Building an Xbox 360 Emulator, part 1: Feasibility/CPU

Questions

Emulators are complex pieces of software and often push the bounds of what's possible by nature of having to simulate different architectures and jump through crazy hoops. When talking about the 360 this gets even crazier, as unlike when emulating an SNES the Xbox is a thoroughly modern piece of hardware and in some respects is still more powerful than most mainstream computers. So there's the first feasibility question: **is there a computer powerful enough to emulate an Xbox 360?** (sneak peak: I think so)

Now assume for a second that a sufficiently fast emulator could be built and all the hardware exists to run it: how would one even know what to emulate? Gaming hardware is almost always completely undocumented and very special-case stuff. There are decades-old systems that are just now being successfully emulated, and some may never be possible! Add to the potential hardware information void all of the system software, usually locked away under super strong NDA, and it looks worse. It's amazing what a skilled reverse engineer can do, but there are limits to everything. **Is there enough information about the Xbox 360 to emulate it?** (sneak peak: I think so)

Research

The Xbox 360 is an embedded system, geared towards gaming and fairly specialized – but at the end of the day it's derived from the Windows NT kernel and draws with DirectX 9. The hardware is all totally custom (CPU/GPU/memory system/etc), but roughly equivalent to mainstream hardware with a 64-bit PPC chip like those shipped in Macs for awhile and an ATI video chipset not too far removed from a desktop card. Although it's not going to be a piece of cake and there are some significant differences that may cause problems, this actually isn't the worst situation.

The next few posts will investigate each core component of the system and try to answer the two questions above. They'll cover the CPU, GPU, and operating system.

CPU (Xenon)

Reference: http://en.wikipedia.org/wiki/Xenon_(processor), http://free60.org/Xenon_(CPU)

- 64-bit PowerPC w/ in-order execution and running big-endian
- 3.2GHz 3 physical cores/6 logical cores
- L1: 32KB instruction/32KB data, L2: 1MB (shared)
- Each core has 32 integer, 32 floating-point, and 128 vector registers
- Altivec/VMX128 instructions for SIMD floating-point math
- ~96GFLOPS single-precision, ~58GFLOPS double-precision, ~9.6GFLOPS dot product

PowerPC

The PowerPC instruction set is RISC – this is a good thing, as it's got a fairly small set of instructions (relative to x86) – it doesn't make things much easier, though. Building a translator for PPC to x86-* is a non-trivial piece of work, but not that bad. There are some considerations to take into account when translating the instruction set and worrying about performance, highlighted below:

- Xenon is 64-bit – meaning that it uses instructions that operate on 64-bit integers. Emulating 64-bit on 32-bit instruction sets (such as x86) is not only significantly more code but also at least 2x slower. May mean x86-64 only, or letting some other layer do the work if 32-bit compatibility is a must.

- Xenon uses in-order execution – great for simple/cheap/power-efficient hardware, but bad for performance. Optimizing compilers can only do so much, and instruction streams meant for in-order processors should always run faster on out-of-order processors like the x86.
- The shared L2 cache, at 1MB, is fairly small considering there is no L3 cache. General memory accesses on the 360 are fast, but not as fast as the 8MB+ L3 caches commonly found in desktop processors.
- PPC has a large register file at 32I/32F/128V relative to x86 at 6I/8F/8V and x86-64 at 12I/16F&V – assuming the PPC compiler is fully utilizing them (or the game developers are, and it's safe to bet they are) this could cause a lot of extra memory swaps.
- Being big-endian makes things slightly less elegant, as all loads and stores to memory must take this into account. Operations on registers are fine (a lot of the heavy math where perf really matters), but because of all the clever bit twiddling hacks out there memory must always be valid. This is the biggest potentially scary performance issue, I believe.

Luckily there is a tremendous amount of information out there on the PowerPC. There are many emulators that have been constructed, some of which run quite fast (or could with a bit of tuning). The only worrisome area is around the VMX128 instructions, but it turns out there are very few instructions that are unique to VMX128 and most are just the normal Altivec ones. (If curious, the v*128 named instructions are VMX128 – the good news is that they've been documented enough to reverse).

Multi-core

'6 cores' sounds like a lot, but the important thing to remember is that they are hardware threads and not physical cores. Comparing against a desktop processor it's really 3 hardware cores at 3.2GHz. Modern Core i7′s have 4-6 hardware cores with 8-12 hardware threads – enough to pin the threads used on a 360 to their own dedicated hardware threads on the host.

There is of course extra overhead running on a desktop computer: you've got both other applications and the host OS itself fighting for control of the execution pipeline, caches, and disk. Having 2x the hardware resources, though, should be plenty from a raw computing standpoint:

- SetThreadAffinityMask/SetThreadIdealProcessor and equivalent functions can control hardware threading.
- The properties of out-of-order execution on the desktop processors should allow for better performance of hardware threads vs. the Xenon.
- The 3 hardware cores are sharing 1MB of L2 on the Xenon vs. 8-16MB L3 on the desktop so cache contention shouldn't happen nearly as often.
- Extra threads on the host can be used to offload tasks that on a real Xenon are sharing time with the game, such as decompression.

Raw performance

The Xbox marketing guys love to throw around their fancy GFLOP numbers, but in reality they are not all that impressive. Due to the aforementioned in-order execution and the strange performance characteristics of a lot of the VMX128 instructions it's almost impossible to hit the reported numbers in anything but carefully crafted synthetic benchmarks. This is excellent, as modern CPUs are exceeding the Xenon numbers by a decent margin (and sometimes by several multiples). The number of registers certainly helps the Xenon out but only experimentation will tell if they are crucial to the performance.

Emulating a Xenon

With all of the above analysis I think I can say that it's not only possible to emulate a Xenon, but it'll likely be sufficiently fast to run well.

To answer the first question above: by the time a Xenon emulation is up to 95% compatibility the target host processors will be plenty fast; in a few years it'll almost seem funny that it was ever questioned.

And is there enough information out there? So far, yes. I spent a lot of nights reverse engineering the special instructions on the PSP processor and the Xenon is about as documented now. The Free60 project has a decent toolchain but is lacking some of the VMX128 instructions which will make testing things more difficult, but it's not impossible.

Combined with some excellent community-published scripts for IDA Pro (which I have to buy a new license of… ack $$$ as much as a new MacBook) the publicly available information and some Redbull should be enough to get the Xbox 360 CPU running on the desktop.

Building an Xbox 360 Emulator, part 2: Feasibility/GPU

Research

Following up from last post, which dove into the Xbox 360 CPU, this post will look at the GPU.

GPU (Xenos)

Reference: http://en.wikipedia.org/wiki/Xenos_(graphics_chip), http://free60.org/Xenos_(GPU)

- ATI R500 equivalent at 500MHz
- 48 shader pipeline processors
- 8 ROPs – 8-32 gigasamples/second
- 6 billion vertices/second, 500 million triangles/second, 48 bilion shader ops/second
- Shader Model 3.0+
- 26 texture samplers, 16 streams, 4 render targets, 4096 VS/PS instructions
- VFETCH, MEMEXPORT

The Xenos GPU was derived from a desktop part right around the time when Direct3D 10 hardware was starting to develop. It's essentially Direct3D 9 with a few additions that enable 10-like features, such as VFETCH. Performance-wise it is quite slow compared to modern desktop parts as it was a bit too early to catch the massive explosion in generally programmable hardware.

Performance

The Xenos is great, but compared to modern hardware it's pretty puny. Where as the Xenon (CPU) is a bit closer to desktop processors, the GPU hardware has been moving forward at an amazing pace and it's clearly visible when comparing the specs.

- Modern (high-end) GPUs run at 800+MHz – many more operations/second.
- Modern GPUs have orders of magnitude more shader processors (multiplying the clock speed change).
- 32-128 ROPs multiply out all the above even more.

- Most GPUs have shader ops measured in trillions of operations per second.
- All stats (for D3D10+) are way over the D3D9 version used by Xenos (plenty of render targets/etc).

Assuming Xenos operations could be run on a modern card there should be no problem completing them with time to spare. The host OS takes a bit of GPU time to do its compositing, but the is plenty of memory and spare cycles to handle a Xenos-maxing load.

VFETCH

One unique piece of Xenos is the vfetch shader instruction, available from both vertex and pixel shaders, which gives shader programs the ability to fetch arbitrary vertex data from samplers set to vertex buffers. This instruction is fairly well documented because it is usable from XNA Game Studio, and some hardcore demoscene guy actually reversed a lot of the patch up details (available here with a bunch of other goodies). It also looks like you can do arbitrary texture fetches (TFETCH?) in both vertex and pixel shaders – kind of tricky.

Unfortunately, the ability to sample from arbitrary buffers is not something possible in Direct3D 9 or GL 2. It's equivalent to the Buffer.Load call in HLSL SM 4+ (starting in Direct3D 10).

