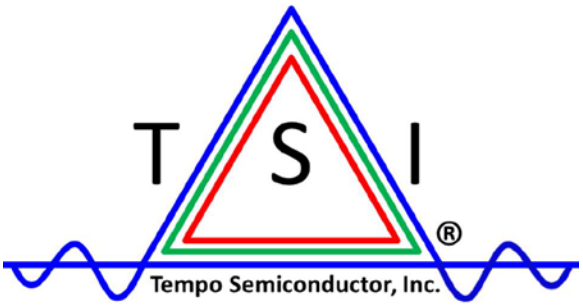
Sample Code for Initializing the ACS422XXX Family of I2S Codecs

Application Note

TEMPO CONFIDENTIAL



1.0 Purpose

The purpose of this application note is to help the system designer understand how to initialize the ACS422XXX I2S/I2C based audio codecs. The code listed in this document is written in the C language and has been proven to work while using a Microchip PIC16F1503, the Hi-Tech C Lite Compiler, and the Microchip MPLAB X IDE software. Any changes to the processor or compiling environment may require code changes that are NOT covered by this document.

Note that the code references some I2C routines for the PIC16F1503 that are given at the end of this document.

This document is NOT intended to provide a complete execution code routine, but merely samples of code that can be used as a complete embedded program.

Additionally, this document does not represent fully optimized code. System designers should alter the code as necessary to allow for optimizations for their specific SoC.

Finally – this example code assumes that the speaker dock is designed to support ANALOG INPUT and either 2 or 4 speakers. If the design uses 2 speakers, then the ACS422XXX will be used and it will employ an internal loopback register allowing for ADC-to-DAC signal routing internally. If 4 speakers are used, the analog signal will first route to the ACS422XXX . The internal loop-back bit will NOT be set. Instead, the outputs from the ADC will simultaneously route to the ACS32201 inputs and the ACS422XXX DAC inputs.

2.0 Initialization of the Codec

The ACS422x67/68 and ACS32201 are pre-configured to support either 12.288MHz or 11.576MHz input clock. Other ACS422XXX parts and/or different input clock frequencies may require additional configuration.

The procedure involves programming the equalizer (EQ) and other enhancements (if desired), setting the gain for the speaker/headphone paths, choosing the input path, setting a loop-back (if desired), setting clocking structures, powering on various portions of the codec, and enabling the volume levels.

Here is a short initialization sequence example that can be used for either part. Note, this assumes that the design will use a single-ended analog input into the IN1 path, and that the analog audio will be re-directed to the speaker output path.

```
void init(void);
void i2c_start(void);
void i2c_restart(void);
void I2CWrite(unsigned int Reg, byte Value);
void i2c_stop(void);
void i2c_wait(void);
byte I2CRead(unsigned int Reg, unsigned char ack);

void main(void)
{
    __delay_ms(50);  //Delay start of code execution to allow power settling
    int vol = 255; // Set up global VOLUME variable
    init();  //Initialization of the I2C Setup
```

```
    SSP1IF = 0; //CLEAR INTERRUPT for I2C Communication

/****** PROGRAM EQ, COMPRESSOR, LIMITER, ENHANCEMENT SETTINGS HERE ******/

    i2c_start();
    I2CWrite(0x04,0x00); //Mute the Left DAC while making changes
    i2c_restart();
    I2CWrite(0x05,0x00); //Mute the Right DAC while making changes
    i2c_restart();
    I2CWrite(0x13, 0x2a); //Set master mode, 48KHZ
    i2c_restart();
    I2CWrite(0x14, 0x03); //Force ADC and DAC to share only the DAC clocks
    i2c_restart();
    I2CWrite(0x02, 0x74); //Set Left Channel Speaker amplifier gain to +3.75dB
    i2c_restart();
    I2CWrite(0x03, 0x74); //Set Right Channel Speaker amplifier gain to +3.75dB
    i2c_restart();
    I2CWrite(0x1A, 0xfc); //turn on ADC power (APPLY ONLY TO ACS422XXX )
    i2c_restart();
    I2CWrite(0x1B, 0x19); //turn on SPK AMP power (If using HP amp on ACS422XXX , use 0x7F)
    i2c_restart();
    I2CWrite(0x16, 0x00); //unmute ADC (APPLY ONLY TO ACS422XXX )
    i2c_restart();
    I2CWrite(0x18, 0x00); //unmute DAC
    i2c_restart();
    I2CWrite(0x69,0x20); //enable analog loopback
    i2c_restart();
    I2CWrite(0x04, VOL); //restore volume to left channel DAC
    i2c_restart();
    I2CWrite(0x05, VOL); //restore volume to right channel DAC
    }
    i2c_stop();
}
```

The code above should provide analog playback from the IN1 path. Note that some register writes may not be necessary, depending on the design or overall system goals (such as speaker amplifier gain, ADC/DAC clock sharing, input selection, etc.). Specifically, registers 02h and 03h for the speaker gain must be set appropriately.

