LSE Blog

Operating systems, computer security, languages theory, and even more!

- **About us**
  - Main website
  - Git repositories
  - @lse_epita

  RSS Feed
- **Categories**
  - Events
  - Hardware
  - Language
  - Reverse Engineering
  - Security
  - System
    - Linux
  - Tutorials
    - Parallelism
    - PythonGDB
  - Writeups
    - CSAW CTF 2012 Quals
    - DEFCON 2013 Quals
    - DEFCON2K12 Prequals
    - Hack.lu CTF 2012
    - Hack.lu CTF 2013
    - NDH2K12 Prequals
    - NDH2K13 Quals
    - Olympic-CTF 2014
    - PlaidCTF 2012
    - SecuInside2K12 Prequals
    - ebCTF 2013
- **Authors**
  - ✉ Samuel Angebault
  - ✉ Remi Audebert
  - ✉ Jean-Loup Bogalho
  - ✉ Pierre Bourdon
  - ✉ Marwan Burelle
  - ✉ Samuel Chevet
  - ✉ Pierre-Marie de Rodat
  - ✉ Ivan Delalande
  - ✉ Corentin Derbois
  - ✉ Nassim Eddequiouaq
  - ✉ Louis Feuvrier
  - ✉ Fabien Goncalves
  - ✉ Nicolas Hureau
  - ✉ Gabriel Laskar
  - ✉ Franck Michea
  - ✉ Bruno Pujos
  - ✉ Clement Rouault
  - ✉ Pierre Surply

Emulating the Gamecube audio processing in Dolphin

Written by Pierre Bourdon
2012-12-03 20:00:00

For the last two weeks, I've been working on enhancements and bug fixes related to audio processing in the Dolphin Emulator (the only Gamecube/Wii emulator that allows playing commercial games at the moment). Through this project I have learned a lot about how audio processing works in a Gamecube. Very little documentation is available on that subject, so I think writing an article explaining how it works might teach some new things to people interested in Gamecube/Wii homebrew development or emulators development. This article was first published in 3 parts on the Dolphin official forums. Before publishing it on the blog, I made some small changes (mostly proof-reading and adding some complementary images) but most explanations are the same.

If you're interested in the code, it is available in the `new-ax-hle` branch on the official Google Code repository.

Let's start this exploration of audio emulation in a Gamecube emulator by looking at how the real hardware processes sound data.

How sound is processed in a Gamecube

There are three main internal components related to sound in a Gamecube: the ARAM, the AI and the DSP:

- ARAM is an auxiliary memory which is used to store sound data. The CPU cannot access ARAM directly, it can only read/write blocks of data from RAM to ARAM (or from ARAM to RAM) using DMA requests. As ARAM is quite large, games often use it to store more than sound data: for example, WWE Day of Reckoning 2 uses it to store animation data (and a bug in DMA handling causes a crash because the data it writes is corrupted).
- The AI (Audio Interface) is responsible for getting sound data from RAM and sending it to your TV. It performs an optional sample rate conversion (32KHz -> 48KHz) and converts the data to an analog signal that is sent through the cables to your audio output device. The input data is read at a regular interval from RAM (not ARAM), usually every 0.25ms 32 bytes of input data is read (each sound sample is 2 bytes, so 32 bytes is 16 sound samples, which is 8 stereo sound samples, and 8 samples every 0.25ms == 32KHz sound).
- The DSP is what processes all the sounds a game wants to play and outputs a single stereo stream. Its job is to perform volume changes on the sounds, sample rate conversion (converting 4KHz sounds which take less space to 32KHz sounds - this is needed because you can't mix together sounds that are not the same rate). It can optionally do a lot of other stuff with the sounds (delaying to simulate 3D sound, filtering, handling surround sound, etc.).

