C bus is used to control the ADC and to read the status. SPI is used to read/ write the ADC registers and hence used for initialization and data retrieval.

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. Transmit and receive FIFOs can be programmed as destination/source addresses in the µDMA module.

In this 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)

```
//
// The SSI0 peripheral must be enabled for use.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);
//
// For this example SSI0 is used with PortA[5:2].  The actual port and pins
// used may be different on your part, consult the data sheet for more
// information.  GPIO port A needs to be enabled so these pins can be used.
// TODO: change this to whichever GPIO port you are using.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
//
// Configure the pin muxing for SSI0 functions on port A2, A3, A4, and A5.
// This step is not necessary if your part does not support pin muxing.
// TODO: change this to select the port/pin you are using.
//
GPIOPinConfigure(GPIO_PA2_SSI0CLK);
GPIOPinConfigure(GPIO_PA3_SSI0FSS);
GPIOPinConfigure(GPIO_PA4_SSI0RX);
GPIOPinConfigure(GPIO_PA5_SSI0TX);

//
```

```
    // Configure the GPIO settings for the SSI pins.  This function also gives
    // control of these pins to the SSI hardware.  Consult the data sheet to
    // see which functions are allocated per pin.
    // The pins are assigned as follows:
    //      PA5 - SSI0Tx
    //      PA4 - SSI0Rx
    //      PA3 - SSI0Fss
    //      PA2 - SSI0CLK
    // TODO: change this to select the port/pin you are using.
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE, GPIO_PIN_5 | GPIO_PIN_4 | GPIO_PIN_3 |
                   GPIO_PIN_2);
    //
    // Configure and enable the SSI port for SPI master mode.  Use SSI0,
    // system clock supply, idle clock level low and active low clock in
    // freescale SPI mode, master mode, 1MHz SSI frequency, and 8-bit data.
    // For SPI mode, you can set the polarity of the SSI clock when the SSI
    // unit is idle.  You can also configure what clock edge you want to
    // capture data on.  Please reference the datasheet for more information on
    // the different SPI modes.
    //
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_1,
                       SSI_MODE_MASTER, 2000000, 8);
    //
    // Enable the SSI0 module.
    //
    SSIEnable(SSI0_BASE);
    SSIEnable(SSI1_BASE);
    //
    // Read any residual data from the SSI port.  This makes sure the receive
    // FIFOs are empty, so we don't read any unwanted junk.  This is done here
    // because the SPI SSI mode is full-duplex, which allows you to send and
    // receive at the same time.  The SSIDataGetNonBlocking function returns
    // "true" when data was returned, and "false" when no data was returned.
    // The "non-blocking" function checks if there is any data in the receive
    // FIFO and does not "hang" if there isn't.
    //
    while(SSIDataGetNonBlocking(SSI0_BASE, &pui32DataRx[0]))
    {
    }
```

SPI communication

The SPI is communicating at 2 Mbps speed. Chip select should be low to communicate and the start pin has to be asserted high, before reading configuration from ADS1248.

After each byte is written, values from ADC are read back into the same register from next cycle.

```
/*****************************************************************************
*    @name        Spi_Read_write_8bit
*    @brief       Send/ reseive 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)
{

       uint32_t temp=0;
       while(SSIDataGetNonBlocking(uiBase,&temp));
       for(temp_count=0;temp_count<Length;temp_count++)
       {
              SSIDataPut(uiBase,*(Pbuff));
              SSIDataGet(uiBase,&temp);
              *(Pbuff++) = temp;
       }
}
```

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

```
void I2C1_Init(void)
{
       unsigned int Status=0;

       SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C1);
       SysCtlPeripheralReset(SYSCTL_PERIPH_I2C1);
       SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

       SysCtlPeripheralClockGating(true);
       SysCtlDelay(2);

       GPIOPinConfigure(GPIO_PA6_I2C1SCL);
       GPIOPinConfigure(GPIO_PA7_I2C1SDA);
       SysCtlDelay(2);
```

```
        GPIOPinTypeI2CSCL(GPIO_PORTA_BASE, GPIO_PIN_6);
        GPIOPinTypeI2C(GPIO_PORTA_BASE, GPIO_PIN_7);
        SysCtlDelay(2);

        ROM_I2CMasterInitExpClk(I2C1_BASE, SYS_CLOCK, false);
        SysCtlDelay(2);

         Status=I2CMasterLineStateGet(I2C1_BASE);
}
```

