TEXAS INSTRUMENTS

Application Report
SLAA265–September 2005

TLV320AIC3x WinCE 5.0 Drivers

Wendy X. Fang *DAP Group, HPA*

ABSTRACT

TLV320AIC3x audio drivers have been developed with SPI or I2C control interfaces and I2S audio streaming. The code was tested on an Intel™ MainStone II platform, running on the Microsoft™Windows™ CE 5.0 operating system. This application report discusses the SPI/I2C and I2S drivers, including the hardware connection between the TLV320AIC33/32EVM and the MainStone II platform, the Windows CE 5.0 drivers' code and structure, and the installations.

Contents

List of Figures

1 Introduction

Texas Instruments (TI) audio devices TLV320AIC3x (or AIC3x for short), including TLV320AIC31, TLV320AIC32, and TLV320AIC33, are low-power, high-performance stereo input and stereo output coder/decoders (codec). These devices are ideal for portable audio and telephony applications, in which an embedded operating system, such as Windows™ CE, often resides and operates.

The Windows CE (WinCE) 5.0 drivers for the AIC3x codecs that were developed are discussed in this application report to assist users to quickly set up, run, and use the codec device on the WinCE 5.0 system.

The AIC3x drivers were coded on the standard device driver's PDD (platform-dependent device) layer, and the PDD layer was further split to have an additional processor-dependent layer (PDL) to make the drivers easy to port into different host processors. See application report *TSC2301 WinCE Generic Drivers* (SLAA187) for details on Windows CE PDD and TI PDL generic drivers.

The drivers were run and tested on an AIC33 EVM board and an Intel™ MainStone II platform with the PXA270 Step B0 processor.

2 Connections

The AIC33 device must be wired and connected to a host processor, where the device driver code is ported and executed. The two buses or ports for AIC33 operation are the control bus and the audio data bus. The control bus on the AIC33 can be either an SPI or an I2C bus, selectable through a hardware pin (SELECT) on the AIC33; the AIC31 or AIC32 can be controlled only through I2C bus. The audio data streams through the I2S bus on the AIC3x.

In developing the AIC33 drivers for this application, the TI AIC33EVM board (SBAU114) and the Intel MainStone II platform with the PXA270 Step B0 processor (see reference 4) were employed.

On the SPI-controlled AIC33, the nine digital signals that are essential for running the AIC33 audio drivers are:

- the SPI bus, four wires: SCLK, $\overline{\text{SS}}$, MOSI, and MISO (at J16 of AIC33EVM board);
- the main audio codec clock, MCLK (at J17 of AIC33EVM board); and
- the I2S bus, four wires: BCLK, WCLK, SDIN and SDOUT (at J17 of AIC33EVM board).

Figure 1 shows the wires and connection between the AIC33 and PXA270 processor for the SPI control interface.

