Intel High Definition Audio

From OSDev Wiki

Intel High Definition Audio refers to the specification released by Intel for delivering high-definition audio that is capable of playing back more channels at higher quality than previous integrated audio codecs like AC97.

Contents

Identifying HDA on a machine

All HDA devices appear on the PCI bus with a specific VendorID and DeviceID. Many HDA devices have a Vendor ID of 8086 (Intel), and a Device ID such as 2668 or 27D8, but other Vendor IDs are also in use, e.g. Vendor ID 1002 (AMD) and Device ID 4383. If you find any others, please add them into this page.

Overview

The HDA specification (link at the bottom of this page) details how to set up devices at the two ends of a link and there is no substitute for working from it, but it takes several readings through its 200+ pages before a clear picture eventually begins to emerge from it, so this overview is aimed at making the most frustrating parts clear from the outset.

At one end is the PCI part of the HDA device, while at the far end are hardware codecs and the many widgets which they contain. Setting things up at the near end is relatively easy, part of it being done by keyhole surgery through a couple of ordinary ports in the PCI device configuration space, while the rest is done through memory mapped ports located at an address found at index 10-17h (though the lowest four bits should be taken as zeros). You may not need to change anything in the PCI configuration space at all, but there's a fair bit to do with the memory mapped ports to set up and control DMA engines. Setting things up at the codec end is much more complicated though as you have to interrogate them to find out what they are and what functionality they offer, and then you have to work out how to set them up correctly to create paths between devices (speakers and mic.s) and DACs/ADCs. All your communications with the codecs and the many widgets they contain will be done by sending special commands via the link.

Data and commands are sent across the link in frames with strict timings and bit limits, but the work of packaging the different kinds of data into packets to go into these frames is all done for you, so all you need to do is set up a number of buffers. Two of these buffers are called CORB (command output ring buffer) and RIRB (response input ring buffer) - each buffer has a DMA engine dedicated to it which will in one case send commands from the CORB buffer across the link to codecs, and in the other case will write responses from codecs into the RIRB buffer. There will in many implementations also be an immediate command port which allows you to send commands to codecs/widgets and to receive responses from them without going through the CORB and RIRB mechanisms, but this route should not be used at the same time as CORB/RIRB as they may conflict, so it should really be reserved for initial exploration while designing your driver. The purpose of CORB and RIRB is to allow large numbers of these relatively slow communications to take place in the background while the processor goes off to do something else.

There are also buffers and DMA engines dedicated to four input streams and four output streams (at least, it's four of each in current implementations, but your software ought to check the actual number), each stream needing a descriptor buffer which must contain two or more descriptors (up to 256) which define a list of data buffers used by that stream, and the data buffers which these descriptors define will contain the actual sound samples (or have samples written into them) structured like the content of .wav files (though 20 and 24-bit samples must be padded out with zeros at the lsb end to make them all 32-bits long). The combined length of the sound data buffers can be anything up to 4GiB, so you could set things up to play or record a very long sound file and leave it going all by itself. In reality though, you'll probably work with chunks of memory a just few megabytes in size (or smaller) as one megabyte gives you room for about six seconds of 16-bit stereo data at 41.1KHz. For performance reasons, making the length of these data buffers a multiple of 128 bytes is recommended.

With all these buffers, the DMA engines jump back to the start and carry on running from there infinitely until you stop them, although with CORB there is a register which stores the last valid command which the DMA engine must stop at (after sending that command) and it will only move on again when that register is modified (by you) to enable more commands to be sent. It is the job of your software to collect data from input buffers before they are overwritten on the next lap. The set of sound data buffers defined by descriptors for a single stream collectively comprise a cyclic stream buffer, but it's divided up into chunks defined by descriptors to enable an interrupt to be generated at the end of each chunk (interrupt optional) to help you write new data into an output buffer that's just been sent before the DMA engine returns to that buffer on the next lap, or to copy data out of an input buffer that's just been written by a DMA engine to make way for more data to be written on the next lap.

At the codec end, you will need to start out by interrogating the root node of each of 15 possible codecs. The STATESTS register at offset 0Eh indicates which codec addresses have codecs at the end of them. The verb F00h will then be used with an 8-bit parameter to request information such as vendor ID, device ID and the starting node number and number of nodes for the function groups in the codec. Having found the function groups, you can use the same verb to interrogate them to find out the starting node number and number of nodes of their widgets, and also the type of the function group itself - AFG (audio function group) is the one you probably want unless you're looking for a modem. You can then interrogate each widget to ask it what its type is (e.g. output converter (DAC), input converter (ADC), mixer, selector, pin complex, power widget, volume knob). Another verb, F02h, allows you to get a connection list of other widgets in the same function group directly connected to the widget you're interrogating, though you need to use verb F00h first and the parameter at 0Eh to get the connection list length. On the Netbook I program on there is only one codec, one function group (AFG) and 37 widgets (about half of which are vendor defined audio widgets, though many of those don't exist on the codec manufacturer's datasheet as they are just holding places for real widgets used on more advanced sound cards). Don't expect there to be a volume control widget: the volume can be controlled by setting different amplifier levels at the input and output controllers instead.