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. I2C bus is also used to switch the ADC chip select and to monitor for /DRDY

I2C send/ receive routines

```
/******************************************************************************
 *    @name        I2C_Send
 *    @brief       This function sends data on SDA Line of I2C Protocol.
 *                           After Start bit MSB is transmitted first. Therefore, 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);
    }

    // Wait until transmission completes
    // To avoid staying here on no daughter card and to finish transmission
    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);

		// Wait until transmission completes
	//	while(I2CMasterBusy(base)){}
	//	usecWait(300);                              // To avoid staying here on no
daughter card and to finish transmission
		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;
}
```

1.3 ADS1248

Generation of Excitation current

- The internal voltage reference should be ON (MUX1 register, VREFCON) to generate the excitation current on the DACs.
- ADS1248 generates two excitation currents that could be output to pins AIN0…AIN7 or IEXT1/ IEXT2 using register IDAC1.
- By default, the excitation current is disconnected and can be turned ON using IDAC1.
- The excitation current can be varied between 50uA to 1500uA using register IDAC0. By default the excitation currents are turned off.

ADC reference

The ADC internal reference is always ON. This can be done with register MUX1. The external reference input pair REF1 (REFSELT1) and normal mode of operation (MUXCAL) is selected.

Chip Select

- Whenever it is intended to communicate to ADS1248, it is necessary to assert CS low. After communication is over it is asserted high.
- Start pin has to be high to communicate with the ADC.

```
void Set_CS_Low(void)
{
        uint8_t I2CData = 0;
        I2CData = Read_I2C_Expander_Data();
        I2CData &=0xEF;                                  // Clear other bits (except CS)

        i2c_write_data[0]=I2CData;

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE; //reset to pull low //check
        Send_I2C1_Expander_Cmd(i2c_write_data);
}

void Set_CS_High(void)
{
        uint8_t I2CData = 0;
        I2CData = Read_I2C_Expander_Data();
        I2CData &=0xEF;                                  // Clear other bits (except CS)

        i2c_write_data[0]=I2CData | 0x10;                //set CS

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE; //reset to pull low //check
        Send_I2C1_Expander_Cmd(i2c_write_data);
}
```

Convert Start

- Start can be pulled high all the time to enable continuous sampling happens. The other way is to time the start signal and assert the pin high for a short duration (more than 3 ADC clock cycles). When the start pin is low, it is not possible to read the configuration registers.
- Allow 32 ADC clock cycles for ADC to settle.
- Wait for DRDY/
- Assert the chip select pin
- Issue a READ command..

```
void Set_StartConv_High(void)
{
        uint8_t I2CData = 0;
        I2CData = Read_I2C_Expander_Data();
        I2CData &=0xF7;                       // Clear other bits (except START)

        i2c_write_data[0]=I2CData | 0x08;//SET_ADC_START_CONV

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE;
        Send_I2C1_Expander_Cmd(i2c_write_data);
        SysCtlDelay(100);
}


void Set_StartConv_Low(void)
{
        uint8_t I2CData = 0;
        I2CData = Read_I2C_Expander_Data();
        I2CData &=0xF7;                    // Clear other bits (except START)

        i2c_write_data[0]=I2CData;
        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE;
        Send_I2C1_Expander_Cmd(i2c_write_data);
}
```

