Implementation Guide
For
x86Lib
by Jordan Earls

Hello There! So, you want to use this weird library thing I made... Well, in this little guide, I'll try to teach you how to use it, and even a little bit on how to get started emulating... I love to use example code rather than just plain text, so we are going to code this document up! Along with this guide, there is a /sample/ directory which will have the finished example code and makefile....

First thing you'll need to do is build x86Lib. It is a very simple thing to build, as it is almost OS-Independent, and only uses the C and C++ standard libraries.
Note you can skip this if you have Windows and have downloaded a binary release
Just download the latest release or if you want get the latest from SVN and extract it to whatever directory.
No configuration is needed, just goto bash or cmd.exe and type “make release” and it will build libx86lib.a and a test program using it(this is to make sure it links properly)

Ok, lets get started! Now, I am assuming that you will be using C++ for your emulation project. It is possible to use most any other language, but a proper “layer file” must be used in order to get rid of the C++ness... What I aim to do in this guide is make an example emulator with a “put on screen” display port thing, a getkey port, an instruction “timer” thing that causes interrupts, 1MB of addressable memory and load a binary file from disk and place it in 0xF000:0000. Also, we will handle triple fault exceptions and such...

Lets start with a little template
[code: ./1_template/main.cpp]

```
#include <iostream>
#include <x86Lib.h>

using namespace std;
using namespace x86Lib; //the x86Lib namespace...just for simplicity, use the whole thing..


/**We just need to define the functions to create a x86CPU class...**/
PhysMemory::PhysMemory(){
	size=0xFFFFF; //in PhysMemory, there is a variable named size, this isn't some magic thing that came out of nowhere...lol
	ptr=new uint8_t[size];
	memset(ptr,0x66,size); //initialize it all to 0x66, an invalid opcode(so we catch bugs early)
}
PhysMemory::~PhysMemory(){
	delete []ptr;
}

uint8_t PhysMemory::ReadByte(uint32_t off){
```

```
        return 0; //avoid warnings...
}

uint16_t PhysMemory::ReadWord(uint32_t off){
        return 0; //avoid warnings...
}

uint32_t PhysMemory::ReadDword(uint32_t off){
        return 0; //avoid warnings...
}

void PhysMemory::WriteByte(uint32_t off,uint8_t val){

}

void PhysMemory::WriteWord(uint32_t off,uint16_t val){

}

void PhysMemory::WriteDword(uint32_t off,uint32_t val){

}

uint32_t PhysMemory::GetSize(){
        return size;
}

bool PhysMemory::CanRead(uint32_t off){
        return 0; //avoid warnings...
}

bool PhysMemory::CanWrite(uint32_t off){
        return 0; //avoid warnings...
}


/**These don't have to be defined until much later...**/
x86Ports::x86Ports(){

}

x86Ports::~x86Ports(){

}


void x86Ports::WriteByte(uint16_t port,uint8_t val){

}

void x86Ports::WriteWord(uint16_t port,uint16_t val){

}

void x86Ports::WriteDword(uint16_t port,uint32_t val){ //not used yet, but must be
defined
}

uint8_t x86Ports::ReadByte(uint16_t port){
```

```
        return 0; //avoid warnings...
}

uint16_t x86Ports::ReadWord(uint16_t port){
        return 0; //avoid warnings...
}

uint32_t x86Ports::ReadDword(uint16_t port){ //not used yet, but must be defined
        return 0; //avoid warnings...
}

PhysMemory physical_memory;
x86Ports dev_ports;
x86CPU cpu_ctrl(&physical_memory,&dev_ports,0,CPU_DEFAULT);
//Intialize cpu_ctrl with our physical memory, ports, no flags, and default CPU
level...

int main(){

    return 0;
}
```
[/code]
The template code should be pretty simple and easy to understand...
I will go over it anyway though..

The PhysMemory and x86Ports classes are prototyped in x86Lib.h, but the functions are not made in x86Lib, so you must define all of them....(or else you will get a linker error)
The PhysMemory class is of course the class that handles physical memory. It's functions are used when you do most anything with memory, including a single CPU cycle. Basically, the CPU requests to read/write to a 32bit flat address(you need not worry about segments and such internal CPU things) and then you give it back the byte at that address, or you write a byte to that address...

x86Ports is what controls the ports of the CPU. It is called when you do an in/out instruction. These will be discussed later...

