

**AP-648**

**APPLICATION  
NOTE**

**USB Audio Using the  
8x930Ax/Hx Controller  
and an Audio Codec**

December, 1997

Order Number: 292206-003

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com>.

Copyright © Intel Corporation 1997. \*Third-party brands and names are the property of their respective owners.

- 1.0 INTRODUCTION ..... 1**
- 2.0 USB and Digital Audio ..... 1**
  - 2.1 Universal Serial Bus ..... 1
  - 2.2 Intel's 8x930Ax/Hx USB Controller ..... 1
  - 2.3 Traditional PC Audio ..... 1
    - 2.3.1 Possible Drawbacks to Using PC Sound Cards ..... 1
  - 2.4 USB Audio ..... 2
- 3.0 Codec Background and Selection ..... 2**
  - 3.1 Codec Basics ..... 2
  - 3.2 ARCHITECTURE CONSIDERATIONS ..... 3
  - 3.3 Performance ..... 3
  - 3.4 Features ..... 3
  - 3.5 Interface Signals ..... 3
    - 3.5.1 Serial ..... 4
    - 3.5.2 Parallel ..... 4
- 4.0 8x930Ax/Hx Interface ..... 4**
  - 4.1 USB to 8x930Ax/Hx Interface ..... 4
  - 4.2 8x930Ax/Hx to Codec Interface ..... 5
- 5.0 8x930Ax/Hx to Codec Interface Example ..... 5**
  - 5.1 HIGH LEVEL SYSTEM CONSIDERATIONS ..... 5
  - 5.2 AD1845 Details ..... 6
  - 5.3 8x930Ax/Hx to AD1845 Interface Example ..... 7
  - 5.4 Considerations and Recommendations ..... 10
    - 5.4.1 USB SOF ..... 10
    - 5.4.2 Data Request ..... 10
    - 5.4.3 Matching Data Rates ..... 10
    - 5.4.4 Offsetting the SOF ..... 10
- 6.0 Conclusion ..... 11**
- 7.0 References ..... 11**
- 8.0 Firmware Template for Communication Between the 8x930Ax/Hx and Audio Codec ..... 12**
  - 8.1 Firmware Guideline ..... 12
    - 8.1.1 Initialize ..... 12
    - 8.1.2 SOF ISR ..... 13
    - 8.1.3 Codec Request ISR ..... 14
  - 8.2 Sample Subroutines ..... 14
- 9.0 Sample Firmware Subroutines for Communication Between the 8x930Ax/Hx and Audio Codec . 15**
- 10.0 Schematics ..... 20**

**Figures**

- 1. Generic Coding and Decoding Block Diagram ..... 2
- 2. 8x930Ax/Hx and AD1845 Block Diagram ..... 7
- 3. AD1845 DMA Timing Requirements ..... 8
- 4. 8x930Ax/Hx Parallel Port Write Timing ..... 8
- 5. Read Timings for AD1845 ..... 8
- 6. Read Timings for 8x930Ax/Hx (1 Wait State) ..... 9
- 7. 8x930Ax/Hx and AD1845 Offset ..... 11

**Tables**

- 1. Sample Wire Map Between 8x930Hx and Audio Codec ..... 12



## 1.0 INTRODUCTION

This application note provides an introduction to interfacing the 8x930Ax/Hx with an audio codec to perform playback of digital audio, presents an overview of USB as it relates to digital audio, and discusses codec background and selection. It also describes 8x930Ax/Hx interfacing, providing an example with design recommendations.

## 2.0 USB and Digital Audio

### 2.1 Universal Serial Bus

The Universal Serial Bus (USB) is an industry standard interconnection bus designed to support a wide range of peripherals around the PC. The USB topology has three elements: host, hubs, and functions. The PC is the host and peripherals are the functions. Peripherals interface to the host PC via USB cables and protocol. To enable a variety of peripherals, the USB protocol defines four transfer types: Control, Isochronous, Interrupt and Bulk.

Every peripheral will need to support control transfers so that configuration and command/status information can flow between the host PC and peripheral. Isochronous transfers provide guaranteed bus access and constant data rate to support CTI (computer-telephone integration) and audio systems. Interrupt transfers are designed to support human input devices such as joysticks, mice, and keyboards. These devices need to communicate small amounts of data infrequently, but with bounded service periods. Bulk transfers are designed for peripherals such as printers and digital cameras. These devices communicate large amounts of data to the PC as bandwidth becomes available.

The USB implements a blocking bandwidth allocation scheme that denies access to a peripheral if the peripheral exceeds the current bandwidth allocation or latency requirements. USB allows up to 90% of the bus bandwidth to be used by isochronous and interrupt transfers. The remaining 10% is reserved for control transfers. Bulk transfers only occur on a bandwidth-available basis.

### 2.2 Intel's 8x930Ax/Hx USB Controller

Intel's 8x930Ax/Hx is a single chip, USB Specification Rev. 1.0 compliant, peripheral controller. The presence of

a rich combination of integrated features makes the 8x930Ax/Hx peripheral controller flexible and powerful. It contains a MCS® 251 microprocessor core, four 8-bit I/O ports, three 16-bit timers, hardware watchdog timer, Programmable Counter Array (PCA), and a Serial I/O port. The purpose of this application note is to provide an introduction on interfacing the 8x930Ax/Hx with an audio codec to perform playback of digital audio.

### 2.3 Traditional PC Audio

There are currently two predominant methods for generating audio using a PC. The first is CD audio. Similar to home stereo compact disk players, the CD-ROM drive reads the compact disk and produces an analog output. This analog signal is inputted to a PC sound card, processed further, and then outputted from the PC using a standard RCA plug.

The second method of generating audio via a PC is performed by reading a file from memory into the sound card. The sound card performs any decoding or processing, converts the digital data to analog, and outputs the analog signal from the PC via the standard RCA plugs. This is the procedure used when a user downloads an audio file or initiates a playback from the hard drive.

#### 2.3.1 Possible Drawbacks to Using PC Sound Cards

There are several drawbacks to using sound cards that install into a PC card slot. The sound card utilizes PC resources like interrupts and ports. It is also an additional cost. Since the sound card has to be inserted into the PC box, reconfiguration is required which can be viewed as difficult by many users.

The interior of a PC box is also noisy with electromagnetic interference (EMI). This may limit the sound quality of 'inside the box' solutions, as there is an analog signal that is sent from the CD-ROM to the sound card. Due to design complexity, sound cards can be limited in channels. This can hamper their scalability and preclude their use in multiple-channel surround sound systems without the addition of processing power to the speakers.

## 2.4 USB Audio

USB, however, lends itself nicely to audio applications because it is ‘outside the box’. The audio data remains digital until it is outside the PC. It is converted back to analog just prior to the speaker amplification circuitry, resulting in improved sound quality. Digital audio typically has higher fidelity, which is obvious in high-end speakers. A USB based solution is also scalable allowing the speaker vendor to increase the quality of audio without the addition of any hardware between the host and the USB controller.

For instance, it is possible to have more than two channels of audio data, enriching the user’s multimedia experience. Similarly, the ease of plug and play makes a USB implementation attractive. The data processing can be done on the host, providing a cost savings to the speaker manufacturer. The user need only buy a pair of USB capable speakers.

USB has also demonstrated the ability to handle the bandwidth required by audio applications. A compact disk has a sample rate of 44.1kHz and 16 bits per sample. Since the USB frame rate is 1ms, the audio peripheral will receive nine frames containing 44 samples and one frame containing 45 samples. In other words, over a time period of 10ms, the peripheral will receive  $9 \times 44 + 45 = 441$  samples, which averages to 44.1 samples per millisecond and equals the sample rate of 44.1kHz.