3.0 Programming the EQ, Compressor, Limiter, and Enhancer Settings

Tempo provides a software programming tool known as ASC (Audio System Configurator) that allows for tuning of the EQ bands, Compressor, Limiter, and Enhancer settings. For more details on this tool, please see the relevant app note available on the Tempo Semiconductor website (AN-02 Audio Algorithms).

The ASC tool will provide a C-Code based output for easy integration of code into the design. This code should be inserted into the main code as indicated above. An example of that code is provided below. Please DO NOT use the coefficients for the EQ, compressor, limiter, and enhancements as provided below! Make sure to use coefficients designed and tuned for the target system!

Note: any changes to the EQ settings should be done with the DAC volume.

```
// Program the EQ!!!
// Actual coefficients calculated from current EQ and Enhancer configuration
// 0x0-0x7F = EQ Coefficients
// 0x80-0xAF = Bass, Treble and 3D Coefficients

const long Coeff[176] =
{
0x003fa018, 0xff80bfcf, 0x003fa018, 0x007f3fc5, 0xffc0bf63, 0x003f870d, 0xff80f1e5, 0x003f870d,
0x007f0d36, 0xffc0f102, 0x003fed27, 0xff804f96, 0x003fc749, 0x007fb06a, 0xffc04b8f, 0x003f9154,
0xff8273f9, 0x003e16e9, 0x007d8c07, 0xffc257c3, 0x003e2595, 0xff909eac, 0x00352995, 0x006f6154,
0xffccb0d6, 0x003edcad, 0xff97e5cc, 0x00395852, 0x00681a34, 0xffc7cb01, 0x00000000, 0x00232768,
0x003fa018, 0xff80bfcf, 0x003fa018, 0x007f3fc5, 0xffc0bf63, 0x003f870d, 0xff80f1e5, 0x003f870d,
0x007f0d36, 0xffc0f102, 0x003fed27, 0xff804f96, 0x003fc749, 0x007fb06a, 0xffc04b8f, 0x003f9154,
0xff8273f9, 0x003e16e9, 0x007d8c07, 0xffc257c3, 0x003e2595, 0xff909eac, 0x00352995, 0x006f6154,
0xffccb0d6, 0x003edcad, 0xff97e5cc, 0x00395852, 0x00681a34, 0xffc7cb01, 0x00000000, 0x00232768,
0x003fe66b, 0xff813e5e, 0x003f20d6, 0x007ec1a2, 0xffc0f8be, 0x0074841f, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x003f144b, 0xff82784c, 0x003e7629, 0x007d87b4, 0xffc2758c, 0x00400000,
0xffa27dfe, 0x002bf93a, 0x005d8202, 0xffd406c6, 0x00400000, 0xffad94df, 0x0027e2de, 0x00526b21,
0xffd81d22, 0x00400000, 0xffb93b7f, 0x00241499, 0x0046c481, 0xffdbeb67, 0x00000000, 0x00400000,
0x003fe66b, 0xff813e5e, 0x003f20d6, 0x007ec1a2, 0xffc0f8be, 0x0074841f, 0x00000000, 0x00000000,
0x00000000, 0x00000000, 0x003f144b, 0xff82784c, 0x003e7629, 0x007d87b4, 0xffc2758c, 0x00400000,
0xffa27dfe, 0x002bf93a, 0x005d8202, 0xffd406c6, 0x00400000, 0xffad94df, 0x0027e2de, 0x00526b21,
0xffd81d22, 0x00400000, 0xffb93b7f, 0x00241499, 0x0046c481, 0xffdbeb67, 0x00000000, 0x00400000,
0x0000b2b6, 0x00000000, 0xffff4d49, 0x007e8197, 0xffc1656d, 0x0000b2b6, 0x00000000, 0xffff4d49,
0x007e8197, 0xffc1656d, 0x00000000, 0x00000000, 0x000010ce, 0x0000219d, 0x000010ce, 0x007a65c3,
0xffc55703, 0x003dcdc3, 0xff846478, 0x003dcdc3, 0x007b85d5, 0xffc44ec7, 0x00000000, 0x0027dcfd,
0xffb04605, 0x0027dcfd, 0x00468a27, 0xffe71633, 0x0027dcfd, 0xffb04605, 0x0027dcfd, 0x00468a27,
0xffe71633, 0x00000000, 0x00000000, 0x000e718b, 0x001ce317, 0x000e718b, 0xffe811b5, 0x001e281d,
0x00400000, 0xff847a2b, 0x003bb139, 0x007b85d5, 0xffc44ec7, 0x00000000, 0x00000000, 0x00400000
};

// Mute DAC output while writing EQ coefficients
i2c_start();
I2CWrite(0x04,0x00);
i2c_restart();
I2CWrite(0x05,0x00);
i2c_stop();
byte lowByte;
byte midByte;
byte hiByte;
byte addrByte;
byte temp;

  for(int i=0; i < 176; i++)
  {
```

```
