I2S/I2C Driver Sample Code Reference Guide

V1.00.001

Publication Release Date: Sep. 2011

Support Chips:

ISD9160

Support Platforms:

NuvotonPlatform_Keil

The information in this document is subject to change without notice.

The Nuvoton Technology Corp. shall not be liable for technical or editorial errors or omissions contained herein; nor for incidental or consequential damages resulting from the furnishing, performance, or use of this material.

Table of Contents

1. Introduction 4
   1.1 Feature 4
      1.1.1 I2S 4
      1.1.2 I2C 4
2. Block Diagram 5
3. Calling Sequence 6
4. Code Section –Smpl_DrvI2S.c 7
5. Execution Environment Setup and Result 18
6. Revision History 19

1. Introduction

This sample code will demo I2S/I2C IP on ISD9160 chip.

1.1 Feature

1.1.1 I2S

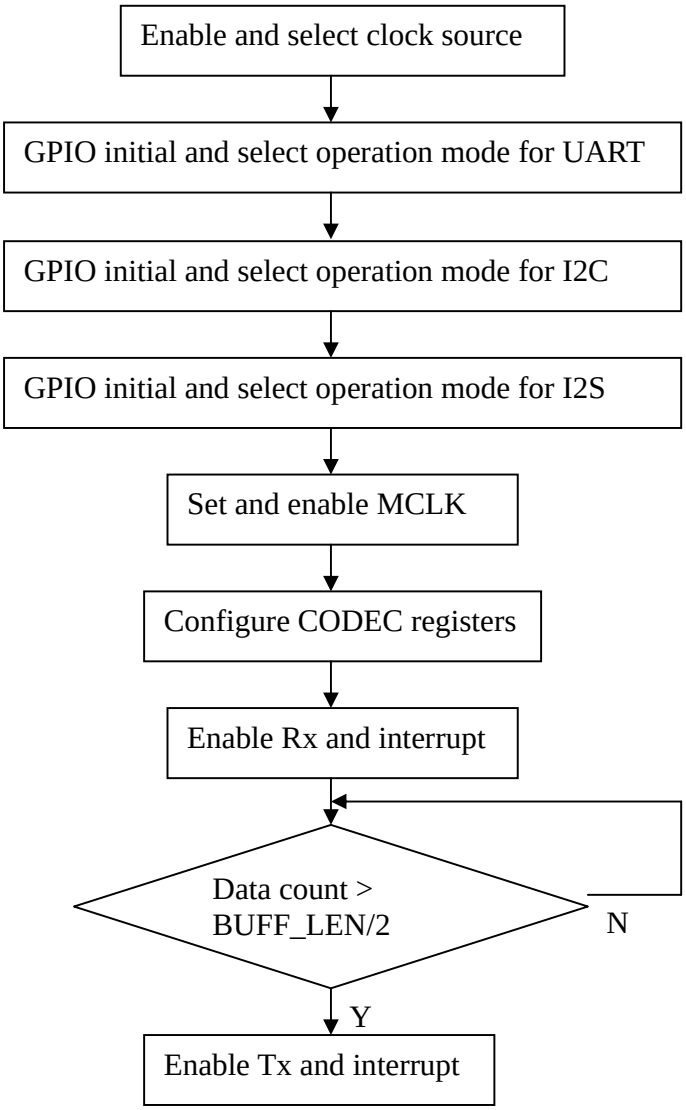
- I2S can operate as either master or slave
- Master clock generation for slave device synchronization.
- Capable of handling 8, 16, 24 and 32 bit word sizes.
- Mono and stereo audio data supported.
- I2S and MSB justified data format supported.
- 8 word FIFO data buffers for transmit and receive.
- Generates interrupt requests when buffer levels crosses programmable boundary.
- Two DMA requests, one for transmit and one for receive.