Assuming two channels of audio data, 10 bytes for protocol overhead, and a worst-case bit stuffing of 16%, the playback system will need to handle  $(45 \times 16 \times 2 + 10) \times 1.16 = 1682$  bits/ms, or approximately 210 bytes/ms. Each USB frame has a data payload of 1024 bytes. Therefore, playback of a stereo audio signal would

consume approximately 20% of the available USB bandwidth. Out of each 210B packet, there will be  $(45 \times 16 \times 2) = 180B$  of raw audio data.

## 3.0 Codec Background and Selection

### 3.1 Codec Basics

Given that the digital-to-analog conversion will occur inside the USB peripheral, it is beneficial to cover the operation of codecs. In general, codecs are used to CODE and/or DECODE data. The term is somewhat general, in that codecs are available for different types of data; such as audio or video. It is also used on occasion to refer to COMPRESSION and DECOMPRESSION, in which the data is encoded according to an algorithm (MPEG for example). For PC audio, the industry accepted data format is linear Pulse Code Modulation (PCM). More specifically, two channels of data (stereo) that is represented by a 16-bit twos complement digital word. Figure 1 shows a general block diagram of coding and decoding.

Coding is done when an analog signal is sampled by the analog-to-digital converter (ADC) and quantized to a digital word (in this example, a 16-bit word with a twos complement format). Depending on the application, digital signal processing (DSP) can then be performed on the digital samples for the purpose of filtering, compression, etc. PCM in its basic form doesn’t use compression, so there is no DSP block for our application. Decoding of the digital samples is done by the digital-to-analog converter (DAC). The output is the reconstructed analog signal. It should be noted that for a two-channel stereo application, there are two separate data paths with duplicate DSP and DAC blocks so that each channel can be controlled individually.

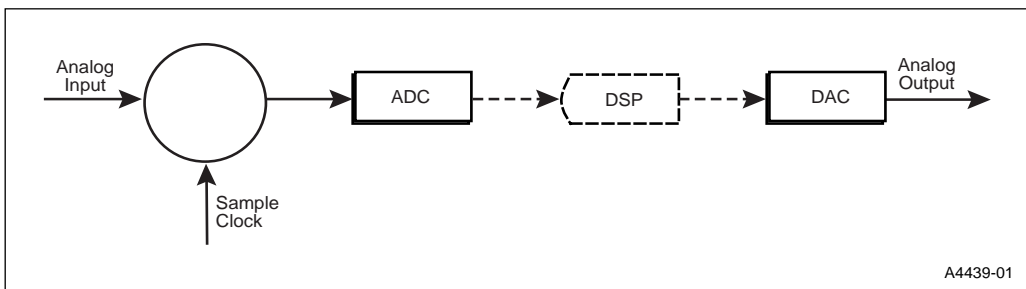


Figure 1. Generic Coding and Decoding Block Diagram

An example of coding is the recording of an audio waveform which entails the first half of the system shown in Figure 1 on page 2. An analog waveform is digitized, compressed if necessary, and then stored on a medium such as a CD. Audio playback is the second half of the system, where the digital words are read from the medium, decompressed if necessary, and then converted from digital to analog and played through a speaker.

### 3.2 ARCHITECTURE CONSIDERATIONS

Several different circuit architectures exist for performing the analog-to-digital and digital-to-analog conversions. For 16-bit digital audio, it appears that sigma delta ADCs and DACs are the industry favorite. As one might expect, there are design trade-offs that have to be made by codec manufacturers. The sigma delta architecture offers the appropriate balance between speed of conversion and resolution of bits.

It is not necessary for the casual user to understand the fine details of how the conversion is done. Conversely, the process shouldn't be considered a black box. Sigma delta converters typically employ integrators, comparators, and digital filters (decimation). The theory of operation is rather involved, but the main principle is that input can be oversampled, threshold approximations can be made, and filtering can be done to extract the output. Since oversampling is used, the codec user should expect to supply a reference frequency substantially higher than the sample rate of the audio data.

### 3.3 Performance

Another main consideration that should be examined is the signal-to-noise ratio (SNR). SNR<sup>1</sup> is used as a figure of merit for ADCs and DACs and represents the 'cleanliness' of the output. It is typically computed via the Fast Fourier Transform (FFT) of the output signal under a pure sinusoidal input. A rule of thumb is that high quality audio requires a SNR of at least 75dB. One can typically find converters with a SNR of 80dB. However, one must understand that the system into which the codec is placed has a great effect on the sound quality.

---

1. SNR is the ratio of the fundamental frequency compared to all the other frequencies at the output.

### 3.4 Features

The codec chip usually includes all the necessary filters and support circuitry used by the converters. They generally do not provide the amplifier circuits that drive the speakers. The output of a typical codec is a line level output (RCA type connectors). Therefore, standard audio amplifier circuitry can be used. Many codecs also output voltage references to help with biasing of the amplifiers. Other features to consider when choosing a codec are: serial or parallel interface, bi-directional transfer capability, and buffering.

Some codecs support a byte-wide parallel transfer while others support a bit-wide serial exchange. As previously mentioned, a 1ms USB frame may contain up to 180 bytes of audio data. Therefore, the USB controller must deliver data at a rate of 180KBps to the codec. The designer of the USB audio system must match the processing needs of the codec to the processing capabilities of the USB controller. If the controller is required to process more than just audio data, it may be beneficial to use a parallel interface. This would increase the controller resources available to other processes.

Bi-directional capability can also increase a codec's cost and complexity. Support for bi-directional transfers means that both coding and decoding can be done; although not simultaneously. Typically, a reconfiguration is necessary to switch between playback and record modes. Bi-directional support will increase the interface complexity as well.

Finally, on chip buffering is a feature to consider. Some codecs provide FIFO buffers to ease the interface and timing requirements to microprocessors. Since data delivery must be guaranteed, the buffer allows some tolerance in servicing the codec's need for data. This is especially important if the controller will process more than one task. The clock rate of the controller will be much higher than the audio sample rate, but the system designer must take precautions to assure that the task switch overhead can be accommodated.

### 3.5 Interface Signals

The number of interface signals required by the codec depends on the number of features and type of interface. Typically, there are two sets of signals connecting the codec and controller: data and control. It is possible to find codecs that require relatively few interface signals,

but this typically means that control and data information is multiplexed. Because multiplexing eases the number of interconnects, it requires more management by the controller and greater attention to timing and coding.

### 3.5.1 Serial

A serial interface will most likely require from 7 to 11 interface signals. The data and control bundles are usually separate. Control information is passed via 3 signals: CDATA, CSHIFT, and CLATCH. CDATA is serial control data. This data is used to configure the codec for the sampling rate, data format, filter programming, master clock frequency, etc. CSHIFT is a clock signal that clocks each bit, serially, into a control buffer. CLATCH is a signal that latches the control word into a register and initiates reconfiguration.

Data is passed in a similar manner by SDATA, SSHIFT, and LRCLK. SDATA is the serial data. SSHIFT is the clock signal that shifts each data bit into a buffer. LRCLK indicates whether the data word is for the left or right channel. For example, 16 bits of left channel information is clocked into the buffer using SSHIFT. A transition on the LRCLK latches the data word and then signals the next 16 bits as right channel data.

The remaining signals vary from codec to codec. There will be at least one chip enable or initialization signal. There could also be separate control signals for features like muting, power management, or transfer direction. Support for bi-directional transfers may require additional interface signals. This sometimes depends on whether the codec becomes a bus master or remains a slave. Keep in mind the number of control signals and the complexity of control logic.

