How To Write a Computer Emulator

by Marat Fayzullin

I wrote this document after receiving large amounts of email from people who would like to write an emulator of one or another computer but do not know where to start. Any opinions and advices contained in the following text are mine alone and should not be taken for an absolute truth. The document mainly covers so-called "interpreting" emulators, as opposed to "compiling" ones, because I do not have much experience with recompilation techniques. It does have a pointer or two to the places where you can find information on these techniques.

If you think that this document is missing something or want to make a correction, feel free to email me your comments. I **do not** answer to flames, idiocy, and requests for ROM images though. I'm badly missing some important FTP/WWW addresses in the resources list in this document so if you know any worth putting there, tell me about them. Same goes for any frequently asked questions that are not in this document.

This document has been translated to Japanese by Bero. There is also Chinese translation available, courtesy of Shun-Yuan Chou, and another Chinese translation by Jean-Yuan Chen. The French translation is made by Maxime Vernier. An older French translation by Guillaume Tuloup may or may not be available at the moment. Spanish translation of the HOWTO has been made by Santiago Romero. Italian translation has been made by Mauro Villani. Finally, a Brazilian Portugese translation by Leandro is available.

Contents

So, you decided to write a software emulator? Very well, then this document may be of help to you. It covers a few common technical questions people ask about writing emulators. It also provides you with "blueprints" for emulator internals which you can follow in some degree.

General

Implementation

- How do I emulate a CPU?
- How do I handle accesses to emulated memory?
- Cyclic tasks: what are they?

Programming Techniques

- How do I optimize C code?
- What is low/high-endiness?
- How to make program portable?
- Why should I make program modular?

- *More to come here*

What can be emulated?

Basically, anything which has a microprocessor inside. Of course, only devices running more or less flexible program are interesting to emulate. Those include:

- Computers
- Calculators
- Videogame Consoles
- Arcade Videogames
- etc.

It is necessary to note that you can emulate **any** computer system, even if it is very complex (such as Commodore Amiga computer, for example). The perfomance of such emulation may be very low though.

What is "emulation" and how does it differ from "simulation"?

Emulation is an attempt to imitate the internal design of a device. Simulation is an attempt to imitate functions of a device. For example, a program imitating the Pacman arcade hardware and running real Pacman ROM on it is an emulator. A Pacman game written for your computer but using graphics similar to the real arcade is a simulator.

Is it legal to emulate the proprietary hardware?

Although this matter lies in the "gray" area, it appears to be legal to emulate proprietary hardware, as long as the information on it hasn't been obtained by illegal means. You should also be aware that it is illegal to distribute system ROMs (BIOS, etc.) with an emulator if the are copyrighted.

What is "interpreting" emulator and how does it differ from "recompiling" emulator?

There are three basic schemes which can be used for an emulator. They can be combined for the best result.

- **Interpretation**
  An emulator reads emulated code from memory byte-by-byte, decodes it, and performs appropriate commands on the emulated registers, memory, and I/O. The general algorithm for such emulator is following:

  ```
  while(CPUIsRunning)
  {
    Fetch OpCode
    Interpret OpCode
  }
  ```

  Virtues of this model include ease of debugging, portability, and ease of synchronization (you can simply count clock cycles passed and tie the rest of your emulation to this cycle count).

  A single, big, and obvious weakness is the low perfomance. The interpretation takes a lot of CPU time and you may require a pretty fast computer to run your code at a decent speed.

- **Static Recompilation**
  In this technique, you take a program written in the emulated code and attempt to translate it into the assembly code of your computer. The result will be a usual executable file which you can run on your computer without any special tools. While static recompilation sounds very nice, it is not always possible. For example, you cannot statically recompile self-modifying code as there is no way to tell what it will become without running it. To avoid such situations, you may try combining static recompiler with an interpreter or a dynamic recompiler.

- **Dynamic Recompilation**
  Dynamic recompilation is essentially the same thing as the static one, but it occurs during program execution. Instead of trying to recompile all the code at once, do it on the fly when you encounter CALL or JUMP instructions. To increase speed, this technique can be combined with the static recompilation. You can read more on dynamic recompilation in the white paper by Ardi, creators of the recompiling Macintosh emulator.

