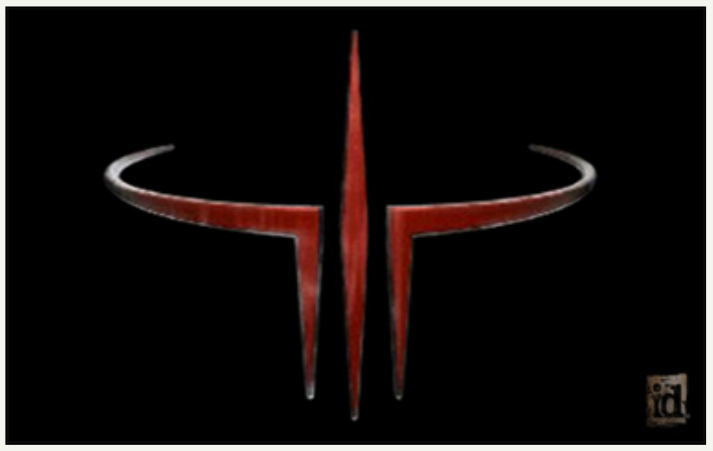
JUNE 30, 2012

QUAKE 3 SOURCE CODE REVIEW: VIRTUAL MACHINE (PART 4 OF 5) >>



If previous engines delegated only the gameplay to the Virtual Machine, idtech3 heavily rely on them for essential tasks. Among other things:

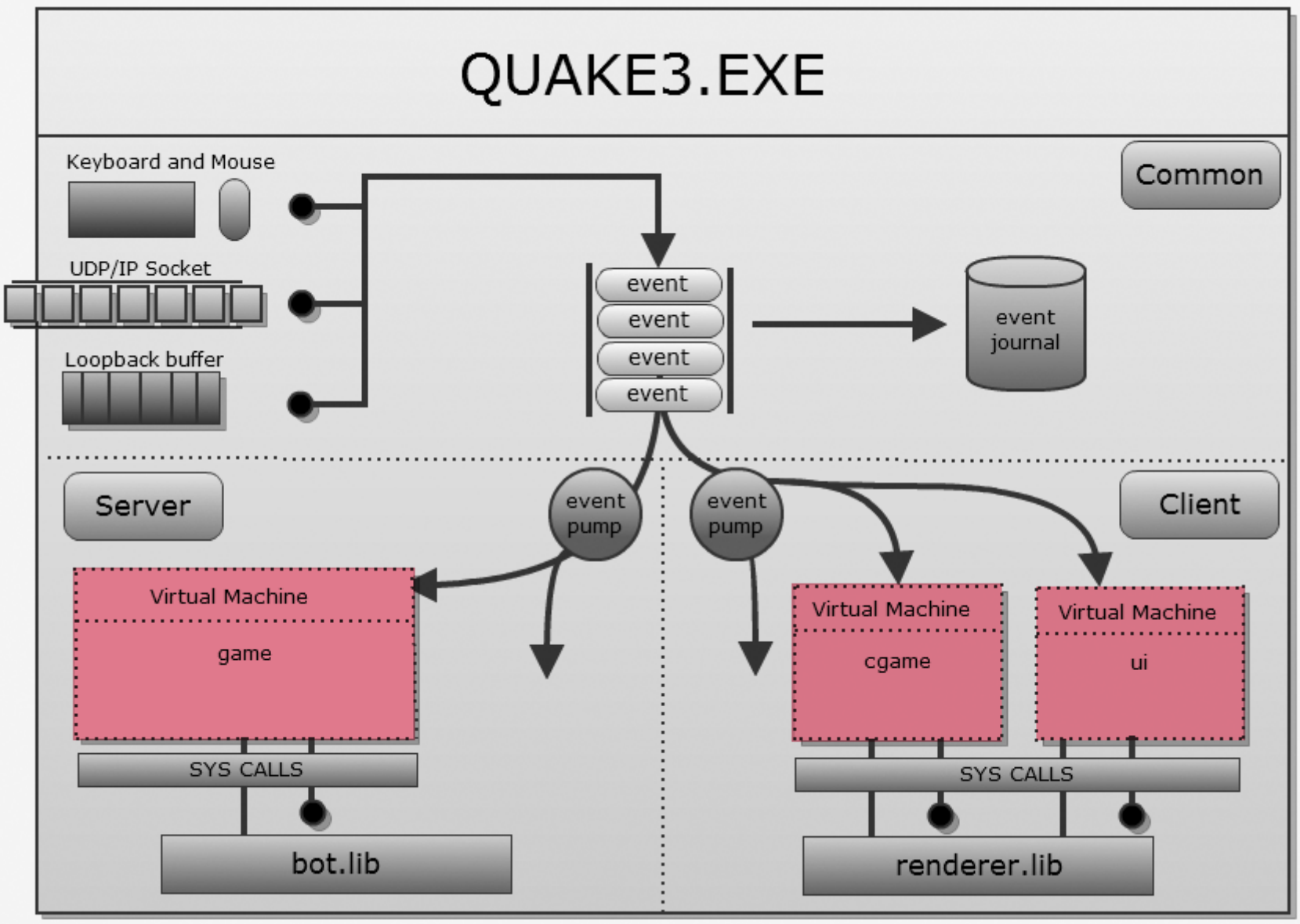
- Rendition is triggered from the Client VM.
- The lag compensation mechanism is entirely in the Client VM.