### 3.5.2 Parallel

A parallel interface will require approximately 12 - 20 interface signals. Control and data information is usually multiplexed over the parallel bus. Data information is passed via DATA[7:0], DCLK, and LRCLK. An entire 8-bit word will be latched using the CLK signal. LRCLK may or may not be required to signify separation of left/right data. Control data may also be output onto DATA[7:0]. One or two additional control signals are asserted to notify the codec as to whether the incoming byte is audio data or control information. There will also be a reset or initialization signal.

As with the serial case, the number of interconnects can climb quickly with additional features. Some parallel codecs are capable of interfacing directly to the ISA bus which may complicate the interface due to bus control issues. Likewise, bi-directional support could increase the number of interconnects.

## 4.0 8x930A/Hx Interface

### 4.1 USB to 8x930A/Hx Interface

The implementation of USB on the 8x930A/Hx can be divided into four sections: first-in, first-out (FIFO), Function Interface Unit (FIU), Serial Bus Interface Engine (SIE), and the transceiver.

The 8x930A/Hx has a total of eight FIFOs: four transmit FIFOs and four receive FIFOs. The transmit/receive FIFOs support four function endpoints (0-3). Endpoint 0 is 16 bytes and is dedicated for control transfers. Endpoint 1 is user configurable up to 1024 bytes, and Endpoint 2 and 3 are 16 bytes. Endpoints 1, 2, and 3 can be used for interrupt, isochronous, control, or bulk transfer types.

The transmit and receive FIFOs are circulating FIFOs which support up to two separate data sets of variable sizes and contain byte count registers that access the number of bytes in the data sets. They also have flags that detect a full or empty FIFO and have the capability of retransmitting the current data set.

The FIU, SIE, and transceiver make up the rest of the 8x930A/Hx's USB interface. The transceiver circuitry detects and drives signaling on the USB data lines. The serial bus interface engine is the USB protocol interpreter. It is responsible for bit stuffing/unstuffing and for ensuring that transmissions across USB cables are least significant bit (LSb) first. The FIU controls operation of the FIFOs and monitors the data transaction. The operation of these three units isn't of much concern for the user. However, more information can be found in the *8x930Ax, 8x930Hx Universal Serial Bus User's Manual* (1).

Given the data payload size, audio applications will use Endpoint 1. More specifically, the 1024 byte FIFO will be configured into two 512 byte sections. During transmission of an audio stream, data from frame X will be read into section A. At the start of frame (X+1), data in section A will be valid and incoming data from the host will be read into section B. At the start of frame (X+2),

data in section B will be valid and incoming data from the host will overwrite section A.

Note that the re-transmit feature does not apply to isochronous transfers. Also note that the USB protocol requires least significant bits and least significant bytes first. The SIE takes care of transmitting the least significant bit to most significant bit in a given byte, so the user need not worry about LSB first. It is the user's responsibility, however, to ensure that multi-byte data packets are transmitting least significant bytes (LSB) first.

## 4.2 8x930Ax/Hx to Codec Interface

Once data has been captured from the USB cable into the transceiver FIFOs, there are two methods in which to transfer data from the 8x930Ax/Hx to the codec. The first is through the Serial I/O port. The second is through the system bus (8 bit I/O ports 0 and 2). For isochronous transfers, the 8x930Ax/Hx is configured as a high speed device and runs internally at 12MHz.

The serial I/O port supports communication with modems and other external peripheral devices. It can operate in 3 full duplex asynchronous modes or 1 half duplex synchronous mode. In half duplex mode (mode 0), the clock is output on the TXD pin while data is received and transmitted on the RXD pin. Transfers are controlled by using the Serial Port Control (SCON) and Serial Buff (SBUFF) registers. Similar to the transmit/receive FIFO's, data is transferred LSB first. The user must be aware that a peripheral cycle is not equal to 8x930Ax/Hx's clock cycle. Peripheral cycles run at a rate of  $F_{clk}/6$ . An internal clock rate of 12MHz equates to a 2MHz clock rate on the TXD pin.

Additional interface signals can be controlled using the remaining I/O's on Port 1 and 3. The TI bit in the SCON register can be polled to determine the completion of a data sample transfer and therefore trigger a LRCLK signal. Since the 8x930Ax/Hx has only one Serial I/O port, the transfer of control information needs to be programmed explicitly.

The system bus (ports 0 and 2) is used to communicate with external memory. The external bus supports 16 bits of addressing and 8 bits of data by using two 8 bit ports for the interface. One port is used for 8 bits of address information while the other 8-bit port is multiplexed with address and data. From a codec's perspective, it does not

matter if the 8x930Ax/Hx is configured for page mode or non-page mode.

A codec that uses a parallel interface can be considered a second memory device. The codec may use the control interface to request a data sample or control information. After the controller detects the request, it writes data onto the system bus. A great deal of care must be taken with the address decoding and bus timing. The designer needs to make sure that reads/writes to memory are not acknowledged by the codec and vice versa.

Using the 17- and 18-bit addressing modes of the 8x930Ax/Hx can be useful. An external PAL or logic gate can decode these two signals to generate the codec's enable signals. The transfer of control signals can be done using ports 1 and 3. Similar to the Serial I/O method, the signal transitions on these pins must be programmed explicitly.

## 5.0 8x930Ax/Hx to Codec Interface Example

The following example shows what one would need to implement the playback of digital audio using the 8x930Ax/Hx and an AD1845 Parallel Port Stereo Codec from Analog Devices. This is not a recommendation but only a design application example. Many codecs with parallel interfaces could conform to this example. It is not intended to cover all details of designing a USB audio system, but merely to address some of the issues faced by system designers.

### 5.1 HIGH LEVEL SYSTEM CONSIDERATIONS

One of the goals is to provide a simple overall design for the playback of stereo digital audio data. The serial codecs considered for this example required that data be sent MSb first. As previously discussed, serial output from the 8x930Ax/Hx is LSB first. This requires that bit reordering be done. In theory, one could use bit operations to individually write bits to one of the 8x930Ax/Hx's output pins. However, it is assumed that the timing restrictions and coding overhead are too complex to make this a viable solution.

Another possible solution would be to use the serial port in conjunction with some external components. The maximum peripheral clock rate from the serial port is 2MHz. The audio data rate is 180 bytes/ms = 1.44

Mbits/s. Therefore, 72% of the serial port's clock cycles need to be available for transferring data. The program code and interrupt routines can be streamlined to satisfy this throughput requirement. For instance, the program code could be simplified so that its sole responsibility would be to transfer data from the 8x930Ax/Hx receive FIFO to the serial port. An external FIFO or buffer, capable of reordering the bits in each byte, could be used and a state machine could be designed on a FPGA to control the interface between the FIFO and codec.

This would remove the logic complexity from the 8x930Ax/Hx. The FPGA would be responsible for the codec's configuration and regulate the flow of data to the codec. Care would have to be taken, since the codec and speaker system would essentially run asynchronous to the 8x930Ax/Hx and USB system. The 8x930Ax/Hx would simply pump data out the serial port and into the FIFO. The FPGA would have to monitor the state of the codec and fill level of the FIFO and coordinate the flow of data between the two.

If the serial interface is utilized for other purposes, a parallel codec solution can also be implemented. Although a parallel codec is typically more expensive than a serial codec, some of the additional features can prove beneficial and provide a lower overall cost. For example, it supports a burst type transfer that simplifies handshaking. This helps to ease the latency requirements with the controller. The AD1845 is also a flexible part in terms of system configuration. It can be programmed to accept several data formats and data rates, and is operational with a range of clock rates. Finally, many parallel codecs support bi-directional transfers and power down modes, leaving room for future development.

