

AC97

AC'97 (Audio Codec '97) is an audio standard published by Intel in 1997. It is mainly used by on-board chips and less often by sound cards (or modems). Since 2004, it is more and more replaced by High Definition Audio Interface (HD Audio).

In this (more or less) tutorial, I assume that you have a PCI driver that works as a whole, from which you have the BARs (in this case, I / O rooms) from a device with a specific manufacturer and device ID, Can be found. In addition, you should be familiar with the structure of sound files (WAV, uncompressed) and thus also with terms like PCM or Samplerate.

Compatible devices

AC'97 compatible devices appear as PCI devices with the base class code 04h (multimedia controller) and the subclass code 01h (audio). However, these are only sound cards and corresponding chips in general, such devices can also be HD audio sound cards or other incompatible audio devices. Therefore, here is an incomplete list of AC'97 devices:

Vendor	Device
0x8086 (Intel)	0x2415 (82801AA - Intel ICH)
0x8086 (Intel)	0x2425 (82801AB - Intel ICH0)
0x8086 (Intel)	0x2445 (82801BA - Intel ICH2)
0x1106 (VIA)	0x3058 (vmtl. auf allen VIA-Mainboards verbauter AC'97-Chip)

Unfortunately, the interfaces for controlling the cards are different from manufacturer to manufacturer, so that they are discussed in separate sections.

Intel sound cards

QEMU (0.10.0) and VirtualBox (2.1.4) use ICH. The codes presented in this tutorial seem to work only there.

Initialization

The devices have two I / O rooms, the first being NAM-BAR (Native Audio Mixer BAR) and the second NABM-BAR (Native Audio Bus Master BAR). The NAM-BAR controls control the mixer (ie volume, samplerate, ...), which is used to control playback in NABM-BAR (play / pause, sound buffer, ...). To enable these I / O rooms and start the bus master, the value of the "COMMAND" register in the PCI configuration room must be 0x05 "ge-Or-t". An example:

```
pciConfigWrite(Bus,Device,Function,PCI_COMMAND,pciConfigRead(Bus,Device,Function,PCI_COMMAND)|5)
```

The device must then be initialized as follows:

- Reset
- Adjust the volume
- If necessary, set sampler dataDies kann zum Beispiel so geschehen:

```
void delay(int ms); // Waits "ms" milliseconds
uint16_t inw(int port); // Reads a value of I/O port "port"
void outb(int port, uint8_t value); // Specifies the value "I / O port" port
void outw(int port, uint16_t value); // Specifies the value "I / O port" port

// These two variables must be fed with data from the PCI driver (first or second I / O space)

int nambar; //NAM-BAR
int nabmbar; //NABM-BAR

int volume; // Volume; Attention: 0 is full volume, 63 is as good as mute!

// Code (in any function):
outw(nambar + PORT_NAM_RESET, 42); // Resets (any value is possible here)
outb(nabmbar + PORT_NABM_GLB_CTRL_STAT, 0x02); // Also here - 0x02 is mandatory
delay(100);
volume = 0; // The loudest!
outw(nambar + PORT_NAM_MASTER_VOLUME, (volume<<8) | volume); // Other Volume (left and right)
outw(nambar + PORT_NAM_MONO_VOLUME, volume); // Volume for mono output (unnecessary)
outw(nambar + PORT_NAM_PC_BEEP, volume); // Volume for the PC speaker (unnecessary if not used)
outw(nambar + PORT_NAM_PCM_VOLUME, (volume<<8) | volume); // Volume for PCM (left and right)
delay(10);
if (!(inw(nambar + PORT_NAM_EXT_AUDIO_ID) & 1))
{ /* Sample rate fixed to 48 kHz */ }
else
{
```

```

    outw(nambar + PORT_NAM_EXT_AUDIO_STC, inw(nambar + PORT_NAM_EXT_AUDIO_STC) | 1); // Enable variable
rate audio
    delay(10);
    outw(nambar + PORT_NAM_FRONT_SPLRATE, 44100); // Other Sample Rate: 44100 Hz
    outw(nambar + PORT_NAM_LR_SPLRATE, 44100); // Stereo sample rate: 44100 Hz
    delay(10);
    // Actual sampler data is now in PORT_NAM_FRONT_SPLRATE or PORT_NAM_LR_SPLRATE
}

```

After that, the device should be able to work and play sounds.