I want to write an emulator. Where should I start?

In order to write an emulator, you must have a good general knowledge of computer programming and digital electronics. Experience in assembly programming comes very handy too.

1. Select a programming language to use.
2. Find all available information about the emulated hardware.
3. Write CPU emulation or get existing code for the CPU emulation.
4. Write some draft code to emulate the rest of the hardware, at least partially.
5. At this point, it is useful to write a little built-in debugger which allows to stop emulation and see what the program is doing. You may also need a disassembler of the emulated system assembly language. Write your own if none exist.
6. Try running programs on your emulator.
7. Use disassembler and debugger to see how programs use the hardware and adjust your code appropriately.

Which programming language should I use?

The most obvious alternatives are C and Assembly. Here are pros and cons of each of them:

- **Assembly Languages**

```
+ Generally, allow to produce faster code.
+ The emulating CPU registers can be used to directly
  store the registers of the emulated CPU.
+ Many opcodes can be emulated with the similar
  opcodes of the emulating CPU.
- The code is not portable, i.e. it can not be run on
  a computer with different architecture.
- It is difficult to debug and maintain the code.
```

- **C**

```
+ The code can be made portable so that it works on
  different computers and operating systems.
+ It is relatively easy to debug and maintain the
  code.
+ Different hypothesis of how real hardware works
  can be tested quickly.
- C is generally slower than pure assembly code.
```

Good knowledge of the chosen language is an absolute necessity for writing a working emulator, as it is quite complex project, and your code should be optimized to run as fast as possible. Computer emulation is definitely **not** one of the projects on which you learn a programming language.

Where do I get information on the emulated hardware?

Following is a list of places where you may want to look.

Newsgroups

- **comp.emulators.misc**
  This is a newsgroup for the general discussion about computer emulation. Many emulator authors read it, although the noise level is somewhat high. Read the c.e.m FAQ before posting to this newsgroup.
- **comp.emulators.game-consoles**
  Same as comp.emulators.misc, but specifically dealing with videogame console emulators. Read the c.e.m FAQ before posting to this newsgroup.
- **comp.sys./emulated-system/**
  The comp.sys.* hierarchy contains newsgroups dedicated to specific computers. You may obtain a lot of useful technical information by reading these newsgroups. Typical examples:

  ```
  comp.sys.msx       MSX/MSX2/MSX2+/TurboR computers
  comp.sys.sinclair  Sinclair ZX80/ZX81/ZXSpectrum/QL
  comp.sys.apple2    Apple ][
  etc.
  ```

  Please, check the appropriate FAQs before posting to these newsgroups.
- **alt.folklore.computers**
- **rec.games.video.classic**

FTP

Console and Game Programming site in Oulu, Finland
Arcade Videogame Hardware archive at ftp.spies.com
Computer History and Emulation archive at KOMKON

WWW

 My Homepage
 Arcade Emulation Programming Repository
 Emulation Programmer's Resource

---

How do I emulate a CPU?

First of all, if you only need to emulate a standard Z80 or 6502 CPU, you can use one of the CPU emulators I wrote. Certain conditions apply to their usage though.

For those who want to write their own CPU emulation core or interested to know how it works, I provide a skeleton of a typical CPU emulator in C below. In the real emulator, you may want to skip some parts of it and add some others on your own.

```
Counter=InterruptPeriod;
PC=InitialPC;

for(;;)
{
  OpCode=Memory[PC++];
  Counter-=Cycles[OpCode];

  switch(OpCode)
  {
    case OpCode1:
    case OpCode2:
    ...
  }

  if(Counter<=0)
  {
    /* Check for interrupts and do other */
    /* cyclic tasks here                 */
    ...
    Counter+=InterruptPeriod;
    if(ExitRequired) break;
  }
}
```

First, we assign initial values to the CPU cycle counter (`Counter`), and the program counter (`PC`):

```
Counter=InterruptPeriod;
PC=InitialPC;
```

The `Counter` contains the number of CPU cycles left to the next suspected interrupt. Note that interrupt should not necessarily occur when this counter expires: you can use it for many other purposes, such as synchronizing timers, or updating scanlines on the screen. More on this later. The `PC` contains the memory address from which our emulated CPU will read its next opcode.