        // Poll to make sure the write logic is available
        do
            {
              temp = I2CRead(0x8a, 0);
        }
        while(temp != 0);

        // 24-bit coefficient is broken up into bytes
        lowByte = (Coeff[i] & 0xff);
        midByte = ((Coeff[i] >> 8) & 0xff);
        hiByte =  ((Coeff[i] >> 16) & 0xff);
        addrByte = (byte)i;

        // Write the address followed by the byte coefficients
        i2c_start();
        I2CWrite(0x40, addrByte);
        i2c_restart();
        I2CWrite(0x3a, lowByte);
        i2c_restart();
        I2CWrite(0x3b, midByte);
        i2c_restart();
        I2CWrite(0x3c, hiByte);
        i2c_stop();
        // Hi byte write initiates write sequence
}

// Program Compressor, program loopback, turn on power, and Restore DAC gain to
// saved values or full scale
  i2c_start();
  I2CWrite(0x26, 0x02); //Set make up gain
  i2c_restart();
  I2CWrite(0x27, 0xe9); // Compressor Threshold
  i2c_restart();
  I2CWrite(0x28, 0x03); // Compressor Ratio
  i2c_restart();
  I2CWrite(0x29, 0x67); // Compressor Attack Time (Low)
  i2c_restart();
  I2CWrite(0x2a, 0xb1); // Compressor Attack Time (High)
  i2c_restart();
  I2CWrite(0x2b, 0x62); // Compressor Release Time (Low)
  i2c_restart();
  I2CWrite(0x2c, 0xfa); // Compressor Release Time (High)
  i2c_restart();
  I2CWrite(0x2d, 0xf1); // Limiter Threshold
  i2c_restart();
  I2CWrite(0x2e, 0xf1); // Limiter Target
  i2c_restart();
  I2CWrite(0x2f, 0xb3); // Limiter Attack Time (Low)
  i2c_restart();
  I2CWrite(0x30, 0xd8); // Limiter Attack Time (High)
  i2c_restart();
  I2CWrite(0x31, 0x36); // Limiter Release Time (Low)
  i2c_restart();
  I2CWrite(0x32, 0xff); // Limiter Release Time (High)
  i2c_restart();
```

```
I2CWrite(0x33, 0x94); // Expander Threshold
i2c_restart();
I2CWrite(0x34, 0x01); // Expander Ratio
i2c_restart();
I2CWrite(0x35, 0xb3); // Expander Attack Time (Low)
i2c_restart();
I2CWrite(0x36, 0xd8); // Expander Attack Time (High)
i2c_restart();
I2CWrite(0x37, 0x9b); // Expander Release Time (Low)
i2c_restart();
I2CWrite(0x38, 0xff); // Expander Release Time (High)
i2c_stop();
```

Once the enhancements are programmed, they must be enabled or disabled as desired. Each enhancement has its own enable/disable bit, allowing for flexibility in the system design.

3.1 Enabling the EQ

The Equalizer function of the ACS422XXX is divided into two separate banks of 6 or 12 total EQ bands. There are 6 bands per channel in each bank. The banks can be configured to be used simultaneously, or each band can be enabled sequentially. Register 20h controls the EQ enables. The register is defined as:

CONFIG1 (20h)

Address	Register	Default
20h	CONFIG1	00h

Field Name	Bits	R/W	Default	Description
EQ2_EN	7	RW	0h	EQ2 Enable: 0 = EQ2 bypassed 1 = EQ1 enabled
EQ2_BE	6:4	RW	0h	EQ2 Band Enable: 0h = Pre-scale only 1h = Pre-scale + EQ band 0 2h = Pre-scale + EQ bands 0 to 1 ... 6h = Pre-scale + EQ bands 0 to 5 7h = Reserved
EQ1_EN	3	RW	0h	EQ1 Enable: 0 = EQ1 bypassed 1 = EQ1 enabled
EQ1_BE	2:0	RW	0h	EQ1 Band Enable: 0h = Pre-scale only 1h = Pre-scale + EQ band 0 2h = Pre-scale + EQ bands 0 to 1 ... 6h = Pre-scale + EQ bands 0 to 5 7h = Reserved

To enable all EQ bands in both banks, register 20h must be written a value of EEh. Alternatively, if only the first two bands of EQ1 and EQ2 are desired, register 20h must be written a value of AAh.