Tone (etc.) output

If tones (or any other things in this direction) are played, they must be split into individual pieces with a maximum of 65536 samples each. These must be stored in the memory as follows:

Offset	Content
+0x00	16-Bit-Sample (int16_t) left
+0x02	16-Bit-Sample (int16_t) right

Note: These are two samples - one for each channel. This results in a maximum size of $65536 * 2$ bytes = 128 kilobytes per buffer. A maximum of 32 buffers can be passed to the device, which is played one after the other. This is then a maximum of 4 MB ($128 \text{ kB} * 32$) or 23.8 seconds (44.1 kHz) or 21.8 seconds (48 kHz). It can be seen that this is a large amount of data at a relatively short period of time - so much more than 32 buffers are probably exaggerated from memory consumption. The solution is to fill data banks that have already been played with new data and to play them again. Thus, the device runs through all buffers again and again, while the driver refreshes its contents over and over again.

Let us now come to practice. The buffers are described by 8-byte-long buffer descriptors, which are designed in this way:

Offset	Data type	Content
+0x00	uint32_t	Physical pointer to the buffer (aligned to DWord boundaries)
+0x04	uint16_t	Length of the buffer in Samplen (stereo PCM: maximum 0xFFFF, 0x0000 means: no data)
+0x06	uint16_t	Parameters: 0x8000 = IOC; 0x4000 = BUP; Rest is reserved

- IOC (Interrupt On Completion): If set, the device sends an interrupt when this buffer has been played.
- BUP (Buffer Underrun Policy): This bit is important when this buffer is not yet ready for the last buffer to be played or the next buffer: If the bit is not set, the last sample of this buffer is held after playback is set, "0" is sent. Typically it is set at the last buffer to be played.

The buffer descriptors are entered into a list, which is (logically) named Buffer Descriptor List. Here, as I said, fit a maximum of 32 descriptors purely (thus 256 bytes maximum length). The initial address of this list is then communicated to the device, then the number of the last valid element (at most 31), and finally (symbolically) the play button can be pressed. For example, this could look like this, so you can play a maximum of 23.8 seconds:

```

struct buf_desc
{
    void *buffer;
    unsigned short length;
    int reserved : 14;
    unsigned int bup : 1;
    unsigned int ioc : 1;
} __attribute__((packed));

struct buf_desc *BufDescList; //Buffer Descriptor List

// The following may again be in any function...
int size; // Length of the data to be played in bytes
int i; //Index
int final; // Last valid buffer
for (i = 0; (i < 32) && size; i++)
{
    BufDescList[i].buffer = /* Physical address of the respective buffer */;
    if (size >= 0x20000) // Even more than 128 KB, so the buffer becomes full
    {
        // Maximum length is 0xFFFFE and NOT 0xFFFF! Left and right
        // must be the same number of samples, so this number must be straight.
        BufDescList[i].length = 0xFFFFE;
        size -= 0x20000; //128 kB away
    }
    else
    {
        // Half the length in bytes because 16-bit samples need two bytes
        BufDescList[i].length = size >> 1;
        size = 0; // Nothing more now
    }
}

```

```
BufDescList[i].ioc = 1;
```

```

    if (size) // Another buffer
        BufDescList[i].bup = 0;
    else // No more buffer
    {
        BufDescList[i].bup = 1;
        final = i; // Last valid buffer is this one
    }
}
outl(nabmbar + PORT_NABM_POBDBAR, (uint32_t)/* Physical address of BufDescList */);
outb(nabmbar + PORT_NABM_POLVI, final);
outb(nabmbar + PORT_NABM_POCONTROL, 0x15); // Play, and then generate interrupt!

```

Works with QEMU 0.10.0 and VirtualBox 2.1.4 ... As mentioned above, 16-bit stereo samples with a samplerate of 44.1 kHz (or whatever you have set) must be stored in the memory.

Appendix

Required Constants:

```

#define PORT_NAM_RESET          0x0000
#define PORT_NAM_MASTER_VOLUME 0x0002
#define PORT_NAM_MONO_VOLUME   0x0006
#define PORT_NAM_PC_BEEP       0x000A
#define PORT_NAM_PCM_VOLUME    0x0018
#define PORT_NAM_EXT_AUDIO_ID  0x0028
#define PORT_NAM_EXT_AUDIO_STC 0x002A
#define PORT_NAM_FRONT_SPLRATE 0x002C
#define PORT_NAM_LR_SPLRATE    0x0032
#define PORT_NABM_POBDBAR      0x0010
#define PORT_NABM_POLVI        0x0015
#define PORT_NABM_POCONTROL    0x001B
#define PORT_NABM_GLB_CTRL_STAT 0x0060

```

VIA sound cards

Initialization

VIA's AC'97 sound card is Southbridge function 5, which shares its IO address spaces with the integrated modem (MC97) in function 6. We will only deal with AC'97. The relevant IO space for DMA resides in the first PCI-BAR.

