

# USB AUDIO CARD

This document describes a USB AUDIO CARD. Features of this device, hardware architectures, software techniques and skills, theory of USB audio device, structure of USB audio driver in Windows system, and some future proposals and introduced and discussed in this document. Some photos and charts are also listed in the appendix section.

## Introduction

Several years ago I have used 51 series microprocessor such as at89c2051, at89c51 from ATMEL in my project, these MCU are very economic and easy to use. So far they are still running in many fields. Last year because of the need of a new project, I employ an AVR microprocessor At90s8515 from ATMEL, it's high speed and ISP feature help me finish my project quickly.

When I read the introduction of Atmega162L microprocessor, I was deeply attracted by it's powerful features. Here I have to thank this contest giving me a chance to realize the Atmega series microprocessor.

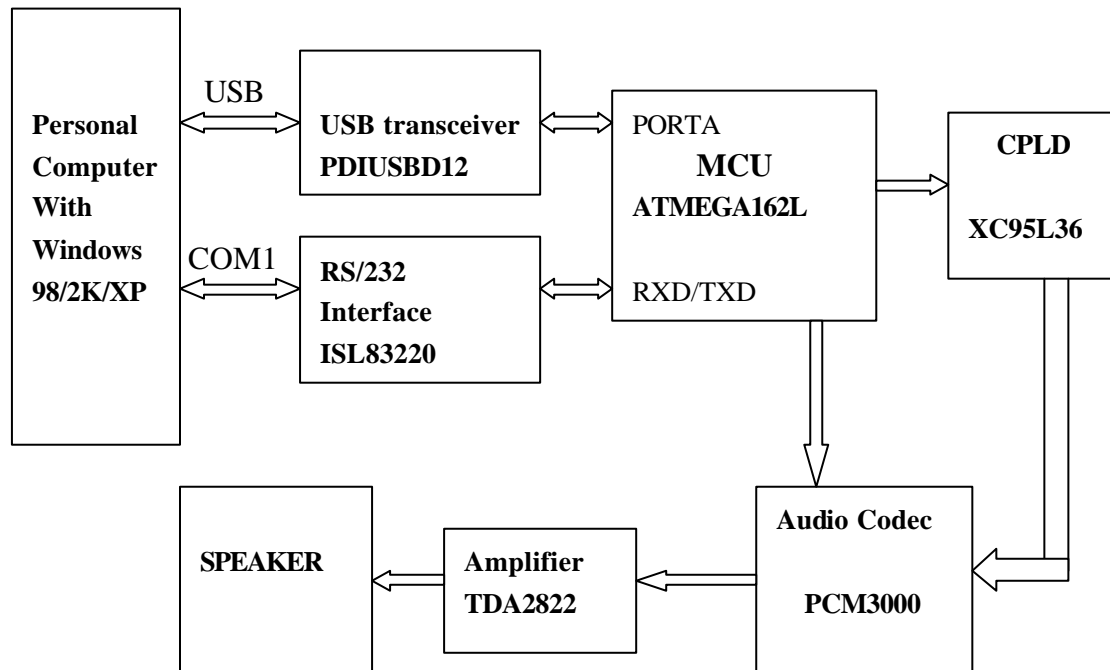
Now many kinds of USB peripheral devices appear in our life. Such as USB keyboard, USB mouse, USB camera and U-disk are very familiar besides your computer. It is obvious that USB interface is very much more convenient than any other interfaces such as PS/2, PCI and IDE.

Atmega162 has 16Kbytes flash and 1Kbytes ram, especially 16MIPS throughput at 16MHZ. It is enough to finish a USB AUDIO CARD. As we know, many audio cards in our computer belong to PCI device, because of the high throughput of PCI interface, many devices select PCI. But now USB2.0 has also high speed to 400Mbps. In fact, the data rate of raw real time stereo audio is very slow, if your sample rate is 48KHz, and one sample has 16bits, the bit rate is only  $48\text{KHz} \times 16\text{bits} = 768\text{Kbps}$ . Do you remember ISA audio card in intel486 computer before? So I select USB1.1 interface chip, which is USB full speed transceiver.

My plan is to design an audio card to play all formats audio data, but because the limit of buffer in the USB transceiver, the highest sample rate in my card has to be 44.1KHz. If you want to let it support 48KHz, you can select another USB transceiver whose ISO buffer is bigger than 128 bytes.

With a USB B-type to A-type cable, my audio card can be easily connected with computer. After a few minutes, Windows system will tell you a new composite USB device is found, and then audio device driver is installed automatically. In the device manager, a USB Audio device appears. After you select the USB audio device as preferred audio device, windows system will transfer audio data to this card. After connecting earphone with this card, you can hear voice when you play a MP3 or WAV file.