Moreover their design is much more elaborated: They combine the security/portability of Quake1 Virtual Machine with the high performances of Quake2's native DLLs. This is achieved by compiling the bytecode to x86 instruction on the fly.

Trivia : The virtual machine was initially supposed to be a plain bytecode interpreter but performances were disappointing so the development team wrote a runtime x86 compiler. According to the .plan from Aug 16, 1999 this was done in one day.

Architecture

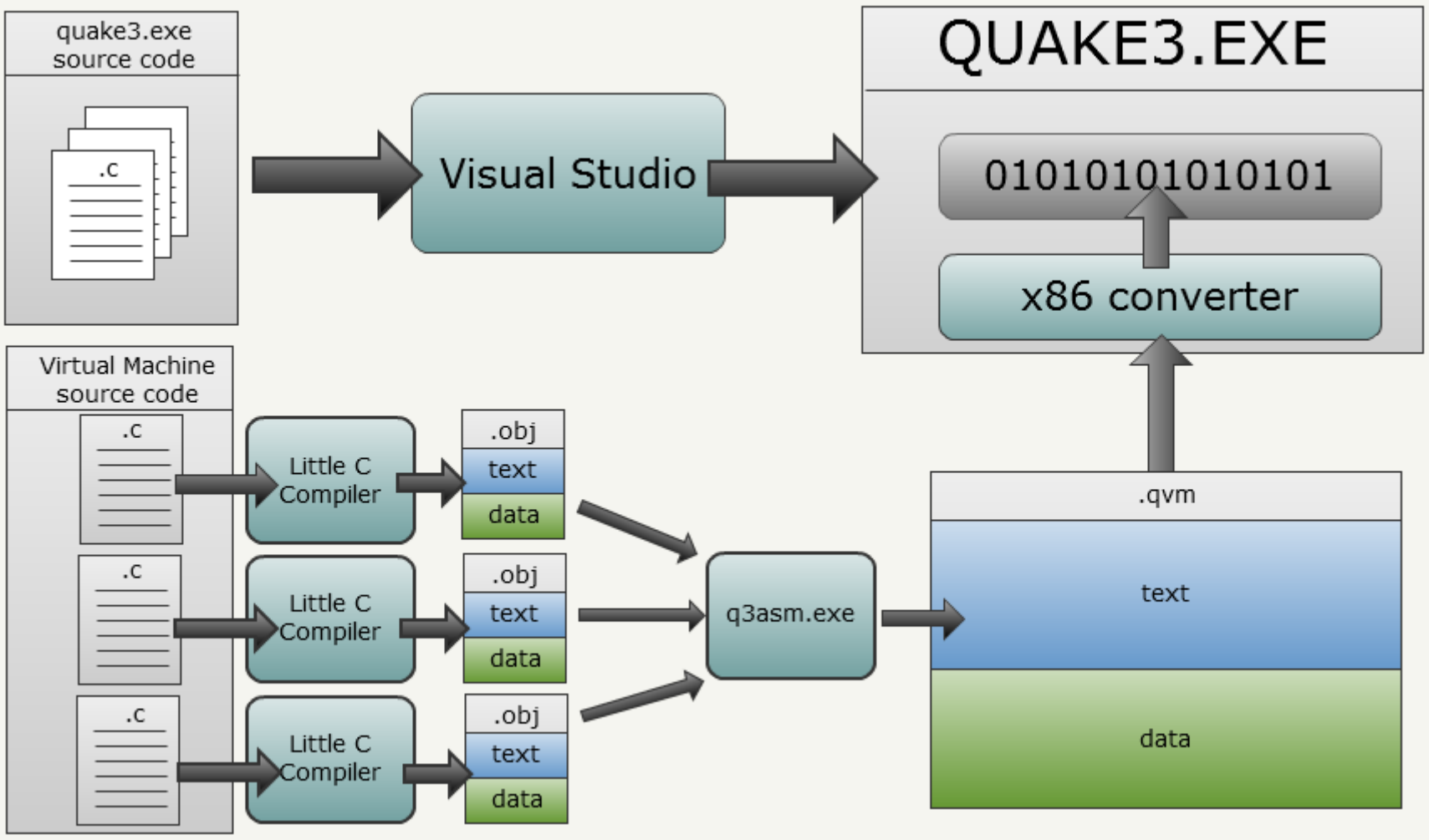
In Quake III a virtual machine is called a QVM: Three of them are loaded at any time:



- Client Side: Two virtual machine are loaded. Message are sent to one or the other depending on the gamestate:
  - `cgame` : Receive messages during Battle phases. Performs entity culling, predictions and trigger `renderer.lib`.
  - `q3_ui` : Receive messages during Menu phases. Uses system calls to draw the menus.
- Server Side:
  - `game` : Always receive message: Perform gamelogic and hit `bot.lib` to perform A.I .

QVM Internals

Before describing how the QVMs are used, let's check how the bytecode is generated. As usual I prefer drawing with a little bit of complementary text:



`quake3.exe` and its bytecode interpreter are generated via Visual Studio but the VM bytecode takes a very different path:

1. Each .c file (translation unit) is compiled individually via LCC.
2. LCC is used with a special parameter so it does not output a PE (Windows Portable Executable) but rather its Intermediate Representation which is text based stack machine assembly. Each file produced features a `text`, `data` and `bss` section with symbols exports and imports.
3. A special tool from id Software `q3asm.exe` takes all text assembly files and assembles them together in one .qvm file. It also transform everything from text to binary (for speed in case the native converted cannot kick in). `q3asm.exe` also recognize which methods are system calls and give those a negative symbol number.
4. Upon loading the binary bytecode, `quake3.exe` converts it to x86 instructions (not mandatory).

LCC Internals

Here is a concrete example starting with a function that we want to run in the Virtual Machine:

```
extern int variableA;
int variableB;
int variableC=0;
int fooFunction(char* string){
    return variableA + strlen(string);
}
```

Saved in `module.c` translation unit, `lcc.exe` is called with a special flag in order to avoid generating a Windows PE object but rather output the Intermediate Representation. This is the LCC .obj output matching the C function above:

```
data
export variableC
align 4
LABELV variableC
byte 4 0
export fooFunction
code
proc fooFunction 4 4
ADDRFP4 0
INDIRP4
ARGP4
ADDRLP4 0
ADDRGP4 strlen
CALLI4
ASGNI4
ARGP4 variableA
INDIRI4
ADDRLP4 0
INDIRI4
ADDI4
RETI4
LABELV $1
endproc fooFunction 4 4
import strlen
bss
export variableB
```

```
align 4
LABELV variableB
skip 4
import variableA
```