MEMEXPORT

Unlike vfetch, this shader instruction is not available in XNA Game Studio and as such is much less documented. There are a few documents and technical papers out there on the net describing what it does, which as far as I can tell is similar to the RWBuffer type in HLSL SM5 (starting in Direct3D 11). It basically allows structured write of resource buffers (textures, vertex buffers, etc) that can then be read back by the CPU or used by another shader.

This will be the hardest thing to fully support due to the lack of clear documentation and the fact that it's a badass instruction. I'm hoping it has some fatal flaw that makes it unusable in real games such that it won't need to be implemented…

Emulating a Xenos

So we know the performance exists in the hardware to push the triangles and fill the pixels, but it sounds tricky. VFETCH is useful enough to assume that every game is using it, while the hope is that MEMEXPORT is hard enough to use that no game is. There are several big open questions that need more investigation to say for sure just how feasible this project is:

- Is it possible to translate compiled shader code from xvs_3_0/xps_3_0 -> SM4/5? (Sure… but not trivial)
- Can VFETCH semantics be implemented in SM4/5? (I think yes, from what I've seen)
- Can MEMEXPORT semantics (whatever they are) be implemented in SM4/5?
- Special z-pass handling may be needed (seen as 'zpass' instruction) – may require replicating draw calls and splitting shaders!

Unlike the CPU, which I'm pretty confident can be emulated, the Xenos is a lot trickier. Ignoring the more advanced things like MEMEXPORT for a second there is a tremendous amount of work that will need to be done to get **anything** rendering once the rest of the emulator is going due to the need to translate shader bytecode. The XNA GS shader compiler can compile and disassemble shaders, which is a start for reversing, but it'll be a pain.

Because of all the GPGPU-ish stuff happening it seems like for at least an initial release Direct3D 11 (with feature level 10.1) is the way to go. I was really hoping to be cross-platform right away, but I'm not positive OpenGL has the necessary support.

So after a day of research I'm about 70% confident I could get **something** rendering. I'm about 20% confident with my current knowledge that a real game that fully utilized the hardware could be emulated. If someone like the guy who reversed the GPU hardware interface decided to play around, though, that number would probably go up a lot ^_^

Building an Xbox 360 Emulator, part 3: Feasibility/OS

Research

The final part of the research phase (previously: CPU, GPU), this post discusses what it would take to emulate the OS on the 360.

I've decided to name the project Xenia, so if you see that name thrown around that's what I'm referring to. I thought it was cute because of what it means in Greek (think host/guest as common terms in emulation) and it follows the Xenon/Xenos naming of the Xbox 360.

Xbox Operating System

Reference: xorloser's x360_imports, Wine, MSDN, ReactOS, various forum posts

The operating system on the Xbox 360 is commonly thought to be a paired down version of the Windows 2000 kernel. Although it has a similar API, it is in fact a from-scratch implementation. The good news is that even though the implementation differs (although I'm sure there's some shared code) the API is really all that matters and that API is largely documented and publicly reverse engineered.

Cross referencing some disassembled executables and xorloser's x360_imports file, there's a fairly even split of public vs. private APIs. For every KeDelayExecutionThread that has nice MSDN documentation there's a XamShowMessageBoxUIEx that is completely unknown. Scanning through many of the imported methods and their call signatures their behavior and function can be inferred, but others (like XeCryptBnQwNeModExpRoot) are going to require a bit more work. Some map to public counterparts that are documented, such as XamGetInputState being equivalent to XInputGetState.

One thing I've noticed while looking through a lot of the used API methods is that many are at a much lower level than one would expect. Since I know games aren't calling kernel methods directly, this must mean that the system libraries are actually statically compiled into the game executables. Let that sink in for a second. It'd be like if on Windows every single application had all of Win32 compiled into it. I can see why this would make sense from a versioning perspective (every game has the exact version of the library they built against always properly in sync), but it means that if a bug is found every game must be updated. What this means for an emulator, though, is that instead of having to implement potentially thousands of API calls there are instead a few hundred simple, low-level calls that are almost guaranteed to not differ between games. This simultaneously makes things easier and harder; on one hand, there are fewer API calls to implement and they should be easier to get right, but on the other hand there may be several methods that would have been much easier to emulate at a higher level (like D3D calls).

Xbox Kernel (xboxkrnl.exe)

Every Xbox has an xboxkrnl module on it, exporting an API similar to ntoskrnl on desktop Windows plus a bunch of additional APIs.

It provide quite a useful set of functionality:

- Program control
- Synchronization primitives (events, semaphores, critical sections)
- Threading
- Memory management
- Common string routines (Unicode comparison, vsprintf, etc)
- Cryptographic providers (DES, HMAC, MD5, etc)
- Raw IO
- XEX file handling and module loading (like LoadLibrary)
- XAudio, XInput, XMA
- Video driver (Vd*)

Of the 800 or so methods present there are a good portion that are documented. Even better, Wine and ReactOS both have most method signatures and quite a few complete implementations.

Some methods are trivial to get emulated – for example, if the host emulator is built on Windows it can often pass down things like (Ex)CreateThread and RtlInitializeCriticalSection right down to the OS and utilize the optimized implementation there. Because the NT API is used there are a lot of these. Some aren't directly exposed to user code (as these are all kernel functions) but can be passed through the user-level functions with only a bit of rewriting. It's possible, with the right tricks, to make these calls directly on desktop Windows (it usually requires the Windows Device Driver Kit to be setup), which would be ideal.

The set that looks like it will be the hardest to properly get figured out are the video methods, like VdInitializeRingBuffer and VdRegisterGraphicsNotification, as it appears like the API is designed for direct writing to a command buffer instead of making calls. This means that, as far as the emulator is concerned, there are no methods that can be intercepted to do useful work – instead, at certain points in time, giant opaque data buffers must be processed to do interesting things. This can make things significantly faster by reducing the call overhead between guest code (the emulated PowerPC instructions) and host code, but makes reverse engineering what's going on much more difficult by taking away easily identifiable call sites and instead giving multi-megabyte data blobs. Ironically, if this buffer format can be reversed it may make building a D3D11/GL3 backend easier than if all the D3D9 state management had to be emulated perfectly.

XAM/Xbox ?? (xam.xex)

Besides the kernel there is a giant library that provides the bulk of the higher-level functionality in Xbox titles. Where the kernel is the tiny set of low-level methods required to build a functioning OS, XAM is the dumping ground for the rest.

- Winsock/XNet Xbox Live networking
- On-screen keyboard
- Message boxes/UI
- XContent package file handling
- Game metadata
- XUI (Xbox User Interface) loading/rendering/etc
- User-level file IO (OpenFile/GetFileSize/etc)
- Some of the C runtime
- Avatars and other user information

This is where things get interesting. Luckily it looks like XAM is **everything** one can do on a 360, including what the dashboard and system uses to do its work (like download queues and such) and not all games use all of the methods: most seem to only use a few dozen out of the 3000 exports.

In the context of getting an emulator up and running almost all of the methods can be ignored. Simple 'hello world' applications link in only about 4, and the games I've looked at largely depend on it for error messages and multiplayer functionality – if the emulator starts without any networking, most of those methods can be stubbed. I haven't seen a game yet that uses XUI for its user interface, so that can be skipped too.

Emulating the OS

Now that the Xbox OS is a bit more defined, let's sketch out how best to emulate it. There are two primary means of emulating system software: low-level emulation (LLE) and high-level emulation (HLE).

Most emulators for early systems (pre-1990′s) use low-level emulation because the game systems didn't include an OS and often had a very minimal BIOS. The hardware was also simple enough (even though often undocumented) that it was easier to model the behavior of device registers than entire BIOS/OS systems – this is why early emulators often require a user to find a BIOS to work.

As hardware grew more complex and expensive to emulate high-level emulation started to take over. In HLE the BIOS/OS is implemented in the emulator code (so no original is required) and most of the underlying hardware is abstracted away. This allows for better native optimizations of complex routines and eliminates the need to rip the copyrighted firmware off a real console. The downside is that for well-documented hardware it's often easier to build hardware simulators than to match the exact behavior of complex operating systems.

I had good luck with building an HLE for my PSP emulator as the hardware was sufficiently complex as to make it impossible to support a perfect simulation of it. The Xbox 360 is the same way – no one outside of Microsoft (or ATI or whoever) knows how a lot of the hardware in the system operates (and may never know, as history has shown). We do know, however, how a lot of the NT kernel works and can stub out or mimic things like the dashboard UI when required. For performance reasons it also makes sense to not have a full-on simulation of things like atomic primitives and other crazy difficult constructs.

So there's one of the first pinned down pieces of the emulator: it will be a high-level emulator. Now let's dive in to some of the major areas that need to be implemented there. Note that these areas are what one would look at when building an operating system and that's essentially what we will be doing. These are roughly in the order of implementation, and I'll be covering them in detail in future posts.