To disable the EQ, simply write 00h to register 20h.

Note: the EQ is applied simultaneously to the headphone and speaker paths. The ACS422XXX contains a bit that controls whether or not the headphone detect pin enables or disables the EQ on the headphone path. For more information, please read Section 4.0.

3.2 Enabling the Compressor/Limiter/Dynamic Range Controller

The Compressor/Limiter/Expander functions are enabled or disabled through register 25h. Register 25h is defined by:

CLECTL (25h)

Address	Register	Default
25h	CLECTL	00h

Field Name	Bits	R/W	Default	Description
	7:5	R	0h	Reserved.
LVL_MODE	4	RW	0h	CLE Level Detection Mode: 0 = Average 1 = Peak
WINDOWSEL	3	RW	0h	CLE Level Detection Window: 0 = Equivalent of 512 samples at the selected Base Rate (~10-16ms) 1 = Equivalent of 64 samples at the selected Base Rate (~1.3-2ms)
EXP_EN	2	RW	0h	Expander Enable: 0 = Disabled 1 = Enabled
LIMIT_EN	1	RW	0h	Limiter Enable: 0 = Disabled 1 = Enabled
COMP_EN	0	RW	0h	Compressor Enable: 0 = Disabled 1 = Enabled

This register not only contains the enhancement enable/disable bits, but also sets the Detection Mode and Detection Window for the Compressor/Limiter/Enhancer block. The Detection Mode and Detection Window bits should be set according to the desired values of the system designer.

To enable the compressor and limiter, but not the expander, the designer should write 03h to register 25h. To enable the compressor and expander but not the limiter, the designer should write 05h to register 25h. To disable all compressor/limiter/expander effects, the designer should write 00h to register 25h.

Note: The designer should be aware that the compressor, limiter, and expander effects are applied simultaneously to the headphone and speaker paths. However, the headphone detection bit does NOT control these functions. If the designer wishes to have these functions applied to the speaker path, but not to the headphone path, then the designer must have the embedded controller detect the headphone presence and shut off the effects by writing 00h to register 25h. Conversely, the embedded controller must also turn these effects back on when the headphones are removed.

3.3 Enabling the Bass/Treble/3D Enhancements

The 3D enhancement, Psychoacoustic Bass enhancement, and Psychoacoustic Treble enhancement features are all enabled/disabled through register 39h. Register 39h is defined by:

FXCTL (39h)

Address	Register	Default
39h	FXCTL	00h

Field Name	Bits	R/W	Default	Description
	7:5	R	0h	Reserved.
3DEN	4	RW	0h	3D Enhancement Enable: 0 = Disabled 1 = Enabled
TEEN	3	RW	0h	Treble Enhancement Enable: 0 = Disabled 1 = Enabled
TNLFBYPASS	2	RW	0h	Treble Non-linear Function Bypass: 0 = Enabled 1 = Bypassed
BEEN	1	RW	0h	Bass Enhancement Enable: 0 = Disabled 1 = Enabled
BNLFBYPASS	0	RW	0h	Bass Non-linear Function Bypass: 0 = Enabled 1 = Bypassed

To enable or disable the 3D effects, the designer must set or clear bit 4 of register 39h. To enable or disable the Treble Enhancement, the designer must set or clear BOTH bits 2 and 3 of register 39h. To enable or disable the Bass Enhancement, the designer must set or clear BOTH bits 0 and 1 of register 39h.

As an example, to enable the Bass Enhancement and the 3D enhancement, the designer would write 0x13 to register 39h.

To disable all enhancements, the designer should write 0x00 to register 39h.

Note: As with the compressor/limiter/enhancer block, the designer should be aware that the 3D/Bass/Treble Enhancements are applied simultaneously to the headphone and speaker paths. However, the headphone detection bit does NOT control these functions. If the designer wishes to have these functions applied to the speaker path, but not to the headphone path, then the designer must have the embedded controller detect the headphone presence and shut off the enhancements by writing 00h to register 39h. Conversely, the embedded controller must also turn these enhancements back on when the headphones are removed.