## 5.2 AD1845 Details

For (16-bit twos) complement PCM data, the AD1845 FIFO considers a 'sample' to be a pair of left/right data samples. Therefore, the on-chip 16-sample FIFO will hold 16 left/right pairs = 64 bytes = 512 bits of audio data. The data payload in each USB frame is 180 bytes, so the AD1845 is capable of holding over a third of the data in a USB frame. Additionally, the AD1845 accepts data which transferred low byte first. It expects LSB left channel, MSB left, LSB right, MSB right. This is the same order that bytes are transmitted by the host, so there is no need for bit or byte reordering.

Addressing for the codec's registers is performed via the ADR1:0 signals. These two bits of input select the Index Address Register (IAR), Indexed Data Register (IDR), Status Register, or PIO Data Register. These four registers are considered to have direct forms of addressing. There are also 32 registers that have indirect addressing. The IAR register holds 4 bits of data that signify the addresses of the indirect registers. The IDR contains data that is to be read/written to the register pointed to by the IAR. In other words, the user selects a particular control register by driving ADR1:0 = 0 and writing the appropriate 8 bits to the parallel port. This loads the IAR with the indirect address. By driving ADR1:0 = 1, the IDR register is selected. The next write to the parallel port loads the data into the particular indirect register that is addressed by the IAR.

Because it is necessary to perform two writes to the data bus and two writes to the ADR signals to update a control register, setup and initialization can become cumbersome. However, the AD1845 supports a Direct Memory Access (DMA) protocol during data transfers. The DMA protocol is similar to a burst type of transfer. If the codec's buffer is not full, the codec requests data. For each request, the controller acknowledges the request and proceeds to output 4 bytes of data (a left/right pair). This type of protocol helps to cut down on the processing overhead. For every interrupt request by the codec, the 8x930Ax/Hx returns 4 bytes of data instead of just one byte.

During playback, the signaling is as follows: The codec requests data by asserting Playback Data Request (PDRQ) high. The controller acknowledges by asserting Playback Data Acknowledge (PDAK) low. The controller then follows by outputting data onto the 8 bit bus. Data is clocked into the codec by the WR#<sup>1</sup> input. WR# is strobed low and data is latched into the codec on the rising edge of WR#. The AD1845 de-asserts PDRQ on the falling edge of the final WR# strobe and the controller de-asserts PDAK after the rising edge of the final WR# strobe. This completes the handshake and the codec will be able to issue another data request.

It should be noted that the ADR inputs are ignored while PDAK is asserted. To stop playback, the user must reset bits in a control register by using IAR and IDR. This must be done in between PDRQ data requests. All PDRQ requests must be acknowledged. If the controller is stopping playback and has loaded the IAR but has not

1. The maximum clock rate for WR# is 6.25 MHz

latched the data into the register, and PDRQ is output by the codec, the controller must acknowledge the request. It must output PDAK and strobe WR# four times. When the codec de-asserts PDRQ, the controller can write to IDR and finish stopping the playback.

The power up initialization of the AD1845 takes approximately 512ms. The IAR reads 80h during initialization and can be polled for a value other than 80h to determine when initialization has completed. This indicates that the controller will have to perform reads from the codec, even if there are no bi-directional transfers. Similarly, when certain parameters are changed during setup, the AD1845 goes through routines that allow filters and other circuitry to settle. The controller must again poll the IAR register.

### 5.3 8x930Ax/Hx to AD1845 Interface Example

A general system block diagram is shown in Figure 2 below. The 8x930Ax/Hx provides 1K of on-chip data memory. All code is executed out of external memory (ROM or RAM) or on-chip ROM. For this design

example, it is assumed that code would be contained on an external ROM chip. The 1K of on-chip memory should suffice for any scratch pad area so there shouldn't be any need to write to external memory.

The first issue to be addressed is the timing of the interface signals. Figure 3 indicates the timing required by the AD1845 and Figure 4 indicates the default timings of the 8x930Ax/Hx parallel port. All timings are in ns.

As one can see, the setup and hold of the data byte with respect to WR# is 10ns; 15ns for the AD1845. The equivalent times for the 8x930Ax/Hx are 68ns and 28ns, indicating that all setup and hold times are compatible.

However, there is a discrepancy in the width of the write pulse. The AD1845 requires a minimum width of 100ns but the 8x930Ax/Hx default configuration only provides 71.8ns. Therefore, the 8x930Ax/Hx must be configured with an extra wait state. This will increase the WR# width to 154ns. Since the program is executing out of ROM, the 8x930Ax/Hx WR# signal is connected only to the codec.

