实验四 语音信号采集及处理综合实验

一、实验目的

1. 掌握基于 DSP 的语音信号采集及回放方法。

2. 掌握数字回声产生原理、编程及参数控制方法。

3. 掌握语音信号的频谱分析方法。

二、实验内容

1. 基本内容：完成语音信号的实时采集与回放（8 分）

2. 综合拓展内容一：将键盘、液晶与基本实验内容进行综合，完成数字回声实验，要求能够通过按键控制回声的延迟时间及回声叠加幅度。（5 分）

3. 综合拓展内容二：选取一段采集到的音频信号，绘制波形并进行播放（1 分）；将上述音频信号进行反序，绘制波形并进行播放（1 分）；计算并绘制原始的及反序的音频信号的数字幅度谱（1 分）；计算并绘制原始的及反序的音频信号的数字相位谱（1 分）；比较原始的及反序的音频信号的数字频谱，并对实验现象进行分析（1 分）。

4. 自主设计完成一个综合实验（2）

三、实验设备

1. 数字计算机

2. DSP 实验箱

3. 手机（自带）

4. 测试导线

四、预习要求

1. 熟悉 TLV320AIC23 的芯片资料。

2. 熟悉 ICETEK-F28335 实验箱的使用方法

3. 复习在 DSP 实验箱上使用按键及液晶显示的方法。

4. 复习 FFT 算法的使用方法。

五、实验原理

1. TLV320AIC23 芯片控制方法

1）语音信号的输入：AIC23 通过其中的 AD 转换采集输入的语音信号，每采集完

一个信号后，将数据发送到 DSP 的 McBSP 接口上，DSP 可以读取到语音数据，每个数据为 16 位无符号整数，左右声道各有一个数值。

2）语音信号的输出：DSP 可以将语音数据通过 McBSP 接口发送给 AIC23，AIC23 的 DA 器件将他们变成模拟信号输出。

DSP 芯片与 TLV320AIC23 的连接关系如图 2.4.1 所示。

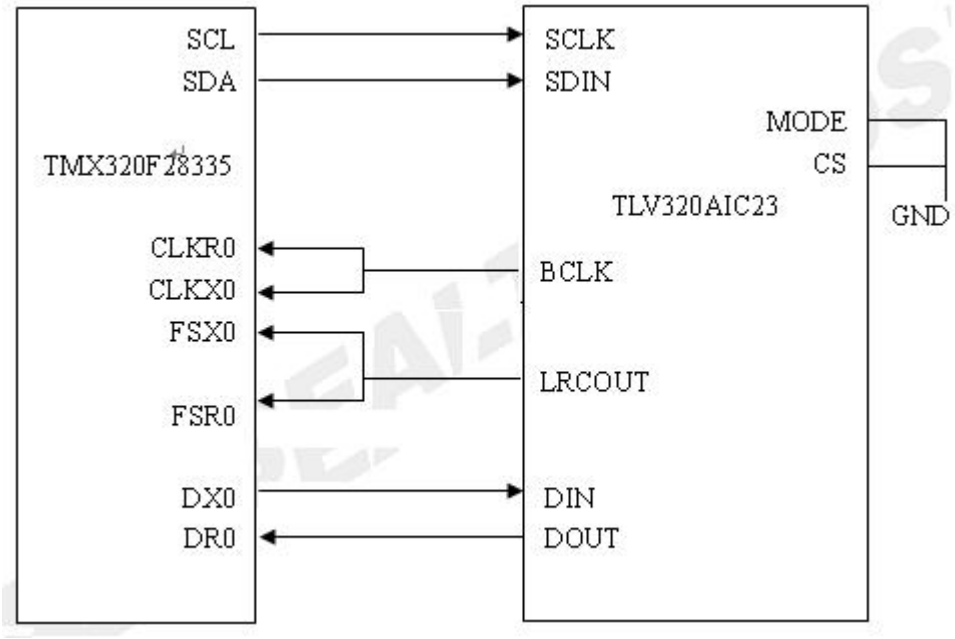

图 2.4.1　DSP 与语音芯片的连接示意图

2. 数字回声原理：在实际生活中，当声源遇到物体时，会发生反射，反射的声波和声源声波一起传输，听者会发现反射声波部分比声源声波慢一些，类似人们面对山体高声呼喊后可以在过一会儿听到回声的现象。声音遇到较远的物体产生的反射会比遇到较近的物体的反射波晚些到达声源位置，所以回声和原声的延迟随反射物体的距离大小改变。同时，反射声音的物体对声波的反射能力，决定了听到的回声的强弱和质量。另外，生活中的回声的成分比较复杂，有反射、漫反射、折射，还有回声的多次反、折射效果。