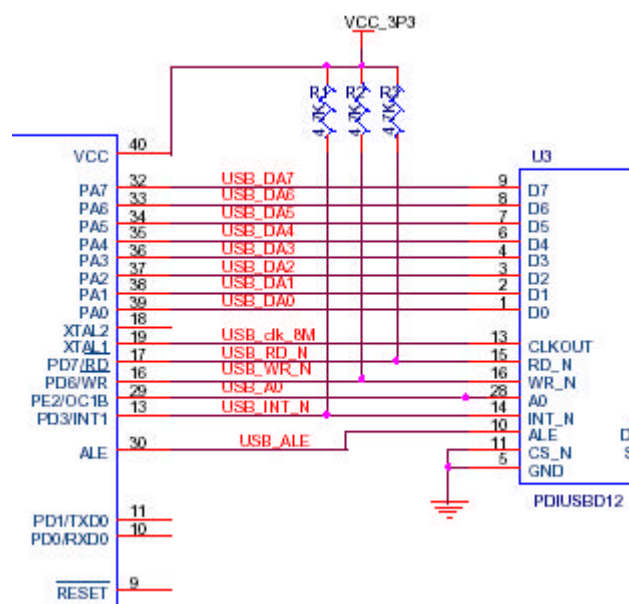
## Hardware Architecture



**Figure 1 the block diagram of the USB AUDIO CARD**

### Interface between USB chip and MCU

This USB audio card has USB1.1 feature. I use the PDIUSB12 from Philip as the USB interface. Whose data transfer rate can achieve 1Mbits/s in isochronous mode. This chip just is a transceiver. So you can change it to be any other USB1.1 or USB2.0 interface chip such as ISP1581 from Philips or CY7C68001 from Cypress.



**Figure 2 Interface between Atmega162 and PDIUSB12**

The PA[7:0] port is used to control USB transceiver(PDIUSBD12).Because the PDIUSBD12 has a programmable PLL, which can generate 4MHz—48MHz clock, I use this CLKOUT to drive Atmega162. The CLKOUT frequency is set to be 8Mhz.And the PDIUSBD12 use a 6MHz crystal. If you don't want to use this CLKOUT signal, you can use the Atmega162 special feature: **Internal Calibrate RC Oscillator**. I have used this function when I am not sure the PDIUSBD12 has worked well. This feature is very useful and interesting, you only need to program the responding FUSE in the Atmega162 to select internal Oscillator.

Atmega162 use PA[7:0] 8 bits bus to transfer data between and D12, PE2 is connected with A0 (D12), which can be used to distinguish COMMAND and DATA when separate address and data bus is configured in D12. In my design, I didn't use this A0 signal through it is connected. ALE is connected with ALE (PDIUSBD12), which is used to distinguish Address phrase and Data phrase, in address phrase, the D12 only judge whether the address is even or odd. The even address indicates the data phrase is DATA, and the odd address indicates the data phrase is COMMAND .D12 has some commands used to control the D12. You can view the PDIUSBD12 datasheet to learn details. So in my C language program I use the following way to write data (0x80) to D12:

```
*(volatile unsigned char *) (0xFF20)=0x80;
```

and use

```
*(volatile unsigned char *) (0xFF21)=0xD0;
```

to write command(0xD0) to D12.When read data from D12 I use:

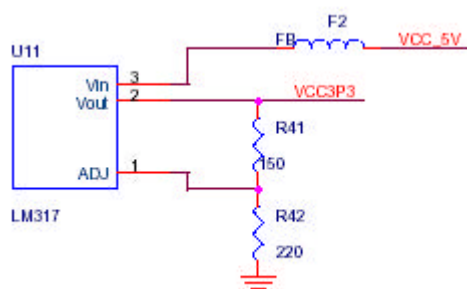
```
(unsigned char )data=*(volatile unsigned char *) (0xFF20);
```

Here I have to mention another feature in Atmega162: **External memory High Mask**, which helps me more. As we know, in At90s8515, when you want to use external memory, the PORTC is used for high address bytes. If your external memory is less than 64Kbytes, some pins of the PORTC are wasted. Now Atmega162 has been improved, the PORTC pins can be released as normal port pins. In my design, I release all PORTC pins to communicate with CPLD.

Interrupt from PDIUSBD12 is connected with INT1 (PD3), which belongs to low level interrupt.

### VCC3.3V Power Supply

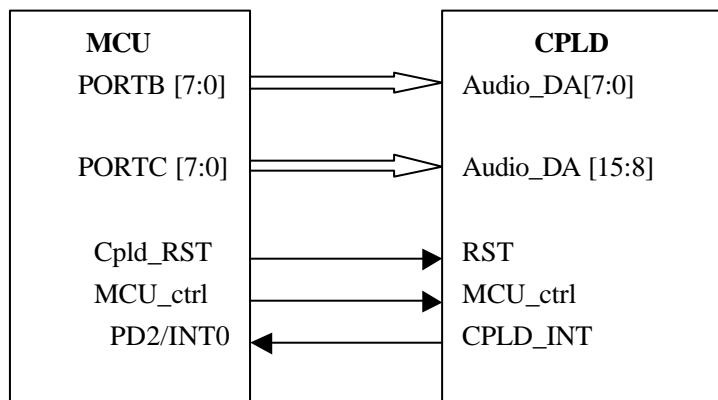
In this design, 3.3V power is required because I use Atmega162L-8PI. USB port only provide 5V voltage, I use LM337 to generate 3.3V:



**Figure 3 VCC3.3V voltage circuit**

### Interface between CPLD and MCU

CPLD is really a frequency divider and a 16bits parallel to serial converter. It generates the synchronous clock and bit clock, and receives 16bits parallel data from Atmega162 and transmits serial data to Audio CODEC.



