Analog Input and Output

Introduction

To move one step closer to creating real-time DSP applications, we now consider interfacing to the A/D–D/A subsystem that is included with the OMAP-L138 Experimenter. The discussion presented here will focus on using the onboard analog interface circuit (AIC), i.e., TLV320AIC3106 denoted as AIC3106.

Analog I/O Basics

- In order for a DSP to process real world analog signals we need some sort of AIC attached to the DSP.
- An idealized AIC is the following:

\[ x_a(t) \rightarrow \text{Antialiasing Lowpass Filter} \rightarrow f_c \approx f_s / 2 \rightarrow \text{High Precision A/D (14 – 24 Bits)} \rightarrow \text{Interface and Control to DSP Chip} \rightarrow y_a(t) \]

\[ f_c \approx f_s / 2 \]

\[ \text{Sampling Clock} \]

\[ f_s \]

\[ \text{High Precision D/A (14 – 24 Bits)} \]
• As with any lowpass sampling theory system, the analog input must not contain any frequency content above the folding frequency, which in this case is $f_s/2$
  – To prevent aliasing and keep unwanted noise out of the system, an antialiasing filter is typically employed
  – For the case of an ideal antialiasing filter, the cutoff frequency is $f_c = f_s/2$

• The output analog signal as obtained from the D/A is postfiltered with a lowpass reconstruction filter to remove spectral translates that exist at frequencies above $f_s/2$

• Again thinking in terms of an ideal AIC, we would like to have:
  – Software control of the sampling rate, possibly asynchronous sampling rates at the A/D and D/A
  – Software control of the input filter cutoff frequency (low end and high end if a bandpass filter is also available, a selectable dc coupled input for certain control applications
  – Software control of the reconstruction filter cutoff frequency

• A more general AIC should also provide two dedicated inputs and two dedicated outputs, e.g., a stereo audio codec (CODer/DECoder)
  – Most stereo audio codecs are at least 16-bits with sampling rates up to 44.1 or 48 kHz supported
Analog I/O on the C6x

• For the C6x to interface with any form of analog I/O, e.g., A/D and D/A converters, we need to understand a little bit about interrupt data processing and the use of the Multichannel audio serial port (McASP)

• From a programming standpoint, the basics of interrupt service routines (ISRs), and maintaining real-time

• A side issue is the possible use of one of the C6x timers to obtain a programmable sampling rate clock
  – An externally supplied sampling rate clock is also possible

Interrupt Basics

• As the name implies, the processor is halted from its normal processing, and required to execute an ISR

• The interrupt time-line is as follows:
  – Configure: The programmer selects sources and creates an interrupt vector table
  – Enable: The programmer enables individual interrupts and enables the global interrupt
  – CPU: Automatically takes care of seeing if a valid interrupt signal exists, setting a flag bit, and if the interrupt is enabled, branching to the appropriate ISR
  – ISR Routine: Written by the programmer in C or assembly (here we are most interested in using C)
Source Selection on the OMAP-L138

- On the C6x there are a total of 12 user selectable interrupts plus two fixed interrupts, \texttt{RESET}, and a nonmaskable interrupt (\texttt{NMI})

<table>
<thead>
<tr>
<th>Priority</th>
<th>Interrupt Name</th>
<th>Interrupt Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>Reset</td>
<td>Reset</td>
</tr>
<tr>
<td></td>
<td>NMI</td>
<td>Nonmaskable</td>
</tr>
<tr>
<td></td>
<td>INT4</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT5</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT6</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT7</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT8</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT9</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT10</td>
<td>Maskable</td>
</tr>
<tr>
<td>Lowest</td>
<td>INT11</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT12</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT13</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT14</td>
<td>Maskable</td>
</tr>
<tr>
<td></td>
<td>INT15</td>
<td>Maskable</td>
</tr>
</tbody>
</table>

- Note that \texttt{RESET} and \texttt{NMI} have the highest priority
- For more information see Chapter 5 of \texttt{spru733a}, \texttt{CPU} & \texttt{Instruction Set Reference Guide}
Interrupt Service Fetch Packet (ISFP)

- The table itself can be reolated to a different address base exclusive of RESET ISFP

Interrupt Vector Table

- We use vectors.asm to place each vector address on a fetch packet boundary; this creates the interrupt service fetch packet address (ISFP)
- In the ASM programming we did not have need for interrupts, so vectors.asm was empty (could have been), except for the branch to c_int00
We will talk about C ISR’s shortly, but we still need to include branches and NOPs in between appropriate labels, e.g., for the OMAP-L138 on-board AIC3106 codec we use vectors.asm to set up the interrupts:

* Welch, Wright, & Morrow,
  * Real-time Digital Signal Processing, 2011

```assembly
/* File name: vectors.asm */
/* Synopsis: Interrupt vector table for OMAP-L138 DSP */

.ref   _c_int00
.ref   _Codec_ISR

.sect  "vectors"
.nocmp ; do not allow 16 bit instructions to be used in the vector table
         ; so everything remains aligned

RESET_RST:
    MVKL .s2 _c_int00, B0
    MVKH .s2 _c_int00, B0
    B          .s2 B0
    NOP  5

NMI_RST:  b  NMI_RST; stall here if interrupt occurs
    NOP  7

RESV1:  b  RESV1; stall here if interrupt occurs
    NOP  7

RESV2:  b  RESV2; stall here if interrupt occurs
    NOP  7

INT4:  b  INT4; stall here if interrupt occurs
    NOP  7

INT5:  b  INT5; stall here if interrupt occurs
    NOP  7

INT6:  b  INT6; stall here if interrupt occurs
    NOP  7

INT7:  b  INT7; stall here if interrupt occurs
    NOP  7

INT8:  b  INT8; stall here if interrupt occurs
    NOP  7

INT9:  b  INT9; stall here if interrupt occurs
    NOP  7

INT10: b  INT10; stall here if interrupt occurs
```
• We will see later that through the configuration tool, associated with DSP/BIOS, we can have the equivalent of `vectors.asm` written automatically, including the ability to easily change interrupt priorities

### Interrupt Control Registers

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSR</td>
<td>Control status register</td>
<td>Allows you to globally set or disable interrupts</td>
</tr>
<tr>
<td>ICR</td>
<td>Interrupt clear register</td>
<td>Allows you to clear flags in the IFR manually</td>
</tr>
<tr>
<td>IER</td>
<td>Interrupt enable register</td>
<td>Allows you to enable interrupts</td>
</tr>
<tr>
<td>IFR</td>
<td>Interrupt flag register</td>
<td>Shows the status of interrupts</td>
</tr>
<tr>
<td>IRP</td>
<td>Interrupt return pointer register</td>
<td>Contains the return address used on return from a maskable interrupt. This return is accomplished via the B IRP instruction.</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt set register</td>
<td>Allows you to set flags in the IFR manually</td>
</tr>
<tr>
<td>ISTP</td>
<td>Interrupt service table pointer</td>
<td>Pointer to the beginning of the interrupt service table</td>
</tr>
<tr>
<td>NRP</td>
<td>Nonmaskable interrupt return</td>
<td>Contains the return address used on return from a nonmaskable interrupt. This return is accomplished via the B NRP instruction.</td>
</tr>
</tbody>
</table>