Memory Management

Before code can be loaded into memory there has to be a way to allocate that memory. In an HLE this usually means implementing the common alloc/free methods of the guest operating system – in the Xbox's case this is the NtAllocateVirtualMemory method cluster. Using the same set of memory routines for all internal host emulator functions (like module loading, mentioned below) as well as requests from game code keeps things simple and reliable. Since the NT-like API of the 360 matches the Windows API it means that in almost all cases the emulator can use the host memory manager for all its request. This ensures performance (as it can't get any faster than that) and safety (as read/write/execute permissions will be enforced). Since everything is working in a sane virtual address space it also means that debugging things is much easier – memory addresses as visible to the emulated game code will correspond to memory addresses in the host emulator space. Calling host routines (like memcpy or video decompression libraries) require no fixups, and embedded pointers should work without the need for translation.

With my PSP emulator I made the mistake of not doing this first and ended up with two completely different memory spaces and ways of referencing addresses. Lesson learned: even though it's tempting to start loading code first, figuring out where to put it is more important.

There are two minor annoyances that make emulating the 360 a bit more difficult than it should be:

Big-Endian Data

The 360 is big-endian and as such data structures will need to be byte swapped before and after system API calls. This isn't nearly as elegant as being able to just pass things around. It's possible to write some optimized swap routines for specific structures that are heavily used such that they get inserted directly into translated code as optimally as possible, but it's not free.

32-bit pointers

On 64-bit Windows all pointers are 64-bits. This makes sense: developers want the large address space that 64-bit pointers gives them so they can make use of tons of RAM. Most applications will be well within the 4GB (or really 2GB) memory limit and be wasting 4 bytes per pointer but memory is cheap so no one cares. The Xbox 360, on the other hand, has only 512MB of memory to be shared between the OS, game, and video memory. 4 wasted bytes per pointer when it's impossible to ever have an address outside the 4 byte pointer range seems too wasteful, so the Xbox compiler team did the logical thing and made pointers 4 bytes in their 64-bit code.

This sucks for an emulator, though, as it means that host pointers cannot be safely round-tripped through the guest code. For example, if NtAllocateVirtualMemory returns a pointer that spills over 4 bytes things will explode when that 8 byte pointer is truncated and forced into a 4 byte guest pointer. There are a few ways around this, none of which are great, but the easiest I can think of is to reserve a large 512MB block that represents all of the Xbox memory at application start and ensure it is entirely within the 32-bit address range. This is easy with NtAllocateVirtualMemory (if I decide to use kernel calls in the host) but also possible with VirtualAlloc, although not as easy when talking about 512MB blocks. If all future allocations are made from this space it means that pointers can be passed between guest and host without worrying about them being outside the allowable range.

Executables and Modules

Operating systems need a way to load and execute bundles of code. On the 360 these are XEX files, which are packages that contain a bunch of resources detailing a game as well as an embedded PE-formatted EXE/DLL file containing the actual code. The emulator will then require a loader that can parse one of these files, extract the interesting content, and place it into memory. Any imports, like references to kernel methods implemented in the emulator, will be resolved and patched up and exports will be cataloged until later used. Finally the code can be submitted to the subsystem handling translation/JIT/etc for actual execution.

There are a few distinct components here:

XEX Parsing

This is fairly easy to do as the XEX file format is well documented and there are many tools out there that can load it. Basic documentation is available on the Free60 site but the best resource is working code and both the xextool_v01 and abgx360 code are great (although the abgx360 source is disgusting and ugly). Some things of note are that all metadata required by various Xbox API calls like XGetModuleSection are here, as well as fun things to pull out that the dashboard usually consumes like the game icon and title information.

PE (Portable Executable) Parsing

The PE file format is how Microsoft stores its binaries (equivalent to ELF on Unix) for both executables and dynamically linked libraries – the only difference between an EXE and a DLL is its extension, from a format perspective. Inside each XEX is a PE file in the raw. This is great, as the PE format is officially documented by Microsoft and kept up to date, and surprisingly they document the **entire** spec including the PowerPC variant.

A PE file is basically a bunch of regions of data (called sections); some are placed there by the linker (such as the TEXT section containing compiled code and DATA section containing static data) and others can be added by the user as custom resources (like localized strings/etc). Two other special sections are IDATA, describing what methods need to be imported from system libraries, and RELOC, containing all the symbols that must be relocated off of the base address of the library.

Loader Logic

Once the PE is extracted from the XEX it's time to get it loaded. This requires placing each section in the PE at its appropriate location in the virtual address space and applying any relocations that are required based on the RELOC section. After that an ahead-of-time translator can run over the instructions in memory and translate them into the target machine format. The translator can use the IDATA section to patch imported routines and syscalls to their host emulator implementations and also log exported code if it will be used later on. This is a fairly complex dance and I'll be describing it in a future post. For now, the things to note are that the translated machine code lives outside of the memory space of the emulator and data references must be preserved in the virtual memory space. Think of the translated code and data as a shadow copy – this way, if the game wants to read itself (code or data) it would see exactly what it expects: PowerPC instructions matching those in the original XEX.

Module Data Structures

After loading and translating a module there is a lot of information that needs to stick around. Both the guest code and the emulator will need to check things in the future to handle calls like LoadLibrary (returning a cached load), GetProcAddress (getting an export), or XGetModuleSection (getting a raw section from the PE). This means that for everything loaded from a XEX and PE there will need to be in-memory counterparts that point at all the now-patched and translated resources.

Interface for Processor Subsystem

One thing I've been glossing over is how this all interacts with the subsystem that is doing the translation/JIT/etc of the PowerPC instructions. For now, let's just say that there has to be a decent way for it to interact with the loader so that it can get enough information to make good guesses about what code does, how the code interacts with the rest of the system, and notify the rest of the emulator about any fixes it performs on that code.

Threads and Synchronization

Because the 360 is a multi-core system it can be assumed that almost every game uses many software threads. This means that threading primitives like CreateThread, Sleep, etc will be required as well as all the synchronization primitives that support multi-threaded applications (locks and such). Because the source API is fairly high-level most of these should be easy to pass down to the host OS and not worry too much about except where the API differs.

This is in contrast to what I had to do when working on my PSP emulator. There, the Sony threading APIs differed enough from the normal POSIX or Win32 APIs that I had to actually implement a full thread scheduler. Luckily the PSP was single-core, meaning that only one thread could be running at a time and a lot of the synchronization primitives could be faked. It also greatly reduced the JIT complexity as only one thread could be generating code at a time and it was easy to ensure the coherency of the generated code.

A 360 emulator must fully utilize a multi-core host in order to get reasonable performance. This means that the code generator has to be able to handle multiple threads executing and potentially generating code at the same time. It also means that a robust thread scheduler has to be able to handle the load of several threads (maybe even a few dozen) running at the same time with decent performance. Because of this I'm deciding to try to use the host threading system instead of writing my own. The code generator will need to be thread safe, but all threading and synchronization primitives will defer to the host OS. Windows, as a host, will have a much better thread scheduler than I could ever write and will enable some fancy optimizations that would otherwise be unattainable, such as pinning threads to certain CPUs and spreading out threads such that they share cores when they would have on the original hardware to more closely match the performance characteristics of the 360.

Raw IO

Unlike threading primitives the IO system will need to be fully emulated. This is because on a real Xbox it's reading the physical DVD where as the emulator will be sourcing from a DVD image or some other file.

Most calls found in games come in two flavors:

Low-Level IO (Io* calls)

These kernel-level calls include such lovely methods as IoCreateDevice and IoBuildDeviceIoControlRequest. Since they are not usually exposed to user code my hope is that full general implementations won't be required and they will be called in predictable ways. Most likely they are used to access read-only game DVD data, so supporting custom drivers that direct requests down to the image files should be fairly easy (this is how tools on Windows that let you mount ISOs work). Once things like memory cards and harddrives are supported things get trickier, but it's not impossible and can be skipped initially.

High-Level IO (Nt* calls)

Roughly equivalent to the Win32 file API, NtOpenFile, NtReadFile, and various other functions allow for easier implementation of file IO. That said, if a full implementation of the low-level Io* routines needs to be implemented anyway it may make sense to implement these as calls onto that layer. The reason is that the Xbox DVD format uses a custom file system that will need to be built and kept in memory and calling down to the host OS file system won't really happen (although there are situations where I could it imagine it being useful, such as hot-patching resources).

Just like the memory management functions are best to be shared throughout both guest and host code, so are these IO functions. Getting them implemented early means less code later on and a more robust implementation.

A lot of the code for these methods can be found in ReactOS, but unfortunately they are GPL (ewww). That means some hack-tastic implementations will probably be written from scratch.

Audio/Video

Once more than 'hello world' applications are running things like audio and video will be required. Due to Microsoft pushing the XNA brand and libraries a lot of the technologies used by the Xbox are the same as they are on Windows. Video files are WMV and play just fine on the desktop and audio is processed through XAudio2 and that's easily mappable to the equivalent desktop APIs.

That said, the initial versions of the emulator will have to try to hard to skip over all of this stuff. Games are still perfectly playable without cut-scenes or music, and it's enough to know it's possible to continue on with implementation.