**Figure 4 Interface between MCU and CPLD**

Here I use Interrupt mode to transfer data between MCU and CPLD. This interrupt belongs to falling edge interrupt. Because the ISO out Endpoint buffer in PDIUSBD12 has only 128 bytes, I have to choose mono channel when the audio format is 44.1KHz\*16bits. Windows system always send 1 audio frame to PDIUSBD12 every one millisecond (you can learn details about this in the USB audio specification), the length of this frame is 88 bytes or 90 bytes, if the audio format is 16KHz\*16bits, then the length of one frame is 32bytes with mono channel, or 64bytes with stereo channel. In the ISR, MCU read 2bytes from PDIUSBD12 main\_ep buffer, and put the low byte on the PORTB [7:0] and put the high byte on the PORTC [7:0] bus. Here the frequency of CPLD\_INT is equal to sample rate, it is generated by a clock divider in the CPLD. Every one cycle (1/44.1KHz) CPLD read the data on the Audio\_DA [15:0] bus and put it into register. The parallel to serial convert module read the register and output serial audio data.

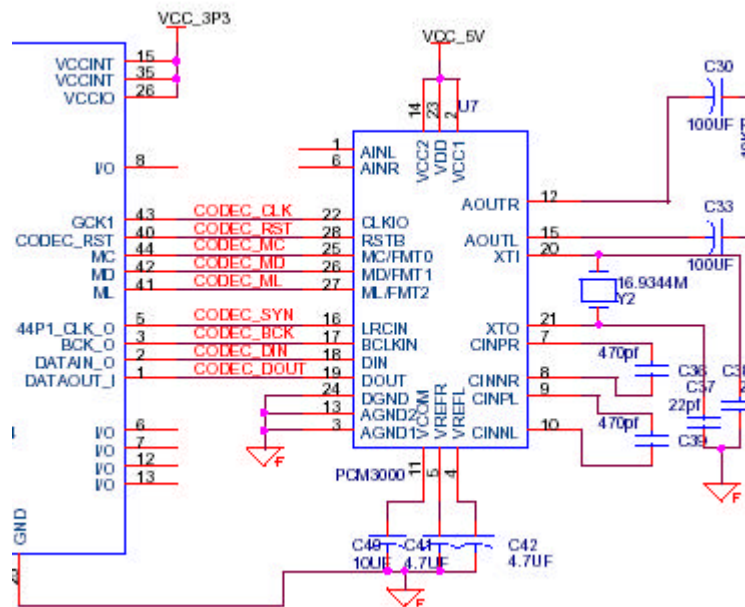
This is another function in the CPLD, when the signal MCU\_Ctrl from Atmega162 is set to be low level, CPLD will connect the PORTB [2:0] with MC, MD, ML, which are 3 control signals for audio codec.

This CPLD is XC95L36 from Xilinx, which has 44 pins. 36 macrocells. Its power supply is 3.3V. And I program it through JTAG port.

At the beginning, I planed to use a 16bit parallel to serial IC or two 8bits parallel to serial Ics to do this function. But these chips can only do P to S, I have to add another clock divider IC to generate SYN, CPLD\_INT and BIT\_CLK signals. That will be very complex, so I choose the XC95L36. It is very flexible and very easy to use.

## Audio Codec

PCM3000E is 16/18-bits stereo audio codec, whose configure registers can be programmed with 3-wire serial interface. Analog audio is produced.



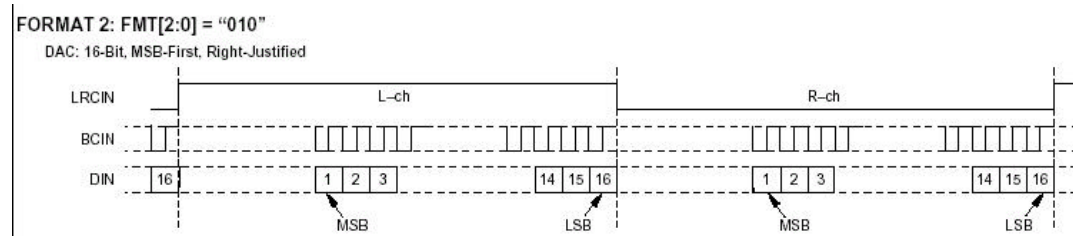
**Figure 5 Interface between CPLD and Audio Codec**

The sample rate of PCM3000E can be up to 48KHZ, the system clk should be  $256 \times fs$ ,  $384 \times fs$  or  $512 \times fs$  ( $fs$  is the sample rate).

In order to accept 44.1KHz/16bits audio data from PC, I choose 16.9344MHz crystal which is equal to  $384fs$  ( $fs=44.1KHz$ );

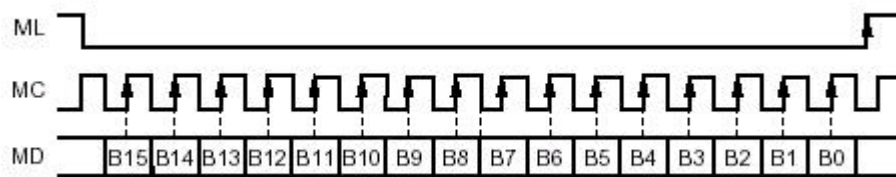
The source clock of CPLD come from PCM3000E's pin CLKIO( which is buffered output of Oscillator).The frequency divided module in the CPLD divide this 16.9344MHz clock and output 44.1KHz SYN clock , 1.4112MHz BIT clock.