1.4 ADC Initialization for RTD sampling (Ratiometric)

- When ADC is initialized for RTD sampling, the excitation currents should be turned ON and routed through IEXC1 and 2.
- The excitation current of 500 uA is chosen.
- Current swapping by routing the same current through the different excitation source and averaging them will cancel out errors.
- Internal reference is always ON.
- The external reference of REF1 pair is selected.
- Gain of 16 and sampling rate of 20 samples per second settings are chosen.
- During initialization self-offset calibration process is initiated, followed by a settling time delay.

```
void ADS1248_Init(void)
{
//SPI pins are already configured

//Reset low is done already

//chip select
        Set_CS_Low();
        SysCtlDelay(10);

//configure the ADC registers
        Spi_write_data[0] = WREG | IDAC0;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x04;//excitation current 500uA (RTD)
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

#ifdef CURRENT_SWAP_TEST  //Alternate between IEXC1 and IEXC2 to cancel errors
        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | IDAC1;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x98;//0xFF - to disconnect //select IDAC output pins
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();
#else
        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | IDAC1;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x89;//0xFF - to disconnect //select IDAC output pins
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();
#endif
```

```
        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | MUX1;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x28;//internal osc, internal reference ON, REF1 input pair
selected, Normal measurement
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();
        msecWait(10);

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | SYS0;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x42;//Select gain 16 and sampling rate 20sps
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        msecWait(10);

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | OFC0;
        Spi_write_data[1] = THREE_REGISTERS_READ_WRITE;
        Spi_write_data[2] = 0x00;//OFFSET calibration
        Spi_write_data[3] = 0x00;
        Spi_write_data[4] = 0x00;
        Spi_Read_write_8bit(Spi_write_data, 5, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

#ifdef GPIO_CONFIG  //To reduce interference on I2C bus, by ADS INT and RESET pins
        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | GPIOCFG;
        Spi_write_data[1] = ONE_REGISTERS_READ_WRITE;
        Spi_write_data[2] = 0x03;//config as GPIO
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | GPIODIR;
        Spi_write_data[1] = ONE_REGISTERS_READ_WRITE;
        Spi_write_data[2] = 0x03;//config as input
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();
#endif

        Set_CS_Low();//self offset calibration
        SysCtlDelay(10);
        Spi_write_data[0] = 0x62;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_Read_write_8bit(Spi_write_data, 2, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        msecWait(802);
}
```

1.5 ADC initialization for ADC mode (without excitation current and internal reference switched on)

To utilize ADS1248 in ADC mode without excitation current,

- the excitation current has to be turned OFF or
- the IDACs have to be disconnected and
- On board internal reference has to be selected.

```
void ADS1248_Init(void)
{
//SPI pins are already configured

//Reset low is done already

//chip select
        Set_CS_Low();
        SysCtlDelay(10);

//configure the ADC registers

        Spi_write_data[0] = WREG | IDAC0;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x04;//500uA for each excitation
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | IDAC1;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0xFF;//select IDAC output pins// Now disconnected
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | MUX1;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x30;//0x28;
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();
        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | SYS0;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = 0x40; //choose sampling rate and gain
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | OFC0;
        Spi_write_data[1] = THREE_REGISTERS_READ_WRITE;
        Spi_write_data[2] = 0x00;//OFFSET calibration
        Spi_write_data[3] = 0x00;
        Spi_write_data[4] = 0x00;
        Spi_Read_write_8bit(Spi_write_data, 5, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        Set_CS_Low();
        SysCtlDelay(10);
        Spi_write_data[0] = WREG | FSC0;
        Spi_write_data[1] = THREE_REGISTERS_READ_WRITE;
        Spi_write_data[2] = 0x00;//calibration coefficient- gain scaling 1
        Spi_write_data[3] = 0x00;
        Spi_write_data[4] = 0x40;
        Spi_Read_write_8bit(Spi_write_data, 5, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        Set_CS_Low();//self offset calibration
        SysCtlDelay(10);
        Spi_write_data[0] = 0x62;
        Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
        Spi_Read_write_8bit(Spi_write_data, 2, SSI0_BASE);
        SysCtlDelay(10);
        Set_CS_High();

        msecWait(10);
}
```

2. Functional Description

2.1 I2C Expander

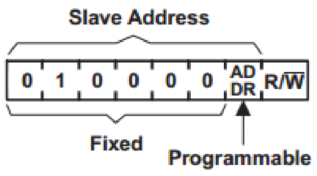


Figure 4. TCA6408A Address

Address Reference

ADDR	I^{2}C 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 has an I2C can be used to configure the pins as inputs or outputs.

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

The bit definitions for the I2C expander pins are shown below.

- The DRDY/ is configured as an input and the other 7 bits are configured as outputs.
- The I2C bus is also used to assert the start, chip select, reset/ pins and the two LEDs. The I2C bus is also used to read the DRDY/ signal from ADC.

```
#define SELECT_RTD_CHANNEL1      0x00   //RTD channel selection
#define SELECT_RTD_CHANNEL2      0x01
#define SELECT_RTD_CHANNEL3      0x02
#define SELECT_RTD_CHANNEL4      0x03

#define CHK_ADC_RDY_N            0x04   //input
#define SET_ADC_START_CONV       0x08
#define SET_ADC_CS_N             0x10
#define SET_ADC_RESET_N          0x20

#define LED1_SET_RESET           0x40
#define LED2_SET_RESET           0x80
```

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^{2}C 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 Initial configuration for I2C port pins