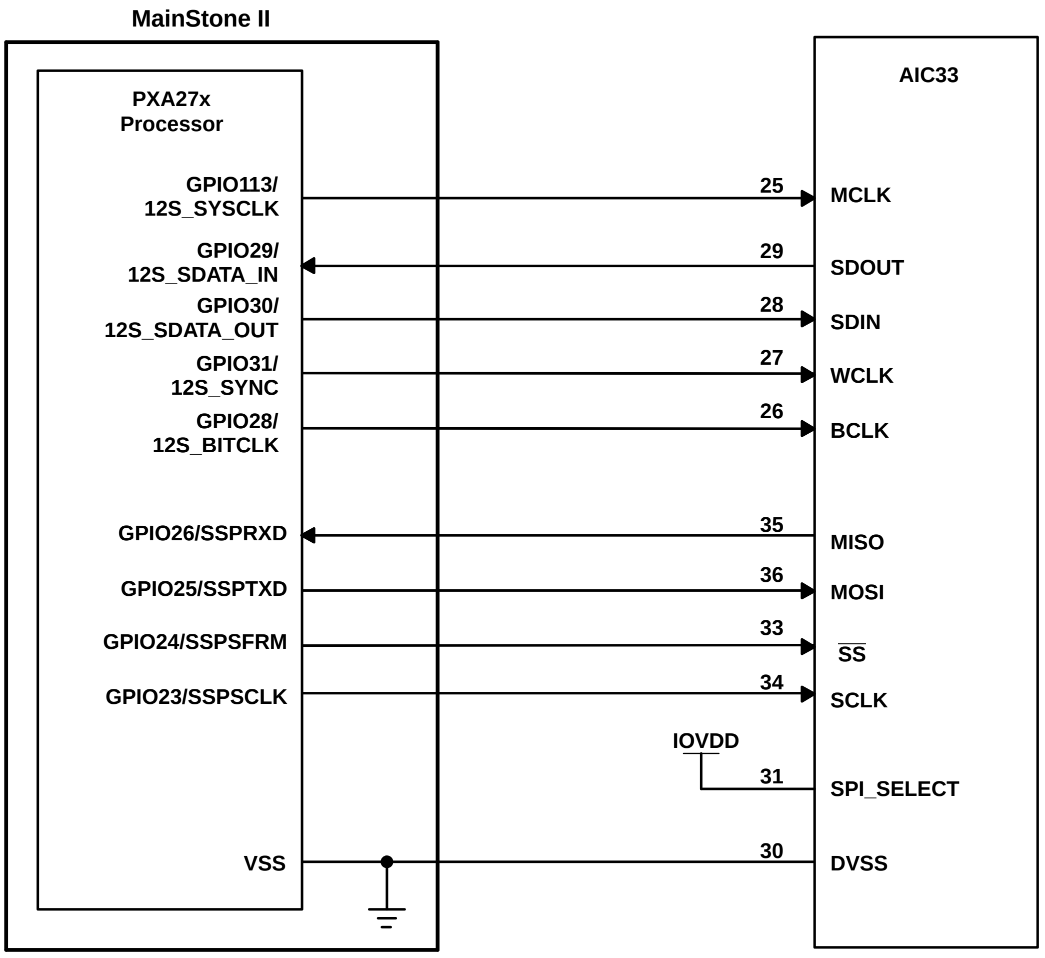

Figure 1. AIC33 Connections to PXA27x Processor With SPI Control Bus

To implement the connection shown in Figure 1 on the AIC33EVM board, JMP10 was installed on pins 3 and 1; JMP11 was installed on pins 3 and 4; and JMP12 was installed on pins 3 and 4. See SBAU114 for the schematic and other details of the EVM board. On the MainStone II system, the original touch/audio module, connected on the MainStone II main board, was removed and replaced with the connections as shown in Figure 1. See the relevant Intel documentation for further information concerning the MainStone II Platform.

On the I2C-controlled AIC3x, the seven digital signals that are essential for running the audio driver are:

- the I2C bus, two wires: SCL, and SDA (at J16 or J17 of AIC33EVM board);
- the main audio codec clock, MCLK (at J17 of AIC33EVM board); and
- the I2S bus, four wires: BCLK, WCLK, SDIN and SDOUT (at J17 of AIC33EVM board).

Figure 2 shows the wires and connection between the AIC3x and PXA270 processor for theI2C control interface.