Notes

Static Linking Verification

As mentioned above it looks like many system methods get linked in to applications at compile-time. To quickly verify that this is happening I disassembled some games and looked at the import for KeDelayExecutionThread (I figured it was super simple). In every game I looked at there was only one caller of this method and that caller was identical. Since KeDelayExecutionThread is essentially sleep I looked at the x360_imports file and found both Sleep and RtlSleep. Sleep, being a Win32 API, is most likely identical to the signature of the desktop version so I assumed it took 1 parameter. The parent method to KeDelayExecutionThread takes 2, which means it can't be Sleep but is likely RtlSleep. The parent of this RtlSleep method takes exactly one parameter, sets the second parameter to 0, and calls down – sounds like Sleep! So then even though xboxkrnl exports Sleep, RtlSleep, and KeDelayExecutionThread the code for both Sleep and RtlSleep are compiled into the game executable instead of deferring to xboxkrnl. I have no idea why xboxkrnl exports these methods if games won't use them (it would certainly make life easier for me if they weren't there), but since it seems like no one is using them they can probably be skipped in the initial implementation.

Patching high-level APIs

Not all things are easier to emulate at a low level for both performance reasons and implementation quality.

To see this clearly take memcpy, a C runtime method that copies large blocks of memory around. Right now this method is compiled into every game which makes it difficult to hook into, unlike CreateThread and other exports of the kernel. Of course it'll work just fine to emulate the compiled PowerPC code (as to the emulator it's just another block of instructions), but it won't be nearly as fast as it could be. I'll dive into this more in a future article, but the short of it is that an emulated memcpy will require thousands to tens of thousands of host instructions to handle what is basically a few hundred instruction method. That's because the emulator doesn't know about the semantics of the method: copy, bit for bit, memory block A to memory block B. Instead it sees a bunch of memory reads and writes and value manipulation and must preserve the validity of that data every step of the way. Knowing what the code is really trying to do (a copy) would enable some optimized host-specific code to do the work as fast as possible.

The problem is that identifying blocks of code is difficult. Every compiler (and every version of that compiler), every runtime (and every version of that runtime), and every compiler/optimizer/linker setting will subtly or non-so-subtly change the code in the executable such that it'll almost always differ. What is memcpy in Game A may be totally different than memcpy in Game B, even though they perform the same function and may have originated from the same source code.

There are three ways around this:

- Ignore it completely
- Specific signature matching
- Fuzzy signature matching

The first option isn't interesting (although it'll certainly be how the emulator starts out).

Matching specific signatures requires a database of subroutine hashes that map to some internal call. When a module is loaded the subroutines are discovered, the database is queried, and matching methods are patched up. The problem here is that building that database is incredibly difficult – remember the massive number of potential variations of this code? It's a good first step and the database/patching functionality may be required for other reasons (like skipping unimplemented code in certain games/etc), but it's far from optimal.

The really interesting method is fuzzy signature matching. This is essentially what anti-virus applications do when trying to detect malicious code. It's easy for virus authors to obfuscate/vary their code on each version of their virus (or even each copy of it), so very sophisticated techniques have been developed for detecting these similar blocks of code. Instead of the above database containing specific subroutine hashes a more complex representation would allow for an analysis step to extract the matching methods. This is a hard problem, and would take some time, but it'd be a ton of fun.

What Next?

We've now covered the 3 major areas of the emulator in some level of detail (CPU, GPU, and OS) and now it's getting to be time to write some code. Before starting on the actual emulator, though, one major detail needs to be nailed down: what does the code translator look like? In the next post I'll experiment with a few different ways of building a CPU emulator and detail their pros and cons.

Building an Xbox 360 Emulator, part 4: Tools and Information

Before going much further I figured it'd be useful to document the setup I've been using to do my reversing. It should make it easier to follow along, and if I forget myself I've got a good reference. Note that I'm going off of publicly searchable information here – everything I've been doing is sourced from Google (and surprisingly sometimes Bing, which ironically indexes certain 360 hacking related information better than Google does!).

I'll try to update this list if I find anything interesting.

Primary Sources

Various Internet Searches

There's a wealth of information out there on the interwebs, although it's sometimes hard to find. Google around for these and you'll find interesting things…

Most of the APIs I've been researching turn up hits on pastebin, providing snippets of sample code from hackers and what I assume to be legit developers. None of it is tagged or labeled, but the API names are unique enough to find exactly the right information. Most of the method signatures and usage information I've gathered so far have come from sources like this.

PowerPC references:

- Altivec_PEM.pdf / AltiVec Technology Programming Environments Manual
- MPCFPE_AD_R1.pdf / PowerPC Microprocessor Family: The Programming Environments
- pem_64bit_v3.0.2005jul15.pdf / PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors
- PowerISA_V2.06B_V2_PUBLIC.pdf / Power ISA Version 2.06 Revision B

- vector_simd_pem_v_2.07c_26Oct2006_cell.pdf / PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual
- vmx128.txt

GPU references:

- R700-Family_Instruction_Set_Architecture.pdf / ATI R700-Family Instruction Set Architecture
- ParticleSystemSimulationAndRenderingOnTheXbox360GPU.pdf / interesting bits about memexport

Xbox 360 specific:

- xbox360-file-reference.pdf / Xbox 360 File Specifications Reference

Free60

- Xenon (CPU) & Xenos (GPU)
- Starting Homebrew Development (toolchain setup, LibXenon, etc)
- XEX file format documentation

The Free60 project is probably the best source of information I've found. The awesome hackers there have reversed quite a bit of the system for the commendable purpose of running Linux/homebrew, and have got a robust enough toolchain to compile simple applications.

I haven't taken the time to mod and setup their stack on one of my 360′s, but for the purpose of reversing and building an emulator it's invaluable to have documentation-through-code and the ability to generate my own executables that are far simpler than any real game. If not for the hard work of the ps2dev community I never would have been able to do my PSP emulator.

Microsoft

There's quite a bit of information out there if you know where to look and what to look for. Although not all specific to the 360, a lot of the details carry over.

mmlite

Microsoft Invisible Computing is a project that has quite a bit of code in it that provides a small RTOS for embedded systems. What makes it interesting (and how I found it) is that it contains several files enabling support for PowerPC architectures. It appears as though the code that ended up in here came from either the same place the Xbox team got their code from, or is even the origin of some of the Xbox toolchain code.

Specifically, the src/crt/md/ppc/xxx.s file has the __savegpr/__restgpr magic that tools like XexTool patch in to IDA dumps. Instead of guessing what all those tiny functions do it's now possible to see the commented code and how it's used. I'm sure there's more interesting stuff in here too (possibly related to the CRT), but I haven't had the need for it yet.

DDI API

A complete listing of all Direct3D9 API calls down to driver mode, with all of the arguments to them. The user-mode D3D implementation on Windows provided by Microsoft calls down into this layer to pass on commands to the driver. On the 360, this API (minus all the fixed function calls) **is** the command buffer format! Although the exact command opcodes are not documented here, there's enough information (combined with tmbinc's code below) to ensure a somewhat-proper initial implementation.

Direct3D Shader Codes

The details of the compiled HLSL bytecode are all laid out in the Windows Driver Kit. Minus some of the Xenos-specific opcodes, it's all here. This should make the shader translation code much easier to write.

DirectX Effect Compiler

By using the command line effect compiler (fxc.exe) it is possible to both assemble and disassemble HLSL for the Xenos – with some caveats. After rummaging through some games I was able to get a few compiled shaders and by munging their headers get the standard fxc to dump their contents (as most shaders are just vs_3_0 and ps_3_0).

The effect compiler in XNA Game Studio does not include the disassembler, but does support direct output of HLSL to binaries the Xenos can run. This tool, combined with the shader opcode information, should be plenty to get simple shaders going.

tmbinc's gpu code / 'gpu-0.0.5.tar.gz'

Absolutely incredible piece of reversing here, amounting to the construction of an almost complete API for the Xenos GPU. Even includes information about how the Xbox-specific vfetch instruction is implemented. When it comes to processing the command buffers, this code will be critical to quickly getting things normalized.

Cxbx

Once thought to be a myth, Cxbx is an Xbox 1 emulator capable of running retail games. A very impressive piece of work, its code provides insights into the 360 by way of the Xbox 1 having a very similar NT-like kernel. Although Cxbx had an easier life by virtue of being able to serve as mainly a runtime library and patching system (most of the x86 code is left untouched), the details of a lot of the NT-like APIs are very interesting. Through some research it seems like a lot have changed between the Xbox 1 and the 360 (almost enough so to make me believe there was a complete rewrite in-between), some of the finer differences between things like security handles and such should be consistent.

abgx360

Although it may be some of the worst C I've ever seen, the raw amount of information contained within is mind boggling – everything one needs to read discs and most of the files on those discs (and the meanings of all of the data) reside within the single-file, 860KB, 16000 line code.

xextool (xor37h/Hitmen)

There are many 'xextool's out there, but this version written way back in 2006 is one of the few that does the full end-to-end XEX decryption and has source available. Having not been updated in half a decade (and having never been fully developed), it cannot decode modern XEX files but does have a lot of what abgx360 does in a much easier-to-read form.