• To control the use interrupts we use 8 registers
Enabling Interrupts

- To enable individual interrupts we set bits in the *interrupt enable register* (**IER**), **INT4–INT15**

- Initially we write a ‘1’ to bit zero which disables all interrupts except the non-maskable interrupt (**NMI**), position 0
- Later to enable an interrupt we write a ‘1’ to the desired interrupt and also to non-maskable interrupt enable (**NMIE**), position 1
- The business of **NMI** and **NMIE** is to insure that at boot time
non-maskable interrupts do not get in before the code itself has properly set up the entire interrupt scheme

- The ultimate *master switch* for interrupts is the global interrupt enable (GIE) bit, which is contained in the control status register (CSR)

  - We write a ‘1’ to GIE to globally enable interrupts
  - The PGIE bit is a safety measure to re-enable GIE should an interrupt occur while we are in the process of disabling interrupts (see the TI docs for more info on this)

- We can use C code to tend to the issue of setting IER and GIE via the CSR

- This is particularly easy if we include the header `<C6x.h>`, since this contains all of the definitions for the CPU registers

- In setting up the AIC3106 codec via Morrow’s functions we have a cascade of function calls beginning with *main.c*:

```c
iscrim status register (CSR)

// FILENAME: main.c
// SYNOPSIS: Main program file for demonstration code
```
Chapter 4 • Analog Input and Output

#include "DSP_Config.h"

int main()
{
    // initialize DSP board
    DSP_Init();
    // call StartUp for application specific code
    // defined in each application directory
    StartUp();

    // main stalls here, interrupts drive operation
    while(1) {
        ;
    }

    • From main we call DSP_Init(), which is in the file OMAPL138_Support_DSP.c

    //////////////////////////////////////////////////////////////////////
    // Purpose: Initializes OMAP-L138 DSP and codec
    // Input: None
    // Returns: Nothing
    // Calls: Init_Interrupts, Init_I2C, Init_AIC3106, Init_McASP0, EnableInterrupts
    // Notes: SampleRateSetting is define in OMAPL138_Config.h
    //////////////////////////////////////////////////////////////////////

    void DSP_Init()
    {
        Init_Interrupts();
        Init_I2C();
        if(!Init_AIC3106(SampleRateSetting)) {
            while(1); // stall on codec init failure
        }
        Init_McASP0();
        EnableInterrupts();
    }

    • Among AIC3106 specific actions, interrupts are initialized
    and finally enabled using additional registers
• Three other registers that we have not mentioned are:
  – The interrupt flag register (IFR): Contains the flags the processor sees when interrupts occur, and later clears back to zero; we can also set/reset these bits ourselves
  – The interrupt clear register (ICR): Used to clear specific IFR bits
  – The interrupt set register (ISR): Used to set specific IFR bits

• A summary of interrupt recognition:

![Diagram of interrupt recognition]

• Morrow’s implementation:

```c
void Init_Interrupts()
///////////////////////////////////////////////////////////////////////
// Purpose:  Initializes interrupts
///////////////////////////////////////////////////////////////////////
// Input:    None
///////////////////////////////////////////////////////////////////////
// Returns:  Nothing
///////////////////////////////////////////////////////////////////////
// Calls:    Nothing
///////////////////////////////////////////////////////////////////////
// Notes:    None
///////////////////////////////////////////////////////////////////////
{
    INTCx *intc = INTC0_Base;
}
```
// ISTP = 0x80000000; // set vector table location shared RAM
ISTP = 0x11800000; // set vector table location DSP RAM
intc->intmux3 = 0x0F0E0D00 | 61; // assign McASP0 irq to INT12

void EnableInterrupts()

///////////////////////////////////////////////////////////////////////
// Purpose: Enables McASP interrupt
//
// Input: None
//
// Returns: Nothing
//
// Calls: Nothing
//
// Notes: None
///////////////////////////////////////////////////////////////////////
{
    IER |= 0x1002; // enable McASP interrupt (12)
    ICR = 0xffff; // clear all pending interrupts
    CSR |= 1; // set GIE
}

Interrupt Service Routines in C

Finally, we get to the code that is to run when the ISR fires.

- The basic rule for writing a C ISR is in the function declaration:

  ```c
  interrupt void myISR (void)
  {
  }
  ```

  - The compiler handles register preservation
  - No input arguments (void)
  - No return values (void)
• Within DSP/BIOS we have the hardware interrupt (HWI) dispatcher (more on this when we get to DSP/BIOS), which allows:
  – Nesting of C ISRs
  – Passing arguments to the ISR
• An ISR example taken from Morrow’s codec interface is the following (myFirstProject+):

```c
#include "DSP_Config.h"

// Data is received as 2 16-bit words (left/right) packed into one
// 32-bit word. The union allows the data to be accessed as a single
// entity when transferring to and from the serial port, but still be
// able to manipulate the left and right channels independently.
#define LEFT 0
#define RIGHT 1

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;

interrupt void Codec_ISR()
```

```c
    // Purpose: Codec interface interrupt service routine
```

// Input: None
//
// Returns: Nothing
//
// Calls: CheckForOverrun, ReadCodecData, WriteCodecData
//
// Notes: None
///////////////////////////////////////////////////////////////////////
{
    WriteDigitalOutputs(1); // Used to debug real-time schedule failures via GPIO
    /* add any local variables here */
    float xLeft, xRight, yLeft, yRight;

    if(CheckForOverrun()) // overrun error occurred (i.e. halted DSP)
        return; // so serial port is reset to recover

    CodecDataIn.UINT = ReadCodecData(); // get input data samples

    /* add your code starting here */

    // this example simply copies sample data from in to out
    xLeft = CodecDataIn.Channel[LEFT];
    xRight = CodecDataIn.Channel[RIGHT];

    yLeft = xLeft;
    yRight = xRight;

    CodecDataOut.Channel[LEFT] = yLeft;
    CodecDataOut.Channel[RIGHT] = yRight;

    /* end your code here */

    WriteCodecData(CodecDataOut.UINT); // send output data to port
    WriteDigitalOutputs(0); // Used to debug real-time schedule failures via GPIO
}
The McASP for Serial Port Audio Communication

- To actually communicate with a data producing device, such as an A/D or D/A, requires the use of a serial port on the C6x, i.e., a McASP (C6713 uses the McBSP–Multichannel Buffered Serial Port)

- For details on the McASP see Chapter 23 (p. 1006) of spruh79a, *C6748 Technical Reference Manual*

- The McASP functions as a general-purpose audio serial port optimized for the needs of multichannel audio applications

- The McASP is useful for time-division multiplexed (TDM) stream, Inter-IC Sound (I2S) protocols, and intercomponent digital audio interface transmission (DIT)

- The McASP consists of transmit and receive sections that may operate synchronized, or completely independently with separate master clocks, bit clocks, and frame syncs, and using different transmit modes with different bit-stream formats

- The McASP module also includes up to 16 serializers that can be individually enabled to either transmit or receive

- In addition, all of the McASP pins can be configured as general-purpose input/output (GPIO) pins
A McASP0 has up to 16 serial data pins, $n = 15$
B One of the DSP's external pins, see your device-specific data manual.