The configuration register (0x03) bits can be set to 1 to make it input or 0 to make it output.
Then using the output register (0x01) appropriate bits can be set to 1 or 0. The input register (0x00) can be used to read the port pin values.

```
uint8_t Set_I2C1_Expander_Config (void)
{
	I2C_DATA_BUFF ADS1248_config_dir_buff = {0};
	unsigned int data_len = 2;
	unsigned char status = I2C_OP_PASS;

	ADS1248_config_dir_buff.i2c_base = I2C1_BASE;
	ADS1248_config_dir_buff.dev_addr = I2C_EXPANDER_ADDR | WRITE;
	ADS1248_config_dir_buff.size = data_len;

	i2c_write_data[0] = 0x04;		//bitwise '0' for output and '1' for Input
	i2c_write_data[1] = I2C_IO_CONFIG;	//Configuration reg of I2c expander
```

```
        ADS1248_config_dir_buff.pBuff = i2c_write_data;

        status = I2C_Send(&ADS1248_config_dir_buff);

        if(status == I2C_OP_FAIL)
        {
                return status;
        }
        return status;
}
```

2.2.1 Setting I2C configuration defaults

```
void    Set_I2C_Expander_Defaults (void)
{
        i2c_write_data[0]=0xD8;
        //Set the value of LEDs to high
        //set chip select to low

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE;
        Send_I2C1_Expander_Cmd(i2c_write_data);

        i2c_write_data[0]=0xE8;
        //Set the value of LEDs to high
        //set chip select to low

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE;
        Send_I2C1_Expander_Cmd(i2c_write_data);
}
```

2.3 Check if ADC conversion is complete using DRDY bit

Before any data is read from the ADC, DRDY/ pin should be low. Only then the read data command can be issued.

```
        // Look for DRDY signal going Low
        // DRDY signal is monitored through the IO Expander
        temp_buff.i2c_base = I2C_base;
        temp_buff.dev_addr =I2C_EXPANDER_ADDR;
        temp_buff.pBuff =&data;
        temp_buff.size =1;
        temp= 0xFF;
        data = 0x00;

        // Loop to wait till DRDY goes low
        while(temp)
        {
                data = I2C_IO_INPUT_READ;
                temp_buff.size =1;
                temp_buff.pBuff =&data;
                I2C_Send(&temp_buff);
                temp_buff.size =1;
                status = I2C_Recv(&temp_buff);
```

```
            if(status == I2C_OP_FAIL)
            {
                   Error_status = TRUE;
                   break;
            }
            temp = data & CHK_ADC_RDY_N;                     // Getting DRDY Bit
            SysCtlDelay(10);
    }
```

2.4 Read the I2C expander port pins

The following code snippet can be used to read the I2C expander port pins.

```
uint8_t Read_I2C_Expander_Data(void)
{
       I2C_DATA_BUFF ADS1248_CS_DRDY_buff = {0};
       unsigned int data_len = 1;
       unsigned char status = I2C_OP_PASS;

       ADS1248_CS_DRDY_buff.i2c_base = I2C1_BASE;
       ADS1248_CS_DRDY_buff.dev_addr = I2C_EXPANDER_ADDR | READ;
       ADS1248_CS_DRDY_buff.size = data_len;

       ADS1248_CS_DRDY_buff.pBuff = i2c_write_data;

       i2c_write_data[0]=I2C_IO_OUTPUT_READ_WRITE;

       status = I2C_Send(&ADS1248_CS_DRDY_buff);
       status = I2C_Recv(&ADS1248_CS_DRDY_buff);

       return i2c_write_data[0];
}
```

2.5 External Multiplexer

The I2C expander pins RTD_SEL0 and RTD_SEL1 offer the channel selection.

RTD_SEL0	RTD_SEL1	Channel selected
0	0	RTD channel 1 (default)
0	1	RTD channel 2
1	0	RTD channel 3
1	1	RTD channel 4

The following code snippet does the external multiplexer's channel selection. Then the IEXC1 and IEXC2 are routed through the appropriate RTD channel. (ex: RTD1_IEXC1)