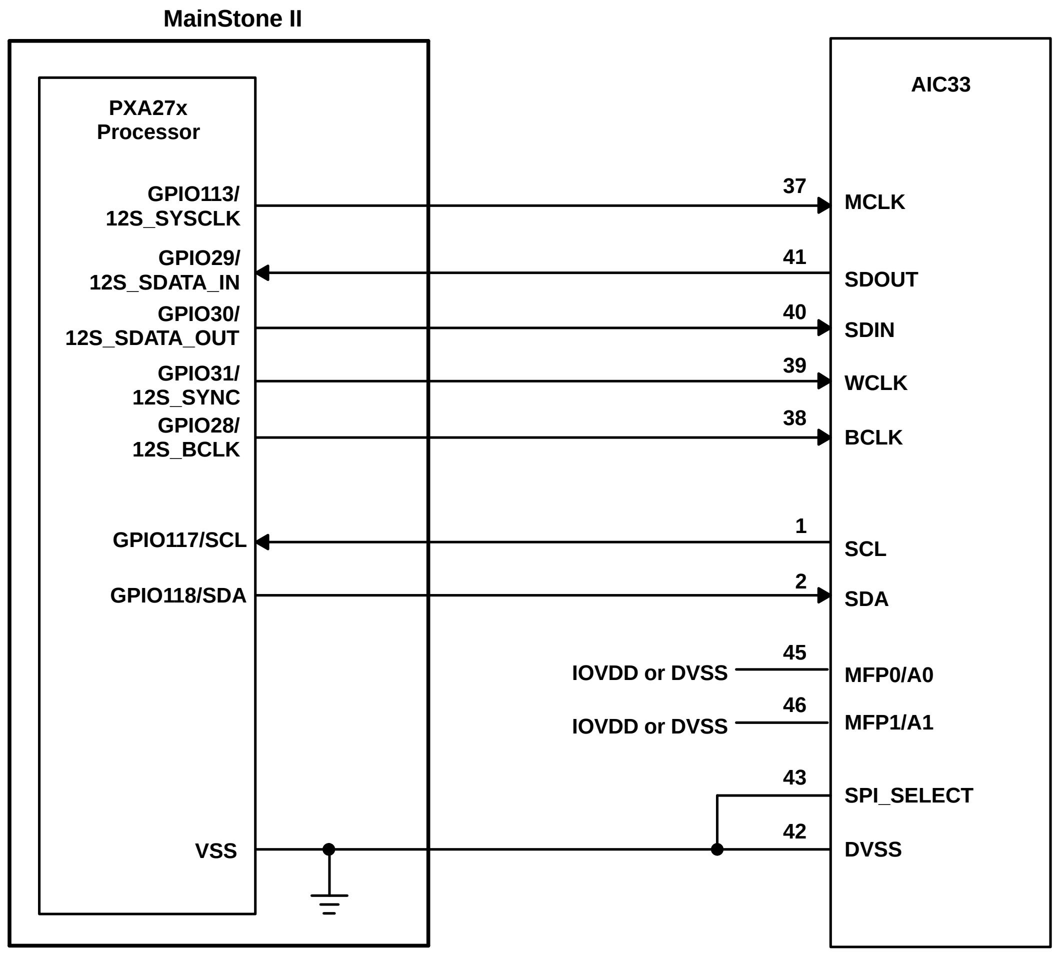

Figure 2. AIC3x Connections to PXA27x Processor with I2C Control

To implement the connection shown in Figure 2, on an AIC33EVM board, JMP10 was installed on pins 3 and 5; JMP11 was installed either on pins 3 and 1 (A0=1) or on pins 3 and 5 (A0=0); and JMP12 was installed either on pins 3 and 1(A1=1) or on pins 3 and 5 (A1=0). See SBAU114 for the schematic and other details of the EVM board and the corresponding data sheets for details on the I2C address.

The connections to the AIC31 or the AIC32 codec is similar to that shown in Figure 2, but note that on AIC31 or AIC32 the only control bus is I2C with default A1=A0=0 as part of the device address (see the corresponding data sheets).

In addition to the connections shown in Figure 1 or Figure 2 is the important and useful hardware reset pin, $\overline{\text{RESET}}$. The $\overline{\text{RESET}}$ pin can be connected to a GPO pin of the host so that the host processor can issue a hardware reset to the AIC3x and put the codec into a known default condition after a power up.

3 Device Drivers

Figure 3 and Figure 4 list the locations of the AIC33 audio device driver files for the SPI and I2C control interfaces, respectively. The files starting with *Host…* are the processor-dependent code or PDL, such as HostAudio.C or HostSPIComm.H.