1.1.2 I2C

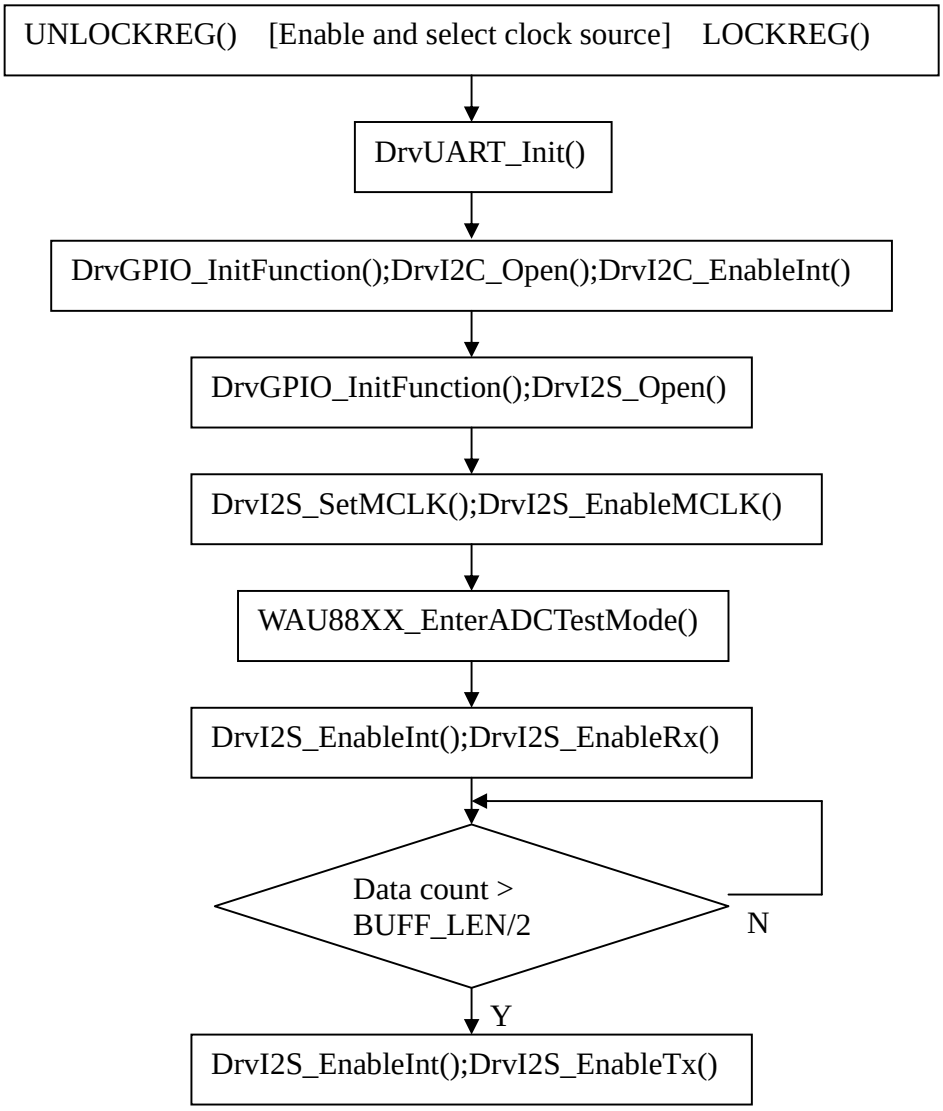
The I2C bus uses two wires (SDA and SCL) to transfer information between devices connected to the bus. The main features of the bus are:

- Master/Slave up to 1Mbit/s
- Bidirectional data transfer between masters and slaves
- Multi-master bus (no central master)
- Arbitration between simultaneously transmitting masters without corruption of serial data on the bus
- Serial clock synchronization allows devices with different bit rates to communicate via one serial bus
- Serial clock synchronization can be used as a handshake mechanism to suspend and resume serial transfer
- Built-in a 14-bit time-out counter will request the I2C interrupt if the I2C bus hangs up and timer-out counter overflows.
- External pull-up are needed for high output
- Programmable clocks allow versatile rate control
- Supports 7-bit addressing mode
- I2C-bus controllers support multiple address recognition (Four slave address with mask option)