- Morrow's McASP init code:

```c
void Init_McASP0()
///////////////////////////////////////////////////////////////////////
// Purpose: Configures McASP to operate with AIC3106 acting as master
// // Input: None
// // Returns: Nothing
// // Calls: Nothing
// // Notes: None
///////////////////////////////////////////////////////////////////////
{
    McASP* mcasp = McASP0_Base;

    // reset McASP
```
mcasp->gblctl = 0;
mcasp->rgblctl = 0;
mcasp->xgblctl = 0;
mcasp->srctl[12] = 0;

wait(10000);

// receiver configuration
mcasp->rmask = 0xffffffff; // no padding used
mcasp->rfmt = 0x000080F8;  // MSB 32bit, 0-delay, no pad, CFGBus
mcasp->afsrctl = 0x00000000; // burst, 1bit, ext fs, rising
mcasp->aclkxctl = 0x00000000; // bit clock from tx side
mcasp->ahclkxctl = 0x00000000; // ext clock (from tx side)
mcasp->rtdm = 0x00000000; // Slot 0
mcasp->rintctl = 0x00000000; // overrun interrupt only
mcasp->rclkchk = 0x00FF0008; // 255-MAX 0-MIN, div-by-256

// transmitter configuration
mcasp->xmask = 0xffffffff; // no padding used
mcasp->xfmt = 0x000080F8;  // MSB 32bit, 0-delay, no pad, CFGBus
mcasp->afsxctl = 0x00000000; // burst, 1bit, ext fs, rising
mcasp->ahclkxctl = 0x00000000; // sync rx clocks
mcasp->ahclkxctl = 0x00000000; // ext clk
mcasp->xtdm = 0x00000000; // slot 0
mcasp->xintctl = 0x00000000; // data interrupts
mcasp->xclkchk = 0x00FF0008; // 255-MAX 0-MIN, div-by-256

// serializer configuration
mcasp->srctl[12] = 0x0000000E; // axr1[12] input
mcasp->pfunc = 0x00000000; // use as mcasp
mcasp->pdir = 0x00000000; // inputs except axr1[11]
mcasp->ditctl = 0x00000000; // not used
mcasp->dibctl = 0x00000000; // not used
mcasp->amute = 0x00000000; // not used

// start clocks
mcasp->rgblctl |= GBLCTL_RHCLKRST;
while(!(mcasp->rgblctl & GBLCTL_RHCLKRST));
mcasp->xgblctl |= GBLCTL_XHCLKRST;
while(!(mcasp->xgblctl & GBLCTL_XHCLKRST));
mcasp->rgblctl |= GBLCTL_RCLKRST;
while(!(mcasp->rgblctl & GBLCTL_RCLKRST));
mcasp->xgblctl |= GBLCTL_XCLKRST;
while(!(mcasp->xgblctl & GBLCTL_XCLKRST));
// clear any pending errors
mcasp->xstat = 0x0000FFFF;
mcasp->rstat = 0x0000FFFF;

// clear serializers
mcasp->rgblctl |= GBLCTL_RSCLR;
while(!(mcasp->rgblctl & GBLCTL_RSCLR));
mcasp->xgbctl |= GBLCTL_XSCLR;
while(!(mcasp->xgbctl & GBLCTL_XSCLR));

// state machine resets
mcasp->rgblctl |= GBLCTL_RSRST;
while(!(mcasp->rgblctl & GBLCTL_RSRST));
mcasp->xgbctl |= GBLCTL_XSRST;
while(!(mcasp->xgbctl & GBLCTL_XSRST));

// prevent output underrun
mcasp->xbuf[11] = 0;

// start frame syncs
mcasp->rgblctl |= GBLCTL_RFRST;
while(!(mcasp->rgblctl & GBLCTL_RFRST));
mcasp->xgbctl |= GBLCTL_XFRST;
while(!(mcasp->xgbctl & GBLCTL_XFRST));

void Init_I2C()
{ 
    I2C *i2c = I2C0_Base;

    i2c->cmdr = 0; // into reset
    i2c->psc = 7; // clock in is 300/4=75MHz, prescaled to 75/8=9.375MHz
    i2c->clk1 = 37; // config for 100kHz I2C clock
    i2c->clkh = 37;
    i2c->cmdr |= ICCMDR_IRS; // out of reset
}
There are a lot of details involved with passing data through the McASP into the AIC3106 and back again (analog I/O)

AIC3106 Stereo Codec

The OMAP-L138 Experimenter includes an AIC3106 stereo audio codec with many advanced features, not found on older DSK’s

FEATURES

- Stereo Audio DAC
  - 102-dBA Signal-to-Noise Ratio
  - 16/20/24/32-Bit Data
  - Supports Rates From 8 kHz to 96 kHz
  - 3D/Bass/Treble/EQ/De-Emphasis Effects
  - Flexible Power Saving Modes and Performance are Available
- Stereo Audio ADC
  - 92-dBA Signal-to-Noise Ratio
  - Supports Rates From 8 kHz to 96 kHz
  - Digital Signal Processing and Noise Filtering Available During Record
- Ten Audio Input Pins
  - Programmable in Single-Ended or Fully Differential Configurations
  - 3-State Capability for Floating Input Configurations
- Seven Audio Output Drivers
  - Stereo Fully Differential or Single-Ended Headphone Drivers
  - Fully Differential Stereo Line Outputs
  - Fully Differential Mono Output
- Low Power: 15-mW Stereo 48-kHz Playback With 3.3-V Analog Supply
- Ultralow-Power Mode with Passive Analog Bypass
- Programmable Input/Output Analog Gains
- Automatic Gain Control (AGC) for Record
- Programmable Microphone Bias Level
- Programmable PLL for Flexible Clock Generation
- Control Bus Selectable SPI or I2C
- Audio Serial Data Bus Supports I²S, Left/Right-Justified, DSP, and TDM Modes
- Alternate Serial PCM/I²S Data Bus for Easy Connection to Bluetooth™ Module
- Concurrent Digital Microphone and Analog Microphone Support Available
- Extensive Modular Power Control
- Power Supplies:
  - Analog: 2.7 V–3.6 V.
  - Digital Core: 1.65 V–1.95 V
  - Digital I/O: 1.1 V–3.6 V
- Packages: 5-mm × 5-mm 80-VFBGA; 7-mm × 7-mm 48-QFN

APPLICATIONS

- Digital Cameras
- Smart Cellular Phones

For more detailed information consult the 102 page TLV320AIC3106 data sheet
• The functional block diagram is shown below:
The advanced capability such as automatic gain control (AGC) is disabled in the present Morrow code, but the controlling registers could be manipulated if needed.

There are also enhancements for playback/output.