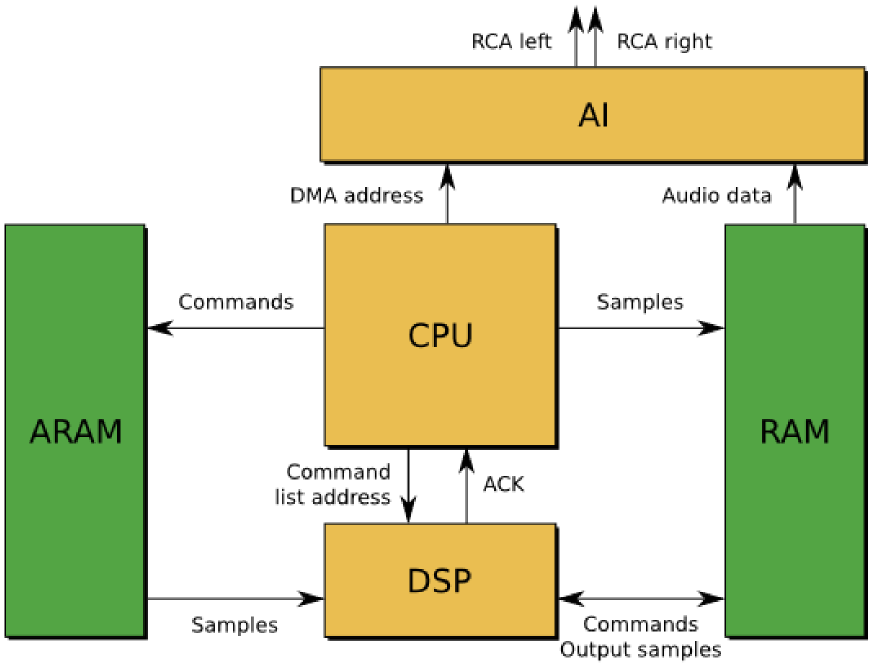

***Figure 1**: Overview of all the components involved in audio processing in a Gamecube*

ARAM and AI are not that hard to emulate: once you understand how they work, they are both simple chips which can only perform one function and don't communicate a lot with the CPU. You just need to have a precise enough timing for AI emulation, and everything is fine.

DSP is a lot harder to emulate properly, for two reasons I have not mentioned yet. First, it is a programmable CPU. All the mixing, filtering, etc. are part of a program that is sent to the DSP by the game, and the DSP behavior varies depending on the program it receives. For example, the DSP is not only used for sound processing, but also to unlock memory cards, and to cipher/decipher data sent to a GBA using the official link cable. Even for sound processing, not every game uses the same DSP code. The second reason is that it can communicate with the main Gamecube CPU, read RAM and ARAM and write to RAM. This allows games to use a complicated communication protocol between CPU and DSP.

We call a program running on the DSP a UCode ("*microcode*"). Because the DSP is programmable, it would seem like the only way to emulate it properly is to use low level emulation: running instructions one by one from a program to reproduce accurately what the real DSP does. However, while it is programmable, there are actually very few different UCodes used by games. On Gamecube, there are only 3 UCodes we know of: the AX UCode (used in most games because it is distributed with Nintendo's SDK), the Zelda UCode (called that way because it's used in Zelda games, but it is also used for some Mario games and some other first party games), and the JAC UCode (early version of the Zelda UCode, used in the Gamecube IPL/BIOS as well as *Luigi's Mansion*). That means if we can reproduce the behavior of these three UCodes, we can emulate the audio processing in most games without having to emulate the DSP instructions.

I started working on AX HLE 3 weeks ago because I want to play *Skies of Arcadia Legends* and *Tales of Symphonia*, two games that had completely broken audio with the previous AX HLE implementation. I added a new hack to "fix" the bug that caused bad music emulation, but fixing this made me even more interested in rewriting the whole thing to make it cleaner. I wasn't following Dolphin development when the current AX HLE was developed. However, it looks to me as if it was written without actually looking at the DSP code, only looking at what is sent to the DSP and what comes out. I don't know if people at the time had the capability to disassemble DSP code, but it is a very bad way to emulate AX anyway: some parts of the emulation code are completely WTF, and the more you understand how AX works the less you understand how the current AX HLE emulation was able to work and output sound in most cases. That's why, two weeks ago I decided I should start from scratch and re-implement AX HLE.