2. Block Diagram



3. Calling Sequence



4. Code Section –Smpl_DrvI2S.c

```
#define __RECPLAY_DEMO__
/*---------------------------------------------------------------------------*/
/* (C) Copyright Information Storage Devices, a Nuvoton Company              */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* External Function Prototypes                                              */
/*---------------------------------------------------------------------------*/
/* Standard ANSI C header files                                              */
/*---------------------------------------------------------------------------*/
#include <stdio.h>
#include "isd9xx.h"
//#include "SemiHost.h"
//#include "CoOS.h"
//#include "defs.h"
/*---------------------------------------------------------------------------*/
/* Global Data Declarations                                                  */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/* Home header file                                                          */
/*---------------------------------------------------------------------------*/
//#include "RecPlay_demo.h"
/*---------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------*/
/* External Declarations                                                     */
/*---------------------------------------------------------------------------*/
//#include "spi_cmd.h"
/*---------------------------------------------------------------------------*/
/* Header files for other modules                                            */
/*---------------------------------------------------------------------------*/
#include "DrvPDMA.h"
#include "DrvUART.h"
#include "DrvSYS.h"
#include "DrvGPIO.h"
#include "DrvSPI.h"
#include "DrvI2S.h"
#include "DrvOSC.h"
#include "DrvI2C.h"

//#include "i2s.h"
//#include "MemManage.h"
//#include "CompEngine.h"
```

```
//#include "../SpiFlash/c2082.h" /* Header file with global prototypes */
//#include "../SpiFlash/Serialize.h" /* Header file with SPI master abstract prototypes */
//#include "dataflash.h"
/*---------------------------------------------------------------------------*/
/* Functions Details:                                                        */
/*---------------------------------------------------------------------------*/
/* Executable functions                                                      */
/*---------------------------------------------------------------------------*/
#define BUFF_LEN        64

uint32_t PcmBuff[BUFF_LEN] = {0};
uint32_t u32BuffPos = 0;
uint32_t u32startFlag;
S_DRVI2S_DATA_T st;
uint8_t u8Divider;
int MclkFreq;

extern uint32_t SystemFrequency;

#define outpw(port,value)      *((volatile unsigned int *)(port))=value
#define GAIN_UPDATE 0x100

extern uint32_t g_timer0Ticks;
extern uint32_t g_timer1Ticks;
extern uint32_t g_timer2Ticks;
extern uint32_t g_timer3Ticks;

extern void TimerInit(void);
extern void PwmInit(void);
extern void I2SInit(void);
extern void UART_INT_HANDLE(uint32_t u32IntStatus);
//extern OS_FlagID UARTRxFlag;
//extern OS_FlagID UARTTxFlag;

uint32_t isr_cnt=0;
uint32_t srv_cnt=0;

#define RXBUFSIZE 64
volatile uint8_t comRbuf[RXBUFSIZE];
volatile uint16_t comRbytes = 0;          /* Available receiving bytes */
volatile uint16_t comRhead    = 0;
volatile uint16_t comRtail      = 0;
volatile int32_t g_bWait        = TRUE;

extern uint32_t GetUartCLk(void);
extern      WAU88XX_EnterADCTestMode(void);
/*---------------------------------------------------------------------------------------------------------*/
```

```
/* Define functions prototype
*/
/*---------------------------------------------------------------------------------------------------------*/
/*---------------------------------------------------------------------------------------------------------*/
/*    I2S Tx Threshold Level Callback Function when Tx FIFO is less than Tx FIFO
Threshold Level                    */
/*---------------------------------------------------------------------------------------------------------*/
void Tx_thresholdCallbackfn(uint32_t status)
{
      uint32_t u32Len, i;
      uint32_t * pBuff;
      pBuff = &PcmBuff[0];

      /* Read Tx FIFO free size */
      u32Len = 8 - _DRVI2S_READ_TX_FIFO_LEVEL();

