

# Using IDA's GDB debugger with QEMU emulator

*Copyright 2009 Hex-Rays SA*

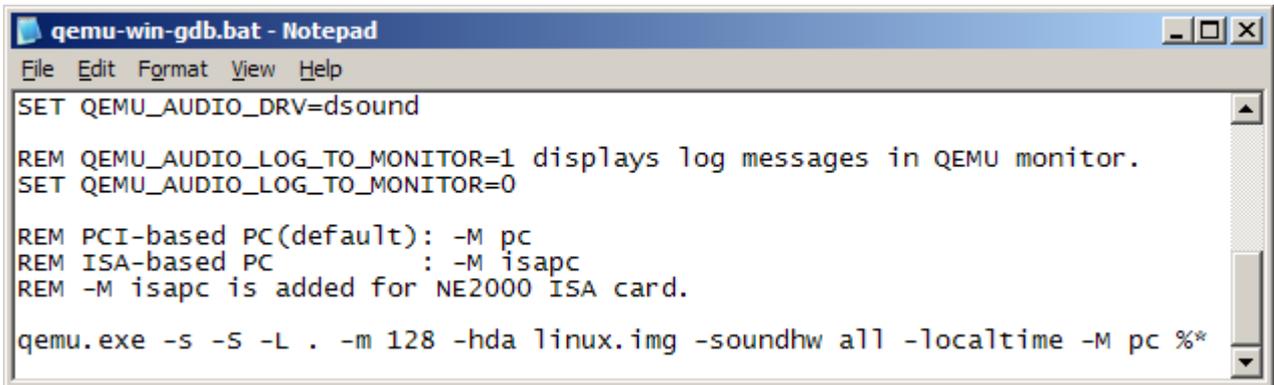
QEMU is a processor emulator which can emulate a handful of processors, including Intel x86 and ARM architectures. It includes a GDB stub which can be used with new GDB debugger plugin in IDA 5.4.

## Getting QEMU

QEMU's home page is at <http://bellard.org/qemu/>. Win32 builds can be downloaded from Takeda Toshiya's page at <http://homepage3.nifty.com/takeda-toshiya/qemu/>. This primer assumes you downloaded QEMU 0.9.1 for Win32 or later. We will debug the small Linux included with QEMU.

## Enabling GDB stub

After unpacking QEMU, make a copy of the `qemu-win.bat` file, for example `qemu-win-gdb.bat` and edit it. Add `-s -S` to the `qemu.exe` call (`-s` enables GDB stub and `-S` instructs QEMU to stop at the system start):



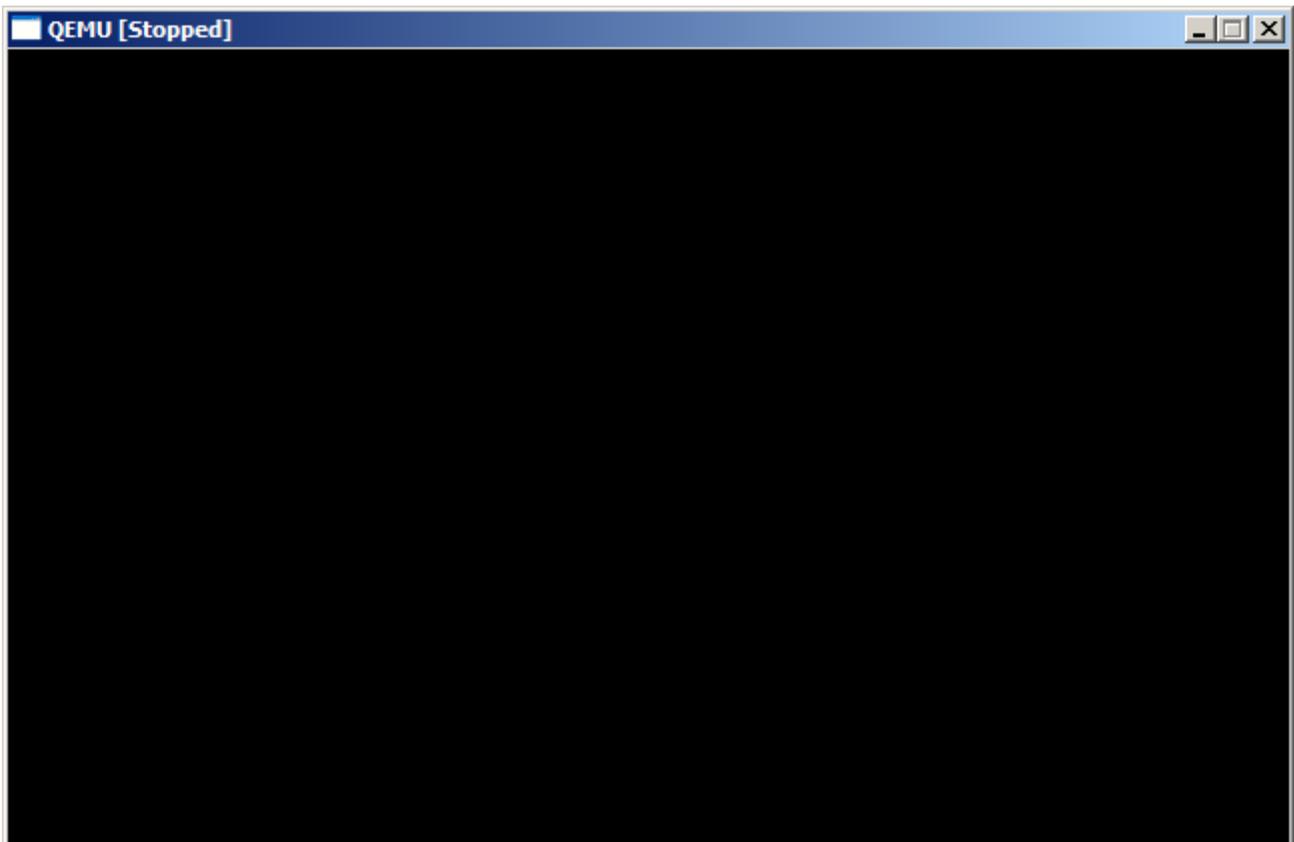
```
qemu-win-gdb.bat - Notepad
File Edit Format View Help
SET QEMU_AUDIO_DRV=dsound

REM QEMU_AUDIO_LOG_TO_MONITOR=1 displays log messages in QEMU monitor.
SET QEMU_AUDIO_LOG_TO_MONITOR=0

REM PCI-based PC(default): -M pc
REM ISA-based PC          : -M isapc
REM -M isapc is added for NE2000 ISA card.

qemu.exe -s -S -L . -m 128 -hda linux.img -soundhw all -localtime -M pc %*
```