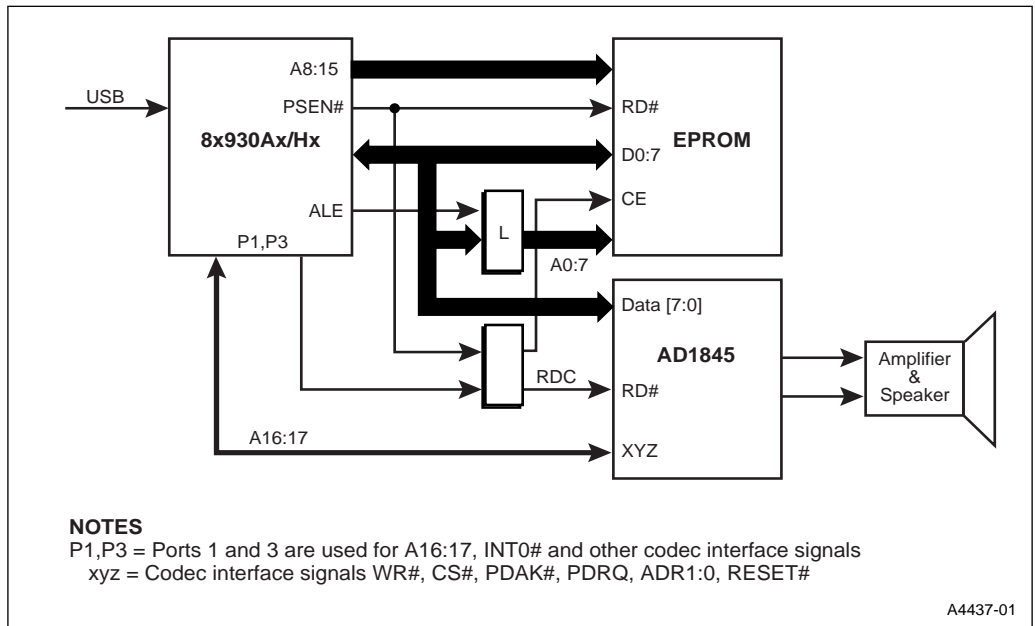


Figure 2. 8x930Ax/Hx and AD1845 Block Diagram

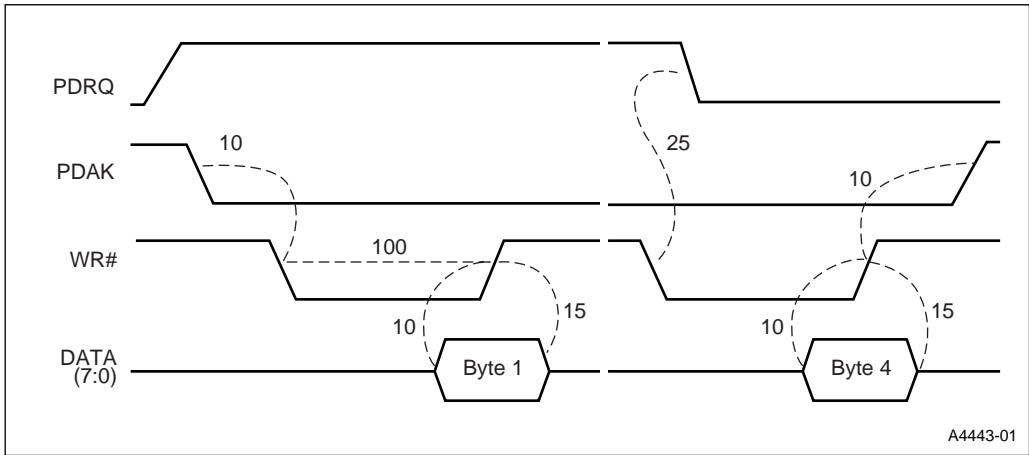


Figure 3. AD1845 DMA Timing Requirements

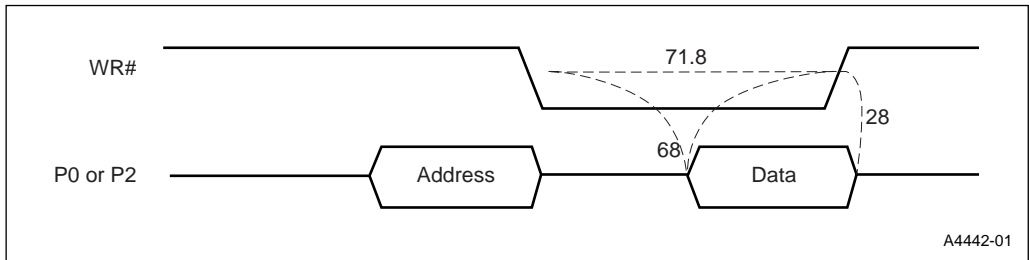


Figure 4. 8x930Ax/Hx Parallel Port Write Timing

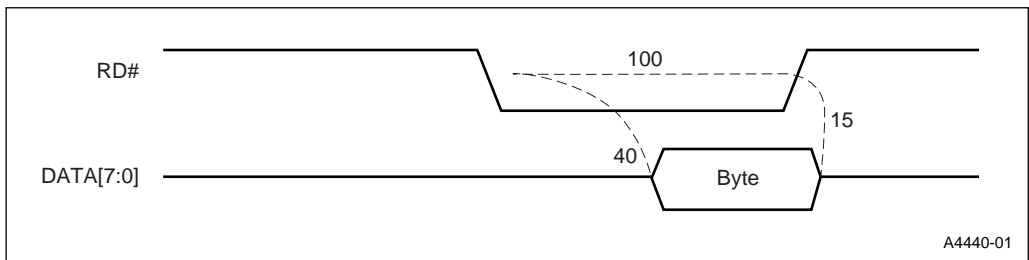
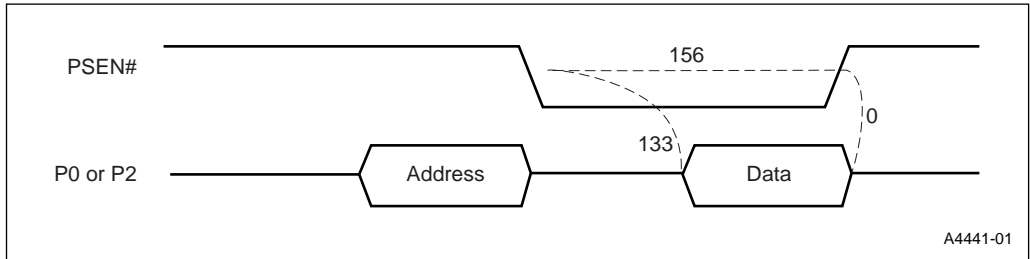


Figure 5. Read Timings for AD1845



**Figure 6. Read Timings for 8x930Ax/Hx (1 Wait State)**

Reading from the codec presents some additional problems since there must be a way of determining if the controller is reading from the memory chip or the codec. One way to do this is to assign the codec to a memory address or memory block. Any read from that particular location or block is ignored by the external ROM.

The 8x930Ax/Hx provides for 16-, 17-, or 18-bit addressing. Under 18-bit addressing, outputs A16 and A17 (pins P1.7 and P3.7) select one of four 64KB memory pages (00, 01, FE, and FF). The program code executing out of ROM should easily fit inside a 64KB block. This program code can reside in page FF. By defining the codec as page 00, one can decode A16 and A17 and generate the codec's read signal. Namely, O-Ring A16, A17, and PSEN# will produce a read signal for the codec. Required read timings are shown above in Figure 5 and Figure 6. All timings are in ns.

When addressing the codec, the 8 bits of address data sent out onto the system bus by the 8x930Ax/Hx are irrelevant. With 1 wait state, the 8x930Ax/Hx requires valid data to be on the bus within 133ns after applying PSEN#. The AD1845 can deliver the data being read in 40ns, so the decoding logic has roughly 90ns to generate the codec's read signal. Note that all reads, including the ones to the external ROM chip, will have 1 wait state. This will degrade program execution to some degree, as there will be an extra cycle of latency from ROM reads.

## 5.4 Considerations and Recommendations

### 5.4.1 USB SOF

Although bandwidth allotment for the isochronous data is guaranteed, there is no guarantee regarding the exact placement of the isochronous data inside a particular frame. Furthermore, the exact byte count of raw data will also vary. Since the placement and exact size of the data packet can vary from frame to frame, the system designer must use the Start of Frame (SOF) interrupt and RXCNT register to determine when new data is valid and how much data has been received.

As previously mentioned, the endpoint 1 receive FIFO is 1024 bytes and is configured as two 512 byte FIFOs. Data in each of the FIFOs is qualified by the SOF. In other words, a SOF tells the user that a new frame has started and that the isochronous data received in the previous frame must be valid. The SOFH register contains the Any Start of Frame (ASOF) bit which indicates that a SOF has been received. This can be set to generate an interrupt. The interrupt routine can then poll the RXCNT register, which contains the byte count for the data packet which is received.

### 5.4.2 Data Request

The 8x930Ax/Hx's external interrupts (INT0# and INT1#) can be used to detect the AD1845's request for data (PDRQ signal). Edge triggered interrupts must be a high-to-low transition and stay low for at least 5 state times (666ns). The PDRQ signal is active high, so it must be inverted. This can be absorbed into the PAL logic that contains the WR# and RD# logic. One must realize that PDRQ has a minimum de-assertion time of 320ns between requests. This should not present a problem, but rather indicate the degree of turnaround that may be necessary when the codec's buffer is not full.

The RXDAT register contains the byte which is currently active in the receive FIFO. FIFO pointers are incremented automatically after each read so the user need only be concerned with moving data from RXDAT to the output port. Data transmission can be done using a loop. The interior of the loop detects PDRQ, outputs PDAK, and performs 4 moves from RXDAT to the output port. RXCNT is the number of bytes received in a particular frame and signifies the loop exit point.

### 5.4.3 Matching Data Rates

The delivery of isochronous data via discrete USB packets presents some other issues that should be addressed. On average, the delivery rate and consumption rate are the same (44.1KHz). However, there is no phase synchronization between the USB clock and the codec's clock. Therefore, there will be phase misalignment. Clock jitter and phase drift due to component variations will cause a temporary mismatch in delivery and consumption rates.