PCM3000E use SYN clock to distinguish left channel or right channel. In the DAC module, it triggers the data at the rising edge of BCKIN, so I have to place data on DIN at the falling edge of BIT clock.



**Figure 6 Audio Data Format**

MC,MD,ML are used to program the PCM3000E, the timing is very simple.



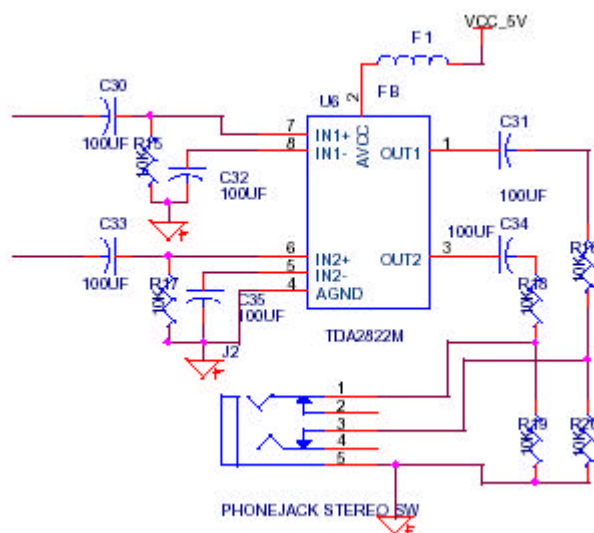
**Figure 7 Control Data Input Format**

These 3 signals are connected with CPLD, but are controlled by Atmega162; this element has been explained above.

PCM3000E has 4 registers, you can use it to select Audio data format, set attenuation level, set DAC soft mute and so on.

### Audio amplifier

TDA2822 is a dual audio power amplifier, which receives the analog audio from PCM3000E then drives the earphone or speaker.



**Figure 8 Audio amplifier**

### RS232 Interface

ISL83220 is a RS232 transceiver, it is only used when debugging the program running in the atmega162L.

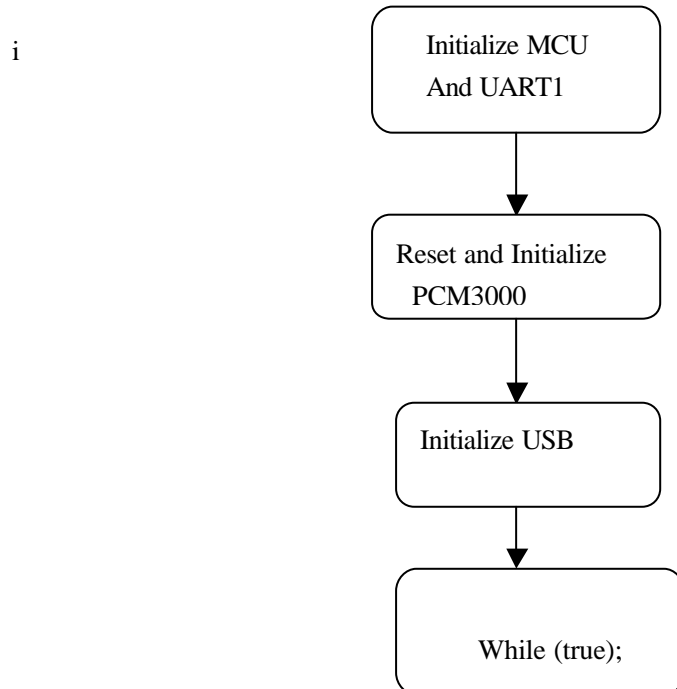
### Summary

ATMEGA162L is a central processor. It mainly processes the USB audio protocol, and transfer data between USB module and AUDIO codec module. In the beginning, it also initializes usb chip, cpld , and audio codec.

## Software techniques and skills

I use ICC\_AVR Compiler tool and AVR STUDIO4 to develop this project.

### 1:Main process



**Figure 9 Main flow chart**

In the Main( ) function, I first initialize the atmega162(port, uart), then reset the Audio codec and configure it's register group using 3 wires:MC,MD,ML,then I initialize the usb interface chip, set endpoints enable and select ISO OUT mode because of the 44.1KHz/16bits audio format.Last enable the interrupt of Atmega162 and make it enter LOOP mode to wait Interrupt.