      if (u32BuffPos >= 8)
      {
            for   (i = 0; i < u32Len; i++)
            {
                  _DRVI2S_WRITE_TX_FIFO(pBuff[i]);
            }

            for (i = 0; i < BUFF_LEN - u32Len; i++)
            {
                  pBuff[i] = pBuff[i + u32Len];
            }

            u32BuffPos -= u32Len;
      }
      else
      {
            for   (i = 0; i < u32Len; i++)
            {
                  _DRVI2S_WRITE_TX_FIFO(0x00);
            }
      }
}

/*---------------------------------------------------------------------------------------------------------*/
/*    I2S Rx Threshold Level Callback Function when Rx FIFO is more than Rx FIFO
Threshold Level                    */
/*---------------------------------------------------------------------------------------------------------*/
void Rx_thresholdCallbackfn(uint32_t status)
{
      uint32_t u32Len, i;
```

```
    uint32_t *pBuff;
    if (u32BuffPos < (BUFF_LEN-8))
    {
        pBuff = &PcmBuff[u32BuffPos];

        /* Read Rx FIFO Level */
        u32Len = _DRVI2S_READ_RX_FIFO_LEVEL();

        for ( i = 0; i < u32Len; i++ )
        {
            pBuff[i] = _DRVI2S_READ_RX_FIFO();
        }

        u32BuffPos += u32Len;

        if (u32BuffPos >= BUFF_LEN)
        {
            u32BuffPos =  0;
        }
    }
}

/*---------------------------------------------------------------------------
   MAIN function
   *---------------------------------------------------------------------------*/
int main (void)
{
    S_DRVI2S_DATA_T st;

    /* Step 1. Enable and select clock source*/
    UNLOCKREG();
    SYSCLK->PWRCON.OSC49M_EN = 1;
    SYSCLK->PWRCON.OSC10K_EN = 1;
    SYSCLK->PWRCON.XTL32K_EN = 1;
    SYSCLK->CLKSEL0.STCLK_S = 3; /* Use internal HCLK */
    SYSCLK->CLKSEL0.HCLK_S = 0; /* Select HCLK source as 48MHz */
    SYSCLK->CLKDIV.HCLK_N    = 0;   /* Select no division                  */
    SYSCLK->CLKSEL0.OSCFSel = 0;    /* 1= 32MHz, 0=48MHz */
    SYSCLK->CLKSEL2.I2S_S        = 2;    // HCLK
    LOCKREG();

    /* Step 2. GPIO initial and select operation mode for UART*/
    ///////
    //UART
    //////
    DrvUART_Init(115200);    //Set UART I/O and UART setting

    printf("+----------------------------------------------------------+\n");
```

```
printf("|                    Development Board Demo Program                    |\n");
printf("|                                                                      |\n");
printf("+----------------------------------------------------------------------+\n");

/* Step 3. GPIO initial and select operation mode for I2C*/
//////
// I2C
//////
DrvGPIO_InitFunction(FUNC_I2C0); //Set I2C I/O
DrvI2C_Open(I2C_PORT0, (DrvSYS_GetHCLK() * 1000), 48000);    //clock = 48Kbps
DrvI2C_EnableInt(I2C_PORT0); //Enable I2C0 interrupt and set corresponding NVIC bit

/* Step 4. GPIO initial and select operation mode for I2S*/
///////
// I2S
//////
DrvGPIO_InitFunction(FUNC_I2S0);    //Set I2S I/O
DrvGPIO_InitFunction(FUNC_MCLK1); //Set MCLK I/O

st.u32SampleRate     = 16000;
st.u8WordWidth             = DRVI2S_DATABIT_16;
st.u8AudioFormat     = DRVI2S_STEREO;
st.u8DataFormat      = DRVI2S_FORMAT_I2S;
st.u8Mode                  = DRVI2S_MODE_SLAVE;
st.u8RxFIFOThreshold = DRVI2S_FIFO_LEVEL_WORD_0;
st.u8TxFIFOThreshold = DRVI2S_FIFO_LEVEL_WORD_8-1;
DrvI2S_Open(&st);

/* Step 5. Set and enable MCLK*/
DrvI2S_SetMCLK(12000000);    //MCLK = 12MHz
DrvI2S_EnableMCLK(1);          //enable MCLK

/* Step 6. Configure CODEC registers*/
WAU88XX_EnterADCTestMode();

/* Step 7. Enable Rx and interrupt*/
//Enable Rx threshold level interrupt and install its callback function
DrvI2S_EnableInt(I2S_RX_FIFO_THRESHOLD, Rx_thresholdCallbackfn);
u32startFlag = 1;

// Enable I2S Rx function to receive data
DrvI2S_EnableRx(TRUE);

while(1)
{
      if (u32startFlag)
      {
```

```
                /* Step 8. Enable Tx and interrupt*/
                // Enable I2S Tx function to send data when data in the buffer is more than
half of buffer size
                if (u32BuffPos >= BUFF_LEN/2)
                {
                    DrvI2S_EnableInt(I2S_TX_FIFO_THRESHOLD,
Tx_thresholdCallbackfn);
                    DrvI2S_EnableTx(TRUE);
                    u32startFlag = 0;
                }
            }
        }

}
```