AX UCode features and internals

AX is a low-level audio library for Gamecube games, which comes with a builtin UCode to perform audio signal processing on the DSP. I'll first talk about what it can do, then explain how the UCode knows what it should do.

Luckily, Nintendo gives us a lot of information about the role of the DSP in a patent filed on Aug 23 2000: US7369665, "Method and apparatus for mixing sound signals". Figures 8, 9A and 9B are especially interesting in our case because they describe precisely what the DSP does internally and how inputs and outputs interact with each other. That helps, but most of this information could already be discovered by reverse engineering the UCode anyway (I learned the existence of this patent pretty late).

The basic role of the DSP is to get several sounds and mix them together to give a single sound. The sounds that it has to mix are provided through a list of *Parameter Blocks* (PB). Each PB corresponds to a sound to be mixed. It contains where to find the input sound data, but also a lot of configuration options: input sample rate, sound volume, where it should be mixed and at what volume (left channel/right channel/surround), if the sounds loop, from where does the loop start, etc.

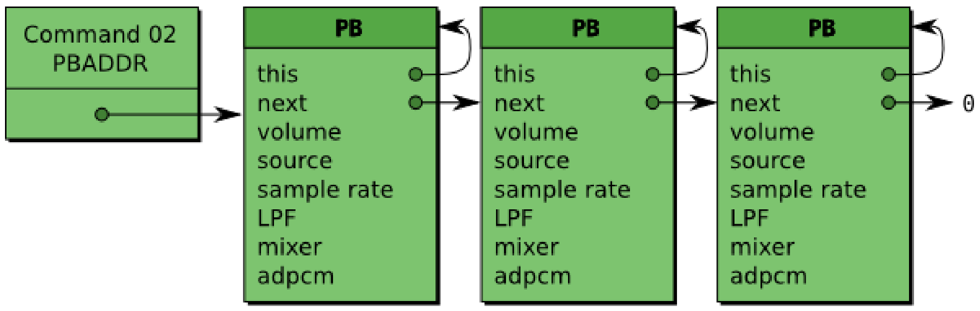

***Figure 2**: List of PBs with example fields. The* `PBADDR` *AX command gives the address of the first PB to the DSP.*

Every 5ms AX gets a list of PB and mixes each PB to 3 channels: Left, Right and Surround. It then sends 5ms of output to the RAM, at an address provided by the CPU. Sometimes being able to change sound data only every 5ms is not enough: to overcome that, each PB has a list of updates to be applied every millisecond. This allows sub-5ms granularity in sound mixing configuration. AX also provides a way to add audio effects on the L/R/S streams through the use of AUX channels. Each PB can be mixed to L/R/S but also to AUXA L/R/S and AUXB L/R/S. Then, the CPU can ask to get the contents of the AUXA and AUXB mixing buffers, replace them with its own data, and ask the DSP to mix AUXA and AUXB with the main L/R/S channels.

That's about it for the main features of AX. Some more things can be done optionally (for example *Initial Time Delay*, used to delay one channel to simulate 3D sound) but they are not used that often by games. Let's see how the CPU sends commands to the DSP.

The DSP has two ways to communicate with the game: through DMA, which allows it to read or write to RAM at any address it wants, and through mailbox registers, which is a more synchronous way to exchange small amounts of data (32 bits at a time) with the CPU. Usually, mailbox registers are used for synchronization and simple commands. For more complicated commands the CPU sends an address to the DSP via mailbox, and the DSP gets the data at this address through DMA.