We are most interested in the basic frequency response of the input and output channels in the default settings of the Morrow codec library.
Chapter 4 • Analog Input and Output

- The ADC specs, without programmable filter alterations are:

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TEST CONDITIONS</th>
<th>MIN</th>
<th>TYP</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUDIO ADC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input signal level (0-dB)</td>
<td>Single-ended input</td>
<td>0.707</td>
<td></td>
<td></td>
<td>V\text{\textsubscript{RSS}}</td>
</tr>
<tr>
<td>Signal-to-noise ratio, A-weighted(^{(1)})(^{(2)})</td>
<td>(f_s = 48\text{ ksp}, 0\text{-dB PGA gain, inputs ac-shorted to ground})</td>
<td>80</td>
<td>92</td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Dynamic range (^{(3)})</td>
<td>(f_s = 48\text{ ksp}, 0\text{-dB PGA gain, }-60\text{ dB full-scale input signal})</td>
<td>91</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>THD Total harmonic distortion</td>
<td>(f_s = 48\text{ ksp}, 0\text{-dB PGA gain, }-2\text{dB full-scale, 1-kHz input signal})</td>
<td>(-88)</td>
<td>(-70)</td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>PSRR Power supply rejection ratio</td>
<td>217-Hz signal applied to DRVDD</td>
<td>49</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Gain error</td>
<td>1-kHz signal applied to DRVDD</td>
<td>0.84</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Input channel separation</td>
<td>(1\text{-kHz, }-2\text{-dB full-scale signal, MIC3L to MIC3R})</td>
<td>(-86)</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>ADC programmable gain amplifier maximum gain</td>
<td>(1\text{-kHz input tone})</td>
<td>59.5</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>ADC programmable gain amplifier step size</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Input resistance</td>
<td>MIC1L/MIC1R inputs routed to single ADC</td>
<td>20</td>
<td></td>
<td></td>
<td>kΩ</td>
</tr>
<tr>
<td></td>
<td>input mix attenuation = 0 dB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIC1L/MIC1R inputs routed to single ADC, input mix attenuation = 12 dB</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIC2L/MIC2R inputs routed to single ADC</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input mix attenuation = 0 dB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIC2L/MIC2R inputs routed to single ADC, input mix attenuation = 12 dB</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIC3L/MIC3R inputs routed to single ADC</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input mix attenuation = 0 dB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIC3L/MIC3R inputs routed to single ADC, input mix attenuation = 12 dB</td>
<td>80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input level control minimum attenuation setting</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Input level control maximum attenuation setting</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Input signal level</td>
<td>Differential input</td>
<td>1.414</td>
<td></td>
<td></td>
<td>V_{\text{MAX}}</td>
</tr>
<tr>
<td>Signal-to-noise ratio, A-weighted(^{(1)})(^{(2)})</td>
<td>(f_s = 48\text{ ksp}, 0\text{-dB PGA gain, inputs ac-shorted to ground, differential mode})</td>
<td>92</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>THD Total harmonic distortion</td>
<td>(f_s = 48\text{ ksp}, 0\text{-dB PGA gain, }-2\text{dB full-scale 1-kHz input signal, differential mode})</td>
<td>(-91)</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
</tbody>
</table>

### ANALOG PASS THROUGH MODE

| Input to output switch resistance, \(\text{r}_{\text{OUT}}\) | MIC1LINE1 to LINE\_OUT | 330 | | | Ω |
| | MIC2LINE2 to LINE\_OUT | 330 | | | |

### ADC DIGITAL DECIMATION FILTER, \(f_s = 48\text{ kHz}\)

| Filter gain from 0 to 0.39 \(f_s\) | \(±0.1\) | | | | dB |
| Filter gain at 0.4125 \(f_s\) | \(-0.25\) | | | | dB |
| Filter gain at 0.45 \(f_s\) | \(-3\) | | | | dB |
| Filter gain at 0.5 \(f_s\) | \(-17.5\) | | | | dB |
| Filter gain from 0.55 \(f_s\) to 64 \(f_s\) | \(-75\) | | | | dB |
| Filter group delay | 17\(f_s\) | | | | s |

\(^{(1)}\) Ratio of output level with 1-kHz full-scale sine-wave input, to the output level with the inputs shorted, measured A-weighted over a 20-Hz to 20-kHz bandwidth using an audio analyzer.

\(^{(2)}\) All performance measurements done with 20-kHz low-pass filter and, where noted, A-weighted filter. Failure to use such a filter may result in higher THD+N and lower SNR and dynamic range readings than shown in the Electrical Characteristics. The low-pass filter removes out-of-band noise, which, although not audible, may affect dynamic specification values.
- The DAC specs, without programmable filter alterations are:

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TEST CONDITIONS</th>
<th>MIN</th>
<th>TYP</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC DIGITAL INTERPOLATION – FILTER $f_s = 48$ kps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pass band</td>
<td></td>
<td>0</td>
<td>0.45 $f_s$</td>
<td></td>
<td>Hz</td>
</tr>
<tr>
<td>Pass-band ripple</td>
<td></td>
<td>±0.06</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Transition band</td>
<td></td>
<td>0.45 $f_s$</td>
<td>0.55 $f_s$</td>
<td></td>
<td>Hz</td>
</tr>
<tr>
<td>Stop band</td>
<td></td>
<td>0.55 $f_s$</td>
<td>7.5 $f_s$</td>
<td></td>
<td>Hz</td>
</tr>
<tr>
<td>Stop-band attenuation</td>
<td></td>
<td>65</td>
<td></td>
<td></td>
<td>dB</td>
</tr>
<tr>
<td>Group delay</td>
<td></td>
<td>21/$f_s$</td>
<td></td>
<td></td>
<td>s</td>
</tr>
</tbody>
</table>
AIC3106 Portion of the OMAP-L138

Note: AC coupled input and output
AIC3106 on the Board
A Simple Interrupt Loop Program

- The myFirstProject_Plus is an example of a simple interrupt driven program for sample-by-sample processing of analog/audio I/O.
- Will mostly consider this as a black box program, however based on the preceding discussion, some additional knowledge has been obtained about the operating principles.
- The code listings are given below, with descriptions following:

```c
#include "DSP_Config.h"

// Function Prototypes
long int rand_int(void);

// Data is received as 2 16-bit words (left/right) packed into one 32-bit word. The union allows the data to be accessed as a single entity when transferring to and from the serial port, but still be able to manipulate the left and right channels independently.

#define LEFT 0
#define RIGHT 1

volatile union {
    Uint32 UINT;
    Int16 Channel[2];
} CodecDataIn, CodecDataOut;
```
*/ add any global variables here */

interrupt void Codec_ISR()
{
// Purpose: Codec interface interrupt service routine
// Input: None
// Returns: Nothing
// Calls: CheckForOverrun, ReadCodecData, WriteCodecData
// Notes: None

/* add any local variables here */
WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse
float xLeft, xRight, yLeft, yRight;
short k;

if(CheckForOverrun()) // overrun error occurred (i.e. halted DSP)
    return; // so serial port is reset to recover