XexTool (xorloser)

The best 'xextool' out there, this command line app does quite a bit. For my purposes it's interesting because it allows the dumping of the inner executable from XEX files along with a .idc file that IDA can use to resolve import names. By using this it's possible to get a pretty decent view of files in IDA and makes reversing much easier. Unfortunately (for me), xorloser has never released the code and it sounds like he never will.

PPCAltivec IDA plugin (xorloser)

Another tool released by xorloser (originally from Dean Ashton), this IDA plugin adds support for all of the Altivec/VMX instructions used by the 360 (and various other PPC based consoles). Required when looking at real code, and when going to implement the decoder for these instructions the source (included) will prove invaluable, as most of the opcodes are undocumented.

Building an Xbox 360 Emulator, part 5: XEX Files

I thought it would be useful to walk through a XEX file and show what's all in there so that when I continue on and start talking about details of the loader, operating system, and CPU things make a bit more sense. There's also the selfish reason: it's been 6 months since I last looked at this stuff and I needed to remind myself ^_^

What's a XEX?

XEX files, or more accurately XEX2 files, are signed/encrypted/compressed content wrappers used by the 360. 'XEX' = **X**box **EX**ecutable, and pretty close to 'EXE' – cute, huh? They can contain resources (everything from art assets to configuration files) and executable files (Windows PE format .exe files), and every game has at least one: default.xex.

When the Xbox goes to launch a title, it first looks for the default.xex file in the game root (either on the disc or hard drive) and reads out the metadata. Using a combination of a shared console key and a game-specific key, the executable contained within is decrypted and verified against an embedded checksum (and sometimes decompressed). The loader uses some extra bits of information in the XEX, such as import tables and section maps, to properly lay out the executable in memory and then jumps into the code.

Of interest to us here is:

- What kind of metadata is in the XEX?
- How do you get the executable out?
- How do you load the executable into memory?
- How are imports resolved?

The first part of this document will talk about these issues and then I'll follow on with a quick IDA walkthrough for reversing a few functions and making sure the world is sane.

Getting a XEX File

Disc Image

There are tons of ways to get a working disc image, and I'm not going to cover them here. The information is always changing, very configuration dependent, and unfortunately due to stupid US laws in a grey (or darker) area. Google will yield plenty of results, but in the end you'll need either a modded 360 or a very specific replacement drive and a decent external SATA adapter (cheap ones didn't seem to work for me).

The rest of this post assumes you have grabbed a valid and legally-owned disc image.

Extracting the XEX

Quite a few tools will let you browse around the disc filesystem and grab files, but the most reliable and easy to use that I've found is wx360. There are some command line ones out there, FUSE versions for OSX/etc, and some 360-specific disc imaging tools can optionally dump files. If you want some code that does this in a nice way, wait until I open up my github repo and look XEGDFX.c.

You're looking for 'default.xex' in the root of the disc. For all games this is where the game metadata and executable code lies. A small number of games I've looked at have additional XEX files, but they are often little helper libraries or just resources with no actual code.

Once you've got the XEX file ready it's time to do some simple dumping of the contents.

Peeking into a XEX

The first tool you'll want to use is xorloser's XexTool. Grab it and throw both it and the included x360_imports.idc file into a directory with your default.xex.

```
02/25/2011  10:10 AM         3,244,032 default.xex 12/04/2010  12:25
AM           173,177 x360_imports.idc 12/07/2010  11:29 PM
185,856 xextool.exe
```

XexTool has a bunch of fun options. Start off by dumping the information contained in default.xex with the -l flag:

```
D:\XexTutorial>xextool.exe -l default.xex XexTool v6.1  -  xorloser
2006-2010 Reading and parsing input xex file...  Xex Info   Retail
Compressed   Encrypted   Title Module   XGD2 Only   Uses Game Voice
Channel   Pal50 Incompatible   Xbox360 Logo Data Present   Basefile
Info   Original PE Name:   [some awesome game].pe   Load Address:
82000000   Entry Point:        826B8B48   Image Size:
B90000   Page Size:            10000   Checksum:           00000000
Filetime:           4539666C - Fri Oct 20 17:14:36 2006   Stack Size:
40000 .... a lot more ....
```

You'll find some interesting bits, such as the encryption keys (required for decrypting the contained executable), basic import information, contained resources, and all of the executable sections. Note that the XEX container has the information about the sections, not the executable. We will see later that most of the PE header in the executable is bogus (likely valid pre-packaging, but certainly not after).

Extracting the PE (.exe)

Next up we will want to crack out the PE executable contained within the XEX by using the '-b' flag. Since we will need it later anyway, also add on the '-i' flag to output the IDC file used by IDA.

```
D:\XexTutorial>xextool -b default.exe -i default.idc default.xex
XexTool v6.1  -  xorloser 2006-2010 Reading and parsing input xex
file... Successfully dumped basefile idc to default.idc Successfully
dumped basefile to default.exe  Load basefile into IDA with the
```

```
following details DO NOT load as a PE or EXE file as the format is
not valid File Type:       Binary file Processor Type:  PowerPC: ppc
Load Address:    0x82000000 Entry Point:     0x826B8B48
```

Take a second to look at the output and note the size difference in the input .xex and output .exe:

```
02/25/2011  10:10 AM         3,244,032 default.xex 08/12/2011  11:04
PM        12,124,160 default.exe
```

In this case the XEX file was both encrypted and compressed. When XexTool spits out the .exe it not only decompresses it, but also pads in some of the data that was stripped when it was originally shoved into the .xex. Thinking about rotational speeds of DVDs and the data transfer rate, a 4x compression ratio is pretty impressive (and it makes me wonder why all PE's aren't packed like this…).

PE Info

You can try to peek at the executable but the text section is PowerPC and most Microsoft tools shipped to the public don't support even looking at the headers in a PPC PE. Luckily there are some 3rd party tools that do a pretty good job of dumping most of the info. Using Matt Pietrek's pedump you can get the headers:

```
D:\XexTutorial>pedump /B /I /L /P /R /S default.exe > default.txt
```

Check out the results and see the PE headers. Note that most of them are incorrect and (I imagine) completely ignored by the real 360 loader. For example, the data directories and section table have incorrect addresses, although their sizes are fairly accurate. During XEX packing a lot of reorganizing and alignment is performed, and some sections are removed completely.

The two most interesting bits in the resulting file from a reversing perspective are the imports table and the runtime function table. The imports table data referenced here is pretty bogus, and instead the addresses from the XEX should be used. The Runtime Function Table, however, has valid addresses and provides a useful resource: addresses and lengths of most methods in the executable. I'll describe both of these structures in more detail later.

Loading a XEX

[I'll patch this up once I open the github repo, but for future reference XEXEX2LoadFromFile, XEXEX2ReadImage, and XEPEModuleLoadFromMemory are useful] Now that's possible to get a XEX and the executable it contains, let's talk about how a XEX is loaded by the 360 kernel.

Sections

Take a peek at the '-l' output from XexTool and down near the bottom you'll see 'Sections'. What follows is a list of all blocks in the XEX, usually 64KB chunks (0×10000), and their type:

```
Sections     0) 82000000 - 82010000 : Header/Resource     1) 82010000
- 82020000 : Header/Resource -- snip --    12) 820C0000 - 820D0000 :
Header/Resource    13) 820D0000 - 820E0000 : Header/Resource    14)
820E0000 - 820F0000 : Code    15) 820F0000 - 82100000 : Code -- snip
--   108) 826C0000 - 826D0000 : Code   109) 826D0000 - 826E0000 :
Code   110) 826E0000 - 826F0000 : Data   111) 826F0000 - 82700000 :
Data .... and many more ....
```

Usually they seem to be laid out as read-only data, code, and read-write data, and always grouped together. All of the fancy sections in the PE are distilled down to these three types, likely due to the fact that the 360 has these three memory protection modes. Everything is in 64KB chunks because that is the allocation granularity for the memory pages. Those two things together explain why the headers in the PE don't match up to reality – the tool that takes .exe -> .xex and does all of this stuff never fixes up the data after it butchers everything. When it comes to actually loading XEX's, all of these transformations actually make things easier. Ignoring all of the decryption/decompression/checksum craziness, loading a XEX is usually as simple as a mapped file read/memcpy. Much, much faster than a normal PE file, and a very smart move on Microsoft's part.