With AX, about the only thing received through mailboxes (excluding UCode switching stuff which is not relevant to sound processing) is an address to a larger block of data which contains commands for the DSP. Here is a few commands that AX understands and that I have reverse engineered:

- Command 00: `SETUP`, initializes internal mixing buffers with a constant value or a value and a delta. Usually just initializes to 0.
- Command 02: `PBADDR`, gives the DSP the address in RAM of the first PB. Each PB contains the address of the next PB, so knowing only the address of the first PB is enough to get the whole list.
- Command 03: `PROCESS`, does all the audio processing and mixes the PBs to internal buffers.
- Command 04: `MIX_AUXA`, sends the contents of the AUXA buffers to the CPU, receives processed AUXA, and mix it with

the main channels.

- Command 05: `MIX_AUXB`, same as `MIX_AUXA` for AUXB
- Command 06: `UPLOAD_LRS`, sends the contents of the main L/R/S channels to the CPU.
- Command 0D: `MORE`, read more commands from RAM and start executing them. I suspect this is used for long command lists, but I've never seen it used.
- Command 0E: `OUTPUT`, interlaces L/R channel, clamp to 16 bits and send to RAM, where it will most likely get picked up by the Audio Interface.
- Command 0F: `END`, signals the end of a command list.

A few more commands exist, but these commands are the main things to handle to get audio working in most games I've found. Actually, only handling `PBADDR`, `PROCESS`, `OUTPUT` and `END` should allow about 90% of games to have some of the audio working (without stuff like AUX effects, used for echo/reverb).

When AX is done handling a command list, it sends an interrupt to the CPU to signal that it is ready to receive more data. This is very important because it is the only way for the CPU to know that the data it requested to be uploaded from the DSP is actually valid and done copying/processing. Then, at the next 5ms tick, the CPU will send a new command list to the DSP, and the cycle repeats.

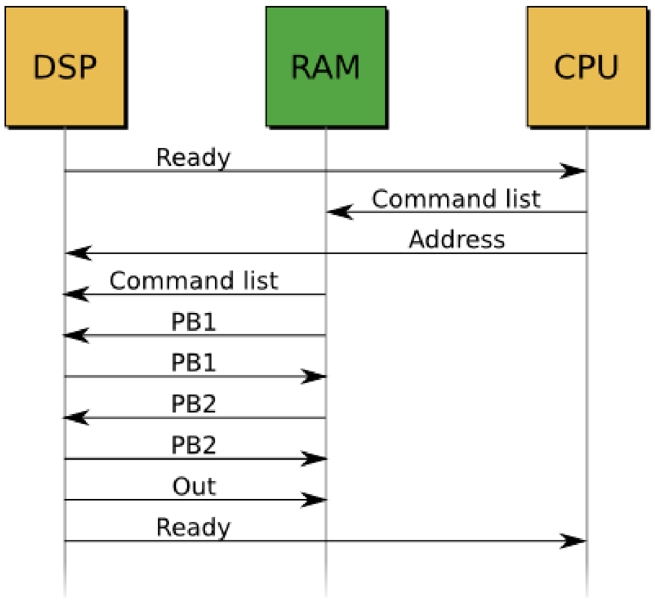

***Figure 3**: Timeline of an AX 5ms frame handling*

AX HLE in Dolphin, previous vs. new

DSP HLE was developed at a time when people did not know much about how the Gamecube DSP worked. It was basically a hack to have sound in games, and more hacks were added on top of that hack to try and fix bugs. The AX UCode emulation is probably the most hacky thing in the DSP HLE code. For example, some of the code that is used looks like this:

```
// TODO: WTF is going on here?!?
// Volume control (ramping)
static inline u16 ADPCM_Vol(u16 vol, u16 delta)
{
        int x = vol;
        if (delta && delta < 0x5000)
                x += delta * 20 * 8; // unsure what the right step is
                //x += 1 * 20 * 8;
        else if (delta && delta > 0x5000)
                //x -= (0x10000 - delta); // this is to small, it's often 1
                x -= (0x10000 - delta) * 20 * 16; // if this was 20 * 8 the sounds in Fire Emblem and Paper Mario
                        // did not have time to go to zero before the were closed
                //x -= 1 * 20 * 16;

         // make lower limits
        if (x < 0) x = 0;
        //if (pb.mixer_control < 1000 && x < pb.mixer_control) x = pb.mixer_control; // does this make
                // any sense?

        // make upper limits
        //if (mixer_control > 1000 && x > mixer_control) x = mixer_control; // maybe mixer_control also
                // has a volume target?
        //if (x >= 0x7fff) x = 0x7fff; // this seems a little high
        //if (x >= 0x4e20) x = 0x4e20; // add a definitive limit at 20 000
        if (x >= 0x8000) x = 0x8000; // clamp to 32768;
        return x; // update volume
}
```