当已知一个数字音源后，可以利用计算机的处理能力，用数字的方式通过计算模拟回声效应。简单地讲，可以在原声音流中叠加延迟一段时间后的声流，实现回声效果。当然通过复杂运算，可以计算各种效应的混响效果。如此产生的回声，我们称之为数字回声。可以通过理想的低通滤波器无失真地恢复成原连续信号。

六、实验步骤

1. 准备硬件及建立工程（注意：请先将 CCS 打开时默认路径文件夹里的所有文件

删除，并且使用局域网共享的最新版 function）。

2. 将个人手机音频输出口与 DSP 实验箱音频输入口 MIC IN 相连（插座 J9 最下方红色插孔），实验箱音频输出口 HEAD PHONE 与（插座 J11 第二个绿色插孔）与信号源上的音响输入相连。连接图如下：

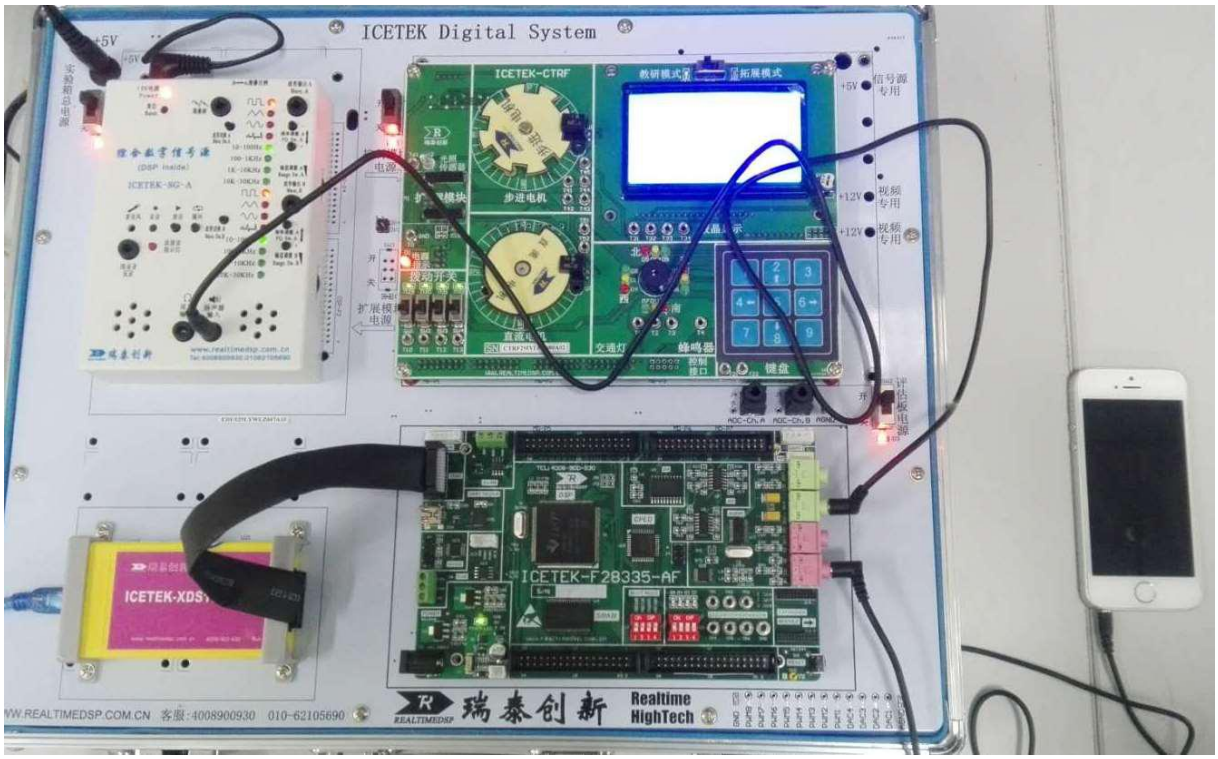

图 2.4.2　实验系统连接示意图

3. 根据实验指导书提供的参考代码完成基本内容。

4. 完成综合拓展内容一。

5. 完成综合拓展内容二。

6. 完成自主设计综合实验。

七、参考程序

```
   #include "DSP2833x_Device.h"     // DSP2833x Headerfile Include File
#include "DSP2833x_Examples.h"   // DSP2833x Examples Include File

void   I2CA_Init(void);
Uint16 I2CA_WriteData(struct I2CMSG *msg);
Uint16 I2CA_ReadData(struct I2CMSG *msg);
interrupt void i2c_int1a_isr(void);

Uint16 AIC23Write(int Address,int Data);
void Delay(int time);
```

```
void pass(void);
void fail(void);
void emif_init(void);
void AIC23Init(void);

#define I2C_SLAVE_ADDR        0x1A
#define I2C_NUMBYTES          20
#define I2C_EEPROM_HIGH_ADDR  0x00
#define I2C_EEPROM_LOW_ADDR   0x17
unsigned int bEcho=0;
unsigned int uDelay,uEffect;
struct I2CMSG I2cMsgOut1={I2C_MSGSTAT_SEND_WITHSTOP,
                          I2C_SLAVE_ADDR,
                          I2C_NUMBYTES,
                          I2C_EEPROM_HIGH_ADDR,
                          I2C_EEPROM_LOW_ADDR,
                          0x02,                  // Msg Byte 1
                          0x17,                  // Msg Byte 2
                          0x05,                  // Msg Byte 3
                          0x79,                  // Msg Byte 4
                          0x07,                  // Msg Byte 5
                          0x79,                  // Msg Byte 6
                          0x08,                  // Msg Byte 7
                          0x15,                  // Msg Byte 8
                          0x0A,                  // Msg Byte 9
                          0x00,                  // Msg Byte 10
                          0x0C,                  // Msg Byte 11
                          0x02,                  // Msg Byte 12
                          0x0E,                  // Msg Byte 13
                          0x43,                // Msg Byte 14
                          0x10,                // Msg Byte 15
                          0x82,                // Msg Byte 16
                          0x12,                // Msg Byte 17
                          0x01,                // Msg Byte 18
                          0x1E,                // Msg Byte 19
                          0x00};                  // Msg Byte 20


struct I2CMSG I2cMsgIn1={ I2C_MSGSTAT_SEND_NOSTOP,
                          I2C_SLAVE_ADDR,