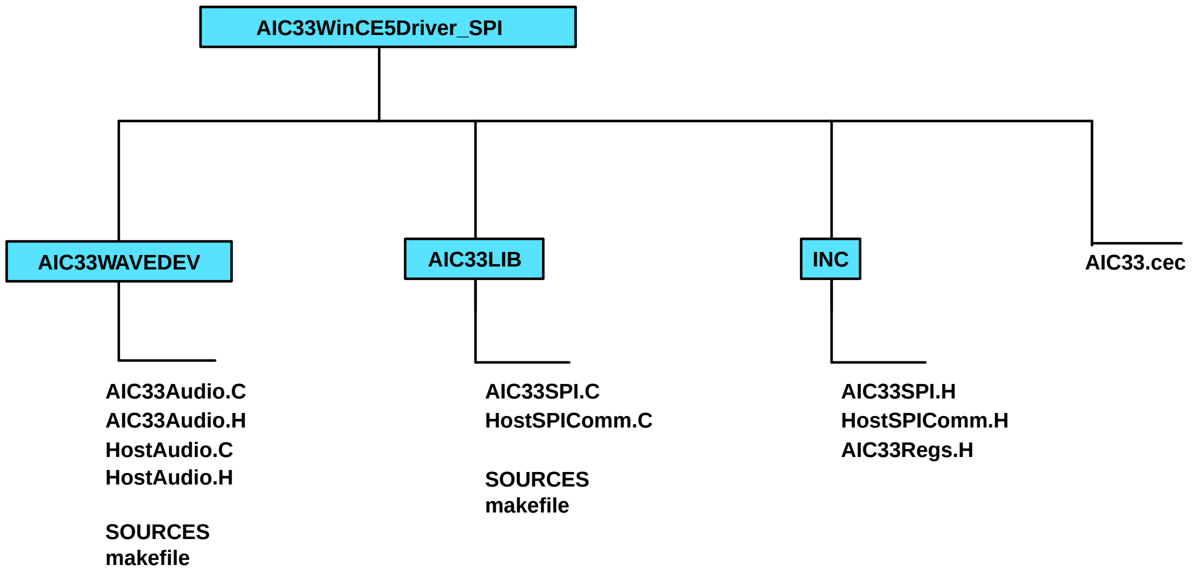

Figure 3. AIC33 WinCE 5.0 Driver Files With SPI Control Interface

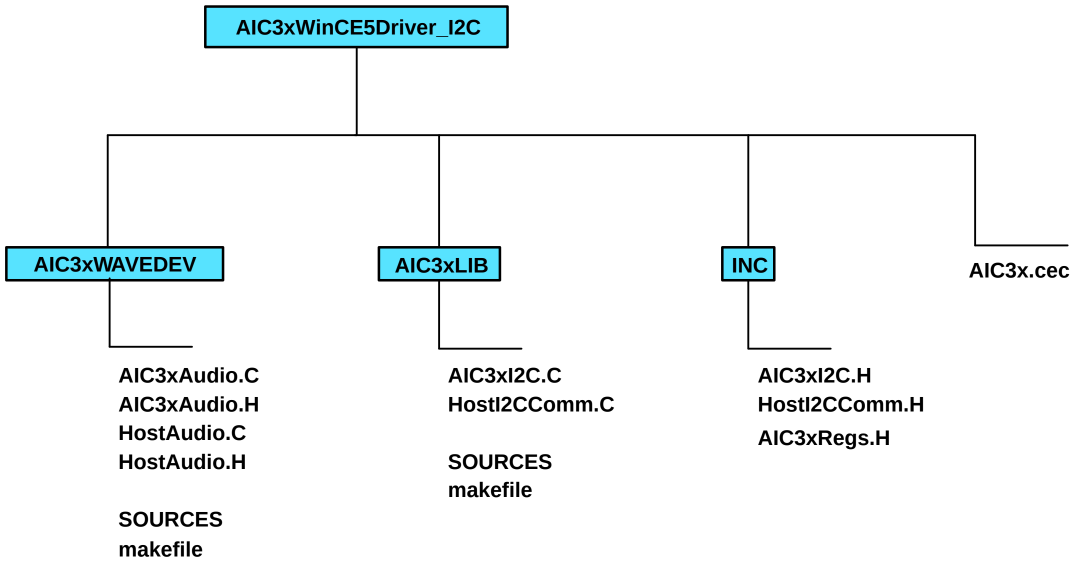

Figure 4. AIC3x WinCE 5.0 Driver Files With I2C Control Interface

3.1 SPI Interface

The four AIC33 pins, SCLK, $\overline{\text{SS}}$, MOSI, and MISO, are connected to the GPIO23 to GPIO26 of the PXA27x processor, respectively.

On the host side, the PXA27x GPIO, SSP, and Clock management control registers are used to set up the SPI interface to communicate with the AIC33's SPI. The setup was implemented at the routine, HWSetupSPI():

```
///////
// Function: void HWSetupSPI(BOOL InPowerHandle)
// Purpose:  This function must be called from the power handler
//           of the respective drivers using this library. This
//           function will configure the GPIO pins according to
//           the functionality shown in the table below
//                Signals      Pin#    Direction      Alternate Function
//                SSPSCLK      GP23    output              2
//                SSPSFRM      GP24    output              0
//                SSPTXD       GP25    output              2
//                SSPRXD       GP26    input               1
///////
void HWSetupSPI(BOOL InPowerHandle)
{
   RETAILMSG(1,(TEXT("Setup Host GPIO & SSP for an SPI Interface... \r\n")));
// disable Unit clock
   g_pClockRegs->cken &= ~XLLP_CLKEN_SSP1;
// disable SSP1
   g_pSSPRegs->sscr0 &= ~SSE_ENABLE;
// Set up the GPIO24=SFRM = 1 (GPSR0)
   g_pGPIORegs->GPSR0 |= ( GPIO_24_SFRM );
// Program direction of the GPIOs (GPDR0)
   // (GPIO23/24/25 as outputs and GPIO26 as input)
   g_pGPIORegs->GPDR0 |= GPIO_23_SCLK;
   g_pGPIORegs->GPDR0 |= GPIO_24_SFRM;
   g_pGPIORegs->GPDR0 |= GPIO_25_MOSI;
   g_pGPIORegs->GPDR0 &= ~GPIO_26_MISO;
// Program GPIO alternate function register (GAFR0_U)
   g_pGPIORegs->GAFR0_U &= 0xFFC03FFF;
   g_pGPIORegs->GAFR0_U |= GPIO_23_AF2_SSPSCLK;
   // GPIO24 is used here as GPO
   g_pGPIORegs->GAFR0_U |= GPIO_25_AF2_SSPTXD;
   g_pGPIORegs->GAFR0_U |= GPIO_26_AF1_SSPRXD;
   // Set up SSP registers (when disabled SSP)
   // set up SSP control register 0 and 1
   g_pSSPRegs->sscr0 = (SCR_590_KHZ | SSE_DISABLE | ECS_INTERNAL |
                        FRF_MOTOROLA | DSS_8_BIT );
   g_pSSPRegs->sscr1 = (RFT_SEVEN | TFT_ZERO | MWDS_16_BIT | SPH_HALF_DELAY |
                        SPO_IDLE_LOW | LBM_DISABLE | TIE_DISABLE | RIE_DISABLE );
   // Enable SSP last
   g_pSSPRegs->sscr0 |= SSE_ENABLE;
   // enable SPI1 Unit clock
   g_pClockRegs->cken |= XLLP_CLKEN_SSP1;
//    DumpRegsGPIO();
//    DumpRegsSSP();
//    DumpRegsClock();
}
```

Note that: (1) The host’s GPIO24 was used as the SPI’s $\overline{\text{SS}}$, but this pin in the PXA27x processor should be programmed as a GPO and be set high and low by SW (not through the PXA27x SSP function); and (2) the SSP1 unit clock should be turned OFF before the setup and turned ON after the setup.

3.2 I2C Interface

The two AIC3x I2C bus pins, SCK and SDA, are connected to the GPIO117 and GPIO118 of the PXA27x processor, respectively.