I don't even know how this code evolved to become what it is displayed here, I just know that it is not a good way to implement AX HLE. Also, some of the design choices in the previous implementation just couldn't allow for accurate HLE.

The first issue is that the audio emulation pipeline was simply not correct: the AI was completely bypassed, and sound went directly from the DSP to the emulated audio mixer, without being copied to RAM at any time. This "kind of" works but completely breaks CPU audio effects... which aren't emulated anyway.

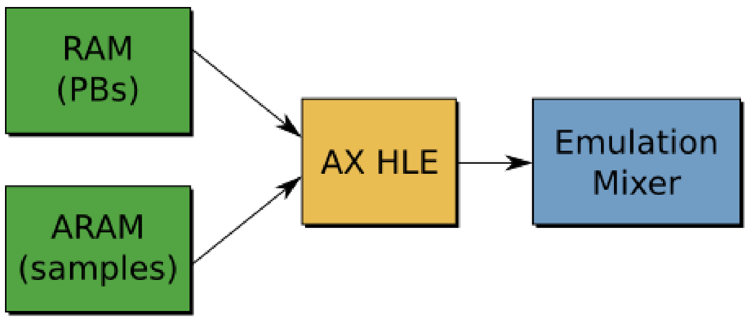

***Figure 4**: Audio emulation pipeline in the previous AX HLE implementation*

But the biggest issue is the timing on which AX HLE was working. On real hardware, the DSP runs on its own clock. At some point the CPU sends commands to it, it processes all of these commands as fast as possible, and sends a message back to the CPU when it's done. The CPU copies the processed data, then when it needs more data (in most cases, 5ms later) it sends new commands to the DSP. In the previous AX HLE implementation, none of that was right. What the emulated AX did was:

- As soon as we get the command that specified the sounds that should be mixed, copy the sound data address somewhere.
- Every 5ms send a message to the CPU saying that we processed the commands (even though no commands were processed)
- When the audio backend (ALSA, XAudio, DirectSound) requires more data, AX HLE mixed the sound and returned audio data.

Basically, nothing was right in the timing. That implementation allows for some cool hacks (like having the audio running at full speed even though the game is not running at 100% speed), but it is inaccurate and bug-prone.

When trying to fix the "missing instruments" bug affecting the games I wanted to play, I noticed all these timing issues and thought about rewriting AX HLE (once again... I always wanted to rewrite AX HLE every time I looked at the code). The hack fix (re4d18e3a8b7c) that I found to compensate for the timing issues really did not satisfy me, and knowing more about AX HLE I noticed that rewriting it was actually not as hard as I thought it would be. After working for 24h streight on `new-ax-hle`, I finally got a first working version which had ok sounds and music in *Tales of Symphonia*.

The design in `new-ax-hle` is in my opinion a lot better than the design used in the previous AX HLE:

- A DSP Thread is created when the UCode is loaded. This thread will be responsible for all the sound mixing work the DSP does.
- When we get commands from the CPU, we copy the command list to a temporary buffer, and wake up the DSP Thread to tell him we have commands to process.
- The DSP Thread handles the commands, sends a message to the CPU when it's done, and goes back to sleep.