The card obtains sound data from multiple buffers described in SGD (Scatter / Gather DMA Table) tables by descriptors of the following format. The physical addresses of the tables must be written into the 32-bit registers AC97_VIA_R_SGD_TABLE_BASE and / or AC97_VIA_W_SGD_TABLE_BASE and aligned to 2-byte boundaries:

```

typedef struct
{
    uint32_t buf;
    uint32_t len      : 24;
    uint32_t reserved : 5;
    uint32_t stop     : 1; // Transfer at the end of the block. To continue, set Bit 2 in Register 0 (Audio
SGD Read Channel Status)
    uint32_t flag      : 1; // Transfer at the end of the block. Raises a FLAG interrupt.
    uint32_t eol       : 1; // Last post. Raises an EOL interrupt.
} __attribute__((packed)) ac97Via_SGDEntry_t;

```

Access to the AC97 codec

Access to the AC97 codec is more difficult than with the o.g. AC97 implementation of Intel, because its registers are not mapped into a separate address space, but must be read and written via the 4 byte-long register at offset 0x80. The following code implements access to these registers:

```
void codec_sendCommand(ac97Via_t* ac97, uint8_t reg, bool primary, uint16_t data)
{
    while (inl(ac97->iobase + AC97_VIA_ACCESS_CODEC) & AC97_VIA_CODEC_BUSY) // Wait until the codec has time
        ;
    if (primary)
        outl(ac97->iobase + AC97_VIA_ACCESS_CODEC, AC97_VIA_CODEC_SEL_PRIM | AC97_VIA_CODEC_WRITE |
((uint32_t)reg << 16) | data);
    else
        outl(ac97->iobase + AC97_VIA_ACCESS_CODEC, AC97_VIA_CODEC_SEL_SEC | AC97_VIA_CODEC_WRITE |
((uint32_t)reg << 16) | data);
}

uint16_t codec_readStatus(ac97Via_t* ac97, uint8_t reg, bool primary)
{
    while (inl(ac97->iobase + AC97_VIA_ACCESS_CODEC) & AC97_VIA_CODEC_BUSY) // Wait until the codec has time
        ;
    if (primary)
    {
        outl(ac97->iobase + AC97_VIA_ACCESS_CODEC, AC97_VIA_CODEC_SEL_PRIM | AC97_VIA_CODEC_READ |
AC97_VIA_CODEC_PRIM_VALID | ((uint32_t)reg << 16));
        while (!(inl(ac97->iobase + AC97_VIA_ACCESS_CODEC) & AC97_VIA_CODEC_PRIM_VALID)) // Wait until
the codec has delivered the data
            ;
        return inl(ac97->iobase + AC97_VIA_ACCESS_CODEC) & 0xFFFF;
    }
    else
    {
        outl(ac97->iobase + AC97_VIA_ACCESS_CODEC, AC97_VIA_CODEC_SEL_SEC | AC97_VIA_CODEC_READ |
AC97_VIA_CODEC_SEC_VALID | ((uint32_t)reg << 16));
        while (!(inl(ac97->iobase + AC97_VIA_ACCESS_CODEC) & AC97_VIA_CODEC_SEC_VALID)) // Wait until
the codec has delivered the data
            ;
        return inl(ac97->iobase + AC97_VIA_ACCESS_CODEC) & 0xFFFF;
    }
}
```

Appendix

Definitions of the constants used above:

```
// Register-Offsets
#define AC97_VIA_R_SGD_CONTROL    0x01

#define AC97_VIA_R_SGD_TABLE_BASE 0x04

#define AC97_VIA_W_SGD_CONTROL    0x11
#define AC97_VIA_W_SGD_TABLE_BASE 0x14
#define AC97_VIA_ACCESS_CODEC     0x80

// Bits of the AC97_VIA_ACCESS_CODEC-Registers
#define AC97_VIA_CODEC_SEL_PRIM    0
#define AC97_VIA_CODEC_PRIM_VALID BIT(25)
#define AC97_VIA_CODEC_SEL_SEC    BIT(30)
#define AC97_VIA_CODEC_SEC_VALID  BIT(27)
#define AC97_VIA_CODEC_BUSY       BIT(24)
#define AC97_VIA_CODEC_WRITE      0
#define AC97_VIA_CODEC_READ       BIT(23)
```