Now we will eventually load a binary assembly file as the BIOS(if you dare call it that) so lets go ahead and make that file, so we have it in our makefile and such...
[code: ./1_template/test.asm]
```
org 0

nop
hlt
;do nothing! w00t just like in real life!
```
[/code]

Note that I use FASM to build this. Basically, this just does nothing and halts the CPU(actually causing a panic, but we'll get to that later...)

Now let's build all of this!
[code: ./1_template/makefile]
```
#x86Lib Sample Implementation Makefile

_CPPFLAGS=-Wall -g -fexceptions -I./../include
```

```
default:
	fasm test.asm test.bin
	g++ $(_CPPFLAGS) -c main.cpp -o main.o
	g++ $(_CPPFLAGS) -o x86Emulator main.o -lx86Lib -L./../lib
```
[/code]

Now we have our template emulator built and coded.

Ok, so next thing to do, code it up so it can actually execute instructions.

First thing we need to do is implement memory IO. Our memory model we will use for our example will be like that of an actual PC. 1MB of memory, and ROM above 0xC0000. If the CPU tries to write data into ROM, we need to throw an exception. We use the exception class of Mem_excp. The constructor of Mem_excp takes just one argument, the offset at which the error occurred.
So lets fill in those functions for PhysMemory.
[code: ./2_basics/main.cpp]
//~Line 8
```
static const uint32_t ROM_START=0xC0000;
```
//~Line 22
```
uint8_t PhysMemory::ReadByte(uint32_t off){
	if(off>size){
		throw Mem_excp(off);
	}
	return ptr[off];
}

uint16_t PhysMemory::ReadWord(uint32_t off){
	if((off+1)>size){ //add 1 to offset because we read 2 bytes(a word), rather than 1 byte
		throw Mem_excp(off);
	}
	return *(uint16_t*)&ptr[off];
}

uint32_t PhysMemory::ReadDword(uint32_t off){
	if((off+3)>size){
		throw Mem_excp(off);
	}
	return *(uint32_t*)&ptr[off];
}

void PhysMemory::WriteByte(uint32_t off,uint8_t val){
	if(off>ROM_START){
		throw Mem_excp(off);
	}
	ptr[off]=val;
}

void PhysMemory::WriteWord(uint32_t off,uint16_t val){
	if((off+1)>ROM_START){
		throw Mem_excp(off);
	}
	*(uint16_t*)&ptr[off]=val;
}

void PhysMemory::WriteDword(uint32_t off,uint32_t val){
```

```
		if((off+3)>ROM_START){
			throw Mem_excp(off);
		}
		*(uint32_t*)&ptr[off]=val;
}

uint32_t PhysMemory::GetSize(){
		return size;
}

bool PhysMemory::CanRead(uint32_t off){
		if(off<size){
			return true;
		}else{
			return false;
		}
}

bool PhysMemory::CanWrite(uint32_t off){
		if(off<ROM_START){
			return true;
		}else{
			return false;
		}
}
//~cut
[/code]
```

This is pretty simple right? Note the use of casting to avoid temp variables. In case you don't understand it, *(uint32_t*)&ptr[off] will get the uint8_t* location of ptr[off] and will then cast that pointer to a uint32_t* and then it will dereference the pointer as a uint32_t.

Now that we can read and write memory, what are we going to do with it? Well, we need to load a ROM BIOS at 0xF0000. Should be pretty simple, just use physical_memory.WriteByte() a few times... But wait! We can't write into ROM with that function! So we need to do it inside of the class so we can access the ptr variable. The only place, and logical place to load the BIOS would be in the constructor. Afterall, ROM can't change, so it only needs to be loaded once.

[code: ./2_basics/main.cpp]

```
//~line 1
#include <iostream>
#include <fstream>
#include <x86Lib.h>

using namespace std;
using namespace x86Lib; //the x86Lib namespace...just for simplicity, use the whole thing..

static const uint32_t ROM_START=0xC0000;
/**We just need to define the functions to create a x86CPU class...**/
PhysMemory::PhysMemory(){
		size=0xFFFFF; //in PhysMemory, there is a variable named size, this isn't some magic thing that came out of nowhere...lol
		ptr=new uint8_t[size];
		memset(ptr,0x66,size); //initialize it all to 0x66, an invalid opcode
		ifstream bios("test.bin",ios::binary); //open it readonly binary
```

```
		bios.read((char*)&ptr[0xF0000],0xFFFF);
}
//~cut
```
[/code]

Quite simple, and logical BIOS loading code...
Now, how do we actually get started executing instructions?