On the host side, the PXA27x GPIO, I2C, and Clock management control registers are used to set up the I2C interface to communicate with the AIC3x’s I2C. The setup was implemented at the routine, HWSetupI2C():

```
///////
// Function: void HWInitI2C(BOOL InPowerHandle)
// Purpose:  This function must be called from the power handler
//           of the respective drivers using this library. This
//           function will configure the GPIO pins according to
//           the functionality shown in the table below
//               Signals    Pin#     Direction       Alternate Function
//                 SCL      GPIO117  output                  1
//                 SDA      GPIO118  output(at init)         1
///////
BOOL HWInitI2C(BOOL InPowerHandle)
{
    RETAILMSG(1,(TEXT("Setup Host GPIO & I2C for an I2C Interface...\r\n")));
    // init I2C control register (disabled I2C unit)
    g_pI2CRegs->icr = 0;
    // enable I2C unit clock (the clock should be enabled first)
    g_pClockRegs->cken |=  XLLP_CLKEN_I2C;
    // set up GPIO
    g_pGPIORegs->GPDR3 |= GPIO_117_SCL;
    g_pGPIORegs->GPDR3 |= GPIO_118_SDA;
    g_pGPIORegs->GAFR3_U &= ~GPIO_I2C_MASK;
    g_pGPIORegs->GAFR3_U |=  GPIO_117_AF1_SCL;
    g_pGPIORegs->GAFR3_U |=  GPIO_118_AF1_SDA;
    // Setup processor I2C slave address (used only when PXA27x is slave)
    g_pI2CRegs->isar =  0x007F;
    // Set Processor I2C as master and enable the I2C
    g_pI2CRegs->icr  = ICR_SCLEA | ICR_IUE;

//  DumpRegsGPIO();
//  DumpRegsI2C();
//  DumpRegsClock();
    return(TRUE);
}
```

Two other important I2C interface routines are the HWI2CWriteRegs() and HWI2CReadRegs(), which allow the PXA27x to write to or read from AIC3x control registers using the I2C bus. The protocol for I2C write and read have been defined (see Figures 5 and 6 of the AIC33 data sheet SLAS480).

```
///////
// Function: HWI2CWriteRegs Routine
// Purpose:  This routine allows the PXA27x to write to AIC33
//           control register(s) using I2C bus.
// Note: The first byte in bytesBuf is the starting address
//       for writing; and the 2nd and on are bytes/contents
//       writing to AIC33
///////
BOOL HWI2CWriteRegs(UINT8 *bytesBuf, UINT8 bytesCount, BOOL InPowerHandle)
{
      UINT32 reg;
      if (!InPowerHandle)
      {
         // write (AIC33) device address + write "0" to I2C bus
         g_pI2CRegs->idbr = I2C_WRITE;
         reg = g_pI2CRegs->icr;
         reg |=  (ICR_START | ICR_TB);
         reg &= ~(ICR_STOP | ICR_ALDIE);
         g_pI2CRegs->icr = reg;
         if (HWI2CTxBusy(1000)) return(FALSE);
         // Send all bytes (1st byte is AIC33 starting address)
         while (bytesCount--)
         {
              g_pI2CRegs->idbr = (UINT32)(*bytesBuf++);
              reg = g_pI2CRegs->icr;
              reg &= ~ICR_START;
              reg |= (ICR_ALDIE | ICR_TB);
```

```
                    if (bytesCount == 0)
                          reg |= ICR_STOP;
                    else
                          reg &= ~ICR_STOP;
                    g_pI2CRegs->icr = reg;
                    if (HWI2CTxBusy(1000)) return(FALSE);
               }
               // Clear the STOP bit always
               g_pI2CRegs->icr &= ~ICR_STOP;
               return(TRUE);
          }
          else
          {
               RETAILMSG(1, (TEXT("HW Tx Error...\r\n")));
               return(FALSE);
          }
      }
      ///////
      // Function: HWI2CReadRegs Routine
      // Purpose:  This routine allows the PXA27x to read from AIC33
      //           control register(s) using I2C bus.
      // Note: The first byte in bytesBuf is the starting address for
      //       reading; and the 2nd and on are values reading from AIC33
      ///////
      BOOL HWI2CReadRegs(UINT8 *bytesBuf, UINT8 bytesCount,
                                BOOL InPowerHandle)
      {
               UINT32 reg;
               if (!InPowerHandle)
               {
                    // write (AIC33) device address + write "0" to I2C bus
                    g_pI2CRegs->idbr = I2C_WRITE;
                    reg = g_pI2CRegs->icr;
                    reg |=  (ICR_START | ICR_TB);
                    reg &= ~(ICR_STOP | ICR_ALDIE);
                    g_pI2CRegs->icr = reg;
                    if (HWI2CTxBusy(1000)) return(FALSE);
                    // write (AIC33) register address to I2C bus
                    g_pI2CRegs->idbr = (UINT32)(*bytesBuf++);
                    bytesCount -= 1;
                    reg = g_pI2CRegs->icr;
                    reg &= ~(ICR_START | ICR_STOP);
                    reg |= (ICR_ALDIE | ICR_TB);
                    g_pI2CRegs->icr = reg;
                    if (HWI2CTxBusy(1000)) return(FALSE);
                    // restart and register address + read "1" to I2C bus
                    g_pI2CRegs->idbr = I2C_READ;
                    reg = g_pI2CRegs->icr;
                    reg |= (ICR_START | ICR_TB);
                    reg &= ~(ICR_STOP | ICR_ALDIE);
                    g_pI2CRegs->icr = reg;
                    if (HWI2CTxBusy(1000)) return(FALSE);
                  // read the AIC33 registers' contents
                   while (bytesCount--)
              {
                   reg = g_pI2CRegs->icr;
                   reg &= ~ICR_START;
                   reg |= (ICR_ALDIE | ICR_TB);
                   if (bytesCount == 0)
                   {
                             reg |= (ICR_ACKNAK | ICR_STOP) ;
                   }
                   else
                   {
                             reg &= ~(ICR_ACKNAK | ICR_STOP);
                   }
                   g_pI2CRegs->icr = reg;
                   if (HWI2CRxBusy(1000)) return(FALSE);
                   reg = (g_pI2CRegs->idbr) & 0xFF;
                   *bytesBuf++ = (UINT8) reg;
                     }
               g_pI2CRegs->icr &= ~(ICR_ACKNAK | ICR_STOP);
               return(TRUE);
       }
       else
       {
               RETAILMSG(1, (TEXT("HW Rx Error...\r\n")));
               return(FALSE);
       }
      }
```