4.0 Programming the Headphone Detection Bit and Controlling the EQ

The ACS422XXX features a detection bit that can indicate the presence of a headphone and control the shutdown/enable of the speaker amplifier portion. Additionally, the EQ1 and EQ2 banks can be enabled/disabled by this pin. This allows for the EQ to be applied to the speaker path only, or it allows for separate EQ to be applied to the speaker path and headphone path. For example, EQ1 can be applied to the speaker path while EQ2 becomes applied to the headphone path. These features are controlled by register 1Ch. Register 1Ch is defined by:

CTL (1Ch)

Address	Register	Default
1Ch	CTL	00h

Field Name	Bits	R/W	Default	Description
HPSWEN	7	RW	0h	Headphone Switch Enable: 0 = Headphone switch disabled 1 = Headphone switch enabled
HPSWPOL	6	RW	0h	Headphone Switch Polarity: 0 = HPDETECT high indicates headphone 1 = HPDETECT high indicates speaker
EQ2SW	5:4	RW	0h	EQ2 behavior due to speaker/headphone output state: 00b = EQ is not disabled due to headphone/speaker logic 01b = EQ is disabled when headphone output is active 10b = EQ is disabled when speaker output is active 11b = EQ is disabled when headphone AND speaker output are active
EQ1SW	3:2	RW	0h	EQ1 behavior due to speaker/headphone output state: 00b = EQ is not disabled due to headphone/speaker logic 01b = EQ is disabled when headphone output is active 10b = EQ is disabled when speaker output is active 11b = EQ is disabled when headphone AND speaker output are active
TSDEN	1	RW	0h	Thermal Shutdown Enable: 0 = Thermal shutdown disabled 1 = Thermal shutdown enabled
TOEN	0	RW	0h	Zero-Cross Time-Out Enable: 0 = Time-out disabled 1 = Time-out enabled (volumes update if no zero-cross event occurs before time-out)

To enable the headphone detection pin, bit 7 must be set to a 1, and bit 6 must reflect the appropriate desired polarity of the HP_DET signal. The signal can be defined as either active HIGH or active LOW. Note that the ACS422XXX features an internal pull-up to PVDD on this pin.

The ACS32201 also features an internal pull-up to PVDD, but the HP_DET pin has been replaced by a “SHUTDOWN#” pin. Therefore, the ACS32201 has register 1Ch pre-set to C0h in the factory. The ACS32201 is expecting that the SHUTDOWN# pin be an active low signal (low = shutdown). However, this polarity can be overwritten by simply writing the desired behavior to register 1Ch.

As mentioned above, the HP_DET signal can also control the behavior of the EQ1 and EQ2 banks. This behavior is defined by the status of bits 2:5 of register 1Ch. By default, these bits are set to 00h, and therefore, the EQ will simultaneously be applied to both the headphone path and the speaker path. By setting the bits appropriately (depending on the switch polarity), the EQ banks can be individually applied to either the headphone path or the speaker path.

For example: in a system with an ACS422XXX codec and an active low HP_DET signal (low = headphone), setting register 1Ch to D8h will cause bank EQ1 to be applied to the speakers while bank EQ2 is applied whenever headphones are sensed.

For the ACS32201, bits 2:5 should remain set to 00h, because the ACS32201 does not feature a headphone path.

Note: As mentioned previously, the designer should be aware that the Compressor / Limiter / Enhancer / 3D / Bass / Treble features are applied simultaneously to the headphone and speaker paths. However, the headphone detection bit does NOT control these functions. If the designer wishes to have these functions applied to the speaker path, but not to the headphone path, then the designer must have the embedded controller detect the headphone presence and shut off the features appropriately. Conversely, the embedded controller must also turn these features back on when the headphones are removed.

5.0 Controlling the Volume

In order to control the volume of the ASC422xXX, it is recommended to use the DAC volume registers (registers 04h and 05h). The Headphone Volume registers (00h and 01h) and the Speaker Gain registers (02h and 03h) should be fixed during initialization according to the system design targets. By default, the headphone volume registers are set to 79h on the ACS422XXX and the speaker gain registers are set to 6Fh. These default values will yield a 1V RMS output with a 0dBFS sine wave from the headphone path and 2W RMS into 4-ohm loads with a 0dBFS sine wave on the speaker path. These registers can be trimmed or adjusted upon initialization.