To execute a single instruction(or possibly prefix) we use the x86CPU::Cycle() function. Basically, what we do is make an infinite loop executing that function. There is just one other thing that must be added other than that though. A try-catch block must be used. As of the time of this writing, the only exception that gets thrown out of the CPU(and at the implementation) is the CpuPanic_excp which implies either a constant stand still(which could actually be ignored) or a triple fault(or on 8086, any fault but divide by zero).

[code: ./2_basics/main.cpp]
```
//~line 124
int main(){
	for(;;){ //infinite loop
		try{
			cpu_ctrl.Cycle();
		}
		catch(CpuPanic_excp error){
			//Oh noes! triple fault or infinite loop of nothing!
			cout << "CPU Panic!" <<endl;
			cout << "Message: " << error.desc << endl; //give some info on the error...
			cout << "Code: 0x" << hex << error.code << endl; //Give the code: for development purposes
			cpu_ctrl.DumpState(cout); //this will dump the cpu state to cout
			return 1;
		}
	}
}
//~cut
```
[/code]

Now you can go ahead and make the program. You should get a dump of the registers, with everything 0 except for CS and IP. CS should be 0xF000 and IP should be 1.

Let's make a bit more extensive code sample for our BIOS...
[code: ./2_basics/test.asm]
```
org 0

cli
mov ax,0
mov ss,ax
mov sp,0x500
mov ax,0x1000
mov ds,ax
mov ax,0
call do_something
call cause_error
```

```
cli
hlt

do_something:
mov cx,0xFFFF
.l1:
mov bx,cx
mov [bx],cx
loop .l1
ret

cause_error:
mov bx,0
mov [cs:bx],word 0 ;write into read-only memory
ret
```
[/code]

Whenever you build and run this, it should display Mem Error—16bit faults. (with code 0x10000) This means that our exceptions and such are working properly.

Now, what else could their be to program in? Well, remember those things we skipped over earlier? x86Ports, yea, we're going to finish implementing their functions now...

Ports can be used for almost anything, including getting IO, debugging assembly code, even internal emulator control... there are informal guidelines based on how PC ports work, but in reality, ports can do most anything you want them to. We are going to use that to our advantage to make some simple functions for our emulator. We are going to make it so programs can put an ASCII char on the screen, and so that programs can get input from the keyboard. Though the keyboard input isn't very powerful, for the intention of this document to be portable, we will only use getchar() and putchar().

Basically, for writing ports(from the programs point of view, out ports) you have two arguments. One is the port number requested. This is so you can make multiple functions and split them off based on the port number. Also, given is the value the program gives us to use. The use of this depends on the port function. For reading ports(or in ports) we have just one argument, but must return a value also. We just return the value from whatever function is given in the port number.

Here is my simple implementation of a write char and get char port functions...

[code: ./3_ports/main.cpp]
```
//~line 95
void x86Ports::WriteByte(uint16_t port,uint8_t val){
        switch(port){
                case 0xD0: //display ASCI char
                putchar(val);
                break;
                //We could add more functions here...

                default:
                cout << "Warning: unknown port" << endl;
                break;
        }
}
```

```
void x86Ports::WriteWord(uint16_t port,uint16_t val){
          //nothing here yet...
}

void x86Ports::WriteDword(uint16_t port,uint32_t val){ //not used yet, but must be
defined
}

uint8_t x86Ports::ReadByte(uint16_t port){
          switch(port){
                    case 0xC0: //get ASCII char
                    return getchar();
                    break;

                    default:
                    cout << "Warning: unknown port" << endl;
                    break;
          }
          return 0;

}

uint16_t x86Ports::ReadWord(uint16_t port){
          //nothing here yet..
          return 0; //avoid warnings...
}

uint32_t x86Ports::ReadDword(uint16_t port){ //not used yet, but must be defined
          return 0; //avoid warnings...
}
```

//~cut
[/code]

The code is all fairly simple and short.