3.3 Audio Driver

From a hardware standpoint, the AIC33 audio driver must have SPI or I2C (for audio control) and I2S (for audio data streaming) buses. The SPI or I2C bus controls the audio codec's operation by writing to AIC33 audio control registers; the I2S bus transfers audio data between the host and the AIC33.

Additionally, the AIC3x MCLK pin should receive an external clock that provides the necessary timing for the AIC3x audio Sigma-Delta ADC and DAC to operate or run. The MCLK to the AIC3x should be generated from the same resource as the I2S clocks, i.e.: the MCLK should also from the host processor, which is the I2S master as described in this application report.

The AIC3x audio driver was built on the standard audio driver, WaveDev, and is located in the directory *AIC3xWaveDev*.

On the host side, the PXA27x GPIO28 to GPIO31 pins were used as the I2S and connected to AIC3x's BCLK, SDOUT, SDIN, and WCLK, respectively (see Figure 1). The GPIO113 is programmed as the I2S SYSCLK and is connected to the MCLK, which is programmed to generate a 11.346-MHz clock. The I2S setup was implemented at the routine, HWEnableI2S(), as follows:

```
//
//-------------------------------------------------------------------
// Function: HWEnableI2S()
//-------------------------------------------------------------------
//
void HWEnableI2S(void)
{
       RETAILMSG(1,(TEXT("Setup Host GPIO & I2S Interface... \r\n")));
       //Basic Outline:
       // configure the GPIO registers and set to I2S mode
       // Set up I2S control registers at default condition

      // insert reset for I2S
       v_pI2SRegs->sacr0 |= 0x00000008;

       // un-insert the reset
       v_pI2SRegs->sacr0 = 0x00007700;

       // disable I2S unit clock
       v_pClockRegs->cken &= ~XLLP_CLKEN_I2S;

      // setup GPIO direction regs
       v_pGPIORegs->GPDR0 |= XLLP_GPIO_BIT_I2SBITCLK     |
                             XLLP_GPIO_BIT_I2S_SYNC      |
                             XLLP_GPIO_BIT_I2S_SDATA_OUT;
       v_pGPIORegs->GPDR0 &= ~XLLP_GPIO_BIT_I2S_SDATA_IN;
       v_pGPIORegs->GPDR3 |= XLLP_GPIO_BIT_I2S_SYSCLK;  // GPIO113: SYSCLK as output

       // configure GPIO alternate function registers
       v_pGPIORegs->GAFR0_U &= 0x00FFFFFF;
       v_pGPIORegs->GAFR0_U |= XLLP_GPIO_AF_BIT_I2SBITCLK_OUT |
                               XLLP_GPIO_AF_BIT_I2S_SDATA_IN  |
                               XLLP_GPIO_AF_BIT_I2S_SDATA_OUT |
                                XLLP_GPIO_AF_BIT_I2S_SYNC;
       v_pGPIORegs->GAFR0_U &= ~XLLP_GPIO_AF_BIT_I2S_SYSCLK_MASK;
       v_pGPIORegs->GAFR3_U |= XLLP_GPIO_AF_BIT_I2S_SYSCLK;

       // configure I2S reg sacr0 but not enable I2S yet
       v_pI2SRegs->sacr0 = 0x00001104;

      // configure system for I2S mode
       v_pI2SRegs->sacr1 = 0x00000000;

      // configure clock divider
       v_pI2SRegs->sadiv = I2SRATE_44_1;  // divider for 44.1kHz audio

       // enable I2S
       v_pI2SRegs->sacr0 |= 0x00000001;

       // enable Unit clock
       v_pClockRegs->cken |= XLLP_CLKEN_I2S;
//     DumpRegsGPIO();
//     DumpRegsSSP();
//     DumpRegsClock();
//     DumpRegsI2S();
       return ;
}
```