After initial values are assigned, we start the main loop:

```
for(;;)
{
```

Note that this loop can also be implemented as

```
while(CPUIsRunning)
{
```

where `CPUIsRunning` is a boolean variable. This has certain advantages, as you can terminate the loop at any moment by setting `CPUIsRunning=0`. Unfortunately, checking this variable on every pass takes quite a lot of CPU time, and should be avoided if possible. Also, **do not** implement this loop as

```
while(1)
{
```

because in this case, some compilers will generate code checking whether `1` is true or not. You certainly don't want the compiler to do this unnecessary work on every pass of a loop.

Now, when we are in the loop, the first thing is to read the next opcode, and modify the program counter:

```
OpCode=Memory[PC++];
```

Note that while this is the simplest and fastest way to read from emulated memory, it is **not always feasible**. A more universal way to access memory is covered later in this document.

After the opcode is fetched, we decrease the CPU cycle counter by a number of cycles required for this opcode:

```
Counter-=Cycles[OpCode];
```

The `Cycles[]` table should contain the number of CPU cycles for each opcode. **Beware** that some opcodes (such as conditional jumps or subroutine calls) may take different number of cycles depending on their arguments. This can be adjusted later in the code though.

Now comes the time to interpret the opcode and execute it:

```
switch(OpCode)
{
```

It is a common misconception that the `switch()` construct is inefficient, as it compiles into a chain of `if() ... else if() ...` statements. While this is true for constructs with a small number of cases, the large constructs (100-200 and more cases) always appear to compile into a jump table, which makes them quite efficient.

There are two alternative ways to interpret the opcodes. The first is to make a table of functions and call an appropriate one. This method appears to be less efficient than a `switch()`, as you get the overhead from function calls. The second method would be to make a table of labels, and use the `goto` statement. While this method is slightly faster than a `switch()`, it will only work on compilers supporting "precomputed labels". Other compilers will not allow you to create an array of label addresses.

After we successfully interpreted and executed an opcode, the comes a time to check whether we need any interrupts. At this moment, you can also perform any tasks which need to be synchronized with the system clock:

```
if(Counter<=0)
{
  /* Check for interrupts and do other hardware emulation here */
  ...
  Counter+=InterruptPeriod;
  if(ExitRequired) break;
}
```

These cyclic tasks are covered later in this document.

Note that we do not simply assign `Counter=InterruptPeriod`, but do a `Counter+=InterruptPeriod`: this makes cycle counting more precise, as there may be some negative number of cycles in the `Counter`.

Also, look at the

```
if(ExitRequired) break;
```

line. As it is too costly to check for an exit on every pass of the loop, we do it only when the `Counter` expires: this will still exit the emulation when you set `ExitRequired=1`, but it won't take as much CPU time.

How do I handle accesses to emulated memory?

The simplest way to access emulated memory is to treat it as a plain array of bytes (words, etc.). Accessing it is trivial then:

```
  Data=Memory[Address1]; /* Read from Address1 */
  Memory[Address2]=Data; /* Write to Address2  */
```

Such simple memory access is not always possible for following reasons though:

- **Paged Memory**
  The address space may be fragmented into switchable pages (aka banks). This is often done to expand memory when the address space is small (64kB).

- **Mirrored Memory**
  An area of memory may be accessible at several different addresses. For example, the data you write into location $4000 will also appear at $6000 and $8000. The ROMs may also be mirrored due to incomplete address decoding.

- **ROM Protection**
  Some cartridge-based software (such as MSX games, for example) tries to write into its own ROM and refuses to work if writing succeeds. This is often done for copy protection. To make such software work on your emulator, you should disable writes into ROM.