CodecDataIn.UINT = ReadCodecData(); // get input data samples

/* add your code starting here */
// this example simply copies sample data from in to out
xLeft = CodecDataIn.Channel[ LEFT];
xRight = CodecDataIn.Channel[ RIGHT];
/**************** Input Noise Testing ******************
//Generate left and right noise samples
xLeft = ((short)rand_int())>>2;
xRight = ((short)rand_int())>>2;
/******************* Input Noise Testing *******************/

yLeft = xLeft;
yRight = xRight;

CodecDataOut.Channel[ LEFT] = yLeft;
CodecDataOut.Channel[RIGHT] = yRight;

for(k=0; k<1; k++)
{
    // Idle loop to load processor inside ISR
More Details

- The sampling rate is set by uncommenting/commenting
  #defines from DSP_Config.h

// Welch, Wright, & Morrow,
// Real-time Digital Signal Processing, 2011

#define SampleRateSetting AIC3106Fs48kHz // 48kHz sample rate
#define SampleRateSetting AIC3106Fs96kHz // 96kHz sample rate
#define SampleRateSetting AIC3106Fs32kHz // 32kHz sample rate
#define SampleRateSetting AIC3106Fs24kHz // 24kHz sample rate
#define SampleRateSetting AIC3106Fs16kHz // 16kHz sample rate
#define SampleRateSetting AIC3106Fs12kHz // 12kHz sample rate
#define SampleRateSetting AIC3106Fs8kHz // 8kHz sample rate
Testing the AIC3106 in the Lab

- Make sure the function StartUp() contains GPIO initialization, i.e.,

```c
void StartUp()
{
    InitDigitalOutputs();
}
```

Testing the AIC3106 in the Lab

- A combination of lab hardware and software can be used to take measurements on the performance of the OMAP-L138
  - With the AIC3106 configured to loop signals through we can characterize the ADC/DAC path
  - If we internally generate a white noise sequence, we can characterize just the DAC output path
- In Assignment #3 these options are explored in detail

Noise Capture to Characterize DAC

- A simple measurement of the DAC path can be obtained by generating white noise in software and sending directly to the output, i.e., from ISRs_Plus.c we can set-up the source with a scale factor of 4

```c
...//*********** Input Noise Testing ***************
//Generate left and right noise samples
    xLeft = ((short)rand_int())>>2;
    xRight = ((short)rand_int())>>2;
//************************************************
...```
- We can then use GoldWave™ or a similar audio capture program, e.g., Audacity
- Record in stereo if you want to capture two channels
- Use a sampling rate that exceeds the DAC sampling rate, i.e. for $f_s = 8000$ Hz in the present example, we capture on the Mac with $f_{\text{capture}} = 48$ ksp for 30 s
  - In general we may use the PC with $f_{\text{capture}} = 48$ kHz as well
  - Note: Some notebook systems do not have a line-in readily available except in a docking station, and the mic input is too sensitive
• Once the wave file is captured you can load it as a matrix into MATLAB (double-click the .wav file in the current directory)

• An estimate of the output power spectrum can be obtained using the `simpleSA()` function

• Since the input is a white noise source, the output power spectrum is a measure of the DAC effective magnitude squared transfer function

• In MATLAB we process the matrix `data` as follows:

```matlab
>> [P, F] = simpleSA(data(:,1),2^10,fs);
>> plot(F,10*log10(P/max(P)))
>> grid
>> xlabel('Frequency (Hz)')
>> ylabel('DAC Transfer Function (dB)')
```
If the input levels on the PC are not set properly, say too much gain, an overload occurs.

The PSD of a saturated recorder input will look much different from the expected results.
ISR Timing Using GPIO

- Moving forward the use of the AIC3106 and the inclusion of real-time DSP algorithms will a mainstay
- The source file `ISRs_Plus.c` includes code to write to the GPIO J15, pin 6

```c
/* add any local variables here */
WriteDigitalOutputs(1); // Write to GPIO J15, pin 6; begin ISR timing pulse

// ISR code to read and write to CODEC and DSP algorithms

WriteDigitalOutputs(0); // Write to GPIO J15, pin 6; end ISR timing pulse
```

- By writing a ‘1’ at the beginning of the ISR and ‘0’ at the end, a timing pulse can be viewed on a logic analyzer, which describes the approximate processor utilization
- Embedded in the above ISR is a for loop to make it clear the finite amount of processor resources are available for real-time processing
- The 100 cycle empty loop does use CPU cycles
• Below are logic analyzer screen shots comparing 8 and 48 kHz sampling rates with a loop count of 1 and 100

Fs = 8 ksp, Loop Count = 100

Fs = 48 ksp, Loop Count = 100

Fs = 48 ksp, Loop Count = 1
VC5505 with AIC3204 Codec

- The VC5505 eZdsp contains a TI AIC3204 audio codec

1.1 Features
- Stereo Audio DAC with 100dB SNR
- 4.1mW Stereo 48ksps DAC Playback
- Stereo Audio ADC with 93dB SNR
- 6.1mW Stereo 48ksps ADC Record
- PowerTune™
- Extensive Signal Processing Options
- Six Single-Ended or 3 Fully-Differential Analog Inputs
- Stereo Analog and Digital Microphone Inputs
- Stereo Headphone Outputs
- Stereo Line Outputs
- Very Low-Noise PGA
- Low Power Analog Bypass Mode
- Programmable Microphone Bias
- Programmable PLL

- Integrated LDO
- 5 mm x 5 mm 32-pin QFN Package

1.2 Applications
- Portable Navigation Devices (PND)
- Portable Media Player (PMP)
- Mobile Handsets
- Communication
- Portable Computing

1.3 Description
The TLV320AIC3204 (sometimes referred to as the AIC3204) is a flexible, low-power, low-voltage stereo audio codec with programmable inputs and outputs, PowerTune capabilities, fixed predefined and parameterizable signal processing blocks, integrated PLL, integrated LDOs and flexible digital interfaces.
• For interfacing the VC5505 DSP stick is has an *Inter-IC Sound* (I2S) as one of its chip peripherals
  – “This peripheral allows serial transfer of full duplex streaming data, usually streaming audio, between DSP and an external I2S peripheral such as an audio codec”
  – See TI document sprufp4a.pdf for more details
• The functional block diagram of this interface is shown below

![Functional block diagram of the I2S interface]

• The interface between the DSP and I2S devices is via four
This interface is much different from McASP used on the OMAP-L138 Experimenter.

A full interrupt driven codec library is not yet available for this board.

**Enhanced VC5505 Polling Routine**

- During Spring 2010 an enhanced polling routine was developed for the VC5505.
- Using this code the sampling rate can be set via defines
  - There are still issues to resolve with this code.