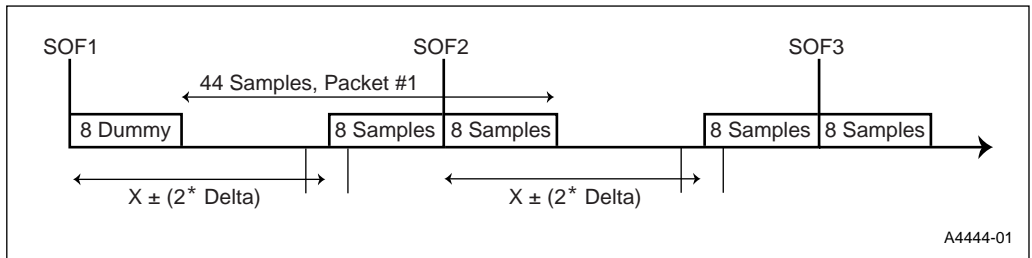
For instance, imagine that 44 samples are delivered in a USB frame. This data must feed the codec for the next 1ms. Since the codec's sampling clock isn't synchronized, it could go through 45 cycles in the 1ms time period. Similarly, the USB frame may deliver 45 samples, and the codec may only request 44.

Since delivery and consumption could be mismatched during any 1ms time period, one frame may suffer from data starvation while the next frame has extra samples that aren't consumed. The system designer must determine how these issues will be handled. Many codecs will replay the last sample if no new data is available. One can easily drop samples by moving to the next receive FIFO when a SOF interrupt is received. Any data that was received in the previous frame and not transferred to the codec is simply not used. This would be the simplest method of handling too much / too little data. Of course, adding or dropping samples without any interpolation or averaging will degrade performance.

### 5.4.4 Offsetting the SOF

The problem with frame starvation lies in the fact that there will be contention at the end of each 1ms time period (end of each USB frame). The 8x930Ax/Hx receives 44 or 45 samples in a data packet and must spread that data over a 1ms time period. This will instigate a race between arrival of the next USB frame and emptying of the codec's buffer.

For example, assume the 8x930Ax/Hx transmits the last byte of data from a packet and fills the AD1845 buffer. The AD1845 will deplete a sample from the buffer and assert PDRQ. The 8x930Ax/Hx will not be able to respond. The AD1845 will continue to deplete samples from the buffer while the 8x930Ax/Hx waits for the next packet of data.



**Figure 7. 8x930Ax/Hx and AD1845 Offset**

A solution for this is to offset the arrival of a USB frame by emptying the AD1845 buffer. This can be accomplished by outputting 8 dummy samples at the start of the isochronous transfer. See Figure 7 above.

When the isochronous transfer is first started, 8 samples worth of dummy data can be sent to the codec (32 bytes). This will delay consumption of the real audio data. The arrival of the next USB frame will occur when the codec's buffer is half empty instead of completely empty.

Additional stewardship will be necessary. Phase mismatch will still exist and samples will have to be padded or dropped. To do this, the system designer will have to determine a point in time when the last sample should be transmitted to the codec. This can be called point X, and a delta around point X can be defined.

A timer count can be used to roughly determine the amount of time that has elapsed since a SOF. If the loop variable hasn't reached RXCNT before the timer count (count + delta), then a sample needs to be dropped. This keeps the offset centered at 8 samples. Likewise, if the loop variable reaches RXCNT before a timer value of (count - delta), then the codec is running fast requiring the addition of an extra sample. Controlling the offset also buys the system designer some time in handling the SOF interrupts. While the 8x930Ax/Hx is attending to the new SOF and new data packet, the codec is coasting on the last 8 samples in its buffer.

## 6.0 Conclusion

As one may conclude, the playback of digital audio data via USB has many advantages. The quality of sound delivered to the end user can be improved while maintaining cost effectiveness and ease of use. The variety of audio codecs and the numerous integrated features of the 8x930Ax/Hx controller make it well suited for codec interfacing. Additionally, the transmission of audio data is well within the capabilities of the USB protocol and the 8x930Ax/Hx controller, allowing future development.

## 7.0 References

- (1) 8x930Ax, 8x930Hx *USB Microcontroller User's Manual* - order number: 272949-001.

## 8.0 Firmware Template for Communication Between the 8x930Ax/Hx and Audio Codec

The purpose of this template is to guide the user in developing the communication code between the 8x930Ax/Hx and the AD1845. Intel assumes no liability whatsoever for use of this template.

Table 1 below indicates the example wire map between the 8x930Hx and the audio codec. Please refer to Section 10.0, Schematics for more details

**Table 1. Sample Wire Map Between 8x930Hx and Audio Codec**

Codec	8x930Hx
Data 7:0	P2: 0:7
WR#	P3.6(WR#)
RD#	P3.7 OR PSEN#
CS#	Ground
PDAK#	P1.0(T2)
PDRQ	P1.3(CEX0)
ADDR 1:0	ADDR 1 to GND and ADDR 0 to P1.1 (T2EX)
RESET#	Tied to Vcc on the codec with an RC timed for the necessary delay required at power up; tied to pin P1.5(CEX2) on the 8x930Hx
PWRDWN#	Vcc
CDACK#	Vcc

The template has been divided into three portions:

1. Initialization
2. Start Of Frame (SOF) ISR
3. CODEC Request ISR

Sample subroutines are provided in Section 8.2 below.

## 8.1 Firmware Guideline

### 8.1.1 Initialize

1. The 8x930Hx powers on and the 8x930Ax/Hx evaluation board initializes.
  - Set the PCA to positive edge triggered interrupts on CEX0
  - Set the SOF interrupt to a priority level above the PCA interrupt
2. Assume
  - The 8x930Hx is in PAGE-MODE for memory bus access.
  - There is an additional state inserted into the memory bus access. The 8x930Hx WR# strobe is lengthened by one state length. This is one way of allowing the bus to sync up with codec's read and write cycle.

3. Initialize the codec with the 8x930Hx
  - Resynch subroutine used at power on
  - Set MODE2 bit
    - Indir\_cntrl\_reg\_write 01001100B, 11011010B
  - Set Crystal Clock select to 25 Mhz
    - 25 Mhz is an arbitrary frequency, other possibilities are: 24.576, 1431818, 24, and 33 Mhz
    - indir\_cntrl\_reg\_write 01011101B, 01100000B
  - Resynch subroutine used again as the codec resets itself
  - Set codec for 16-bit PCM 44.1 Khz sample rate
    - Indir\_cntrl\_reg\_write 01001000B, 01011011B
  - Set codec for DMA transfers (write directly to the codec FIFO), Single channel DMA, Playback DISABLED (i.e. wait till data available from host)
    - Indir\_cntrl\_reg\_write 01001001B, 00000100B
  - Clear MCE Bit (*note: there will be 128 samples of muted output now*)
    - Cntrl\_reg\_write 0B 00000000B

### 8.1.2 SOF ISR

Four possible situations have been considered:

1. There is audio data in the FIFO (check RXCON bit RXFFRC) and currently no playback by the codec
  - Transfer all audio data in the FIFO to the RAM buffer in on chip memory
  - Use the byte count from RXCNT
  - Data goes from RXDAT to RAM buffer one byte at a time
  - When transfer is complete update the ram buffer's data pointer
  - Turn on the playback of the codec
  - Indir\_Cntrl\_reg\_write 00001001B, 00000101B
  - The above write enables codec DMA transfer(write directly to the codec FIFO), Single Channel DMA, Playback ENABLED
  - Set a global variable "HubSamples" = the samples in the RAM buffer i.e. RXCNT / 4
  - Set a global variable "playback" to TRUE
2. There is audio data in the FIFO and playback is occurring
  - Transfer all audio data in the FIFO to the RAM buffer in on chip memory as in a)
  - Update "HubSamples"
  - Update the data pointer in the ram buffer
3. There is no audio data in the FIFO and no playback by the codec
  - Do nothing
4. There is no new audio data in the FIFO and playback is occurring