                          I2C_NUMBYTES,
                          I2C_EEPROM_HIGH_ADDR,
                          I2C_EEPROM_LOW_ADDR};
```

```
struct I2CMSG *CurrentMsgPtr;             // Used in interrupts
Uint16 PassCount;
Uint16 FailCount;
#define AUDIOEN (*(unsigned short int *)0x180002)
#define SRAM_Base_Adress  0x100000
void main(void)
{
   long int nAudioData,nAudioData1;
   float AudioData,AudioData_delay;
   int uWork;
   unsigned long int i;
   CurrentMsgPtr = &I2cMsgOut1;
   InitSysCtrl();
   InitXintf16Gpio();
   InitMcbspaGpio();
   InitI2CGpio();
   emif_init();
   AUDIOEN = 0;
// Disable CPU interrupts
   DINT;

   InitPieCtrl();

// Disable CPU interrupts and clear all CPU interrupt flags:
   IER = 0x0000;
   IFR = 0x0000;

   InitPieVectTable();

   EALLOW;   // This is needed to write to EALLOW protected registers
   PieVectTable.I2CINT1A = &i2c_int1a_isr;

   EDIS;   // This is needed to disable write to EALLOW protected
registers

   I2CA_Init();

// Enable I2C interrupt 1 in the PIE: Group 8 interrupt 1
   PieCtrlRegs.PIEIER8.bit.INTx1 = 1;

// Enable CPU INT8 which is connected to PIE group 8
   IER |= M_INT8;
   EINT;
```

```
   InitMcbspa();          // Initalize the Mcbsp-A
   AIC23Init();
   nAudioData=nAudioData1=0;

   for(i=0;i<=0x48000;i++)
   {
    *(int *)(SRAM_Base_Adress+i)=0;
   }

   uDelay=128;   //回声延迟时间
   uEffect=256;   //回声强度

    for(;;)
   {
       while(!McbspaRegs.SPCR1.bit.RRDY);
       uWork=McbspaRegs.DRR2.all;  // 读取左右声道的数据
       uWork=McbspaRegs.DRR1.all;  // 因为耳机输入左右声道相同，所以读两
次即可


       *(int *)(SRAM_Base_Adress+nAudioData)=uWork;
/*
        //此处代码作为回声实现参考
       if ( bEcho )           // 需要制作数字回声否?
          {
             uEffect%=1024;         // 保证输入在 0-1023 之间
             uDelay%=1024;          // 保证输入在 0-1023 之间

             //音频缓冲区:
[1]..........................................................
[0x48000]
             //                                          |
|
             //                                     |
|
             //                                AudioData_delay(延迟)
|
             //
AudioData(当前音频数据)
             //
             //     回声=  AudioData+AudioData_delay(衰减后)

             nAudioData1=nAudioData-uDelay*100;          // 根据 uDelay
```

```
参数计算，调整指针
            if ( nAudioData1<0 )
               nAudioData1+=0x48000;

            AudioData_delay=*(int *)(SRAM_Base_Adress+nAudioData1);
      // 取得之前保存的音频数据 1
            AudioData=*(int *)(SRAM_Base_Adress+nAudioData);
   // 取得当前音频数据

            AudioData=AudioData_delay*uEffect/512.0+AudioData;
   // 之前音频数据衰减一半后，与当前声音混响
            uWork=AudioData;
         }
      */
      while(!McbspaRegs.SPCR2.bit.XRDY);