Device Registers

Offset (Hex)	Name	Description	
00	GCAP	Global Capabilities	(includes number of DMA engines for input and output streams)
02	VMIN	Minor Version	
03	VMAJ	Major Version	
04	OUTPAY	Output Payload Capacity	(packet size limit for the/each output line)
06	INPAY	Input Payload Capacity	(packet size limit for each input line)
08	GCTL	Global Control	(used to reset the link and codec)
0C	WAKEEN	Wake Enable	
0E	STATESTS	State Change Status	

Setting up the AFG codec

Each programmer will likely have their own way of going about this, but the main aim will be to set out to identify which pin complexes are connected to actual speakers, headphone sockets, microphones and microphone sockets before following the trail of connections back to the most appropriate DAC or ADC to handle the stream. How you end up connecting things up will depend on what you want to do: you could, for example, use the same DAC for the speakers as for the headphone jack, and you might use the same ADC for the built-in microphone as for the microphone jack, but that won't work if you intend to send different outputs to the speakers and headphones at the same time or if you need to collect different inputs from different microphones (perhaps with the idea of using one to collect background noise to subtract from the other input). Here's an suggestion as to how the task might be broken down, but use it only as a rough guide:-

(1) Use verb F00h with parameter 4 and NID 0 (the root node) to find the number of function groups in the codec, then check those nodes using their NID and with verb F00h and parameter 5 to hunt for an AFG function group.

(2) Use verb F00h with parameter 4 and the correct NID for the AFG gunction group to get the start node (first widget) and number of nodes (widgets).

(3) Collect the following data for each widget: param 9h (primarily to get the widget type); Ch (pin capabilities); Dh (input amplifier details); 12h (output amplifier details); Eh (connection list length); 13h (volume knob capabilities). Also, use verb F1Ch to collect the configuration defaults, and verb F02h to collect the first entry in the connection list. Don't worry about whether useful responses are available for each widget - they will send back all zeros wherever they are not relevant.

(4) Sort the results into groups of DACs, ADCs, mixers, pins, etc.

(5) Identify the pins with actual speakers, microphone and jacks attached to them, using the configuration defaults which you collected earlier. If no speakers show up in bits 31-30 you should then search in order of default-association:sequence, while EAPD capability will serve as an additional clue.

(6) Make two lists of pins: one for those you want to send outputs to, and the other for inputs.

(7) Try to find the shortest paths from the speakers and headphone jacks back to DACs, though you may wish to use different DACs for different pins, so it's up to you to apply your own intelligence to solving this problem. The first connection in the connection list for a widgit will help you find the right path, but you may need to explore more items in the connection lists for some of the widgets.

(8) Try to find the shortest paths from the ADCs to any built-in microphone and microphone jack. Again you may wish to use different ADCs for the different inputs or you may wish to share the same one.

(9) Set up the paths you want to use by setting an intitial volume/gain and turning off the mutes at every widget in the chain wherever there is one needing to be set. Use verb 2h to set Format for any DACs and ADCs you're going to use, and verb 706h to set the stream and the lowest channel. Verb 707 must be used to enable input/output at the pins (and read up on VrefEn for mic.s). Remember to set Unsol for the jacks too if you want pins to report headphones/microphone being plugged in or removed.

(10) For each path, create a list of volume controls, mutes and the range of numbers that can be used with each volume/gain control to guide the creation of suitable controls for the user to manipulate in order to control each input/output.

(11) Set EAPD to enable the external amp for the speakers.

After you've done all that, get the near end of the link set up to get the DMA engines to handle the streams (read the specification carefully and act on it) and with luck you'll soon have sound coming in and out. If it doesn't happen that easily, you can set up the DMA position buffer to see if the DMA engines are actually running and looping correctly through the cyclic stream buffers. If all is well there, you might find it useful to use the codec ID to find the datasheet for it just to make sure you aren't missing anything out in the way you're setting it up. Also, feel free to improve the instructions on this page so that others can gain from your solutions to any problems you encounter.

External links

- Intel High Definition Audio Specification (http://www.intel.co.uk/content/www/us/en/standards/high-definition-audio-specification.html)