It is basically the exact same model *DSP LLE on Thread* (another DSP configuration option in Dolphin) uses, with less synchronization (LLE tries to match the number of cycles executed on CPU and DSP, which causes some extra performance hit). This also kind of matches what happens on the real hardware, using 2 chips instead of 2 threads. However, this also means the audio processing speed is tied to the CPU speed: if the CPU cannot keep up, it won't send commands often enough and the audio backend won't receive enough data to avoid stuttering.

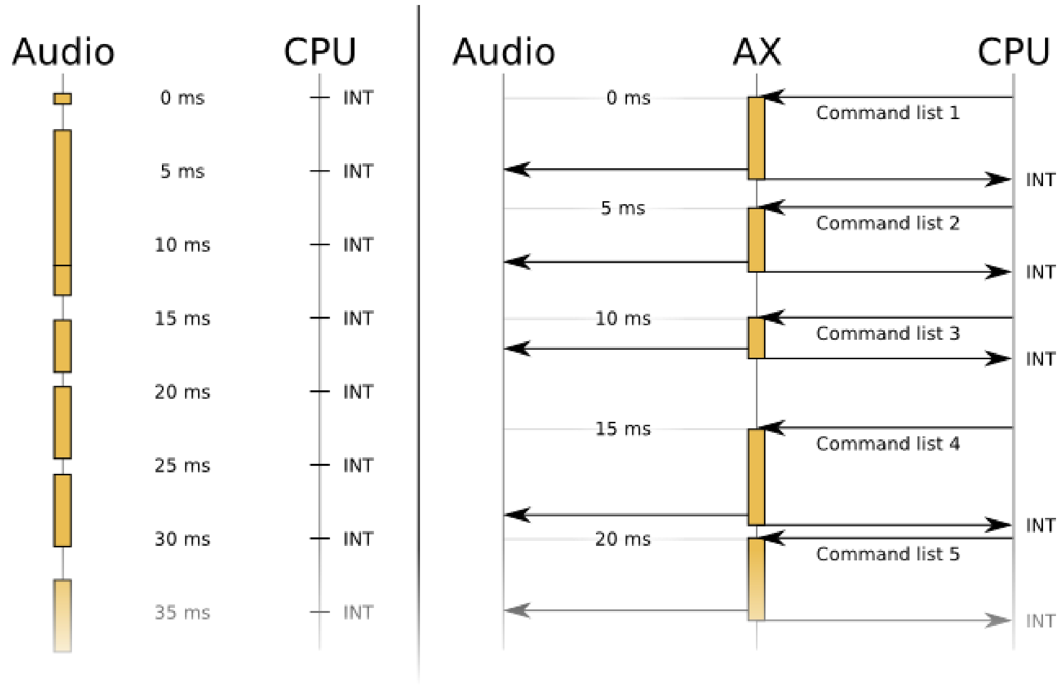

***Figure 5**: Comparison of processing timelines. On the left, previous implementation. On the right, `new-ax-hle`.*

Another change, this time not exactly linked to overall design, is that the `new-ax-hle` now handles most AX commands instead of only the one specifying the first parameter block address like the old AX does. Some of these other commands are used to set up global volume ramping, send data back to the main RAM, mix additional data from the RAM, or output samples to the buffers used by the audio interface. This means new-ax-hle now follows the correct audio emulation pipeline: `ARAM -> DSP -> RAM -> AI -> Output` (instead of the pipeline used before: `ARAM -> DSP -> Output`). This also means some CPU sound effects like echo, reverb, etc. should work fine.