- **Memory-Mapped I/O**
  There may be memory-mapped I/O devices in the system. Accesses to such memory locations produce "special effects" and therefore should be tracked.

To cope with these problems, we introduce a couple of functions:

```
  Data=ReadMemory(Address1);  /* Read from Address1 */
  WriteMemory(Address2,Data); /* Write to Address2  */
```

All special processing such as page access, mirroring, I/O handling, etc., is done inside these functions.

`ReadMemory()` and `WriteMemory()` usually put a lot of overhead on the emulation because they get called very frequently. Therefore, they must be made as efficient as possible. Here is an example of these functions written to access paged address space:

```
static inline byte ReadMemory(register word Address)
{
  return(MemoryPage[Address>>13][Address&0x1FFF]);
}

static inline void WriteMemory(register word Address,register byte Value)
```

```
{
  MemoryPage[Address>>13][Address&0x1FFF]=Value;
}
```

Notice the `inline` keyword. It will tell compiler to embed the function into the code, instead of making calls to it. If your compiler does not support `inline` or `_inline`, try making function `static`: some compilers (WatcomC, for example) will optimize short static functions by inlining them.

Also, keep in mind that in most cases the `ReadMemory()` is called several times more frequently than `WriteMemory()`. Therefore, it is worth to implement most of the code in `WriteMemory()` leaving `ReadMemory()` as short and simple as possible.

- **A little note on memory mirroring:**
  As was said before, many computers have mirrored RAM where a value written into one location will appear in others. While this situation can be handled in the `ReadMemory()`, it is usually not desirable, as `ReadMemory()` gets called much more frequently than `WriteMemory()`. A more efficient way would be to implement memory mirroring in the `WriteMemory()` function.

Cyclic tasks: what are they?

Cyclic tasks are things which should periodically occur in an emulated machine, such as:

- Screen refresh
- VBlank and HBlank interrupts
- Updating timers
- Updating sound parameters
- Updating keyboard/joysticks state
- etc.

In order to emulate such tasks, you should tie them to appropriate number of CPU cycles. For example, if CPU is supposed to run at 2.5MHz and the display uses 50Hz refresh frequency (standard for PAL video), the VBlank interrupt will have to occur every

```
2500000/50 = 50000 CPU cycles
```

Now, if we assume that the entire screen (including VBlank) is 256 scanlines tall and 212 of them are actually shown at the display (i.e. other 44 fall into VBlank), we get that your emulation must refresh a scanline each

```
50000/256 ~= 195 CPU cyles
```

After that, you should generate a VBlank interrupt and then do nothing until we are done with VBlank, i.e. for

```
(256-212)*50000/256 = 44*50000/256 ~= 8594 CPU cycles
```

Carefully calculate numbers of CPU cycles needed for each task, then use their **biggest common divisor** for `InterruptPeriod` and tie all other tasks to it (they should not necessarily execute on every expiration of the `Counter`).

How do I optimize C code?

First, a lot of additional code perfomance can be achieved by choosing right optimization options for the compiler. Based on my experience, following combinations of flags will give you the best execution speed:

```
Watcom C++      -oneatx -zp4 -5r -fp3
GNU C++         -O3 -fomit-frame-pointer
Borland C++
```

If you find a better set of options for one of these compilers or a different compiler, please, let me know about it.

- **A little note on loop unrolling:**
  It may appear useful to switch on the "loop unrolling" option of the optimizer. This option will try to convert short loops into linear pieces of code. My experience shows, though, that this option does not produce any perfomance boost. Turning it on may also break your code in some very special cases.

Optimizing the C code itself is slightly trickier than choosing compiler options, and generally depends on the CPU for which you compile the code. Several general rules tend to apply to all CPUs. Do not take them for absolute truths though, as your mileage may vary:

- **Use the profiler!**
  A run of your program under a decent profiling utility (`GPROF` immediately comes to mind) may reveal a lot of wonderful things you have never suspected before. You may find that seemingly insignificant pieces of code are executed much more frequently than the rest of it and slow the entire program down. Optimizing these pieces of code or rewriting them in assembly language will boost the perfomance.