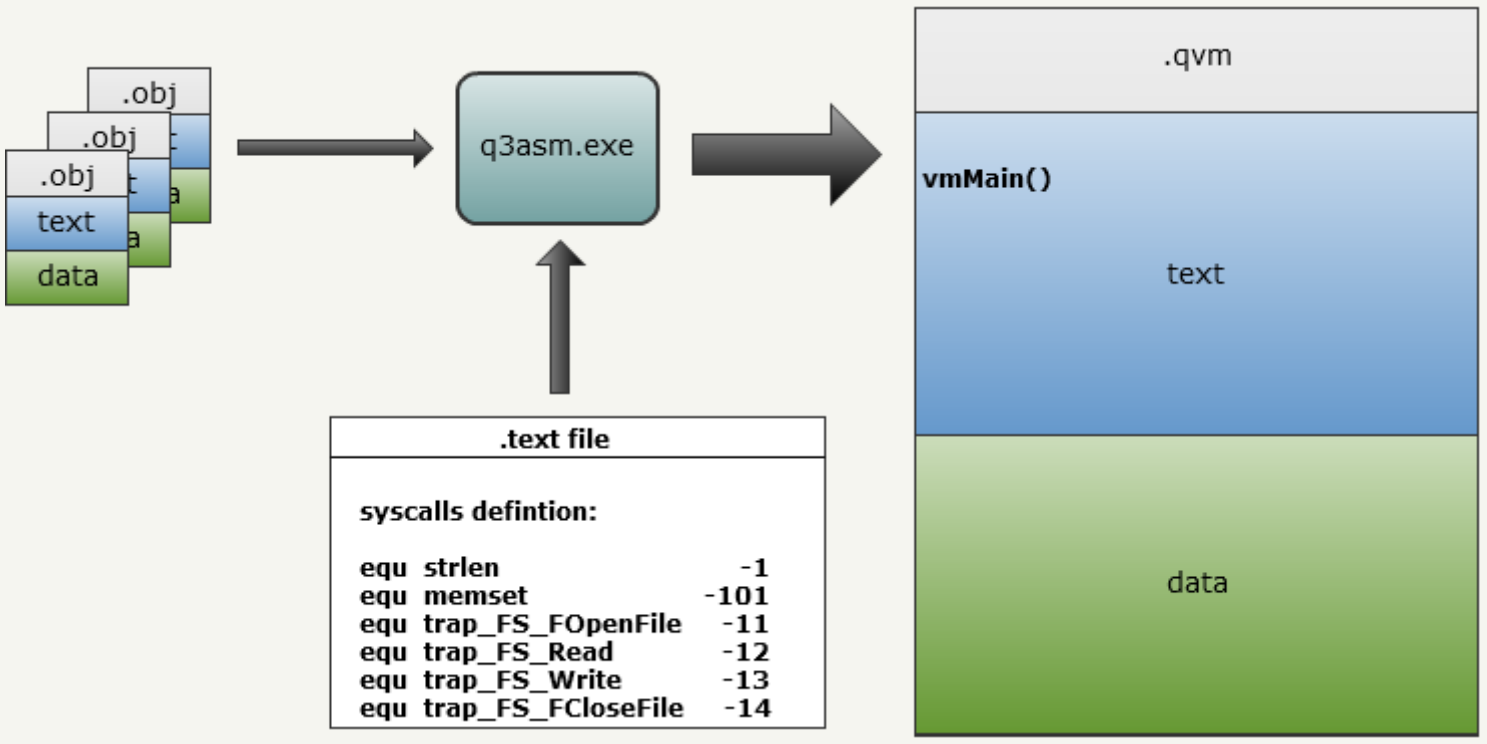
A few observations:

- The bytecode is organized in sections (marked in red): We can clearly see the `bss` (uninitialized variables), `data` (initialized variables) and `code` (usually called `text` but whatever..)
- Functions are defined via `proc`, `endproc` sandwich (marked in blue).
- The Intermediate Representation of LCC is a stack machine: All operations are done on the stack with no assumptions made about CPU registers.
- At the end of the LCC phrase we have a bunch of files importing/exporting variables/functions.
- Each statement starts with the operation type (i.e: `ARGP4`, `ADDRGP4`, `CALLI4`...). Every parameter and result will be passed on the stack.
- Import and Export are here so the assembler can "link" translation units" together. Notice `import strlen`, since neither q3asm.exe nor the VM Interpreter actually likes to the C Standard library, `strlen` is considered a system call and must be provided by the Virtual Machine.

Such a text file is generated for each .c in the VM module.

q3asm.exe Internals

`q3asm.exe` takes the LCC Intermediate representation text files and assembles them together in a .qvm file:

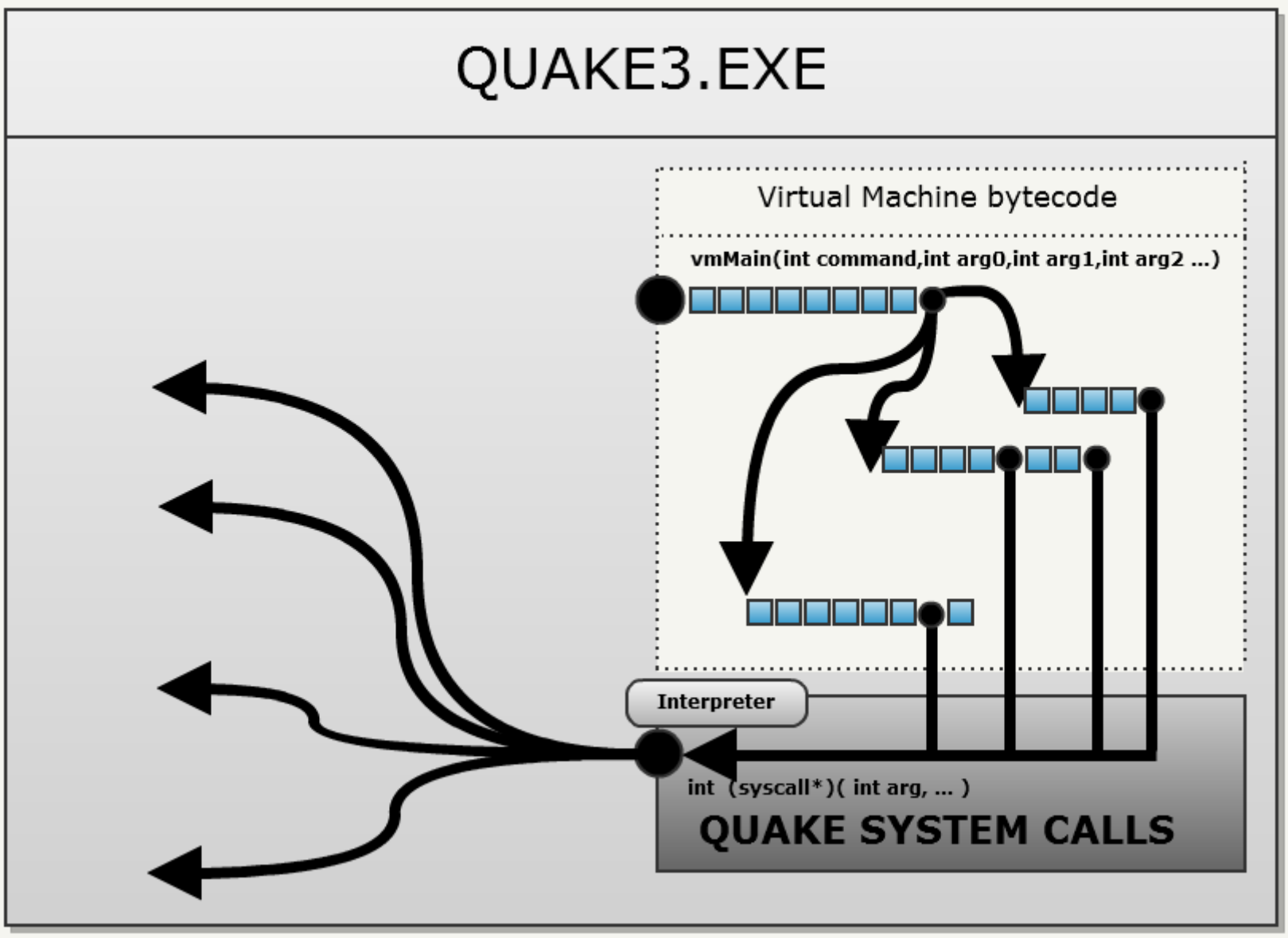


Several things to notice:

- q3asm makes sense of each import/export symbols across text files.
- Some methods are predefined via a system call text file. You can see the syscall for the client VM and for the Server VMs. System calls symbols are attributed a negative integer value so they can be identified by the interpreter.
- q3asm change representation from text to binary in order to gain space and speed but that is pretty much it, no optimizations are performed here.
- The first method to be assembled MUST be `vmMain` since it is the input message dispatcher. Moreover it MUST be located at `0x2D` in the text segment of the bytecode.

QVM: How it works

Again a drawing first illustrating the unique entry point and unique exit point that act as dispatch:



A few details:

Messages (Quake3 -> VM) are send to the Virtual Machine as follow:

- Any part of Quake3 can call `VM_Call( vm_t *vm, int callnum, ... )`.
- `VMCall` takes up to 11 parameters and write each 4 bytes value in the VM bytecode (`vm_t *vm`) from 0x00 up to 0x26.
- `VMCall` writes the message id at 0x2A.
- The interpreter starts interpreting opcodes at 0x2D (where `vmMain` was placed by `q3asm.exe`).
- `vmMain` act as a dispatch and route the message to the appropriate bytecode method.