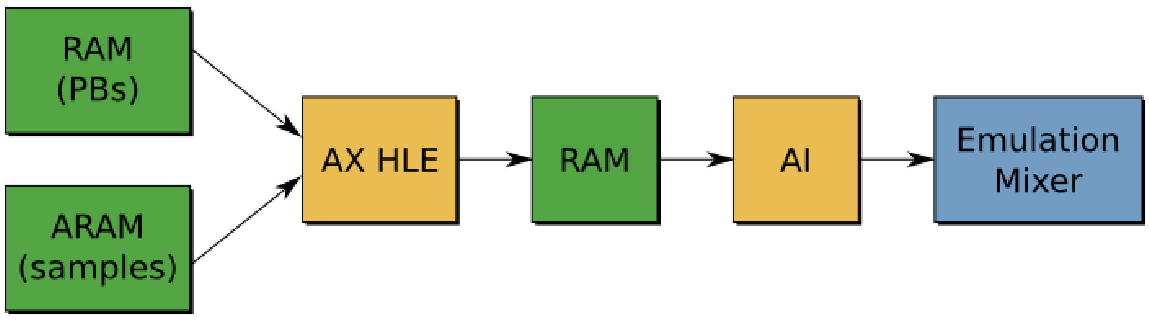

***Figure 6**: Audio emulation pipeline in the new AX HLE implementation*

Overall, the more I fix bugs in `new-ax-hle`, the more I'm amazed the previous AX HLE could work so well. It is a pile of hacks, implementing only 2/19 AX commands (and one of these commands is not even implemented correctly), with a completely wrong timing, and some ugly code that makes no sense. I don't blame the previous authors of this code - at the time, documentation about the DSP was a lot sparser, and analyzing UCodes had to be done with a text editor because there was no awesome IDA plugin for the GC DSP.

Conclusion

At the time I'm writing this article, `new-ax-hle` works a lot better than the previous AX HLE in most Gamecube games, and only a few remaining bugs are known in GC games. The Wii AX code is a bit less mature and is more like a proof of concept: I haven't really worked a lot on it, and after one or two weeks of bug fixing it should also become pretty much perfect, including Wiimote audio emulation (which was only supported with LLE previously). I'm hoping this code will be merged for 4.0, and I'll most likely be working on Zelda UCode HLE next (which has a less ugly implementation but has the same design issues as AX).

Thanks to Pierre-Marie (`pmderodat@lse`) for his nice Inkscape-made pictures.

Tweet 26

Permalink & comments

5 Comments LSE Blog 1 Login

Recommend Share Sort by Best

Join the discussion…

Faviann Di Tullio • 3 years ago
Really interesting read. You're pretty good at simplifying technical stuff. Thanks for taking the time to share the knowledge.
1 • Reply • Share ›

Frank Tackitt • 3 years ago
As someone who has done a tiny bit of hacking on Dolphin, and was scared away due to the old HLE implementation, thank you very much.

Also, when doing the Zelda UCode, if you manage to get the GBA link part working without LLE, you would be my hero
1 • Reply • Share ›

Rowan Bell • 2 years ago
mother fucking post
• Reply • Share ›

Anonymous • 3 years ago
I always thought the GameCube DSP was like a powerful Sony SPC700 in the Super Nintendo where their was hardware channels to be emulated where SNES had 8 and GameCube had 64 with both systems supporting ADPCM as that what specs made it seem to be at first.
• Reply • Share ›

Jonathan Souty • 3 years ago
"After working for 24h streight on new-ax-hle , I finally got a first working version which had ok sounds and music in Tales of Symphonia"

It's possible to get this version of your new ax hle ?
• Reply • Share ›

ALSO ON **LSE BLOG** WHAT'S THIS?

ebCTF 2013: Network challenges: NET100, NET200, NET300
4 comments • 2 years ago
m00! — Well done, another solution was to use tc (cf http://mytestbed.net/projects/... -> "Different delay on each destination"). Of course it was a little bit ...

ebCTF 2013: PWN300
5 comments • 2 years ago
hérvé leblanc — Thanks

Dealing with the pull-up resistors on AVR
2 comments • 2 years ago
Pierre Surply — The temperature sensors aren't connected with Ethernet, but with a RJ45 cable which two wires are used to supply the NTC and one to ...

ebCTF 2013: FOR100
11 comments • 2 years ago
Testuser :) — See my comment above for another user, maybe it helps!

Subscribe Add Disqus to your site Privacy