WAU8822Setup.c

```
#include <stdio.h>
#include "Driver/DrvI2C.h"


//=============================================================
// I2C transaction to set up WAU8822
//=============================================================

uint8_t Device_Addr0;
uint8_t Tx_Data0[3];
uint8_t Rx_Data_High;
uint8_t Rx_Data_Low;
uint8_t DataLen0;
volatile uint8_t EndFlag0 = 0;
typedef enum { kI2CWritingWAU88XX_A, kI2CReadingWAU88XX_A, kI2CIdle}
I2CRWMode_t;
I2CRWMode_t I2CRWMode = kI2CIdle;
#define GAIN_UPDATE 0x100

void I2C0_Callback_Tx(uint32_t status)
{
    if (status == 0x08)                         /* START has been transmitted */
    {
        I2C0->DATA = 0;
        I2C0->DATA = (Device_Addr0<<1);
        DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
    }
    else if (status == 0x18)   /* SLA+W has been transmitted and ACK has been received */
    {
        I2C0->DATA = Tx_Data0[DataLen0++];
        DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
    }
    else if (status == 0x20)   /* SLA+W has been transmitted and NACK has been received */
    {
        DrvI2C_Ctrl(I2C_PORT0, 1, 1, 1, 0);
    }
    else if (status == 0x28)   /* DATA has been transmitted and ACK has been received */
    {
        if (DataLen0 != 2)
        {
            DrvI2C_WriteData(I2C_PORT0, Tx_Data0[DataLen0++]);
            DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
```

```
            }
            else
            {
                DrvI2C_Ctrl(I2C_PORT0, 0, 1, 1, 0);
                EndFlag0 = 1;
            }
        }
        else
        {
            printf("Status 0x%x is NOT processed\n", status);
        }
}
void I2C0_Callback_Rx(uint32_t status)
{
        if (status == 0x08)                 /* START has been transmitted and prepare SLA+W */
        {
            DrvI2C_WriteData(I2C_PORT0, Device_Addr0<<1);
            DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
        }
        else if (status == 0x18)   /* SLA+W has been transmitted and ACK has been received */
        {
            DrvI2C_WriteData(I2C_PORT0, Tx_Data0[DataLen0++]);
            DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
        }
        else if (status == 0x20)   /* SLA+W has been transmitted and NACK has been received */
        {
            DrvI2C_Ctrl(I2C_PORT0, 1, 1, 1, 0);
        }
        else if (status == 0x28)   /* DATA has been transmitted and ACK has been received */
        {
            if (DataLen0 != 1)
            {
                DrvI2C_WriteData(I2C_PORT0, Tx_Data0[DataLen0++]);
                DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
            }
            else
            {
                DrvI2C_Ctrl(I2C_PORT0, 1, 0, 1, 0);                 //repeat start
            }
        }
        else if (status == 0x10)/* Repeat START has been transmitted and prepare SLA+R */
        {
            DrvI2C_WriteData(I2C_PORT0, Device_Addr0<<1 | 0x01);
            DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
        }
        else if (status == 0x40)   /* SLA+R has been transmitted and ACK has been received */
        {
            DrvI2C  Ctrl(I2C  PORT0, 0, 0, 1, 1);
```

```
        }
        else if (status == 0x50)   /* DATA has been received and ACK has been returned */
        {
                Rx_Data_High = DrvI2C_ReadData(I2C_PORT0);
                DrvI2C_Ctrl(I2C_PORT0, 0, 0, 1, 0);
        }
        else if (status == 0x58)   /* DATA has been received and NACK has been returned */
        {
                Rx_Data_Low = DrvI2C_ReadData(I2C_PORT0);
                DrvI2C_Ctrl(I2C_PORT0, 0, 1, 1, 0);