```c
#include "aic3204.h"

// Function Prototypes
long int rand_int(void);

/* Sampling Rates:
   * AIC3204_FS_8KHZ
   * AIC3204_FS_16KHZ
   * AIC3204_FS_24KHZ
   * AIC3204_FS_32KHZ
   * AIC3204_FS_48KHZ
*/
```

---

VC5505 with AIC3204 Codec

wires

```
#include "aic3204.h"

// Function Prototypes
long int rand_int(void);

/* Sampling Rates:
   * AIC3204_FS_8KHZ
   * AIC3204_FS_16KHZ
   * AIC3204_FS_24KHZ
   * AIC3204_FS_32KHZ
   * AIC3204_FS_48KHZ
*/
```
* AIC3204_FS_96KHZ
*/
short fs = AIC3204_FS_24KHZ;

void main(void)
{
    //Define working variables
    short i = 0;
    short l_chan, r_chan;
    short buffer[128];

    // Initialize Polling
    comm_poll();

    while(1)
    {
        // Get sample using inputs
        //input_sample(&l_chan, &r_chan);

        // Get random sample
        r_chan = ((short) rand_int())>>3;

        /* Fill buffer */
        buffer[i] = r_chan;
        i += 1;
        if (i == 128) i = 0;

        /* Write Digital audio input */
        output_sample(l_chan, r_chan);
    }
}

//White noise generator for filter noise testing
long int rand_int(void)
{
    static long int a = 100001;

    a = (a*125) % 2796203;
    return a;
}

// aic3204.c
/*
 * ECE5655
 * Sean O'Brien
 */
#include "aic3204.h"
#include "usbstk5505.h"
#include "usbstk5505_gpio.h"
#include "usbstk5505_i2c.h"

// Polling Masks
#define AIC3204_I2C_ADDR 0x18
#define XmitL 0x10
#define XmitR 0x20
#define RcvR 0x08
#define RcvL 0x04

// Function Prototypes
static Int16 AIC3204_rget(Uint16 regnum, Uint16* regval);
static Int16 AIC3204_rset(Uint16 regnum, Uint16 regval);

// Sampling rate variable
extern short fs;

/* _AIC3204_rget(regnum, regval) */
/* Return value of codec register regnum */
/* */
static Int16 AIC3204_rget(Uint16 regnum, Uint16* regval)
{
    Int16 retcode = 0;
    Uint8 cmd[2];

    cmd[0] = regnum & 0x007F; // 7-bit Device Address
    cmd[1] = 0;

    retcode |= USBSTK5505_I2C_write(AIC3204_I2C_ADDR, cmd, 1);
    retcode |= USBSTK5505_I2C_read(AIC3204_I2C_ADDR, cmd, 1);

    *regval = cmd[0];
    USBSTK5505_wait(10);
    return retcode;
}

/* _AIC3204_rset(regnum, regval) */
/* */
* Set codec register `regnum` to value `regval` *
* * *

```c
static Int16 AIC3204_rset(Uint16 regnum, Uint16 regval)
{
    Uint8 cmd[2];
    cmd[0] = regnum & 0x007F;       // 7-bit Device Address
    cmd[1] = regval;               // 8-bit Register Data

    return USBSTK5505_I2C_write(AIC3204_I2C_ADDR, cmd, 2);
}
```

```c
void comm_poll()
{
    /* Define sampling rate parameters */
    unsigned char pll_dh, pll_dl, madc, mdac, nadc, ndac;

    /* Initialize BSL */
    USBSTK5505_init();

    /* Configure Serial Bus */
    SYS_EXBUSSEL |= 0x0100; // Configure Serial bus 0 for I2S0

    /* Set Sampling Rate */
    if(fs == AIC3204_FS_44_1KHZ)
    {
        // PLL D = 560
        pll_dh = 0x02;
        pll_dl = 0x30;

        // MDAC & MACC = 3
        madc = 0x03;
        mdac = 0x03;

        // NDAC & NADC = 5
        nadc = 0x05;
        ndac = 0x05;
    }
    else // 8, 16, 24, 32, 48, 96 KHz
    {
        // PLL D = 1680
        pll_dh = 0x06;
        pll_dl = 0x90;