- if "HubSamples" = 0 then turn playback by the codec off and set the global variable "playback" to FALSE
- This means that while there is no data in the FIFO there is still data needed for playback in the audio buffer
  - To "turn playback by the codec off"

*NOTE: the codec cannot request data (i.e. PDRQ HI) when trying to write to codec registers.*

- Drive PDAK# low 10 ns before beginning a write to the codec
- Write all four bytes of a "dummy" sample (i.e. all zeros) to the codec
- Drive PDAK# high 10 ns after the write cycle has ended
- Indir\_Cntrl\_write\_reg 00001001B, 00000100B (DMAtransfers, SingleChannelDMA, Playback DISABLED)

### 8.1.3 Codec Request ISR

Interrupt comes from the PCA:

- if "HubSamples" > 0 then service the request with "real" data
- Drive PDAK# low 10 ns before beginning write to the codec
- Write all four bytes to the codec from the ram buffer
- Drive PDAK# high 10 ns after the end of the write
- Update the data pointer to the RAM buffer
- Decrement "HubSamples"

Otherwise, if "HubSamples" = 0 then do nothing and let the codec make use of its feature to replay the last sample.

## 8.2 Sample Subroutines

**cntrl\_reg\_read ADDRESS, TARGET REG{**

Reads from the directly addressable registers on the codec 10 ns before the memory read to the codec the ADDRESS bit is transmitted to the codec over P1.1 and held stable until 10 ns after the read strobe returns high. This fulfills the timing requirements of the codec.

}

*NOTE: Only the first two registers of the four direct registers on the AD1845 are relevant to this application. Pins are conserved on the 8x930Hx by tying the upper bit of ADDR1:0 on the codec to ground. TARGET REG is simply the byte register where the data will be stored.*

**cntrl\_reg\_write ADDRESS, DATA{**

Writes to the directly addressable registers on the codec 10 ns before the memory write to the codec the two bits of ADDRESS are transmitted to the codec over P1.2:1 and held stable until 10 ns after the write strobe returns high. This fulfills the timing requirements of the codec.

}

**resynch{**

Waits for the codec to reset itself

using function `cntrl_reg_read`

poll control register 00B on the codec until it reads 80H

```
}
```

**`indir_cntrl_reg_write ADDRESS, DATA{`**

Writes to the indirectly addressable registers on the codec.

`cntrl_reg_write 0B, ADDRESS // indirect register to write`

`cntrl_reg_write 1B, DATA // the upper bit of the direct register address is always 0B`

```
}
```

## 9.0 Sample Firmware Subroutines for Communication Between the 8x930Ax/Hx and Audio Codec

This sample code is meant to assist the user in developing their communication code between the 8x930Ax/Hx and the AD1845. This is to be used only as a reference and has not been thoroughly tested. Intel assumes no liability whatsoever for use of this code.

## Sample Subroutine code

InitilizeEmbeddedFunction:

```
    Call    INIT_VARIABLES      ;Initialize the RAM space as required
    Call    SV_ResetRoutine
    Call    INIT_FUNCTION_EP0

Call INIT_CODEC

    setb    SOFIE                ; Enable SOF Interrupts
    setb    IEN1.1              ; Enable Function ISR
    ret

; Initialize the USB subsystem
;    lcall   INIT_USERS_CODE     ; Call to Users code for initialization

*****
;Sub Routine Name:      INIT_CODEC
;Brief Description:    Initializes the codec for interface w/ the 8x930Ax/Hx
;Registers Saved:      Standard pushing and popping of registers used
*****

INIT_CODEC:
*****
; Not directly related to the codec but initialize global variables hubsamples and
; playback
; Also configures the PCA to receive data request interrupts from thecodec

    push   WR0

    mov    WR0,#0000h
    mov    playback,WR0
    mov    hubsamples,WR0

    pop    WR0

    mov    CCAPM0,#21h          ; Positive edge trigger and enable interrupt request for
CEX0     mov    IPH1,#01h        ; Bumps SOF interrupt from level 0 to level 1 priority.
    Ensures that                ; SOF has higher priority than Codec data
    request

*****

    come   Resynch              ; make sure that the codec has
                                ; out of reset
                                ; Set the MODE2 BIT
                                ; Set Crystal Clock to
25 Mhz  IndirectCodecControlRegWrite MACRO Addr, Dat
    IndirectCodecControlRegWrite 01001100b,11011010b          ; Set Crystal Clock to

    IndirectCodecControlRegWrite 01011101b,01100000b          ; resynch again to allow the
    codec                                                       ; to reset
    Resynch                                                       ; Set codec for 16-bit PCM
    44.1                                                         ; Khz sample rate
    IndirectCodecControlRegWrite 01001000b,01011011b          ; Set Codec for dma transfers
                                                                ; Single Channel DMA transfers

    IndirectCodecControlRegWrite 01001001b,00000100b          ; Playback DISABLED
    CodecControlRegWrite00 00000000b                          ; Clear the MC3 Bit

    ret

;-----
*****
;                               SOF ISR
;
;-----
COMMENT *-----
Function name      : SOF_ISR
Brief Description  : Service the SOF_ISR Interrupt
                  : This routine simply displays the upper three bytes of the
                  : SOF in the lower three bits of theLEDs on LED_PORT = P1.
                  : It does not affect the other LEDS. Very useful
                  : in determining when the function is receivingSOFs
                  : The service routine also transfers audio data to
                  : a ram buffer for use by thecodec
Regs preserved    : Reg. A is saved
-----*
SCOPE
```

## Sample Subroutine code

```
SOF_ISR:
EmbeddedFunctionSofRoutine:

HUB.   jnb     ASOF,   ExitSofIsr           ; If this ASOF bit not set, the ISR could be a
Go Check.
      call    SV_SOF_ROUTINE
      clr     ASOF
      push   ACC
      IF FUB_BOARD == DISABLED
          mov  A,     LED_PORT
          anl  A,     #0F8h
          mov  LED_PORT, A
          mov  A,     SOFH
          anl  A,     #07h
          orl  LED_PORT, A
      ENDIF
      pop    ACC
ExitSofIsr:

COMMENT *-----
-----
      SOF Service Routine for the Codec
-----*
      push R0                ; Save registers
      push WR2
      mov A,RXCON            ; move RXCON to get at the RXFFRC bit
      anl A,#10h            ; mask everything except RXFFRC
      jnz dat_yes           ; if (data in fifo)
      mov A,playback        ; if (playback == FALSE)
      anl A,#0FFh
      jz bye                ; then (do nothing)
      mov WR2,hubsamples    ; elseif (hubsamples != 0)
      cmp WR2,#0000h        ; i.e. there are samples in the RAM buffer
      jne bye              ; then (do nothing)
      lcall STOP_PLAY_BACK ; else (turn off the codec)
      mov playback,#0000h   ; set playback = FALSE
      sjmp bye              ; we're done!

dat_yes:
      mov A,playback
      anl A,#0FFh          ; if (playback == FALSE)
      jz turn_on           ; then (turn on the codec and transfer data to ram
buffer)
      lcall XFER_DATA      ; else (only transfer data to the ram buffer)
      sjmp bye              ; we're done

turn_on:
      lcall XFER_DATA
      lcall START_PLAY_BACK
      sjmp bye              ; we're done

bye:
      pop WR2
      pop R0

      ret

COMMENT *-----
-----
Function name      : START_PLAY_BACK
Brief Description  : The global playback function is set to TRUE and
                   : the codec is set to playback audio data
Regs preserved    : N/A
-----*

START_PLAY_BACK:
      mov playback,#0FFh   ; set playback = TRUE
      IndirectCodecControlRegWrite 00001001B,00000101B
      ; the above bit sequence ENABLES playback by thecodec

      ret

;***** END START_PLAY_BACK *****

COMMENT *-----
-----
Function name      : STOP_PLAY_BACK
Brief Description  : Playback by the codec is disabled by first
                   : satisfying the codec's data req and then turning
```