Now lets test these new port functions and make sure they work.

[code: ./3_ports/test.asm]

```
org 0

cli
mov ax,0
mov ss,ax
mov sp,0x500
mov ax,0x1000
mov ds,ax
mov ax,0
;;end init code...
mov ax,cs
mov es,ax
mov bx,hello_string

call putstring
nop
call getkey
cmp al,'A'
jne .error
```

```
mov bx,good_string
call putstring

cli
hlt

.error:
mov bx,error_string
call putstring
cli
hlt



;takes no args, returns with AL being the ASCII char
getkey:
in al,0xC0
ret

;ES:BX is string..
putstring:
.loop1:
mov al,[ES:BX]
cmp al,0
je .end
out 0xD0,al
inc bx
jmp .loop1
.end:
ret

hello_string: db 'hello there Mr. World! Type the letter A: ',0
error_string: db "An Error occurred!",0x0A,0
good_string: db "All successful!",0x0A,0
```
[/code]

This will make it ask you to type 'A'. If you type A, it tells you “All Successful.” If you type something else, it tells you “An Error occurred”.

Now, there is really only one thing left to teach... CPU Interrupts.

Interrupts are used by devices to alert the processor(and the program running) that something has happened that needs to be handled. With x86Lib, in order to tell the CPU an interrupt has occurred, you use the function x86CPU::Int(uint8_t). The only argument is which interrupt number to use. Note! This is not like IRQs, this is actual interrupts, you must resolve interrupts to an IRQ range yourself(and it is out of the range of this document). Also note that exceptions can occur in Int(). As of the time of this writing, if IF is not set, and the interrupt is not NMI, the interrupt will be ignored(rather than being put on a stack to be called later). This will hopefully be fixed sometime in the near future.

In our little test emulator, we don't emulate any real hardware, so we will make a little CPU timer function. The program sets this timer to occur every so many instructions. Then, it sets which IRQ number to use(0 will resolve to int 8, 1 to int 9, and so on), and finally, it executes code and it will be interrupted whenever the CPU has executed so many instructions.

[code: ./4_interrrupts/main.cpp]
//~line 83

```
volatile uint8_t counter_int=0x08;
volatile uint16_t counter=0;
volatile uint16_t counter_max=0;



x86Ports::x86Ports(){

}

x86Ports::~x86Ports(){

}


void x86Ports::WriteByte(uint16_t port,uint8_t val){
          switch(port){
                    case 0xD0: //display ASCI char
                    putchar(val);
                    break;
                    //We could add more functions here...

                    case 0x11: //Set Instruction Counter Interrupt number
                    counter_int=(val&0x08)+0x08; //only support interrupts 8-F
                    break;
                    case 0x12: //Reset instruction counter interrupt
                    counter=counter_max;
                    break;

                    default:
                    cout << "Warning: unknown port" << endl;
                    break;
          }
}

void x86Ports::WriteWord(uint16_t port,uint16_t val){
          switch(port){
                    case 0x10: //Set Instruction Counter Interrupt count
                    //note: this should be 0 to disable
                    counter_max=val;
                    counter=counter_max;
                    break;

                    default:
                    cout << "Warning! Undefined Port" << endl;
                    break;
          }
}

void x86Ports::WriteDword(uint16_t port,uint32_t val){ //not used yet, but must be
defined
}

uint8_t x86Ports::ReadByte(uint16_t port){
          switch(port){
                    case 0xC0: //get ASCII char
                    return getchar();
```

```
                    break;
                    case 0x11: //read current instruction timer interrupt number
                    return counter_int-8;
                    break;

                    default:
                    cout << "Warning: unknown port" << endl;
                    break;
          }
          return 0;

}

uint16_t x86Ports::ReadWord(uint16_t port){
          switch(port){
                    case 0x10: //read current instruction count timer
                    return counter_max;
                    break;
                    case 0x13: //Read current instruciton count
                    return counter;
                    break;

                    default:
                    cout << "Warning! Undefined Port" << endl;
                    break;
          }
          return 0; //avoid warnings...
}

uint32_t x86Ports::ReadDword(uint16_t port){ //not used yet, but must be defined
          return 0; //avoid warnings...
}

PhysMemory physical_memory;
x86Ports dev_ports;
x86CPU cpu_ctrl(&physical_memory,&dev_ports,0,CPU_DEFAULT);
//Intialize cpu_ctrl with our physical memory, ports, no flags, and default CPU
level...

int main(){
          for(;;){ //infinite loop
                    try{
                              if(counter!=0){
                                        counter--;
                                        if(counter==0){
                                                  counter=counter_max;
                                                  cpu_ctrl.Int(counter_int);
                                        }
                              }
                              cpu_ctrl.Cycle();
                    }
                    catch(CpuPanic_excp error){
                              //Oh noes! triple fault or infinite loop of nothing!
                              cout << "CPU Panic!" <<endl;
                              cout << "Message: " << error.desc << endl; //give some
info on the error...
                              cout << "Code: 0x" << hex << error.code << endl;
                              cpu_ctrl.DumpState(cout); //this will dump the cpu
state to cout
```

```
                        return 1;
                    }
                }
            }
        }
```
[/code]