      McbspaRegs.DXR2.all=uWork;
      McbspaRegs.DXR1.all=uWork;

      nAudioData++;
      if(nAudioData>0x48000)
         nAudioData=0;
   }
}   // end of main

void emif_init(void)
{
    #define EMIFA_1       0x00006F88
    #define EMIFA_2       0x00006F96
    #define EMIFA_3       0x00006FA6
    #define EMIFA_4       0x00006FA8
   #define emifa_5       0x00007020

    /* EMIFA */
    *(long *)EMIFA_1      = 0xff000000;
    *(long *)EMIFA_2      = 0xfffffff0;
    *(long *)EMIFA_3      = 0xFFFFFFFF;
    *(long *)EMIFA_4      = 0x0000ffff;
   *(long *)emifa_5      = 0x00001000;
}


void I2CA_Init(void)
{
```

```
   // Initialize I2C
   I2caRegs.I2CSAR = 0x001A;     // Slave address - EEPROM control code

   #if (CPU_FRQ_150MHZ)             // Default - For 150MHz SYSCLKOUT
        I2caRegs.I2CPSC.all = 14;   // Prescaler - need 7-12 Mhz on module
clk (150/15 = 10MHz)
   #endif
   #if (CPU_FRQ_100MHZ)             // For 100 MHz SYSCLKOUT
     I2caRegs.I2CPSC.all = 9;    // Prescaler - need 7-12 Mhz on module
clk (100/10 = 10MHz)
   #endif

   I2caRegs.I2CCLKL = 100;          // NOTE: must be non zero
   I2caRegs.I2CCLKH = 100;          // NOTE: must be non zero
   I2caRegs.I2CIER.all = 0x24;      // Enable SCD & ARDY interrupts

//   I2caRegs.I2CMDR.all = 0x0020; // Take I2C out of reset
   I2caRegs.I2CMDR.all = 0x0420;// Take I2C out of reset     //zq
                              // Stop I2C when suspended

   I2caRegs.I2CFFTX.all = 0x6000;   // Enable FIFO mode and TXFIFO
   I2caRegs.I2CFFRX.all = 0x2040;   // Enable RXFIFO, clear RXFFINT,

   return;
}

Uint16 AIC23Write(int Address,int Data)
{


   if (I2caRegs.I2CMDR.bit.STP == 1)
   {
      return I2C_STP_NOT_READY_ERROR;
   }

   // Setup slave address
   I2caRegs.I2CSAR = 0x1A;

   // Check if bus busy
   if (I2caRegs.I2CSTR.bit.BB == 1)
   {
      return I2C_BUS_BUSY_ERROR;
   }
```

```
   // Setup number of bytes to send
   // MsgBuffer + Address
   I2caRegs.I2CCNT = 2;
   /*for (i=0; i<; i++)

   {
      I2caRegs.I2CDXR = *(msg->MsgBuffer+i);
   }*/
   I2caRegs.I2CDXR = Address;
   I2caRegs.I2CDXR = Data;
   // Send start as master transmitter
   I2caRegs.I2CMDR.all = 0x6E20;

   /*for(i=0;i<1024;i++)
    for(j=0;j<1000;j++)
     k++;*/
   return I2C_SUCCESS;

}

Uint16 I2CA_WriteData(struct I2CMSG *msg)
{
   Uint16 i;

   // Wait until the STP bit is cleared from any previous master
communication.
   // Clearing of this bit by the module is delayed until after the SCD
bit is
   // set. If this bit is not checked prior to initiating a new message,
the
   // I2C could get confused.
   if (I2caRegs.I2CMDR.bit.STP == 1)
   {
      return I2C_STP_NOT_READY_ERROR;
   }

   // Setup slave address