Note: Care should be taken to switch appropriate internal ADC MUX settings that match with the external multiplexer settings.

```
void   Select_I2C_Channels (uint8_t channelNum)
{
       uint8_t I2CData = 0;
       I2CData = Read_I2C_Expander_Data();
       I2CData &=0xFC;// Clear the channel selection bits
```

```
        switch (channelNum)
        {
        case 0:
                i2c_write_data[0]=0x00 | I2CData;
                break;
        case 1:
                i2c_write_data[0]=0x01 | I2CData;
                break;
        case 2:
                i2c_write_data[0]=0x02 | I2CData;
                break;
        case 3:
                i2c_write_data[0]=0x03 | I2CData;
                break;
        }

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE;

        Send_I2C1_Expander_Cmd(i2c_write_data);
}

uint8_t Send_I2C1_Expander_Cmd (uint8_t *write_data)
{
        I2C_DATA_BUFF ADS1248_CS_DRDY_buff = {0};
        unsigned int data_len = 2;
        unsigned char status = I2C_OP_PASS;

        ADS1248_CS_DRDY_buff.i2c_base = I2C1_BASE;
        ADS1248_CS_DRDY_buff.dev_addr = I2C_EXPANDER_ADDR | WRITE;
        ADS1248_CS_DRDY_buff.size = data_len;

        ADS1248_CS_DRDY_buff.pBuff = i2c_write_data;

        status = I2C_Send(&ADS1248_CS_DRDY_buff);

        if(status == I2C_OP_FAIL)
        {
                return status;
        }
        return status;
}
```

2.6 RTD channel switching

The following code snippet, switches external multiplexer through I2C bus and the ADC internal multiplexer settings through SPI bus.

For ADC, to process AIN0 as positive and AIN1 as negative, MUX0 register is chosen as 0x01.

```
void select_Channel(uint8_t ChnNum)
{
        Set_CS_Low();
        SysCtlDelay(10);
        switch (ChnNum)
        {
        case 0:
                Select_I2C_Channels(0);
                //first take default readings for AIN0+ and AIN1- channel1
                Spi_write_data[0] = WREG | MUX0;
                Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
                Spi_write_data[2] = 0x01;//AIN0 positive and AIN1 negative
                Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
                break;
        case 1:
                Select_I2C_Channels(1);
                //second channel
                Spi_write_data[0] = WREG | MUX0;
                Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
                Spi_write_data[2] = 0x13;//AIN2 positive and AIN3 negative
                Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
                break;
        case 2:
                Select_I2C_Channels(2);
                //third channel
                Spi_write_data[0] = WREG | MUX0;
                Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
                Spi_write_data[2] = 0x25;//AIN4 positive and AIN5 negative
                Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
                break;
        case 3:
                Select_I2C_Channels(3);
                //fourth channel
                Spi_write_data[0] = WREG | MUX0;
                Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
                Spi_write_data[2] = 0x37;//AIN6 positive and AIN7 negative
                Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
                break;
        }
        SysCtlDelay(10);
        Set_CS_High();
}
```

2.7 Read ADC Data

Wait for /DRDY to be asserted. Then issue a read data command on SPI bus. This is followed by sending 0xFF thrice to send 24 clock cycles to ADC.

```
        temp_array[0] = RDATA;//0x20;
        temp_array[1] = 0xFF;
        temp_array[2] = 0xFF;
        temp_array[3] = 0xFF;

        Set_CS_Low();
        SysCtlDelay(10);

        Spi_Read_write_8bit(temp_array, 4, spiBase);
        // Get 24 bit data from ADC
        for(temp=0;temp<3;temp++)
        {
                data = temp_array[temp];
                Rtd_data = (Rtd_data<<8) | data;
        }
```

2.8 LED indications

The LEDs are used to indicate the present status of the RTD channel that is getting scanned. Binary logic is used to show the RTD channel number via 2 LEDs.

Present Scanning of channel	LED1 (D2) status	LED2 (D3) status
Channel 1	OFF	OFF
Channel 2	OFF	ON
Channel 3	ON	OFF
Channel 4	ON	ON

```
void    Select_I2C_Channels (uint8_t channelNum)
{
        uint8_t I2CData = 0;
        I2CData = Read_I2C_Expander_Data();
        I2CData &= 0x3C; //clear channel selection + LED 0xFC;// Clear the channel selection 
bits
        switch (channelNum)
        {
        case 0:
                i2c_write_data[0]=0x00 | I2CData | 0x00;
                break;
        case 1:
                i2c_write_data[0]=0x01 | I2CData | 0x40;
                break;
        case 2:
                i2c_write_data[0]=0x02 | I2CData | 0x80;
                break;
        case 3:
                i2c_write_data[0]=0x03 | I2CData | 0xC0;
                break;
        }

        i2c_write_data[1]=I2C_IO_OUTPUT_READ_WRITE;

        Send_I2C1_Expander_Cmd(i2c_write_data);
}
```