#### PCM\_INIT

```
#define Set_Bit(val, bitn)    (val |= (1<<(bitn)))
#define Clr_Bit(val, bitn)   (val &= ~(1<<(bitn)))
void pcm_init(unsigned short par)
{
    unsigned short temp;
    unsigned char i;
    PORTB=0xFF;
    Clr_Bit(PORTD,5);    //MCU_CRTL=0; then FMT[0:2]=PORTB[0:2] in the cpld;
    delay_time(10);
    Clr_Bit(PORTB,2);    //ML=0;

    for(i=16;i>0;i--){
        Set_Bit(PORTB,0);    //MC=1;
        delay_time(20);
        Clr_Bit(PORTB,0);    //MC=0;
        temp=par&((unsigned short)(1<<(i-1)));    //MD data
        if(temp) Set_Bit(PORTB,1);
    }
}
```

```

    else Clr_Bit(PORTB,1);
    delay_time(10);
}
Set_Bit(PORTB,0); //MC=1;
delay_time(20);
Set_Bit(PORTB,2); //ML=1;
delay_time(10);
Set_Bit(PORTD,5); //MCU_CRTL=1; then FMT[0:2]!=PORTB[0:2] in the cpld;
}

```

This process control the PCM3000E, I have to use bit operation to generate right timing. It is very easy to realize. These codes cost me only 1 hour, and all is passed.

### USB INIT

```

#define USB_FRQ 0x45 //8MHZ
void d12_init(void)
{ unsigned char intreg1,intreg2;
  out_cmd(0xd0);
  out_data(0x80); //setaddressEnable

  out_cmd(0xd8);
  out_data(0x01); //setEpEnable

  out_cmd(0xf3);
  out_data(0x44); //128 ISO out
  out_data(USB_FRQ); //setmode,disconnect
  delay_time(50);

  out_cmd(0xf3);
  out_data(0x54); //128 ISO out
  out_data(USB_FRQ); //setmode,reconnect
  delay_time(50);

  out_cmd(0xf4); //read int reg;
  intreg1=in_data();
  intreg2=in_data();
}

```

When USB port is connected with PC, power is supplied. You must configure the USB chip PDIUSB12 to be active. Because Windows system always allocate an address to every USB device, you must first do *set\_address\_enable* command. *Set\_Ep\_Enable* will make the Endpoint ready to receive data from PC. The first *Set\_mode* command make D12 soft disconnect with PC, and configure the internal PLL, make the CLOCK\_OUT to be 8MHz. The second *Set\_mode* command make D12 soft connect with PC. Then PC will detect a new USB device inserted and send protocol packet to PDIUSB12.