The codec can be used per the requests of applications. As an example, this application report had initially set up the AIC33 in such a way that:

- About the I2S interface:
  1. The I2S interface is at 16 bits, standard I2S mode, with 44.1-kHz ADC and DAC sample rates.
  2. The AIC3x is the slave because the host is the I2S master (AIC3x can be I2S slave or master; but PXA27x can be only the master).
- About the audio input circuitry:
  1. The left and right ADC input from the stereo, single-ended LINE3 (MICIN3).
  2. ADC input gain is controlled by its PGA with initial gain 0-dB gain.
- About the audio output circuitry:
  1. The left and right DAC results are routed to the stereo-single-ended headphone, HPL/R and with HPLCOM and HPRCOM being shorted as the VCOM.
  2. The headphone output is at the CAPLESS mode.
  3. The DAC gains and the HPL/R output gains were all initialized to 0 dB.
- About bypass:
  1. The differential LINE2 Left is directly fed to the differential MONO_OUT.
- About other functions:
  1. The input high-pass filter has not been enabled.
  2. The output digital boost, emphasis, and 3-D functions have not been enabled.
  3. The PLL is enabled and GPIO1 is used as the PLL output.
  4. The headset detect function is disabled.
  5. The pop-reduction function is set to slowest and enabled.

All AIC33 audio control registers (in Page0 of the AIC33 memory space) were set up or initialized, as previously stated, in the routine **InitAIC33Audio()** and called by the audio PDD routine **PDD_AudioInitialize()**. The audio initialization routine is:

```
// Initalize AIC33 Audio Register at Default
void InitAIC33Audio(BOOL bInPowerHandler)
{
         RETAILMSG(1, (TEXT("InitAIC33Audio.\r\n")));
         // init for digital functions
         AIC33WriteReg(AIC33_RATE, RATE_INIT_VALUE, bInPowerHandler);
         AIC33WriteReg(AIC33_PLLa, PLLa_INIT_VALUE, bInPowerHandler);
         AIC33WriteReg(AIC33_PLLb, PLLb_INIT_VALUE, bInPowerHandler);
         AIC33WriteReg(AIC33_PLLc, PLLc_INIT_VALUE, bInPowerHandler);
         AIC33WriteReg(AIC33_PLLd, PLLd_INIT_VALUE, bInPowerHandler);
         AIC33WriteReg(AIC33_DATAPATH, DATAPATH_INIT_VALUE, bInPowerHandler);
```

```
        AIC33WriteReg(AIC33_INTERFa, INTERFa_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_INTERFb, INTERFb_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_INTERFc, INTERFc_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_DIGFILT, DIGFILT_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_HEDETa, HEDETa_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_HEDETb, HEDETb_INIT_VALUE, bInPowerHandler);

        // init for analog input functions
        AIC33WriteReg(AIC33_ADCPGAL, ADCPGAL_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_ADCPGAR, ADCPGAR_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_MIC3_ADCL, MIC3_ADCL_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_MIC3_ADCR, MIC3_ADCR_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_MICBIAS, MICBIAS_INIT_VALUE, bInPowerHandler);

       // init for analog output functions
        AIC33WriteReg(AIC33_OUTPWR, OUTPWR_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_OUTDRIVE, OUTDRIVE_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_OUTSTAGE, OUTSTAGE_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_OUTPOP, OUTPOP_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_DACLGAIN, DACLGAIN_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_DACRGAIN, DACRGAIN_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_DACL_HPL, DACL_HPL_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_HPLLEVEL, HPLLEVEL_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_DACR_HPR, DACR_HPR_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_HPRLEVEL, HPRLEVEL_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_L2L_MONO, L2L_MONO_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_MONOLEVEL, MONOLEVEL_INIT_VALUE, bInPowerHandler);

       // init GPIO functions
        AIC33WriteReg(AIC33_GPIO1, GPIO1_INIT_VALUE, bInPowerHandler);
        AIC33WriteReg(AIC33_GPIO2, GPIO2_INIT_VALUE, bInPowerHandler);

  AIC33WriteReg(AIC33_CLKGEN, CLKGEN_INIT_VALUE, bInPowerHandler);

  RETAILMSG(1, (TEXT("Done InitAIC33Audio.\r\n")));
}
```

Refer to the file AIC33Regs.H for the actual setup values in the foregoing initialization routine.

4 Installation

This section presents the installation steps to run the AIC33 WinCE 5.0 drivers on Intel MainStone II platform.

Included with the Microsoft™ Windows CE 5.0 platform builder CD ROM is the *Board Support Package* (BSP) of the MainStone II, called \MAINSTONEII\, which may be located in your PC after installing the Platform Builder 5.0 at, for example:

C:\WinCE500\PLATFORM\.