3. Sensor Open/ short testing

When enabled, two burnout current sources flow through the selected pair of analog inputs to the sensor. One sources the current to the positive input channel, and the other sinks the same current from the negative input channel. When the burnout current sources are enabled, a full-scale reading may indicate an open circuit in the front-end sensor, or that the sensor is overloaded. It may also indicate that the reference voltage is absent. A near-zero reading may indicate a short-circuit condition in the sensor.

```
	//
	//	Check the channels in which sensor is present
	//
	Set_CS_Low();
	Set_StartConv_High();

#ifdef SENSOR_DETECT

	SysCtlDelay(10);
	Spi_write_data[0] = 0x40 | 0x0A;//WREG | IDAC0;
	Spi_write_data[1] = 0x00;//ONE_REGISTER_READ_WRITE;
	Spi_write_data[2] = 0x00;//OFF
	Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);


	SysCtlDelay(10);
	Spi_write_data[0] = 0x40 | 0x02; //MUX1
	Spi_write_data[1] = 0x00;
	Spi_write_data[2] = 0x30;
	Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);

	msecWait(10);

	Sensor_Open=0;
	Sensor_Short=0;
	Sensor_Active=0;

	for (ChnNum=0; ChnNum < 4; ChnNum++)
	{
		Select_I2C_Channels(ChnNum);
		msecWait(2);
		CheckSensorPresent(ChnNum);

		msecWait(100);

		Sum_Burnout = 0;

		for (i1=0; i1<250; i1++)
		{
			RTD[ChnNum]= Read_RTD_data(SSI0_BASE, I2C1_BASE,ChnNum);

			Sum_Burnout = Sum_Burnout + RTD[ChnNum];
		}

		if (Sum_Burnout > 8175000)
		{	//Fix an arbitrary threshold to detect sensor open
			//Open circuit or sensor overload
			Sensor_Open |= (1<<ChnNum);
		}
		else
		{
			if (Sum_Burnout <= 217000)short
			{	//Fix an arbitrary threshold to detect sensor
```

```
                        //short circuit
                        Sensor_Short |= (1<<ChnNum);
                    }
                    else
                    {
                        Sensor_Active |= (1<<ChnNum);
                    }
                }

            }
#endif



/*******************************************************************************
*********/
//    void CheckSensorPresent(uint8_t ChnNum)
//
//    Description:
//                        1. Switches RTD channel Mux0 register based on ChnNum parameter
//                        2. Route the Burn out current source to the respective RTD
channel pins
//    Precondition:
//                        Ensure START is high before calling this function
//                        Ensure SPI peripheral and pins are initialized
//                        Ensure the MACRO, SENSOR_DETECT is defined, to use this function
//    Returns:    None
/*******************************************************************************
*********/
void CheckSensorPresent(uint8_t ChnNum)
{
    uint8_t RTDchn=0;
    switch (ChnNum)
    {
    case 0:
        RTDchn = 0x01;
        break;
    case 1:
        RTDchn = 0x25;
        break;
    case 2:
        RTDchn = 0x37;
        break;
    case 3:
        RTDchn = 0x13;
        break;
    default:
        break;
    }
    Spi_write_data[0] = WREG | MUX0;
    Spi_write_data[1] = ONE_REGISTER_READ_WRITE;
    Spi_write_data[2] = 0xC0 | RTDchn;
    Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
```

```
        SysCtlDelay(10);
        Spi_write_data[0] = 0x40 | 0x0B;//WREG | IDAC1;
        Spi_write_data[1] = 0x00;//ONE_REGISTER_READ_WRITE;
        Spi_write_data[2] = RTDchn;
        Spi_Read_write_8bit(Spi_write_data, 3, SSI0_BASE);
}
```

4. GUI interface

4.1 GUI Installation

The GUI has some component dependencies:

Install the software in the following order:

1. LV Runtime - http://www.ni.com/download/labview-run-time-engine-2010-sp1/2292/en/
2. VISA Runtime - http://www.ni.com/download/ni-visa-run-time-engine-5.1.2/2918/en/
3. The GUI ".exe" file can be run now

4.2 Usage

The GUI works only when the MACRO "GUI" is defined. Ensure that the GUI is not in simulation mode by unchecking the "checkbox.