### Enable Interrupt and enter loop

```

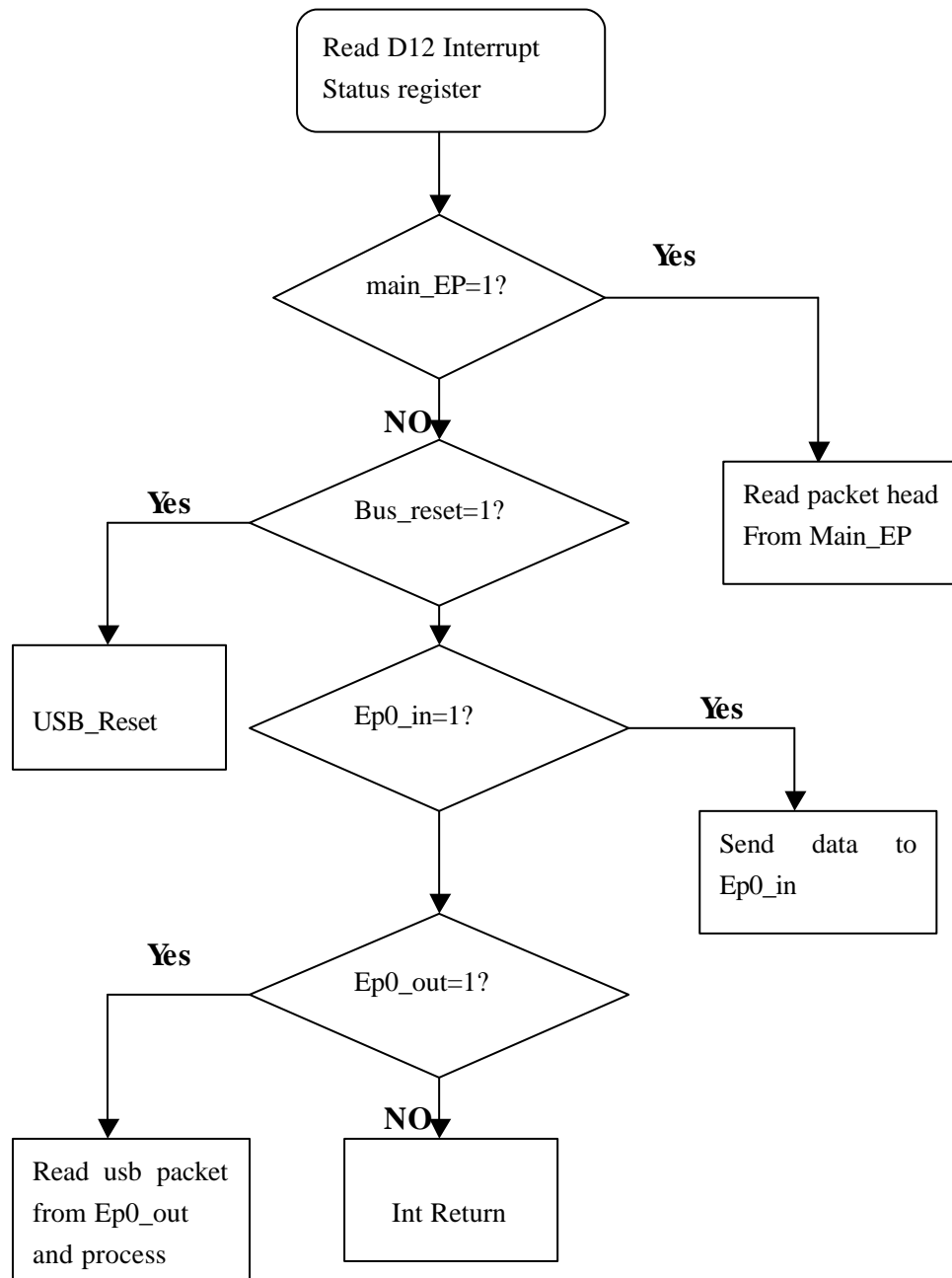
SEI();

```



```
while(1);
```

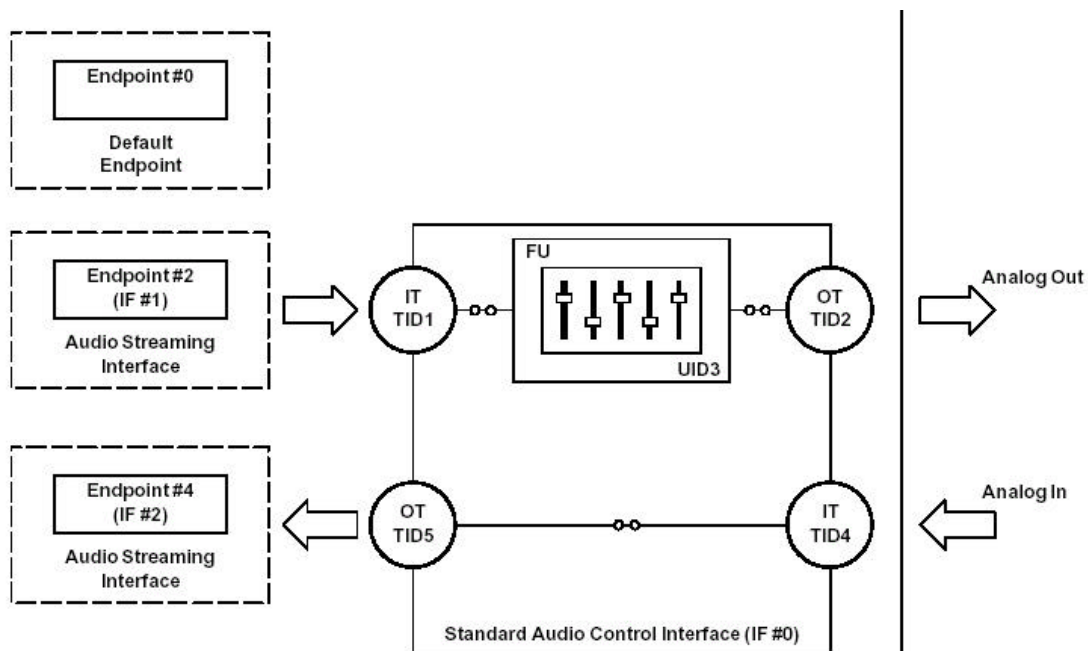
2:ISR\_number3 process (interrupt from USB D12)



**Figure 10 Interrupt 1 flow chart**

Once the MCU set the Soft connect flag in the USB chip, Windows system will send setup packets to the USB control endpoint(Ep0) one by one. Then USB interrupt MCU to process these packets. MCU will read packet from USB EP0 buffer and acknowledge these request such as get\_device\_descriptor, get\_config\_descriptor and so on. In this process, MCU notify Windows system that it is a usb audio device. If Windows system finds this is an available device, USB audio device driver will be loaded. At the late time, you can use the USB audio device as your PCI audio card.

- 1) The first interrupt from D12 is always **BUS\_RST**, I did nothing in this process and return at once, you can add some init\_data in this process.
- 2) The second interrupt from D12 is from EP0\_out, which indicates there are packet in this buffer, then read this packet into RAM and analyze it. This packet is a 8-bytes setup packet from Windows systems, you can view the USB specification to know the definition of these 8 bytes.
- 3) The first packet is always **Get\_device\_descriptor** request, and the correct acknowledge packet should be send back to Ep0\_in buffer. Then D12 will send this packet to Windows system immediately. The acknowledge packet has 18 bytes, but the buffer of Ep0\_in has only 16bytes. So you must first send 16bytes to Ep0\_in and then send the leaving 2bytes to Ep0\_in. when the data in Ep0\_in buffer is sent, D12 will generate a Ep0\_in interrupt, I send the leaving data in this interrupt process.
- 4) Once Windows system received a legal acknowledge packet, it will send **Set\_address** packet to D12, in this process, the USB device address is allocate, and should be written to address\_register in D12. Then a zero\_length packet should be sent to PC , which notifies Windows system the new address has been set up.
- 5) Then Windows systems will send **Get\_device\_descriptor** request again, this process is same as above.
- 6) Next, **Get\_configure\_descriptor** request will be received, the first request only require 9 bytes. It is easy to be done.
- 7) Once the 9 bytes is received, Windows system will send another **Get\_configure\_descriptor** request. At this time , all the configure\_descriptor should be sent to Windows System. This packet's length is 0xda in my design. This length value is also included in the packet, if it is different from the real length of the packet you sent. Windows system will think your acknowledge is not corrected, and will send **Get\_device\_descriptor** and **Get\_configure\_descriptor** three times.