To install the AIC3x Windows CE 5.0 touch and audio drivers into one of MainStone II WorkSpaces, perform the following steps.

4.1 Step I: Copy

1. Copy \AIC3xWinCE5Drivers\AIC3x.cec file into:
   C:\WINCE500\PUBLIC\COMMON\OAK\CATALOG\CEC\
2. Copy all files inside \AIC3xWinCE5Drivers\INC\ into:
   C:\WINCE500\PLATFORM\MAINSTONEII\SRC\INC\
3. Copy the directories **AIC3xLIB** and **AIC3xWaveDev** into:
   C:\WINCE500\PLATFORM\MAINSTONEII\SRC\DRIVERS\

4.2 Step II: Set Up

This step sets up the catalog to include the AIC3x device drivers.

1. Run **Platform Builder 5.0**, and the Platform Builder IDE appears.
2. At the Platform Builder 5.0 IDE, open **Manage Catalog Items** from the menu **File\Manage Catalog Items …**. When the **Manage Catalog Items** window appears, click on **Import** button on the right side of the window, navigate, find, and select **AIC3x.cec** in the directory

   C:\WINCE500\PUBLIC\COMMON\OAK\CATALOG\CEC\,

   and then click on **Open** so that the item is ported in.
3. Click and drag to select all *.cec files in the **Manage Catalog Items** Window, and then click on the **Refresh** button to make sure the new item is loaded.
4. Close the **Manage Catalog Items** window by clicking on its **OK** button.

4.3 Step III: Open

This step, in the Platform Builder 5.0 IDE, opens a new or existing MainStone II workspace per the application. The procedure is ignored here.

4.4 Step IV: Add

This step adds the AIC3x device drivers from the **Catalog** into the existing **OS design**.

1. In the **Catalog** window of the **Platform Builder 5.0 IDE**, find **TI AIC3x Audio CODEC Driver**, right-click on it, and select **Add to OS Design** to add the audio driver to the OS.
2. As a result, the audio device driver should appear under the **Device Drivers** section at the **OSDesignView** window of the WorkSpace.

4.5 Step V: Modify

This step modifies the building device drivers so as to include TI AIC3x drivers.

1. Open the **dirs** file in the directory:

   C:\WINCE500\PLATFORM\MAINSTONEII\SRC\DRIVERS\
2. Add on the **AIC3xLIB** and **AIC3xWAVEDEV**. For example: the **dirs** file could be:

```
DIRS=\
    AIC3xLIB \
    AIC3xWAVEDEV \
# @CESYSGEN IF CE_MODULES_POINTER
#   touch \
# @CESYSGEN ENDIF CE_MODULES_POINTER
# @CESYSGEN IF CE_MODULES_DEVICE
# @CESYSGEN IF CE_MODULES_USBD
    hcd   \
# @CESYSGEN ENDIF CE_MODULES_USBD
# @CESYSGEN IF CE_MODULES_SERIAL
    serial \
# @CESYSGEN ENDIF CE_MODULES_SERIAL
.....
.....
```

3. Save and close the modified **dirs** file.

4.6 Step VI: Update

This step updates the Hardware Specific Files, so that the operating system will use AIC33 device drivers.

1. Open the existing **platform.reg** file from **Hardware Specific** section of the **ParameterView** window of the workspace.
2. Edit the **platform.reg** file such as to delete the old audio **dll** and to add in the AIC33 audio:

```
; -------------------------------------------------------------------------------
; @CESYSGEN IF CE_MODULES_WAVEAPI
IF BSP_NOAUDIO !
```

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WaveDev]
   "Prefix"="WAV"
;    "Dll"="pxa27x_wavedev.dll"
   "Dll"="wavedev.dll"
   "Index"=dword:1
   "Order"=dword:0
   "Priority256"=dword:95
   "Sysintr"=dword:19
...
```

3. Save and close the updated **platform.reg** file.
4. Similarly, edit the **platform.bib** file such as:

```
; -----------------------------------------------------------------------------
; @CESYSGEN IF CE_MODULES_WAVEAPI
IF BSP_NOAUDIO !
;     pxa27x_wavedev.dll     $(_FLATRELEASEDIR)\pxa27x_wavedev.dll    NK  SH
      wavedev.dll     $(_FLATRELEASEDIR)\wavedev.dll    NK  SH
ENDIF BSP_NOAUDIO !
; @CESYSGEN ENDIF CE_MODULES_WAVEAPI
; -----------------------------------------------------------------------------
```

5. Save and close the updated **platform.bib** file.

5 WinCE 5.0 AIC3x Driver Code

To obtain the the WinCE 5.0 AIC3x driver code, contact the TI DAP Application Support Group at e-mail address *dataconvapps@list.ti.com*.

6 References

1. *TSC2301 WinCE Generic Drivers* application report (SLAA187)
2. *TLV320AIC33, Low Power Stereo Audio Codec for Portable Audio/Telephony* data sheet (SLAS480)
3. *TLV320AIC33EVM User's Guide* (SBAU114)
4. Intel PXA27x Processor Developer’s Kit, order number 278827-005