Imports

Both the XEX and PE declare that they have imports, but the XEX ones are correct. Stored in the XEX is a multi-level table of import library (such as 'xboxkrnl.exe') where each library is versioned and then references a set of import entries. You can see the import libraries in the XexTool output:

```
Import Libraries     0) xboxkrnl.exe   v2.0.3529.0  (min v2.0.2858.0)
1) xam.xex        v2.0.3529.0  (min v2.0.2858.0)
```

It's way out of scope here to talk about **how**the libraries are embedded, but you can check out XEXEX2.c in the XEX_HEADER_IMPORT_LIBRARIES bit and later on in the XEXEX2GetImportInfos method for more information. In essence, there is a list of ordinals exported by the import libraries and the address in the loaded executable at where the resolved import should be placed. For example, there may be an import for ordinal 0x26A from 'xboxkrnl.exe', which happens to correspond to 'VdRetrainEDRAMWorker'. The loader will place the real address of 'VdRetrainEDRAMWorker' in the location specified in the import table when the executable is loaded.

The best way to see the imports of an executable (without writing code) is to look at the default.idc file dumped previously by XexTool with the '-i' option. For each import library there will be a function containing several SetupImport* calls that give the location of the import table entry in the executable, the location to place the value in, the ordinal, and the library name. Cross reference x360_imports.idc to find the ordinal name and you can start to decode things:

```
SetupImportFunc(0x8200085C, 0x826BF554, 0x074, "xboxkrnl.exe"); -->
KeInitializeSemaphore SetupImportFunc(0x82000860, 0x826BF564, 0x110,
"xboxkrnl.exe"); --> ObReferenceObjectByHandle
SetupImportFunc(0x82000864, 0x826BF574, 0x1F7, "xboxkrnl.exe"); -->
XAudioGetVoiceCategoryVolumeChangeMask
```

An emulator would perform this process (a bit more efficiently than you or I) and resolve all imports to the corresponding functions in the emulated kernel.

Static Libraries

An interesting bit of information included in the XEX is the list of static libraries compiled into the executable and their version information:

```
Static Libraries     0) D3DX9          v2.0.3529.0     1) XGRAPHC
v2.0.3529.0     2) XONLINE       v2.0.3529.0 .... plus a few more
....
```

In the future it may be possible to build a library of optimized routines commonly found in these libraries by way of fingerprint and version information. For example, being able to identify specific versions of memcpy or other expensive routines could allow for big speed-ups.

IDA Pro

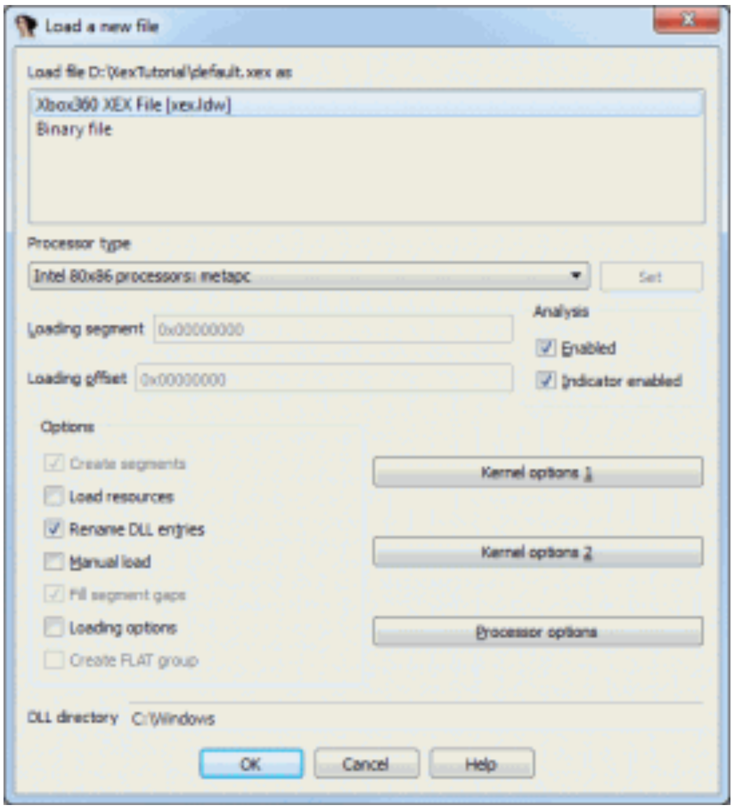
That's about it for what can be done by hand – now let's take a peek at the code!

Getting Setup

I'm using IDA Pro 6 with xorloser's PPCAltivec plugin (required) and xorloser's Xbox360 Xex Loader plugin (optional, but makes things much easier). If you don't want to use the Xex Loader plugin you can load the .exe and then run the .idc file to get pretty much the same thing, but I've had things go much smoother when using the plugin.

Go and grab a copy of IDA Pro 6, if you don't have it. No really, go get it. It's only a foreign money wire of a thousand dollars or two. Worth it. It takes a few days, so I'll wait…

Install the plugins and fire it up. You should be good to go! Close the wizard window and then drag/drop the default.xex file into the app. It'll automatically detect it as a XEX, don't touch anything, and let it go. Although you'll start seeing things right away I've found it's best to let IDA crunch on things before moving around. Wait until 'AU: idle' appears in the bottom left status tray.



Load file D:\XexTutorial\default.xex as

Xbox360 XEX File [xex.ldw]
Binary file

Processor type: Intel 80x86 processors: metapc

Loading segment: 0x00000000

Loading offset: 0x00000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [✓] Create segments
- [ ] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [✓] Fill segment gaps
- [ ] Loading options
- [ ] Create FLAT group

DLL directory: C:\Windows

XEX Import Dialog

IDA Pro Initial View

Navigating

XEX files traditionally have the following major elements in this order (likely because they all come from the same linker):

1. Import tables – a bunch of longs that will be filled with the addresses of imports on load
2. Generic read-only data (string tables, constants/etc)
3. Code (.text)
4. Import function thunks (at the end of .text)
5. Random security code

IDA and xorloser’s XEX importer are nice enough to organize most things for you. Most functions are found correctly (although a few aren’t or are split up wrong), you’ll find the __savegpr/__restgpr blocks named for you, and all import thunks will be resolved.

The Anatomy of an Xbox Executable

I’m not going to do a full walkthrough of IDA (you either know already or can find it pretty easily), but I will show an example that reveals a bit about what the executables look like and how they function. It’s fun to see how much you can glean about the tools and processes used to build the executable from the output!

Reversing Sleep

Starting simple and in a well-known space is generally a good idea. Scanning through the import thunks in my executable I noticed ‘KeDelayExecutionThread’. That’s an interesting function because it is fairly simple – the perfect place to get a grounding. Almost every game should call this, so look for it in yours (your addresses will differ).

```
.text:826BF9B4 KeDelayExecutionThread:                 # CODE XREF:
sub_826B9AF8+5C□p .text:826BF9B4                li      %r3, 0x5A
.text:826BF9B8                  li      %r4, 0x5A .text:826BF9BC
mtspr   CTR, %r11 .text:826BF9C0                bctr
```

All of the import thunks have this form – I'm assuming the loader overwrites their contents with the actual jump calls and none of the code here is used. Time to check out the documentation. MSDN shows KeDelayExecutionThread as taking 3 arguments and returning one:

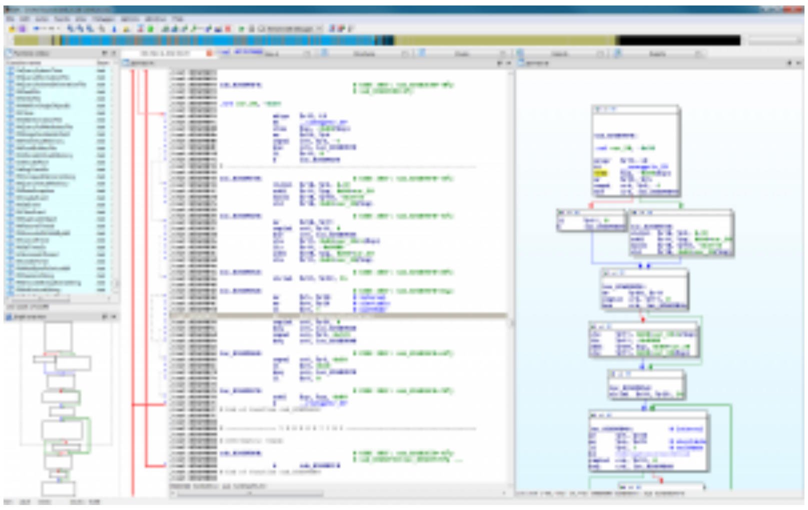
```
NTSTATUS KeDelayExecutionThread(   __in  KPROCESSOR_MODE WaitMode,
__in  BOOLEAN Alertable,   __in  PLARGE_INTEGER Interval );
```

For a function as core as this it's highly likely that the signature has not changed between the documented NT kernel and the 360 kernel. This is not always the case (especially with some of the more complex calls), but is a good place to start. Right click and select 'Set function type' (or hit Y) and enter the signature:

```
int __cdecl KeDelayExecutionThread(int waitMode, int alertable,
__int64* interval)
```

Because this is a kernel (Ke*) function it's unlikely that it is being called by user code directly. Instead, one of the many static libraries linked in is wrapping it up (my guess is 'xboxkrnl'). IDA shows that there is one referencing routine – almost definitely the wrapper. Double click it to jump to the call site.