## Sample Subroutine code

```
                : playback off.
Regs preserved  : DR4 is preserved
-----*
-----*

STOP_PLAY_BACK:
    push DR4                ; save regs
    mov WR4,#0000h ; move codec addr into double word
    mov WR6,#0000h
    clr 1.0                ; drop PDAK# LOW
    mov @DR4,WR4           ; move two "dummy" samples to the codec
    mov @DR4,WR4           ; i.e. all zeros
    setb 1.0               ; drive PDAK# HIGH

    IndirectCodecControlRegWrite 00001001B,00000100B
    ; the above bit sequence DISABLES playback by thecodec

    pop DR4

    ret
;***** END STOP_PLAY_BACK *****

COMMENT *-----*
-----*
Function name      : XFER_DATA
Brief Description  : Transfer ISOC data from the eplfifo to a ram buffer.
                   : In the process hubsamples is updated.
Regs preserved    : WR2, WR4, R6
-----*
-----*

XFER_DATA:
    push WR2                ; save registers
    push WR4
    push R6

    mov WR2,#0000h         ; zero the register
    mov WR2,RXCNTL
    srl WR2                ; divide byte count by 4 to get the sample count
    srl WR2
    mov hubsamples,WR2    ; write back the new value of hubsamples
    mov WR4,audio_buf     ; initialize the audio buffer pointer

move_it:
    ; begin loop
    cmp WR2,#0000h        ; if (sample count == 0)
    je see_ya             ; then (we are done)
    mov R6,RXDAT          ; begin moving 1 sample to the RAM buffer
    mov @WR4,R6
    inc WR4,#01d
    mov R6,RXDAT          ; get the audio data from fifo
    mov @WR4,R6           ; transfer audio data to the RAM buffer
    inc WR4,#01d          ; increment the buffer pointer
    mov R6,RXDAT
    mov @WR4,R6
    inc WR4,#01d
    mov R6,RXDAT
    mov @WR4,R6           ; end move 1 sample to the RAM buffer
    inc WR4,#01d
    dec WR2,#01d          ; one sample down (WR2) to go
    sjmp move_it          ; end loop
    mov audio_data_ptr,audio_buf ; reset the pointer to the beginning of the buffer

and new data
see_ya:
    pop R6
    pop WR4
    pop WR2

    ret
;***** END XFER_DATA *****

COMMENT *-----*
-----*
Function name      : CODEC_ISR
Brief Description  : Services the Codec's request for data one sample at a time
Regs preserved    : WR0,R2-5,WR6,DR12
-----*
-----*
*

CODEC_ISR:
```

## Sample Subroutine code

```
    push WR0          ;preserve register values
    push R2
    push R3
    push R4
    push R5
    push WR6
    push DR12

    mov WR0,hubsamples    ; if (hubsamples == 0)
    cmp WR0,#0000h        ; then exit
    je finis              ; i.e. there is no audio data
                          ; so let the codec repeat last sample

; set up the data for the entire sample
    mov R2,audio_data_ptr ; left low byte
    mov R3,audio_data_ptr + 1; left high byte
    mov R4,audio_data_ptr + 2; right low byte
    mov R5,audio_data_ptr + 3; right high byte
    mov WR12,#0000h
    mov WR14,#0000h        ; load the address for the codec into DR12
    clr P1.0              ; drop PDAK# low
    mov @DR12,R2          ; left low byte to codec
    mov @DR12,R3          ; left high byte to codec
    mov @DR12,R4          ; right low byte to codec
    mov @DR12,R5          ; right high byte to codec
    setb P1.0             ; drive PDAK# back high

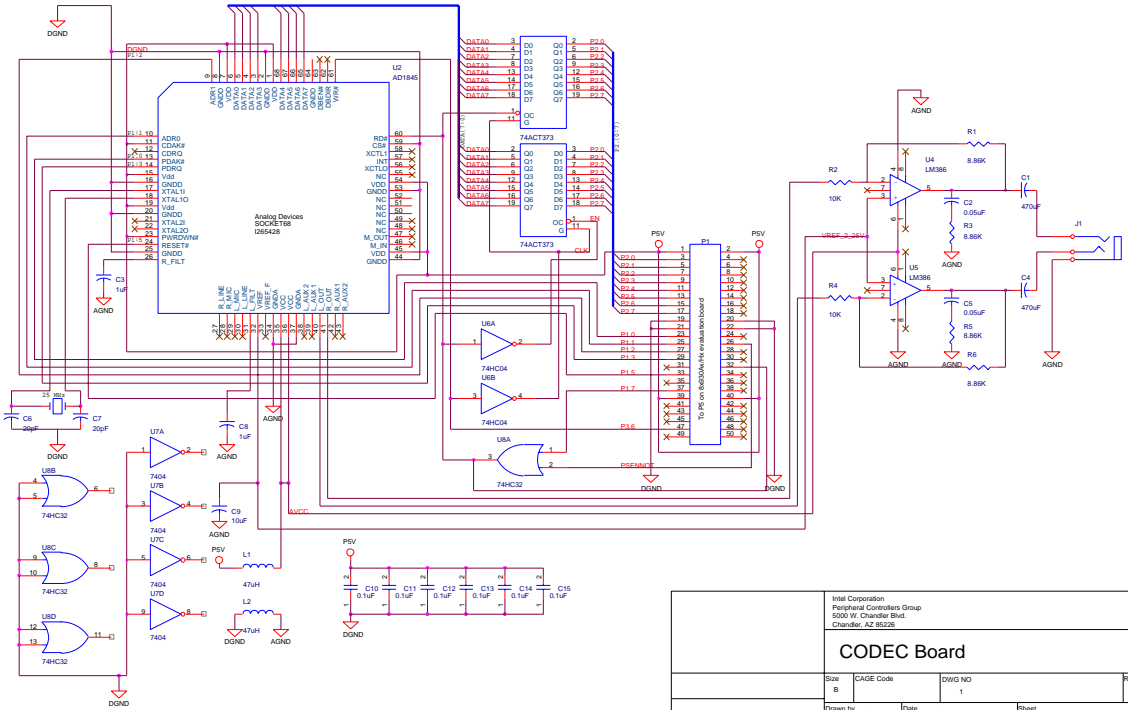
    dec WR0,#1D           ; decrement the hubsamples
    mov hubsamples,WR0    ; update hubsamples
    mov WR6,audio_data_ptr
    add WR6,#04d          ; increment the pointer to the next sample
    mov audio_data_ptr,WR6
    mov CCON,#00h        ; clear the PCA Compare/Capture Flag at CCON.0

finis:
    pop DR12
        pop WR6
        pop R5
        pop R4
        pop R3
        pop R2
        pop WR0

    ret

; ***** END CODEC_ISR *****
```

# 10.0 Schematic



Intel Corporation Peripheral Controllers Group 5000 W. Chandler Blvd. Chandler, AZ 85226			
<b>CODEC Board</b>			
Size B	CAGE Code	OWS NO 1	Rev A
Drawn by LCT		Friday, September 11, 1997	Sheet 1 of 1