Again, standard volume control should be handled through the DAC volume registers, which will affect both the speaker and headphone paths (if available). The DAC volume registers are defined by:

DACVOLL (04h)

Address	Register	Default
04h	DACVOLL	FFh

Field Name	Bits	R/W	Default	Description
DACVOL_L	7:0	RW	FFh	Left DAC Volume (0.375dB Steps): FFh = 0dB FEh = -0.375dB ... 01h = -95.625dB 00h = Mute. Note: If DACVOLU is set, this setting will take effect after the next write to the Right DAC Volume register.

DACVOLR (05h)

Address	Register	Default
05h	DACVOLR	FFh

Field Name	Bits	R/W	Default	Description
DACVOL_R	7:0	RW	FFh	Right DAC Volume (0.375dB Steps): FFh = 0dB FEh = -0.375dB ... 01h = -95.625dB 00h = Mute

The DAC volume registers reach maximum volume when set to FFh. For each bit value less than FFh, the volume will decrease by 0.375dB. For example, setting these registers to FEh will yield -0.375dB. The exact desired step size for volume increments or decrements can be defined by the system designer. However Tempo recommends

using 1.125dB steps with code similar to the following (note that VOLUME_DOWN and VOLUME_UP are active low buttons):

```
if(VOLUME_DOWN==0){

     if(vol>0){
         vol=vol-3;
     }
     i2c_start();
     I2CWrite(0x04,vol);
     i2c_restart();
     I2CWrite(0x05,vol);
     i2c_stop();
     __delay_ms(10);
 }
 if(VOLUME_UP==0){
     if(vol<255){
         vol=vol+3;
     }
     i2c_start();
     I2CWrite(0x04,vol);
     i2c_restart();
     I2CWrite(0x05,vol);
     i2c_stop();
     __delay_ms(10);
 }
```

5.1 Setting the Left and Right DAC Volume Simultaneously

The ACS422XXX has a feature that will allow the DAC volume to only be set after both the Left and Right channel volume registers have been written. This may help avoid any discrepancy in volume between the left and right channels when changing volume. This feature is controlled by setting bit 4 of register 0Ah. Register 0Ah is defined by:

VUCTL (0Ah)

Address	Register	Default
0Ah	VUCTL	C0h

Field Name	Bits	R/W	Default	Description
ADCFADE	7	RW	1h	ADC Volume Fade Enable: 1 = ADC volumes fade between old/new value 0 = ADC volumes change immediately
DACFADE	6	RW	1h	DAC Volume Fade Enable: 1 = DAC volumes fade between old/new value 0 = ADC volumes change immediately
	5	R	0h	Reserved.
INVOLU	4	RW	0h	Input Volume Update Control: 1 = Left Input Volume held until Right Input Volume register is written 0 = Left Input Volume updated immediately
ADCVOLU	3	RW	0h	ADC Volume Update Control: 1 = Left ADC Volume held until Right ADC Volume register is written 0 = Left ADC Volume updated immediately
DACVOLU	2	RW	0h	DAC Volume Update Control: 1 = Left DAC Volume held until Right DAC Volume register is written 0 = Left DAC Volume updated immediately
SPKVOLU	1	RW	0h	Speaker Volume Update Control: 1 = Left Speaker Volume held until Right Speaker Volume register is written 0 = Left Speaker Volume updated immediately
HPVOLU	0	RW	0h	Headphone Volume Update Control: 1 = Left Headphone Volume held until Right Headphone Volume register is written 0 = Left Headphone Volume updated immediately

By setting the DACVOLU bit high, volume control for both the left and right channel DACs will only be effective after the right channel volume is written. Therefore, the LEFT channel DAC volume must always be written first!

It is recommended that the ADCFADE and DACFADE bits remain set high at all times in order to avoid possible pop noises when making rapid changes in volume. Thus if the system designer wishes to use the DACVOLU feature, register 0Ah should be set to C4h.

5.2 Muting the Volume

To mute the volume, registers 04h and 05h must be set to 00h

```
i2c_start();
I2CWrite(0x04, 0); //mute the LEFT DAC volume
i2c_restart();
I2CWrite(0x05, 0); //mute the RIGHT DAC volume
i2c_stop();
```

5.3 Unmuting the Volume

To unmute the volume, the registers 04h and 05h must be restored to their previously known value. Assuming the designer has created a global variable reflecting the desired system volume (called “int vol”), the volume can be restored by the following code:

```
i2c_start();
I2CWrite(0x04, vol); //restore the LEFT DAC volume
i2c_restart();
I2CWrite(0x05, vol); //restore the RIGHT DAC volume
i2c_stop();
```

6.0 Power Savings Mode

The system designer may wish to place the ACS422XXX into a low-power mode when system audio is not present. This would be desired in a battery-operated system, in order to save battery life.