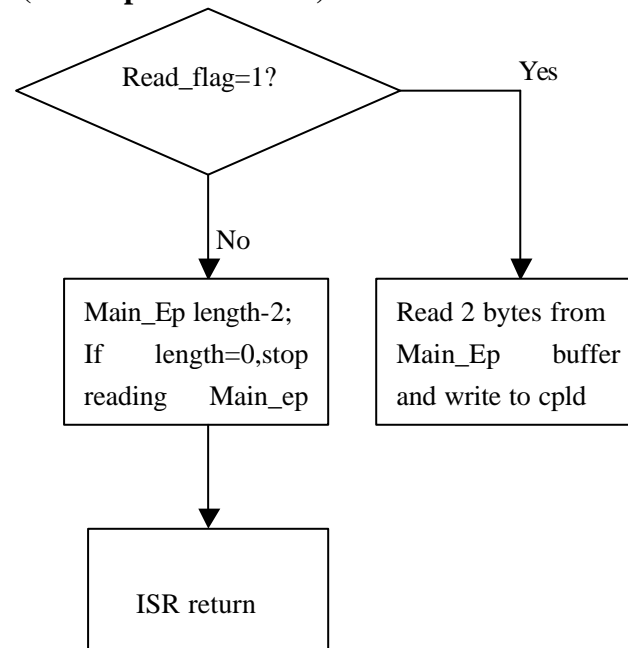


**Figure 11** USB AUDIO function topology

8) The configure\_descriptor packet has all the information about this USB device. It tells Windows system I am an USB audio device, there are all 3 interfaces, how many audio stream format I supported and so on.

After these are finished, Windows system will send *Get\_feature* ,*Get\_interface*, *Set\_feature*, *Set\_interface* ,*FU\_set*, *FU\_get*, *Set\_Endpoint request*, all these processes can be looked up in the USB AUDIO specification.

### 3:ISR\_number2 process(interrupt from CPLD)



**Figure 12 CPLD interrupt process**

This interrupt indicates that CPLD request 2 bytes on the audio\_data[15:0] bus. In my project, I select 44.1KHz\*16bits\*Mono data format because most high quality audio source use 44.1KHz\*16bits data format. So the frequency of SYN signal (LRCIN of PCM3000E) is set to be 44.1KHz, and the frequency of CPLD\_INT will be 44.1KHz\*2=88.2KHz. However, I only support mono, in the even interrupt, MCU read 2 bytes from the Main\_EP buffer and put it on the audio\_data[15:0] bus, in the odd interrupt, MCU do nothing but subtract 2 from buffer length. If the length is zero, then MCU will return at once.

### 4: CPLD MODULE

CPLD is XC95L36 from XILINX. There are 2 functions in this CPLD, including frequency divider, parallel to serial converter. The GCK is 16.9344Mhz from CLKOUT of PCM3000E.

In order to generate BCLK(1.4112MHz), GCK is divided by 12(16.9344MHz/12). Then BCLK is divided by 16(1.4112MHz/16), CPLD\_INT(88.2KHz) is generated, SYN(44.1KHz) is easy to be produced using CPLD\_INT (SYN=CPLD\_INT/2).

Another, CPLDs interconnect PB[2:0] with control signals MC, MD, ML of PCM3000E when MCU active the MCU\_CTRL signal.

## USB AUDIO Theory and driver structure

The USB is very well suited for transport of audio (voice and sound). PC-based voice telephony is one of the major drivers of USB technology. In addition, the USB has more than enough bandwidth for sound, even high-quality audio. Many applications related to voice telephony, audio playback, and recording can take advantage of the USB.

In principle, a versatile bus specification like the USB provides many ways to propagate and control digital audio. For the industry, however, it is very important that audio transport mechanisms be well defined and standardized on the USB. Only in this way can interoperability be guaranteed among the many possible audio devices on the USB. Standardized audio transport mechanisms also help to keep software drivers as generic as possible. The **Audio Device Class** satisfies those requirements. It is written and revised by experts in the audio field.

An essential issue in audio is synchronization of the data streams. Indeed, the human ear easily detects the smallest artifacts. Therefore, a robust synchronization scheme on isochronous transfers has been developed and incorporated in the *USB Specification*. The Audio Device Class definition adheres to this synchronization scheme to transport audio data reliably over the bus.