In this code, we add six port functions.
Write Byte 0x11 is the function to set the IRQ on which the timer interrupt occurs. It can only be interrupts 8-F or IRQs 0-7, respectively.
Write Byte 0x12 is the function to reset the timer. (it takes no argument, it is just a flip flop thing)
Write Word 0x10 is the function to set how many instructions to execute before causing a timer interrupt. It should be set to 0 to disable the timer.
Read Byte 0x11 is the function to read the IRQ set by Write Byte 0x11.
Read Word 0x10 is the function to read the how many instructions it will execute before causing a timer intterupt. (the timer max)
Read Word 0x13 is the function to read how many instructions have executed since the last timer interrupt.

In the main CPU loop, we added a thing to check if the counter is 0. If it is not 0, then it checks and increments the timer.

Now, lets add some test code.

[code: ./4_interrupts/test.asm]
```
org 0

cli
mov ax,0
mov ss,ax
mov sp,0x500
mov ax,0x1000
mov ds,ax
mov ax,0

call install_interrupts
;;end init code...

mov ax,cs
mov es,ax
mov bx,hello_string

call putstring
call getkey
cmp al,'A'
jne .error
mov bx,good_string
call putstring

mov ax,60000
out 0x10,ax
;cause an interrupt every 60,000 instructions
jmp $ ;infinite loop

cli
hlt
```

```
.error:
mov bx,error_string
call putstring
cli
hlt



;takes no args, returns with AL being the ASCII char
getkey:
in al,0xC0
ret

;ES:BX is string..
putstring:
.loop1:
mov al,[ES:BX]
cmp al,0
je .end
out 0xD0,al
inc bx
jmp .loop1
.end:
ret

install_interrupts:
mov ax,0
mov es,ax

mov bx,0x08*4
mov [es:bx],cs
add bx,2
mov [es:bx],word timer_int
sti
ret


timer_int:
push ax
push bx
push es
mov ax,cs
mov es,ax
mov bx,string1
call putstring
pop es
pop bx
pop ax
iret


string1: db 'X',0
hello_string: db 'Hello there Mr. World! Type the letter A: ',0
error_string: db "An Error occurred!",0x0A,0
good_string: db "All successful!",0x0A,0
```

[/code]

This code installs an interrupt that prints 'X' on the screen every time there is an IRQ 0. IRQ 0 is our timer interrupt, so every time the timer goes off, it will print an 'X'.

After setting up the timer, we just have an infinite loop of instructions. And the timer occurs during this infinite loop printing 'X's on the screen rather rapidly.

Now that you have completed reading this little guide/tutorial you should know enough to make a fairly decent emulator. This code can be used as just a template or whatever. It is all under the BSD license as x86Lib. This is all the power you can get with x86Lib without adding the huge x86Lib_internal.h file... This document will not go into that file, as it is usually not needed in simple emulators. If you wish to use it however, then read the x86Lib Development Guide(which, as of the time of this writing, is no where near finished).

There are many more simple things that can be added to this code. Everything from memory mapped devices, to better devices and port functions, and something to help with debugging...

Thank you for reading this, and you can see http://x86lib.sf.net for more information and documents.