The lowest power mode for each codec (while maintaining I2C communication) is obtained by writing the following code sequence:

```
i2c_start();
I2CWrite(0x04, 0x00); //mute the LEFT DAC volume
i2c_restart();
I2CWrite(0x05, 0x00); //mute the RIGHT DAC volume
i2c_restart();
I2CWrite(0x18, 0x08); //mute the DAC
i2c_restart();
I2CWrite(0x16, 0x08); //mute the ADC (Apply only to ACS422XXX )
i2c_restart();
I2CWrite(0x1B, 0x01); //power down the HP and BTL, leave VREF on.
i2c_restart();
I2CWrite(0x1A, 0x00); //power down the ADC, leave master clock
i2c_restart();
__delay_ms(50); //delay by 50ms here.
I2CWrite(0x1B, 0x00); //kill VREF
i2c_restart();
I2CWrite(0x15, 0x3F); //Disable internal pull-downs
i2c_restart();
I2CWrite(0x1F, 0x50); //Decrease modulator rates, supply detect off
i2c_restart();
I2CWrite(0x1C, 0x00); //Disable headphone detection bit
i2c_restart();
I2CWrite(0x63, 0x00);
       //Disable reference clock output (Apply to ACS422XXX  ONLY, and
       //apply ONLY after ACS32201 is fully disabled!)
i2c_restart();
I2CWrite(0x42, 0x00); //Disable ULVO and PWM QT
i2c_restart();
I2CWrite(0x71, 0x06); //Disable HP Current Limit
i2c_stop();
I2CWrite(0x73, 0x06); //Reduce analog currents
i2c_stop();
I2CWrite(0x74, 0x01); //Power down d2a
i2c_restart();
I2CWrite(0x1A, 0x01); //kill master clock
i2c_stop();
```

NOTE: in a system using the ACS422XXX and the ACS32201 simultaneously, the ACS32201 should be powered down completely, prior to beginning the power down sequence of the ACS422XXX ! This is because in a dual-amplifier system, the ACS422XXX will supply the clock to the ACS32201.

6.1 Resuming from Power Savings Mode

Resuming from power savings mode is a simple as backing out the changes made during the entrance of power savings mode. Note that the ACS422XXX should be powered up PRIOR to trying to power up the ACS32201 in dual-amplifier systems. Power-up code should resemble the following:

```
i2c_start();
I2CWrite(0x1A, 0x00); //enable master clock
i2c_restart();
I2CWrite(0x74, 0x00); //Enable HP Current Limit
i2c_stop();
I2CWrite(0x73, 0x00); //Full analog currents
i2c_stop();
I2CWrite(0x71, 0x00); //Power up d2a
i2c_restart();
I2CWrite(0x42, 0xD4); //Enable ULVO and PWM QT
i2c_restart();
I2CWrite(0x63, 0x04);
       //Enable reference clock output (Apply to ACS422XXX  ONLY)
i2c_restart();
I2CWrite(0x1C, 0xC0);
       //Note this register should be restored to desired system design
       //value.  See section 4.0 for more details
i2c_restart();
I2CWrite(0x1F, 0xA1); //Full modulator rates, supply detect on.
i2c_restart();
I2CWrite(0x15, 0x00); //Enable internal pull-downs
i2c_restart();
I2CWrite(0x1B, 0x01); //power up VREF.
i2c_restart();
__delay_ms(50); //Delay by 50 ms.
I2CWrite(0x1A, 0xFC); //power up the ADC, enable master clock
i2c_restart();
I2CWrite(0x1B, 0x19); //power up BTL. (If using HP amp on ACS422XXX , use 0x7F)
i2c_restart();
I2CWrite(0x18, 0x00); //unmute the DAC
i2c_restart();
I2CWrite(0x16, 0x00); //unmute the ADC (Apply only to ACS422XXX )
i2c_restart();
I2CWrite(0x04, vol); //restore the LEFT DAC volume
i2c_restart();
I2CWrite(0x05, vol); //restore the RIGHT DAC volume
i2c_stop();
```

7.0 Sample Rate Considerations

The ACS422XXX and ACS32201 are designed to work with 12.288MHz clock inputs and are pre-set in the factory to a 48KHz base rate. The parts will also work satisfactorily with an 11.576MHz input clock, but it may be desired to change the sample rate to 44.1KHz if this is done.