You can find the list of Message that can be sent to the Client VM and Server VM (at the bottom of each file).

System calls (VM -> Quake3) go out this way:

- The interpreter execute the VM opcodes one after an other (`VM_CallInterpreted`).
- When it encounters a `CALLI4` opcode it checks the int method index.
- If the value is negative then it is a system call.
- The "system call function pointer" (`int (*systemCall)( int *parms )`) is called with the parameters.
- The function pointed to by `systemCall` acts as a dispatch and route the system call to the right part of quake3.exe

You can find the list of system calls that are provided by the Client VM and the Server VM (at the top of each file).

Trivia : Parameters are always very simple types: Either primitives types (char,int,float) or pointer to primitive types (char*,int[]). I suspect this was done to minimize issues due to struct alignment between Visual Studio and LCC.

Trivia : Quake3 VM does not perform dynamic linking so a developer of a QVM mod had no access to any library, not even the C Standard Library (strlen, memset functions are here...but they are actually system calls). Some people still managed to fake it with preallocated buffer: Malloc in QVM !!

Unprecedented freedom

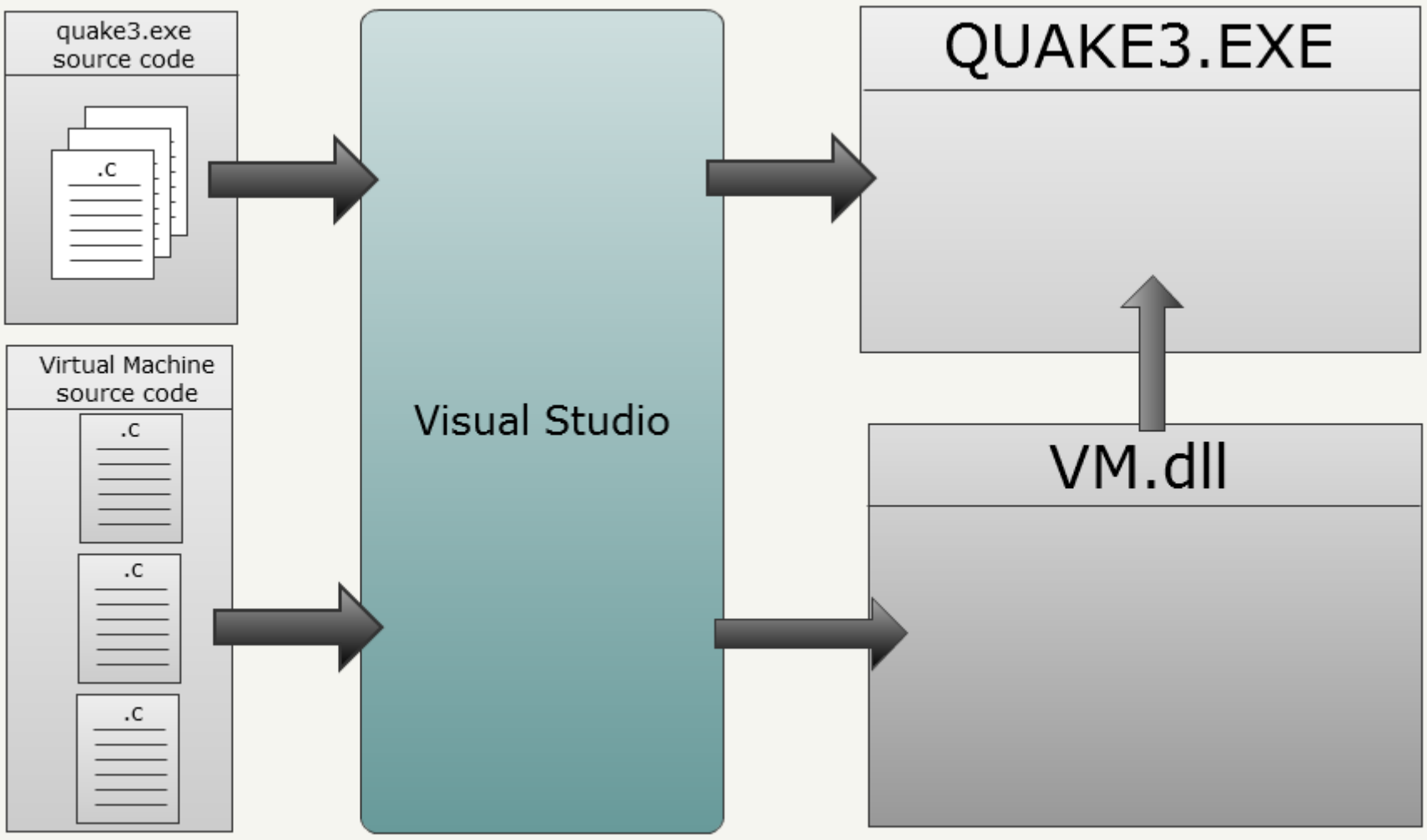
With the kind of task offset to the Virtual Machine the modding community was able to perform much more than modding. The prediction system was rewritten with "backward reconciliation" in Unlagged by Neil "haste" Toronto.