Now we are looking at a much larger function wrapping the call to KeDelayExecutionThread:



IDA Pro View of KeDelayExecutionThread wrapper

Tracing back the parameters to KeDelayExecutionThread, waitMode is always constant but both interval and alertable come from the parameters to the function. Note that argument 0 ends up as 'interval' in KeDelayExecutionThread, has special handling for -1, and has a massively large multiplier on it (bringing the interval from ms to 100-ns). Down near the end you can see %r3 being set, which indicates that the function returns a value. From this, we can guess at the signature of the function and throw it into the 'Set function type' dialog:

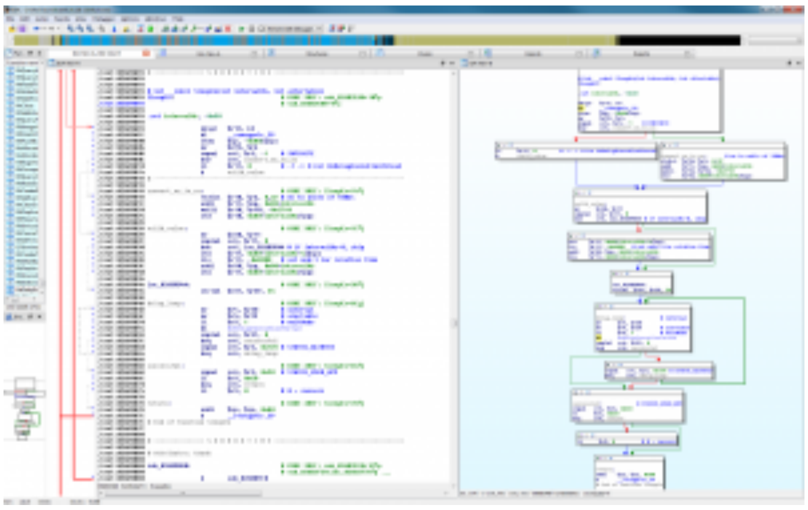
```
int __cdecl DelayWrapper(int intervalMs, int alertable)
```

We can do one better than just the signature, though. This has to be some Microsoft user-mode API. Thinking about what KeDelayExecutionThread does and the arguments of this wrapper, the first thought is 'Sleep'. Win32 Sleep only takes one argument, but SleepEx takes two and matches our signature exactly!

Check out the documentation to confirm: MSDN SleepEx. Pretty close to what we got, right?

```
DWORD WINAPI SleepEx(   __in  DWORD dwMilliseconds,   __in  BOOL
bAlertable );
```

Rename the function (hit N) and feel satisfied that you now have one of hundreds of thunks completed!


Reversed SleepEx

Now do all of the rest, and depth-first reverse large portions of the game code. You now know the hell of an emulator author. Hackers have it easy – they generally target very specific parts of an application to reverse… when writing an emulator, however, breadth often matters just as much as depth x_x

Building an Xbox 360 Emulator, part 6: Code Translation Techniques

One of the most important pieces of an emulator is the simulation of the guest processor – it drives every other subsystem and is often the performance bottleneck. I'm going to spend a few posts looking into how to build a (hopefully) fast and portable PowerPC simulator.

Overview

At the highest level the emulator needs to be able to take the original PowerPC instructions and run them on the host processor. These instructions come in the form of a giant assembled binary instruction stream loaded into memory – there is no concept of 'functions' or 'types' and no flow control information. Somehow, they must be quickly translated into a form the host processor can understand, either one at a time or in batches. There are many techniques for doing this, some easy (and slow) and many hard (and fastish). I've learned through experience that that the first choice one makes is almost always the wrong one and a lot of iteration is required to get the right balance.

The techniques for doing this translation parallel the techniques used for systems programming. There's usually a component that looks like an optimizing compiler, a linker, some sort of module format (even if just in-memory), and an ABI (application binary interface – a.k.a. calling convention).

The tricky part is not figuring out how these components fit together but instead deciding how much effort to put into each one to get the right performance/compatibility trade off.

I recommend checking out Virtual Machines by James Smith and Ravi Nair for a much more in-depth overview of the field of machine emulation, but I'll briefly discuss the major types used in console emulators below.

Interpreters

Implementation: Easy
Speed: Slow
Examples: BASIC, most older emulators

Interpreters are the simplest of the bunch. They are basically exactly what you would build if you coded up the algorithm describing how a processor works. Typically, they look like this:

- While running:
  - Get the next instruction from the stream
  - Decode the instruction
  - Execute the instruction (maybe updating the address of the next instruction to execute)

There are bits that make this a bit more complex than this, but generally they are implementation details about the guest CPU (how does it handle jumping around, etc). Ignoring the code that actually executes each instruction, a typical interpreter can be just a few dozen lines of easy-to-read, easy-to-maintain code. That's why when performance doesn't matter you see people go with interpreters – no one wants to waste time building insanely complex systems when a simple one will work (or at least, they shouldn't!).

Even when an interpreter isn't fast enough to run the guest code at sufficient speeds there are still many advantages to use one. Mainly, due to the simplicity of the implementation, it's often possible to ensure correctness without much trouble. A lot of emulation projects start of with interpreters just to ensure all the pieces are working and then later go back and add a more complex CPU core when things are all verified correct. Another great advantage is that it's much easier to build debuggers and other analysis tools, as things like stepping and register inspection can be one or two lines in the main execution loop.

Unfortunately for this project, performance does matter and an interpreter will not suffice. I would like to build one just to make it easier to verify the correctness of the instruction set, however even with as (relatively) simple it is there is still a large time investment that may never pay off.

Pros

- Easy to implement
- Easy to build debuggers/step-through
- Can get something running fairly fast
- Snapshotting/save states trivial to implement

Cons

- Several orders of magnitude slower than the host machine (think 100-1000x)

JITs

Implementation:	Sort-of easy
Speed:	Sort-of fast
Examples:	Modern Javascript, .NET/JVM, most emulators

This technique is the most common one used in emulators today. It's much more complex than an interpreter but still relatively easy to get implemented. At a high level the flow is the same as in an interpreter, but instead of operating on single instructions the algorithm handles basic blocks, or a short sequence of instructions excluding flow control (jumps and branches).

- While running:
  - Get the next address to process
  - Lookup the address in the code cache
  - If cached basic block present:
    - Execute cached block
  - Otherwise:
    - Translate the basic block
    - Store in code cache
    - Execute block

The emulator spins in this loop and in the steady state is doing very little other than lookups in the cache and calls of the cached code. Each basic block runs until the end and then returns back to this control method to let it decide where to go next.

There is a bit of optimization that can be done here, but because the unit of work is a basic block a lot don't make sense or can't be used to full advantage. The best optimizations often require much more context than a couple instructions in isolation can give and the most expensive code is usually the stuff in-between the basic blocks, not inside of it (jumps/branches/etc).

Compared to the next technique this simple JIT does have a few advantages. Namely, if the guest system is allowed to dynamically modify code (think of a guest running a guest) this technique will get the best performance with as little penalty for regeneration as possible. On the Xbox 360, however, it is impossible to dynamically generate code so that's a whole big area that can be ignored.

In terms of the price/performance ratio, this is the way to go. The analysis is about as simple as the interpreter, the translation is straightforward, and the code is still pretty easy to debug. There's a performance penalty (that can often be eliminated with tricks) for the cache lookups and dynamic translation, but it's still much faster than interpreting.

That said, it's still not fast enough. To emulate 3 3GHz processors with an advanced instruction set on x86-64, every single cycle needs to count. It's also boring – I've built a few before, and building another would just be going through the motions.

Pros

- Pretty fast (10-100x slower than native)
- Pretty simple
- Allows for recompilation if code is changed on-the-fly

Cons

- Few optimizations possible
- Debugging is harder

Recompilers / 'Advanced' JITs

Implementation: Hard
Speed: Fastest possible
Examples: V8, .NET ngen/AOT

Recompilers (often referred to as 'binary translation' in literature) are the holy grail of emulation. The advantages one gets from being able to do full program analysis, ahead-of-time compilation, and trivially debuggable code cannot be beat… except by how hard they usually are to write.

Where as an interpreter works on individual instructions and a simple JIT works on basic blocks, a recompiler works on methods or even entire programs. This allows for complex optimizations to be used that, knowing the target host architecture, can yield even faster code than the original instruction stream.

I split this section between 'advanced' JITs and recompilers, but they are really two different things implemented in largely the same way. An advanced JIT (as I call it) is a simple JIT extended to work on methods and using some simple intermediate representation (IR) that allows for optimizations, but at the end of the day is still dynamically compiling code at runtime on demand. A full recompiler is usually a system that does a lot of the same analysis and uses a similar intermediate representation, but does it all at once and before attempting to execute the code. In some ways a JIT is easier to think about (still operating on small units of code, still doing it inside the program flow), but in many others the recompiler simplifies things. For example, being able to verify an entire program is correct and optimize as much as possible before the guest executes allows for much better tooling to be created. Depending on what library/tools are being used you can also get reflection, debugging, and things usually reserved for original source-level programming like profile-guided optimization.