                EndFlag0 = 1;
        }
        else
        {
                printf("Status 0x%x is NOT processed\n", status);
        }
}
void Write_WAU88XX_A (uint8_t addr, uint32_t data)
{
        if (I2CRWMode != kI2CWritingWAU88XX_A)
        {
                I2CRWMode = kI2CWritingWAU88XX_A;
                DrvI2C_UninstallCallBack(I2C_PORT0, I2CFUNC);
                DrvI2C_InstallCallback(I2C_PORT0, I2CFUNC, I2C0_Callback_Tx);
        }
        Device_Addr0 = 0x1a;
        Tx_Data0[0] = (addr<<1) | ((data>>8) & 0x1);
        Tx_Data0[1] = data & 0xff;
        DataLen0 = 0;
        EndFlag0 = 0;
        DrvI2C_Ctrl(I2C_PORT0, 1, 0, 0, 0);
        while (EndFlag0 == 0);
        EndFlag0 = 0;
}
uint32_t Read_WAU88XX_A (uint8_t addr)
{
        if (I2CRWMode != kI2CReadingWAU88XX_A)
        {
                I2CRWMode = kI2CReadingWAU88XX_A;
                DrvI2C_UninstallCallBack(I2C_PORT0, I2CFUNC);
                DrvI2C_InstallCallback(I2C_PORT0, I2CFUNC, I2C0_Callback_Rx);
        }
        Device_Addr0 = 0x1a;
        Tx_Data0[0] = (addr<<1);
        DataLen0 = 0;
        EndFlag0 = 0;
        DrvI2C_Ctrl(I2C_PORT0, 1, 0, 0, 0);
        while (EndFlag0 == 0);
```

```
    EndFlag0 = 0;
    return (Rx_Data_High <<8 | Rx_Data_Low);
}
void Delay(int count)
{
    volatile uint32_t i;
    for (i = 0; i < count ; i++);
}
void WriteVerify_WAU88XX_A (uint8_t addr, uint32_t data)
{
    uint32_t retVal;

    Write_WAU88XX_A(addr, data);
    retVal = Read_WAU88XX_A ( addr );
    if(retVal != data)
        printf("I2C - Addr %x Expect %x got %x\n",addr,data,retVal);
}

void    WAU88XX_EnterADCTestMode()
{
    //NUC140    setting
    WriteVerify_WAU88XX_A(0x00,    0x000);     /* Reset all registers */
    Delay(0x200);
    WriteVerify_WAU88XX_A(0x01,    0x02F);
    WriteVerify_WAU88XX_A(0x02,    0x1B3);     /* Enable L/R Headphone, ADC Mix/Boost, ADC */
    WriteVerify_WAU88XX_A(0x03,    0x00F);     /* Enable L/R main mixer, DAC */

    WriteVerify_WAU88XX_A(0x04,    0x010);     /* 16-bit word length, I2S format, Stereo */
    WriteVerify_WAU88XX_A(0x05,    0x000);   /* Companding control and loop back mode (all disable) */
    WriteVerify_WAU88XX_A(0x06,    0x1AD);     /* Divide by 6, 16K */
    WriteVerify_WAU88XX_A(0x07,    0x006);     /* 16K for internal filter cofficients */
    WriteVerify_WAU88XX_A(0x0a, 0x008); /* DAC softmute is disabled, DAC oversampling rate is 128x */
    WriteVerify_WAU88XX_A(0x0e, 0x108); /* ADC HP filter is disabled, ADC oversampling rate is 128x */
    WriteVerify_WAU88XX_A(0x0f, 0x1EF);     /* ADC left digital volume control */
    WriteVerify_WAU88XX_A(0x10, 0x1EF);     /* ADC right digital volume control */

    WriteVerify_WAU88XX_A(0x2c, 0x000);     /* LLIN/RLIN is not connected to PGA */
    WriteVerify_WAU88XX_A(0x2f, 0x050);     /* LLIN connected, and its Gain value */
    WriteVerify_WAU88XX_A(0x30, 0x050);     /* RLIN connected, and its Gain value */
    WriteVerify_WAU88XX_A(0x32, 0x001);     /* Left DAC connected to LMIX */
    WriteVerify_WAU88XX_A(0x33, 0x001);     /* Right DAC connected to RMIX */
}
void ADCGainIs(uint8_t gain)
{
    Write_WAU88XX_A(0x2d, GAIN_UPDATE|gain);
    printf("ADC gain =0x%x\n", Read_WAU88XX_A(0x2d));
}
```

```
uint32_t ADCGain()
{
	return Read_WAU88XX_A(0x2d);
}
```

5. Execution Environment Setup and Result

- Prepare a ISD9160 board.
- Compile the sample code.
- Input audio source to ADC of WAU8822 and connect DAC of WAU8822 to speaker.
- Hear correct audio from speaker.

6. Revision History

Version	Date	Description
V1.00.01	Sep. 2011	Created