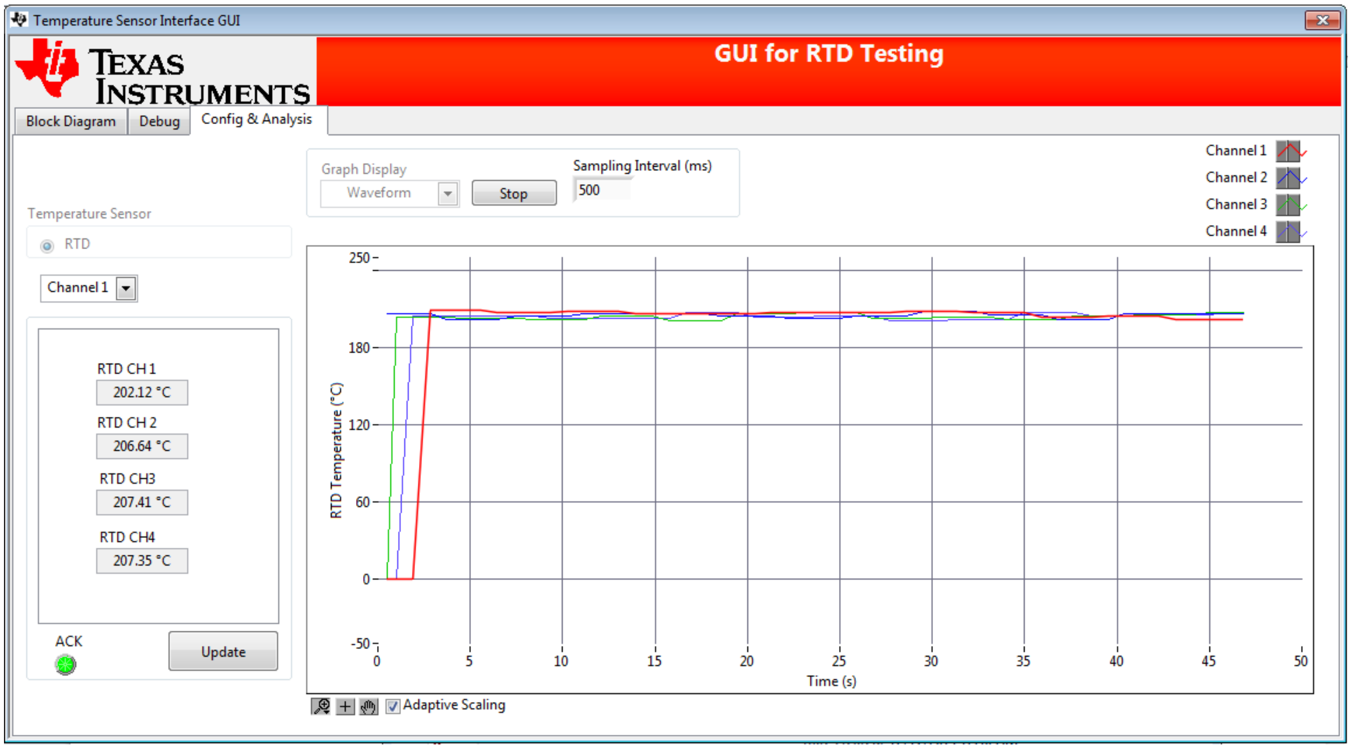
The GUI interface uses USB bulk data transfer to communicate with Tiva Launchpad.

Automatic update

Periodically, it switches between the channels and continues to display the readings as a graph.

Manual update

Instead of periodically switching the channels, it is possible to read the data by as and when it is required, using GUI. The user sets the RTD channel and reads the corresponding ADC data by selecting update button. If the user intends to read data from all 4 channels, he shall select all and then select update.



Sampling Interval (ms): 500

Graph Display: Waveform

Temperature Sensor: RTD

Channel 1

RTD CH 1	202.12 °C
RTD CH 2	206.64 °C
RTD CH3	207.41 °C
RTD CH4	207.35 °C

About Author

Vivek Gopalakrishnan is a Firmware Architect at Texas Instruments India where he is responsible for developing reference design solutions for the Smart Grid within Industrial Systems. Vivek brings to his role his experience in firmware architecture design and development. Vivek earned his Master's degree in Sensor Systems Technology from VIT University, India. He can be reached at vivek.g@ti.com