The hard part of this technique is actually analyzing the instruction stream to try to figure out how the code is organized. Tools like IDA Pro will do this to a point, but are not meant to be used to generate executable code. How this process is normally done is largely undocumented – either because it's considered a trade secret or no one cares – but I puzzled out some of the tricks and used them to build my PSP emulator. There I implemented an advanced JIT, generating code on the fly, but that's only because of the tools that I had at the time and not really knowing what I wanted.

Recompilers are very close to full-on decompilers. Decompilers are designed to take machine instructions up through various representations and (hopefully) yield human-readable high level source code. They are constructed like a compiler but in reverse: compilers usually contain a frontend (input source code -> intermediate representation (IR)), optimizers/analyzers, and a backend (IR -> machine code (MC)); decompilers have a frontend (MC -> IR), optimizers/analyzers, and a backend (IR -> source, like C). The decompiler frontend is fairly trivial, the analysis much more complex, and the backend potentially unsolvable. What makes recompilers interesting is that at no point do they aim to high human-readable output – instead, a recompiler has a frontend like a decompiler (MC -> IR), analyzers like a decompiler, optimizers like a compiler, and a backend like a compiler (IR -> MC).

Pros

- As close to native as possible (1-10x slower, depending on architectures)
- No translation overhead while running (unless desired)
- Debuggable using real tools

- Still pretty novel (read: fun!)

Cons

- Incredibly hard to write
- Can't handle dynamically modifiable code (well)

Building a Recompiler

There are many steps down the path of building a recompiler. Some people have tried building general purpose binary translation toolkits, but that's a lot of work and requires a lot of good design and abstraction. For this project I just want to get something working and I have learned that after I'm done I will never want to use the code again – by the time I attempt a project like this again, I will have learned enough to consider it all garbage I'll be focusing on a Power PC frontend and reusing the PPC instruction set (+ tags) as my intermediate representation (IR) – this simplifies a lot of things and allows me to bake assumptions about the source platform into the entire stack without a lot of nasty hacks. One design concession I will be making is letting the backend (IR -> MC) be pluggable. From experience, the frontend rarely changes between different implementations while the backend varies highly – source PPC instructions are source PPC instructions regardless of whether you're implementing an interpreter, a JIT, or a recompiler. For now I'm planning on using LLVM for the backend (as it also gives me some nice optimizers), but may re-evaluate this later on and would like not to have to reimplement the frontend.

Frontend (MC -> IR)

Assuming that the source module has already been loaded (see previous posts on loading XEX files), the frontend stage includes a few major components:

- Disassembler
- Basic block slicing
- Control Flow Graph (CFG) construction
- Method recognition

From this stage a skeleton of the program can be generated that should fairly closely model the original structure of the code. This includes a (mostly correct) list of methods, tagged references to global variables, and enough information to identify the control flow in any given method.

When working on my PSP emulator I didn't factor out the frontend correctly and ended up having to reimplement it several times. For this project I'll be constructing this piece myself and ensuring that it is reusable, which will hopefully save a lot of time when experimenting with different backends.

Disassembler

A simple run-of-the-mill disassembler that is able to take a byte stream and produce an instruction stream. There are a few table generation toolkits out there that can make this process much faster at runtime but initially I'll be sticking with the tried and true chained table lookup. A useful feature to add to early disassemblers is pretty printing of the instructions, which enables much better output from later parts of system and makes things significantly easier to debug.

Basic Block Slicing / Control Flow Graph Construction / Method Recognition

Basic blocks in this context refer to a sequence of instructions that starts at an instruction that is jumped to by another basic block and ends at the first flow control instruction. This gets the instruction stream into a form that enables the next step.

Once basic blocks are identified they can be linked together to form a Control Flow Graph (CFG). Using the CFG it is possible to identify unique entry and exit points of portions of the graph and call those 'methods'. Sometimes they match 1:1 with the original input code, but other times due to optimizing compilers (inlining, etc) may not – it doesn't matter to a recompiler (but does to a decompiler). Usually the process of CFG generation is combined with the basic block slicing step and executed in multiple passes until all edges have been identified.

There are some tricky details here that make this stage not 100% reliable, namely function pointers. Simple C function pointer passing (callbacks/etc) as well as things like C++ vtables can prevent a proper whole-program CFG from being constructed. In these cases, where the target code may appear to have never been called (as there were no jumps/branches into it) it is important to have a method recognition heuristic to identify them and bring them into the recompiled output. The first stage of this is scanning for holes in the code address space: if after all processing has been done there are still regions that are not assigned to methods they are now suspicious – it's possible that the contents of the holes are actually data or padding and it's important to have a set of rules to follow to identify that. Popular heuristics are looking for function prologues and ensuring a region has all valid instructions. Once found the recompiler isn't done, though, as even if the method gets to the output there is still no way to connect it up to the original callers accessing it by pointer. One way to solve this is to make all call-by-pointer sites instead look up the function in a table of all functions in the module. It can be significantly slower than the native call, but caches can help.

Analysis

The results of the frontend are already fairly usable, however to generate better output the recompiler needs to do a bit of analysis on the source instructions. What comes out of the frontend is a literal interpretation of the source and as such is missing a lot of the extra information that the backend can potentially use to optimize the output. There are hundreds of different things that can be done at this stage as required, but for recompilers there are a few important ones:

- Data Flow Analysis (DFA)
- Control Flow Analysis (CFA)

Data Flow Analysis

Since PPC is a RISC architecture there is often a large number of instructions that work on intermediate registers just for the sake of accomplishing one logical instruction. For example, look at this bit of disassembly (what would come out of the frontend):

```
.text:8210E77C                  lwz     %r11, 0x54(%r31)
.text:8210E780                  lwz     %r9, 0x90+var_40(%sp)
.text:8210E784                  lwz     %r10, 0x50(%r31)
.text:8210E788                  mullw   %r11, %r11, %r9 .text:8210E78C
add     %r11, %r11, %r10 .text:8210E790                  lwz     %r10,
0x90+var_3C(%sp) .text:8210E794                  add     %r11, %r11,
%r10 .text:8210E798                  stw     %r11, 0(%r29)
```

A simple decompilation of this is:

```
var x = (r31)[0x54]; var y = (sp)[0x90+var_40]; var z = (r31)[0x50];
x *= y; x += z; z = (sp)[0x90+var_3C]; x += z; (r29)[0] = x;
```

If you used this as output in your recompiler, however, you would end up with many more instructions being executed than required. A simple data flow analysis would enable result propagation (x = x * y + z, etc) and SSA (knowing that the second lwz into %r10 is a different use of z). Performing this step would allow an output that is much more simple and easier for down-stream optimization passes to deal with:

```
(r29)[0] = ((r31)[0x54] * (sp)[0x90+var_40]) + (r31)[0x50] +
(sp)[0x90+var_3C];
```

Control Flow Analysis

With a constructed Control Flow Graph it's possible to start trying to identify what the original flow control constructs were. It's not required to do this in a recompiler, as the output of the compiler will still be correct if passed through directly to the backend, however the more information that can be provided to the backend the better. Compilers will often change for- and while-loops into post-tested loops (do-while) or specific processor forms, such as the case of PPC which has a special branch counter instruction. By inspecting the structure of the graph it's possible to figure out what are loops vs. conditional branches, and just where the loops are and what basic blocks are included inside the body of the loop. By encoding this information in the IR the backend can do better local variable allocation by knowing what variables are accessible from which pieces of code, better code layout by knowing which side of the branch/loop is likely to be executed the most, etc.

CFA is also required for correct DFA – you could imagine scenarios where registers or locals are set before a jump and changed in the flow. You would not be able to perform the data propagation or collapsing without knowing for certain what the potential values of the register/local could be at any point in time.

Backend (IR -> MC)

I'll be doing an entire post (or more) about what I'm planning on doing here for this project, but for completeness here's an overview:

Backends can vary in both type and complexity. The simplest backend is an interpreter, executing the instruction stream produced by the frontend one instruction at a time (and ignoring most of the metadata attached). JITs can use the information to produce either simple basic block-based code chunks or entire method chunks. Or, as I'm aiming for, a compiler/linker can be used to do a whole bunch more.

Right now I'm targeting LLVM IR (plus some custom code) for the backend. This enables me to run the entire LLVM optimization pass over the IR to produce some of the best possible output, use the LLVM interpreter or JIT to get runtime translation, or the offline LLVM tools to generate executables that can be run on the host machine. The complex part of this process is getting the IR used in the rest of this process into LLVM IR, which is at a much higher level than machine instructions. Luckily others have already used LLVM for purposes like this, so at least it's possible!