Productivity issue and solution

With such a long toolchain, developing VM code was difficult:

- The toolchain was slow.
- The toolchain was not integrated to Visual Studio.
- Building a QVM involved using commandline tools. It was cumbersome and interrupted the workflow.
- With so many elements in the toolchain it was hard to identify which part was at fault in case of bugs.

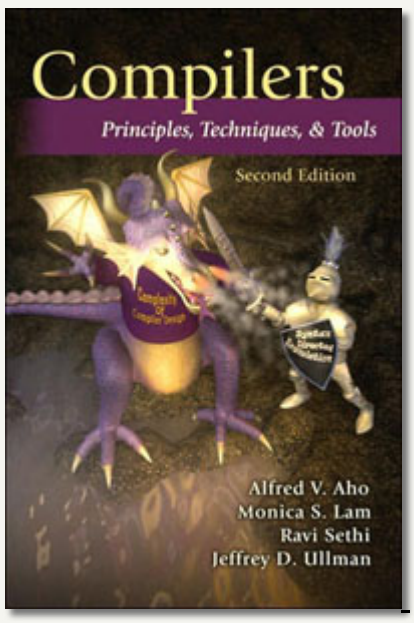
So idTech3 also have the ability to load a native DLL for the VM parts and it solved everything:

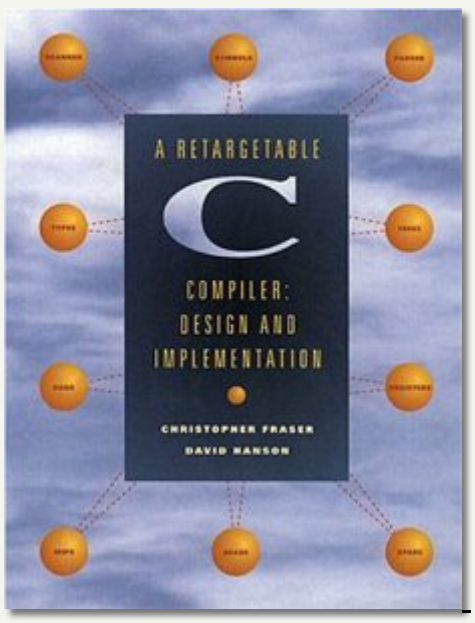


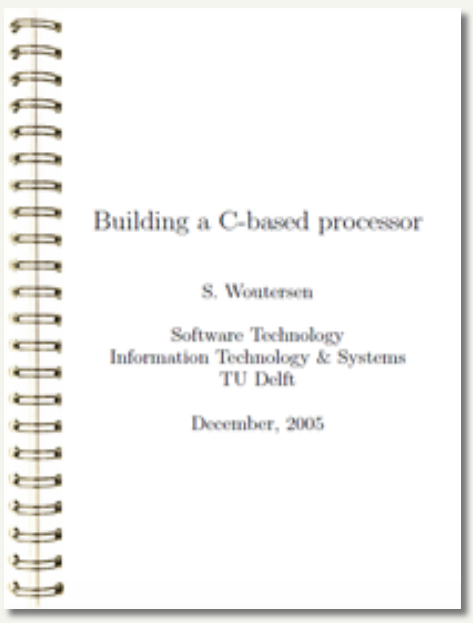
Overall the VM system is very versatile since a Virtual Machine is capable of running:

- Interpreted bytecode
- Bytecode compiled to x86 instructions
- Code compiled as a Windows DLL

Recommended readings







Next part

The A.I Model