   I2caRegs.I2CSAR = msg->SlaveAddress;

   // Check if bus busy
   if (I2caRegs.I2CSTR.bit.BB == 1)
   {
      return I2C_BUS_BUSY_ERROR;
```

```
   }

   // Setup number of bytes to send
   // MsgBuffer + Address
   I2caRegs.I2CCNT = msg->NumOfBytes+2;

   // Setup data to send
   I2caRegs.I2CDXR = msg->MemoryHighAddr;
   I2caRegs.I2CDXR = msg->MemoryLowAddr;
// for (i=0; i<msg->NumOfBytes-2; i++)
   for (i=0; i<msg->NumOfBytes; i++)

   {
      I2caRegs.I2CDXR = *(msg->MsgBuffer+i);
   }

   // Send start as master transmitter
   I2caRegs.I2CMDR.all = 0x6E20;

   return I2C_SUCCESS;
}


Uint16 I2CA_ReadData(struct I2CMSG *msg)
{
   // Wait until the STP bit is cleared from any previous master
communication.
   // Clearing of this bit by the module is delayed until after the SCD
bit is
   // set. If this bit is not checked prior to initiating a new message,
the
   // I2C could get confused.
   if (I2caRegs.I2CMDR.bit.STP == 1)
   {
      return I2C_STP_NOT_READY_ERROR;
   }

   I2caRegs.I2CSAR = msg->SlaveAddress;

   if(msg->MsgStatus == I2C_MSGSTAT_SEND_NOSTOP)
   {
      // Check if bus busy
      if (I2caRegs.I2CSTR.bit.BB == 1)
      {
```

```
        return I2C_BUS_BUSY_ERROR;
      }
      I2caRegs.I2CCNT = 2;
      I2caRegs.I2CDXR = msg->MemoryHighAddr;
      I2caRegs.I2CDXR = msg->MemoryLowAddr;
      I2caRegs.I2CMDR.all = 0x2620;       // Send data to setup EEPROM
address
   }
   else if(msg->MsgStatus == I2C_MSGSTAT_RESTART)
   {
      I2caRegs.I2CCNT = msg->NumOfBytes;  // Setup how many bytes to
expect
      I2caRegs.I2CMDR.all = 0x2C20;       // Send restart as master
receiver
   }

   return I2C_SUCCESS;
}

interrupt void i2c_int1a_isr(void)     // I2C-A
{
   Uint16 IntSource, i;

   // Read interrupt source
   IntSource = I2caRegs.I2CISRC.all;

   // Interrupt source = stop condition detected
   if(IntSource == I2C_SCD_ISRC)
   {
      // If completed message was writing data, reset msg to inactive state
      if (CurrentMsgPtr->MsgStatus == I2C_MSGSTAT_WRITE_BUSY)
      {
         CurrentMsgPtr->MsgStatus = I2C_MSGSTAT_INACTIVE;
      }
      else
      {
         // If a message receives a NACK during the address setup portion
of the
         // EEPROM read, the code further below included in the register
access ready
         // interrupt source code will generate a stop condition. After
the stop
         // condition is received (here), set the message status to try
```

```
again.
        // User may want to limit the number of retries before generating
an error.
        if(CurrentMsgPtr->MsgStatus == I2C_MSGSTAT_SEND_NOSTOP_BUSY)
        {
           CurrentMsgPtr->MsgStatus = I2C_MSGSTAT_SEND_NOSTOP;
        }