Run the `.bat` file. QEMU will stop and wait for the debugger.

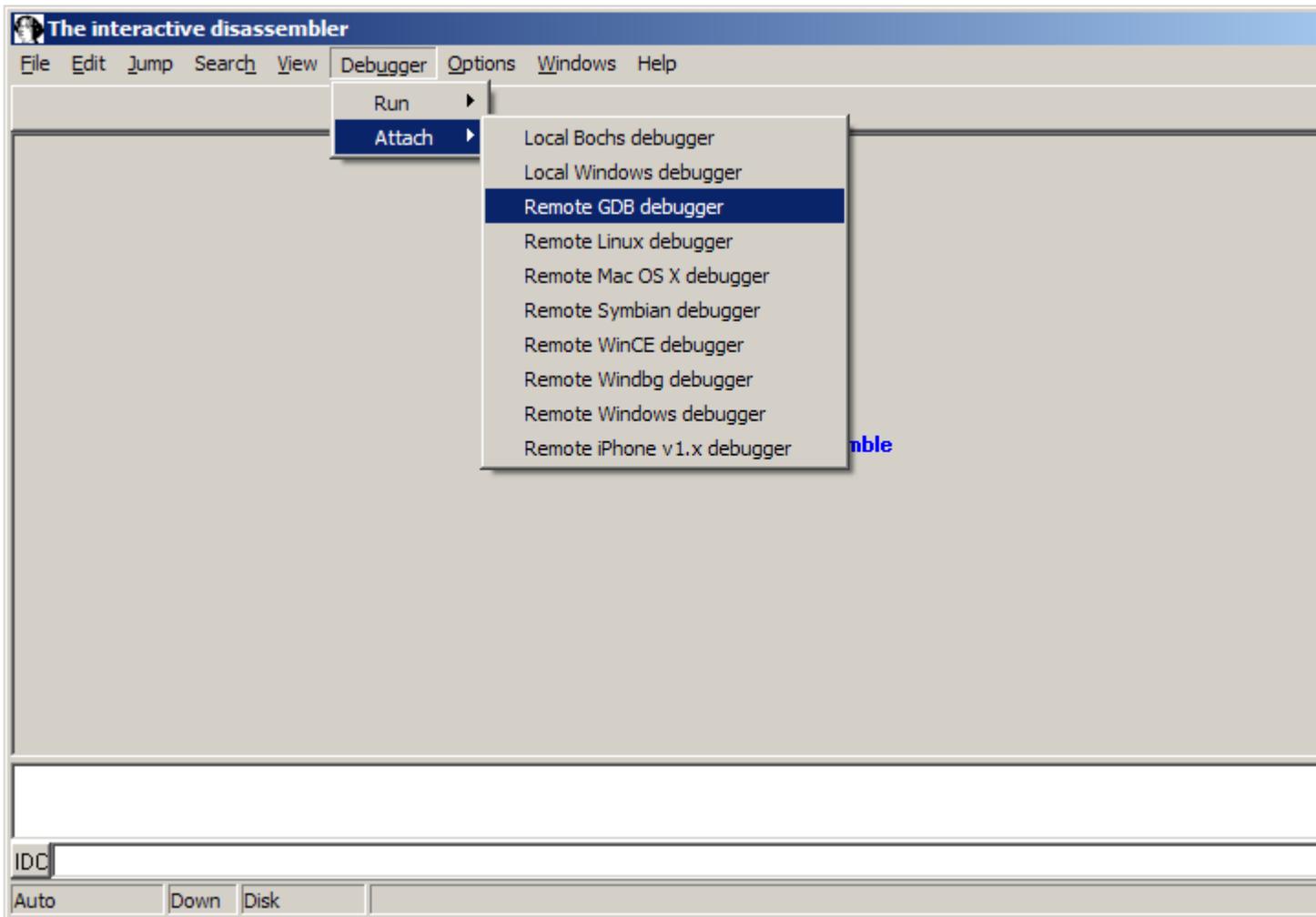


## Debugging with IDA

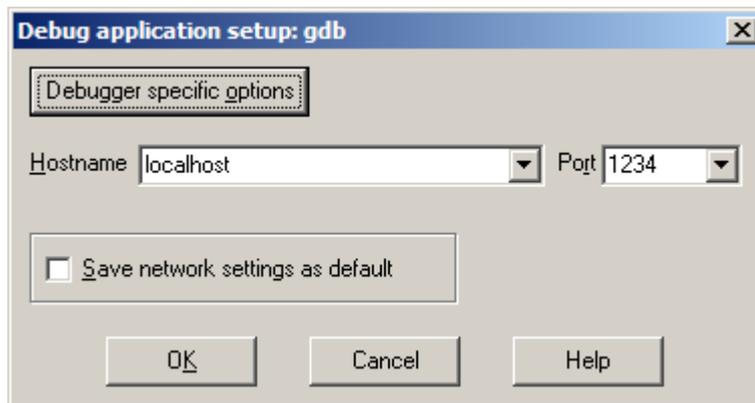
Start IDA.



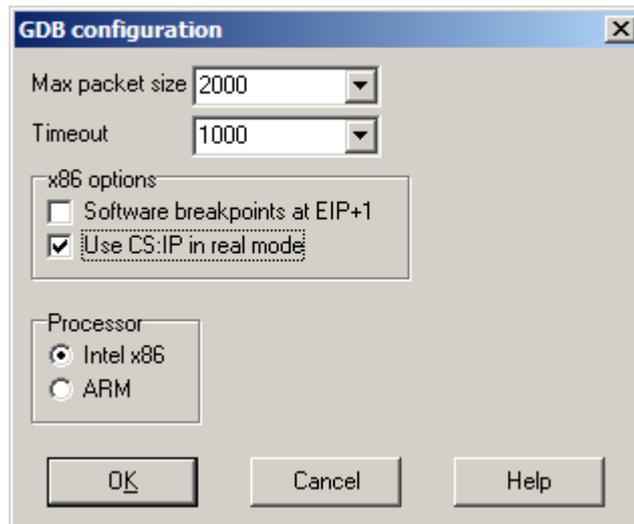
If you get the welcome dialog, choose "Go".



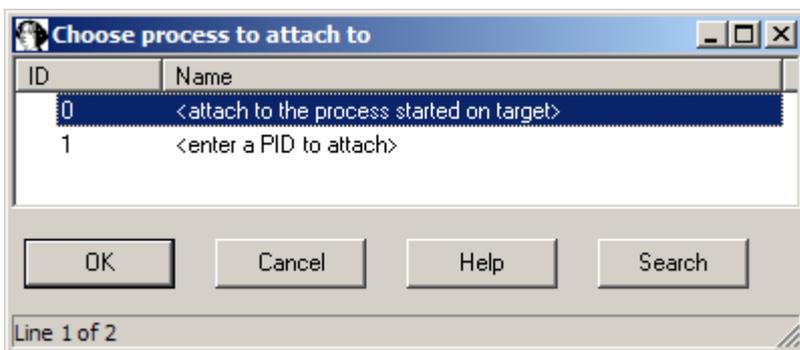
Choose Debugger | Attach | Remote GDB debugger.



Enter "localhost" for hostname and 1234 for the port number. Click "Debugger specific options".