        switch(fs) {
            case AIC3204_FS_8KHZ:
                madc = 0x0C;
                mdac = 0x0B;
                nadc = 0x05;
                ndac = 0x05;
                break;
        }
    }
```
mdac = 0x0C;
break;
case AIC3204_FS_16KHZ:
    madc = 0x06;
    mdac = 0x06;
    break;
case AIC3204_FS_24KHZ:
    madc = 0x04;
    mdac = 0x04;
    break;
case AIC3204_FS_32KHZ:
    madc = 0x03;
    mdac = 0x03;
    break;
case AIC3204_FS_48KHZ:
    madc = 0x02;
    mdac = 0x02;
    break;
case AIC3204_FS_96KHZ:
    madc = 0x01;
    mdac = 0x01;
    break;
}

// NDAC & NADC = 7
nadc = 0x07;
ndac = 0x07;

/* Configure AIC3204 */
AIC3204_rset( 0, 0 );       // Select page 1
AIC3204_rset( 1, 1 );       // Reset codec
AIC3204_rset( 0, 1 );       // Point to page 1
AIC3204_rset( 1, 8 );       // Disable crude AVDD generation from DVDD
AIC3204_rset( 2, 1 );       // Enable Analog Blocks, use LDO power
AIC3204_rset( 0, 0 );

/* PLL and Clocks config and Power Up */
AIC3204_rset(27, 0x1d);// BCLK and WCLK is set as o/p to AIC3204(Master)
AIC3204_rset(28, 0x00);// Data offset = 0
AIC3204_rset( 4, 3 );      // PLL setting: PLLCLK <- MCLK, CODEC_CLKIN <- PLL CLK
AIC3204_rset( 6, 0x07 );   // Set PLL J
AIC3204_rset(7, pll_dh);// PLL setting: D HI_BYTE(5:0)
AIC3204_rset(8, pll_dl);// PLL setting: D LO_BYTE(7:0)
AIC3204_rset(30, 0x88);// For 32 bit clocks per frame in Master mode ONLY
                        // BCLK=DAC_CLK/N

(12288000/8) = 1.536MHz = 32*fs
AIC3204_rset(5, 0x91);//PLL setting: Power up PLL, P=1 and R=1
AIC3204_rset(13, 0x00);// DOR Hi_Byte(1:0) = 0
AIC3204_rset(14, 0x80);// DOR Lo_Byte(7:0) = 128
AIC3204_rset(20, 0x80);// AOSR (7:0) = 128
AIC3204_rset(11, 0x80 | ndac);// Power up NDAC and set NDAC value
AIC3204_rset(12, 0x80 | mdac);// Power up MDAC and set MDAC value
AIC3204_rset(18, 0x80 | nadc);// Power up NADC and set NADC value
AIC3204_rset(19, 0x80 | madc);// Power up MADC and set MADC value

    /* DAC ROUTING and Power Up */
AIC3204_rset( 0, 1 ); // Select page 1
AIC3204_rset( 0xc, 8 ); // LDAC AFIR routed to HPL
AIC3204_rset( 0xd, 8 ); // RDAC AFIR routed to HPR
AIC3204_rset( 0, 0 );  // Select page 0
AIC3204_rset( 64, 2 ); // Left vol=right vol
AIC3204_rset( 65, 0 ); // Left DAC gain to 0dB VOL; Right tracks Left
AIC3204_rset( 63, 0xd4 ); // Power up left,right data paths and set channel
AIC3204_rset( 0, 1 ); // Select page 1
AIC3204_rset( 0x10, 10 ); // Unmute HPL, 10dB gain
AIC3204_rset( 0x11, 10 ); // Unmute HPR, 10dB gain
AIC3204_rset( 9, 0x30 ); // Power up HPL,HPR
AIC3204_rset( 0, 0 );  // Select page 0
USBSTK5505_wait( 100 ); // wait
/* ADC ROUTING and Power Up */
AIC3204_rset( 0, 1 );  // Select page 1
AIC3204_rset( 0x34, 0x30 ); // STEREO 1 Jack
    // IN2_L to LADC_P through 40 kohm
AIC3204_rset( 0x37, 0x30 ); // IN2_R to RADC_P through 40 kohm
AIC3204_rset( 0x36, 3 );  // CM_1 (common mode) to LADC_M through 40 kohm
AIC3204_rset( 0x39, 0xc0 ); // CM_1 (common mode) to RADC_M through 40 kohm
AIC3204_rset( 0x3b, 0 ); // MIC_PGA_L unmute
AIC3204_rset( 0x3c, 0 ); // MIC_PGA_R unmute
AIC3204_rset( 0, 0 );   // Select page 0
AIC3204_rset( 0x51, 0xc0 ); // Powerup Left and Right ADC
AIC3204_rset( 0x52, 0 ); // Unmute Left and Right ADC
AIC3204_rset( 0, 0 );
USBSTK5505_wait( 100 ); // Wait
/* I2S settings */
I2S0_SRGR = 0x0;
I2S0_CR = 0x8010;   // 16-bit word, slave, enable I2C
I2S0_ICMR = 0x3f;  // Enable interrupts
}

void input_sample(short *l_chan, short *r_chan)
{
    /* Read Digital audio input */
while((RcvR & I2S0_IR) == 0); // Wait for receive interrupt to be pending
*l_chan = I2S0 W0 MSW R;
*r_chan = I2S0 W1 MSW R;
}

void output_sample(short l_chan, short r_chan)
{
    while((XmitR & I2S0_IR) == 0); // Wait for transmit interrupt to be pending
    I2S0_W0_MSW_W = l_chan; // 16 bit left channel transmit audio data
    I2S0_W1_MSW_W = r_chan; // 16 bit right channel transmit audio data
}

• CCS 5.1 Project

Files available in the ZIP package
vc5505_aic3204_polling.zip
on the course Web Site

• A GEL file for this board is available for the target configuration from within the CCS 5.1 directory tree

DAC Noise Characterization with Fs = 16 ksps
• Here we configure the sample program to output noise on the right channel
short fs = AIC3204_FS_16KHZ;

void main(void)
{
    // Define working variables
    short i = 0;
    short l_chan, r_chan;
    short buffer[128];

    // Initialize Polling
    comm_poll();

    while(1)
    {
        // Get sample using inputs
        // input_sample(&l_chan, &r_chan);

        // Get random sample
        r_chan = ((short) rand_int()) >> 3;

        /* Fill buffer */
        buffer[i] = r_chan;
        i += 1;
        if (i == 128) i = 0;

        /* Write Digital audio input */
        output_sample(l_chan, r_chan);
    }
}
VC5505 AIC3204 DAC Noise Characterization

$\text{DAC Transfer Function (dB)}$

$\text{Frequency (Hz)}$

$\text{DAC} = 16 \text{ ksp}$

$\text{~8 kHz}$
A Few Notes on the C5515 eZdsp USB Stick

- Note this codec is the same as the VC5505
- Can we use the same basic codec codec?
  - Testing is under way now to verify this
  - The board support library (BSL) is however different as their are different/additional peripherals on this board
The Board Layout

- USB Connector
- Expansion Connectors
- C5515 DSP
- NOR Flash
- Expansion Connector
- AIC3204
- Stereo In
- Stereo Out
- LEDs
- Display
- Pushbuttons
- Micro SD Connector
- USB Emulation Connector
- Expansion Connector
- Stereo In
- Stereo Out
- Display Connector
Chapter 4 • Analog Input and Output

Enhanced C5515 Polling Routine

• The advanced polling routine developed sp2010 for the vc5505 was modified for use on the C5515 eZdsp USB Stick
• The codec code is unchanged, as the AIC3204 is again used on this board
• The only difference is the BSL library, which is enhanced due to the presence of additional peripherals

// Codec Library Functions
#include "aic3204.h"

// Function Prototypes
long int rand_int(void);

/* Sampling Rates:
 * AIC3204_FS_8KHZ
 * AIC3204_FS_16KHZ
 * AIC3204_FS_24KHZ
 * AIC3204_FS_32KHZ
 * AIC3204_FS_48KHZ
 * AIC3204_FS_96KHZ
 */
short fs = AIC3204_FS_8KHZ;

void main(void)
{
    //Define working variables
    short i = 0;
    short l_chan, r_chan;
    short buffer[128];

    // Initialize Polling
    comm_poll();

    while(1)
    {
        // Get sample using inputs
        //input_sample(&l_chan, &r_chan);

        // Get random sample
        r_chan = ((short) rand_int())>>3;
A Few Notes on the C5515 eZdsp USB Stick