        // If completed message was reading EEPROM data, reset msg to
inactive state
        // and read data from FIFO.
        else if (CurrentMsgPtr->MsgStatus == I2C_MSGSTAT_READ_BUSY)
        {
           CurrentMsgPtr->MsgStatus = I2C_MSGSTAT_INACTIVE;
           for(i=0; i < I2C_NUMBYTES; i++)
           {
             CurrentMsgPtr->MsgBuffer[i] = I2caRegs.I2CDRR;
           }
        {
        // Check recieved data
        for(i=0; i < I2C_NUMBYTES; i++)
        {
           if(I2cMsgIn1.MsgBuffer[i] == I2cMsgOut1.MsgBuffer[i])
           {
               PassCount++;
           }
           else
           {
               FailCount++;
           }
        }
        if(PassCount == I2C_NUMBYTES)
        {
           pass();
        }
        else
        {
           fail();
        }


     }

   }
     }
```

```
   }  // end of stop condition detected

   // Interrupt source = Register Access Ready
   // This interrupt is used to determine when the EEPROM address setup
portion of the
   // read data communication is complete. Since no stop bit is commanded,
this flag
   // tells us when the message has been sent instead of the SCD flag.
If a NACK is
   // received, clear the NACK bit and command a stop. Otherwise, move
on to the read
   // data portion of the communication.
   else if(IntSource == I2C_ARDY_ISRC)
   {
      if(I2caRegs.I2CSTR.bit.NACK == 1)
      {
         I2caRegs.I2CMDR.bit.STP = 1;
         I2caRegs.I2CSTR.all = I2C_CLR_NACK_BIT;
      }
      else if(CurrentMsgPtr->MsgStatus ==
I2C_MSGSTAT_SEND_NOSTOP_BUSY)
      {
         CurrentMsgPtr->MsgStatus = I2C_MSGSTAT_RESTART;
      }
   }  // end of register access ready

   else
   {
      // Generate some error due to invalid interrupt source
      asm("   ESTOP0");
   }

   // Enable future I2C (PIE Group 8) interrupts
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
}

void AIC23Init(void)
{
    AIC23Write(0x00,0x00);
    Delay(100);
    AIC23Write(0x02,0x00);
    Delay(100);
    AIC23Write(0x04,0x7f);
```

```
    Delay(100);
    AIC23Write(0x06,0x7f);
    Delay(100);
    AIC23Write(0x08,0x14);
    Delay(100);
    AIC23Write(0x0A,0x00);
    Delay(100);
    AIC23Write(0x0C,0x00);
    Delay(100);
    AIC23Write(0x0E,0x43);
    Delay(100);
    AIC23Write(0x10,0x23);
    Delay(100);
    AIC23Write(0x12,0x01);
    Delay(100);     //AIC23Init
}

void Delay(int time)
{
 int i,j,k=0;
 for(i=0;i<time;i++)
  for(j=0;j<1024;j++)
   k++;
}
void pass()
{
    asm("   ESTOP0");
    for(;;);
}

void fail()
{
    asm("   ESTOP0");
    for(;;);
}
```

八、实验报告要求

实验报告要求体现各个程序的流程图。

实验报告格式参照自控原理的实验报告模板，也可自由发挥。

九、参考资料

1. ICETEK-F28335-AF 评估板及教学实验系统实验指导书 V5 版，瑞泰创新。

2. TLV320AIC23 芯片资料。