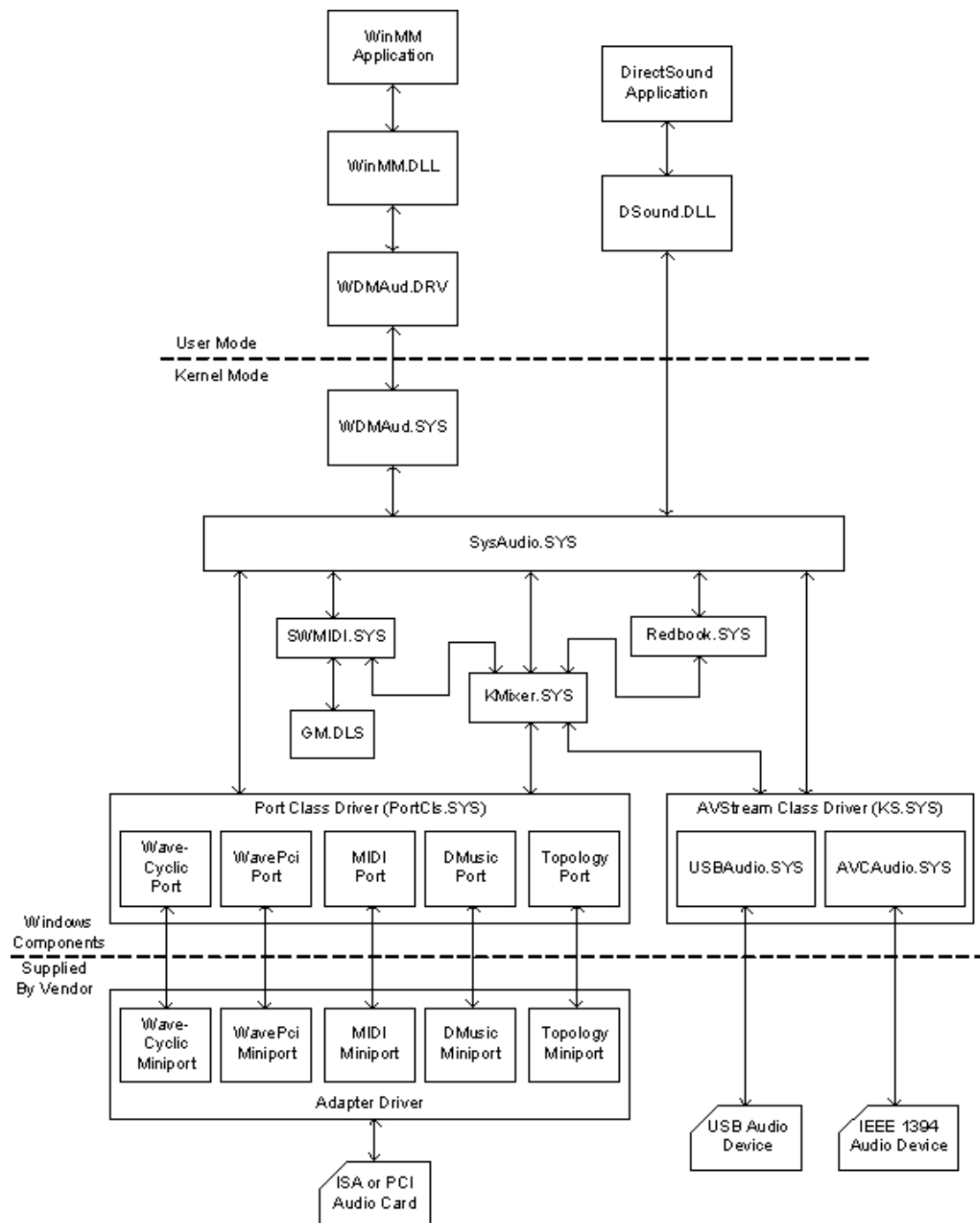
The USB Audio class system driver, Usbaudio.sys, is an AVStream minidriver that provides driver support on Microsoft® Windows® operating systems for USB Audio devices. You can learn more from Windows Platform Design Note <http://www.microsoft.com/whdc/device/audio/usbaud.msp>. To make audio hardware drivers easier to write, the WDM audio driver model isolates the hardware interface issues from the filter-implementation issues. It accomplishes this by organizing the driver code into components that address these issues separately and by precisely defining the interfaces between these components. This paper refers to all the components as drivers, though not all are recognized as such by the operating system. The following figure presents a simplified view of the WDM audio driver architecture in Windows XP.

An AVStream minidriver is a peer of the AVStream class driver and can call WDM support routines (corresponding to the default function pointers in a KSDEVICE\_DISPATCH structure) as necessary. The minidriver does not create a device object, however. Instead, it uses the class driver's device object to register support routines. Typically, minidrivers do not need to implement WDM support routines because all the necessary functionality is available from the class driver.

AVStream is the appropriate class driver to use for an audio device when the driver's complexity exceeds the capabilities of the port drivers supplied by Microsoft (for example, a device combining audio and video) or the driver represents a class of devices with a single minidriver (for example, USB audio). It is also appropriate for supporting some audio functionality in a driver that primarily supports other streaming functionality such as video capture and decoding of MPEG2 and AC3. The vast majority of audio vendors should use the port class/miniport driver model, but Microsoft provides the AVStream class/minidriver model to enable proprietary monolithic audio drivers and drivers for A/V devices with basic audio functionality.

If, during Plug and Play device enumeration, an audio device on a USB or IEEE 1394 bus identifies itself as compliant with the USB or IEEE 1394 audio specifications, the system automatically loads the USBAudio.SYS or AVCAudio.SYS minidriver to drive the device. The

system-supplied minidriver drives the device directly, and no additional vendor-supplied adapter driver is needed.



**Figure 13 WDM audio driver architecture**