/* Fill buffer */
buffer[i] = r_chan;
i += 1;
if (i == 128) i = 0;

/* Write Digital audio input */
output_sample(l_chan, r_chan);
}
}

// White noise generator for filter noise testing
long int rand_int(void)
{
    static long int a = 100001;
    a = (a * 125) % 2796203;
    return a;
}

// aic3204.c
/*
 * ECE5655
 * Sean O'Brien
 * Modified for C5515 eZdsp by Mark Wickert, March 2012
 */

#include "aic3204.h"
#include "usbstk5515.h"
#include "usbstk5515_gpio.h"
#include "usbstk5515_i2c.h"

// Polling Masks
#define AIC3204_I2C_ADDR 0x18
#define XmitL 0x10
#define XmitR 0x20
#define RcvR 0x08
#define RcvL 0x04

// Function Prototypes
static Int16 AIC3204_rget(Uint16 regnum, Uint16* regval);
static Int16 AIC3204_rset(Uint16 regnum, Uint16 regval);

// Sampling rate variable
extern short fs;
/* _AIC3204_rget(regnum, regval) */
* Return value of codec register regnum
*
*/

static Int16 AIC3204_rget(Uint16 regnum, Uint16* regval)
{
    Int16 retcode = 0;
    Uint8 cmd[2];

    cmd[0] = regnum & 0x007F;       // 7-bit Device Address
    cmd[1] = 0;

    retcode |= USBSTK5515_I2C_write(AIC3204_I2C_ADDR, cmd, 1);
    retcode |= USBSTK5515_I2C_read(AIC3204_I2C_ADDR, cmd, 1);

    *regval = cmd[0];
    USBSTK5515_wait(10);
    return retcode;
}

/* _AIC3204_rset(regnum, regval) */
* Set codec register regnum to value regval
*
*/

static Int16 AIC3204_rset(Uint16 regnum, Uint16 regval)
{
    Uint8 cmd[2];
    cmd[0] = regnum & 0x007F;       // 7-bit Device Address
    cmd[1] = regval;                // 8-bit Register Data

    return USBSTK5515_I2C_write(AIC3204_I2C_ADDR, cmd, 2);
}

void comm_poll()
{
    /* Define sampling rate parameters */
    unsigned char pll_dh, pll_dl, madc, mdac, nadc, ndac;

    /* Initialize BSL */
    USBSTK5515_init();
/* Configure Serial Bus */
SYS_EXBUSSEL |= 0x0100; // Configure Serial bus 0 for I2S0

/*/ Set Sampling Rate */
if(fs == AIC3204_FS_44_1KHZ)
{
   // PLL D = 560
   pll_dh = 0x02;
   pll_dl = 0x30;

   // MDAC & MACC = 3
   madc = 0x03;
   mdac = 0x03;

   // NDAC & NADC = 5
   nadc = 0x05;
   ndac = 0x05;
}
else // 8, 16, 24, 32, 48, 96 KHz
{
   // PLL D = 1680
   pll_dh = 0x06;
   pll_dl = 0x90;

   switch(fs) {
      case AIC3204_FS_8KHZ:
         madc = 0x0C;
         mdac = 0x0C;
         break;
      case AIC3204_FS_16KHZ:
         madc = 0x06;
         mdac = 0x06;
         break;
      case AIC3204_FS_24KHZ:
         madc = 0x04;
         mdac = 0x04;
         break;
      case AIC3204_FS_32KHZ:
         madc = 0x03;
         mdac = 0x03;
         break;
      case AIC3204_FS_48KHZ:
         madc = 0x02;
         mdac = 0x02;
         break;
      case AIC3204_FS_96KHZ:
         madc = 0x01;
mdac = 0x01;
break;
}

// NDAC & NADC = 7
nadc = 0x07;
ndac = 0x07;
}

/* Configure AIC3204 */
AIC3204_rset( 0, 0 );  // Select page 1
AIC3204_rset( 1, 1 );  // Reset codec
AIC3204_rset( 0, 1 );  // Point to page 1
AIC3204_rset( 1, 8 );  // Disable crude AVDD generation from DVDD
AIC3204_rset( 2, 1 );  // Enable Analog Blocks, use LDO power
AIC3204_rset( 0, 0 );

/* PLL and Clocks config and Power Up */
AIC3204_rset(27, 0x1d);  // BCLK and WCLK is set as o/p to AIC3204(Master)
AIC3204_rset(28, 0x00);  // Data offset = 0
AIC3204_rset(4, 3);    // PLL setting: PLLCLK <- MCLK, CODEC_CLKIN <-PLL CLK
AIC3204_rset(6, 0x07);  // Set PLL J
AIC3204_rset(7, pll_dh); // PLL setting: D HI_BYTE(5:0)
AIC3204_rset(8, pll_dl); // PLL setting: D LO_BYTE(7:0)
AIC3204_rset(30, 0x88); // For 32 bit clocks per frame in Master mode ONLY  
                        // BCLK=DAC_CLK/N
=(12288000/8) = 1.536MHz = 32*fs
AIC3204_rset(5, 0x91);  //PLL setting: Power up PLL, P=1 and R=1
AIC3204_rset(13, 0x00); // DOSR Hi_Byte(1:0) = 0
AIC3204_rset(14, 0x80); // DOSR Lo_Byte(7:0) = 128
AIC3204_rset(20, 0x80); // AOSR (7:0) = 128
AIC3204_rset(11, 0x80 | ndac); // Power up NDAC and set NDAC value
AIC3204_rset(12, 0x80 | mdac); // Power up MDAC and set MDAC value
AIC3204_rset(18, 0x80 | nadc); // Power up NADC and set NADC value
AIC3204_rset(19, 0x80 | madc); // Power up MADC and set MADC value

/* DAC ROUTING and Power Up */
AIC3204_rset( 0, 1 );  // Select page 1
AIC3204_rset( 0x0c, 8 ); // LDAC AFIR routed to HPL
AIC3204_rset( 0x0d, 8 ); // RDAC AFIR routed to HPR
AIC3204_rset( 0, 0 );  // Select page 0
AIC3204_rset( 64, 2 ); // Left vol=right vol
AIC3204_rset( 65, 0 ); // Left DAC gain to 0dB VOL; Right tracks Left
AIC3204_rset( 63, 0xd4 ); // Power up left,right data paths and set channel
AIC3204_rset( 0, 1 );  // Select page 1
AIC3204_rset( 0x10, 10 ); // Unmute HPL , 10dB gain
AIC3204_rset( 0x11, 10 ); // Unmute HPR , 10dB gain
A Few Notes on the C5515 eZdsp USB Stick

AIC3204_rset( 9, 0x30 );  // Power up HPL,HPR
AIC3204_rset( 0, 0 );    // Select page 0
USBSTK5515_wait( 100 );  // wait
/* ADC ROUTING and Power Up */
AIC3204_rset( 0, 1 );    // Select page 1
AIC3204_rset( 0x34, 0x30 );  // STEREO 1 Jack
                     // IN2_L to LADC_P through 40 kohm
AIC3204_rset( 0x37, 0x30 ); // CM_1 (common mode) to LADC_M through 40 kohm
AIC3204_rset( 0x39, 0x30 );  // CM_1 (common mode) to RADC_M through 40 kohm
AIC3204_rset( 0x3b, 0 );    // MIC_PGA_L unmute
AIC3204_rset( 0x3c, 0 );    // MIC_PGA_R unmute
AIC3204_rset( 0, 0 );      // Select page 0
AIC3204_rset( 0x51, 0xc0 );  // Powerup Left and Right ADC
AIC3204_rset( 0x52, 0 );    // Unmute Left and Right ADC

void input_sample(short *l_chan, short *r_chan)
{
/* Read Digital audio input */
    while((RcvR & I2S0_IR) == 0); // Wait for receive interrupt to be pending
    *l_chan = I2S0_W0_MSW_R;
    *r_chan = I2S0_W1_MSW_R;
}

void output_sample(short l_chan, short r_chan)
{
    while((XmitR & I2S0_IR) == 0); // Wait for transmit interrupt to be pending
    I2S0_W0_MSW_W = l_chan;     // 16 bit left channel transmit audio data
    I2S0_W1_MSW_W = r_chan;     // 16 bit right channel transmit audio data
Chapter 4 • Analog Input and Output

• CCS 5.1 Project

Files available in the ZIP package
C5515_aic3204_polling.zip
on the course Web Site

• A GEL file for this board is available for the target configuration from within the CCS 5.1 directory tree

DAC Noise Characterization with Fs = 8 ksps
• Here we configure the sample program to output noise on the right channel

short fs = \texttt{AIC3204 FS 8KHZ};

void main(void)
{
    //Define working variables
    short i = 0;
    short l_chan, r_chan;
    short buffer[128];

    // Initialize Polling
    comm_poll();
The observed noise bump is unexplained at this point