- **Avoid C++**
  Avoid using any constructs which will force you to compile your program with a C++ compiler instead of plain C: C++ compilers usually add more overhead to the generated code.

- **Size of integers**
  Try to use only integers of the base size supported by the CPU, i.e. `int` ones as opposed to `short` or `long`. This will reduce amount of code compiler generates to convert between different integer lengths. It may also reduce the memory access time, as some CPUs work fastest when reading/writing data of the base size aligned to the base size address boundaries.

- **Register allocation**
  Use as few variables as possible in each block and declare most frequently used ones as `register` (most new compilers can automatically put variables into registers though). This makes more sense for CPUs with many general-purpose registers (PowerPC) than for ones with a few dedicated registers (Intel 80x86).

- **Unroll small loops**
  If you happen to have a small loop which executes a few times, it is always a good idea to manually unroll it into a linear piece of code. See the note above about the automatic loop unrolling.

- **Shifts vs. multiplication/division**
  Always use shifts wherever you need to multiply or divide by 2^n (`J/128==J>>7`). They execute faster on most CPUs. Also, use bitwise AND to obtain the modulo in such cases (`J%128==J&0x7F`).

What is low/high-endianess?

All CPUs are generally divided into several classes, depending on how they store data in memory. While there are some very peculiar specimens, most CPUs fall into one of two classes:

- **High-endian** CPUs will store data so that higher bytes of a word always occur first in memory. For example, if you store `0x12345678` on such CPU, the memory will look like this:

```
 0  1  2  3
+--+--+--+--+
|12|34|56|78|
+--+--+--+--+
```

- **Low-endian** CPUs will store data so that lower bytes of a word always occur first in memory. The example from above will look quite differently on such CPU:

```
 0  1  2  3
+--+--+--+--+
|78|56|34|12|
+--+--+--+--+
```

Typical examples of high-endian CPUs are 6809, Motorola 680x0 series, PowerPC, and Sun SPARC. Low-endian CPUs include 6502 and its successor 65816, Zilog Z80, most Intel chips (including 8080 and 80x86), DEC Alpha, etc.

When writing an emulator, you have to be aware of the endianess of both your emulated and emulating CPUs. Let's say that you want to emulate a Z80 CPU which is low-endian. That is, Z80 stores its 16-bit words with lower byte first. If you use a low-endian CPU (for example, Intel 80x86) for this, everything happens naturally. If you use a high-endian CPU (PowerPC) though, there is suddenly a problem with placing 16-bit Z80 data into memory. Even worse, if your program must work on both architectures, you need some way to sense the endiness.

One way to handle the endiness problem is given below:

```
typedef union
{

  short W;        /* Word access */

  struct          /* Byte access... */
  {
#ifdef LOW_ENDIAN
    byte l,h;     /* ...in low-endian architecture */
#else
    byte h,l;     /* ...in high-endian architecture */
#endif
  } B;

} word;
```

As you see, a word can be accessed as whole using `W`. Every time your emulation needs to access it as separate bytes though, you use `B.l` and `B.h` which preserves order.

If your program is going to be compiled on different platforms, you may want to test that it was compiled with correct endiness flag before executing anything really important. Here is one way to perform such a test:

```
  int *T;

  T=(int *)"\01\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";
  if(*T==1) printf("This machine is high-endian.\n");
  else      printf("This machine is low-endian.\n");
```

How to make program portable?

To be written.

Why should I make program modular?

Most computer systems are made of several large chips each of which performs certain part of system functions. Thus, there is a CPU, a video controller, a sound generator, and so forth. Some of these chips may have their own memory and other hardware attached to them.

A typical emulator should repeat the original system design by implementing each subsystem functions in a separate module. First, this makes debugging easier as all bugs are localized in the modules. Second, the modular architecture allows you to reuse modules in other emulators. The computer hardware is quite standarized: you can expect to find the same CPU or video chip in many different computer models. It is much easier to emulate the chip once than implement it over and over for each computer using this chip.