The sample rates for the DAC and ADC are controlled through registers 17h and 19h. These registers are defined by:

ADCSR (17h)

Address	Register	Default
17h	ADCSR	12h

Field Name	Bits	R/W	Default	Description
ABCM	7:6	RW	0h	ADC Bit Clock Mode (for ADCBCLK generation in master mode): 0h = Auto 1h = 32x Fs 2h = 40x Fs 3h = 64x Fs
	5	R	0h	Reserved.
ABR	4:3	RW	2h	ADC Base Rate: 0h = 32kHz 1h = 44.1kHz 2h = 48kHz 3h = Reserved
ABM	2:0	RW	2h	ADC Base Rate Multiplier: 0h = 0.25x 1h = 0.5x 2h = 1x 3h = 2x 4h-7h = Reserved

DACSR (19h)

Address	Register	Default
19h	DACSR	12h

Field Name	Bits	R/W	Default	Description
DBCM	7:6	RW	0h	DAC Bit Clock Mode (for DACBCLK generation in master mode): 0h = Auto 1h = 32x Fs 2h = 40x Fs 3h = 64x Fs
	5	R	0h	Reserved.
DBR	4:3	RW	2h	DAC Base Rate: 0h = 32kHz 1h = 44.1kHz 2h = 48kHz 3h = Reserved
DBM	2:0	RW	2h	DAC Base Rate Multiplier: 0h = 0.25x 1h = 0.5x 2h = 1x 3h = 2x 4h-7h = Reserved

The sample rate can be changed to suit the system designer's needs, but it should be noted that the ACS422XXX and ACS32201 are specifically designed to support either 48KHz or 44.1Kz. Other base rates are easily supported, but may require more significant changes to the internal PLL structures of the codecs.

To change the base rate of the ACS422XXX from 48KHz to 44.1KHz, simply write 0Ah to registers 17h and 19h. The ACS32201 would only need to write 0Ah to register 19h.

If the base rate is changed, please note that the desired system base rate must also be entered into the Audio System Configurator tool in order for the tool to generate the proper EQ coefficients for the exact base rate being used. Please contact Tempo for more details on this process, or refer to AN-02 Audio Algorithms.

8.0 I2C Reference Code

This code is provided solely as a reference of working code for a MICROCHIP PC16F1503 in the HI-TECH C compiler and should be considered REFERENCE only. This code is not warranted, guaranteed, or optimized in any way.

```
void init(void){
    PORTA = 0x00;   //Clear PORTA register
    LATA = 0x00;    //Clear LATA register
    ANSELA = 0x00;  //Set PORTA as digital inputs
    TRISA = 0x00;   //All port A as outputs
    LATC = 0x00;
    PORTC = 0x00;
    TRISC = 0x3F;       //Enable RC0 (I2C CLK) as an input, enable RC1 (I2C DAT) as an input,
                        //en RC2, RC3, RC4 as inputs.
    ANSELC = 0x00;      //Make sure RC0 and RC1 are digital inputs
    SSPCON1 = 0x28;     //Enable SSP module and Enable I2C Master mode,
                        //clock = FOSC/(4 * (SSPADD + 1))
    SSPCON2 = 0x00;     //Clear the SSPCON2 register
    SSPSTAT = 0x80;     //Disable slew rate control and Disable SMBUS Compliance
    SSPADD = 0x09;      //100Khz @ 4Mhz Fosc
}

void i2c_start(void){
    SEN = 1;            //Generate Start Condition
    i2c_wait();
}

void I2CWrite(unsigned int Reg, byte Value){
    SSPBUF = 0xD2;      //write the codec address! Note, this value changes to D4 for ACS32201
    i2c_wait();
    SSPBUF = Reg;
    i2c_wait();
    SSPBUF = Value;
    i2c_wait();
}

void i2c_stop(void){
    PEN = 1;             //Generate Stop Condition
    i2c_wait();
}

void i2c_wait(void) {
    while (!SSP1IF);
    SSP1IF = 0;
}

void i2c_restart(void){
    RSEN = 1;
    i2c_wait();
}

byte I2CRead(unsigned int Reg, unsigned char ack){
   byte ReadVal = 0x00;
        i2c_start();
```

```
        SSPBUF = 0xD2;
        i2c_wait();
        SSPBUF = Reg;
        i2c_wait();
        i2c_restart();
        SSPBUF = 0xD3;
        i2c_wait();
        RCEN = 1;
        i2c_wait();
        ReadVal = SSPBUF;
        while ((SSPCON2 & 0x1F)| R_nW){};
        if(ack){
            ACKDT=0;
        }
        else{
            ACKDT=1;
        }
        ACKEN = 1;
        i2c_wait();
        i2c_stop();
   return ReadVal;
}
```