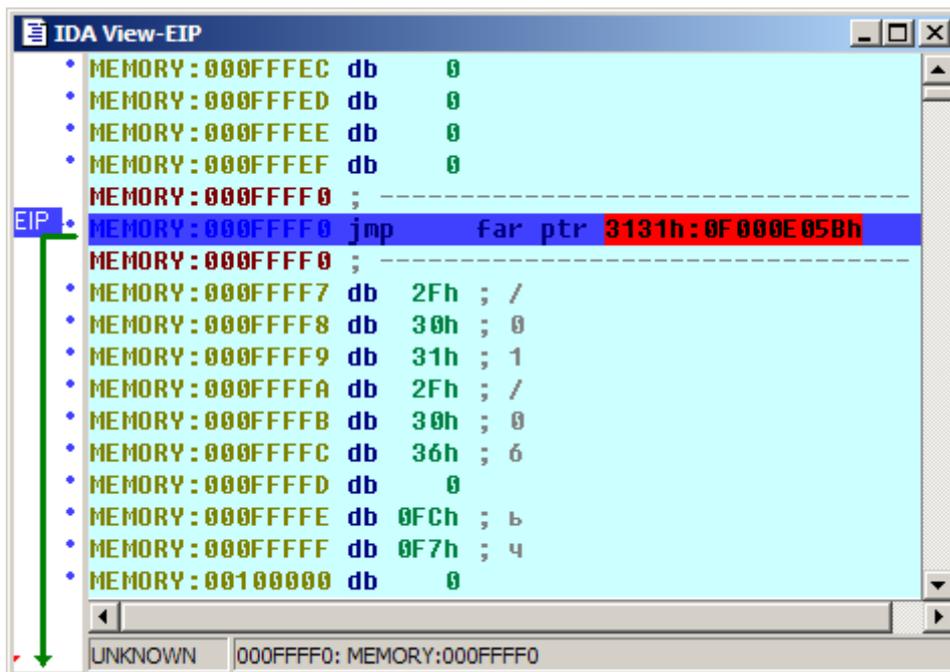


QEMU needs special configuration because it behaves slightly differently from other GDB stubs. Uncheck "Software breakpoints at EIP+1" and check "Use CS:IP in real mode". Make sure Processor is set to "Intel x86". Click OK, then click OK in "Debug application setup" dialog.



Choose <attach to the process started on target> and click OK.

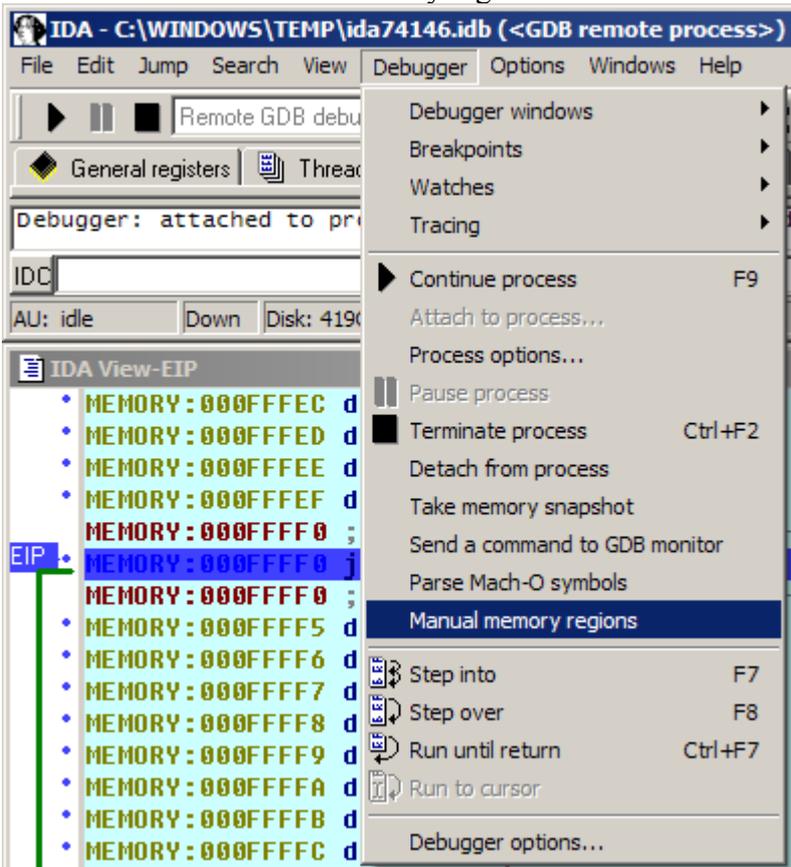
## Debugging the BIOS



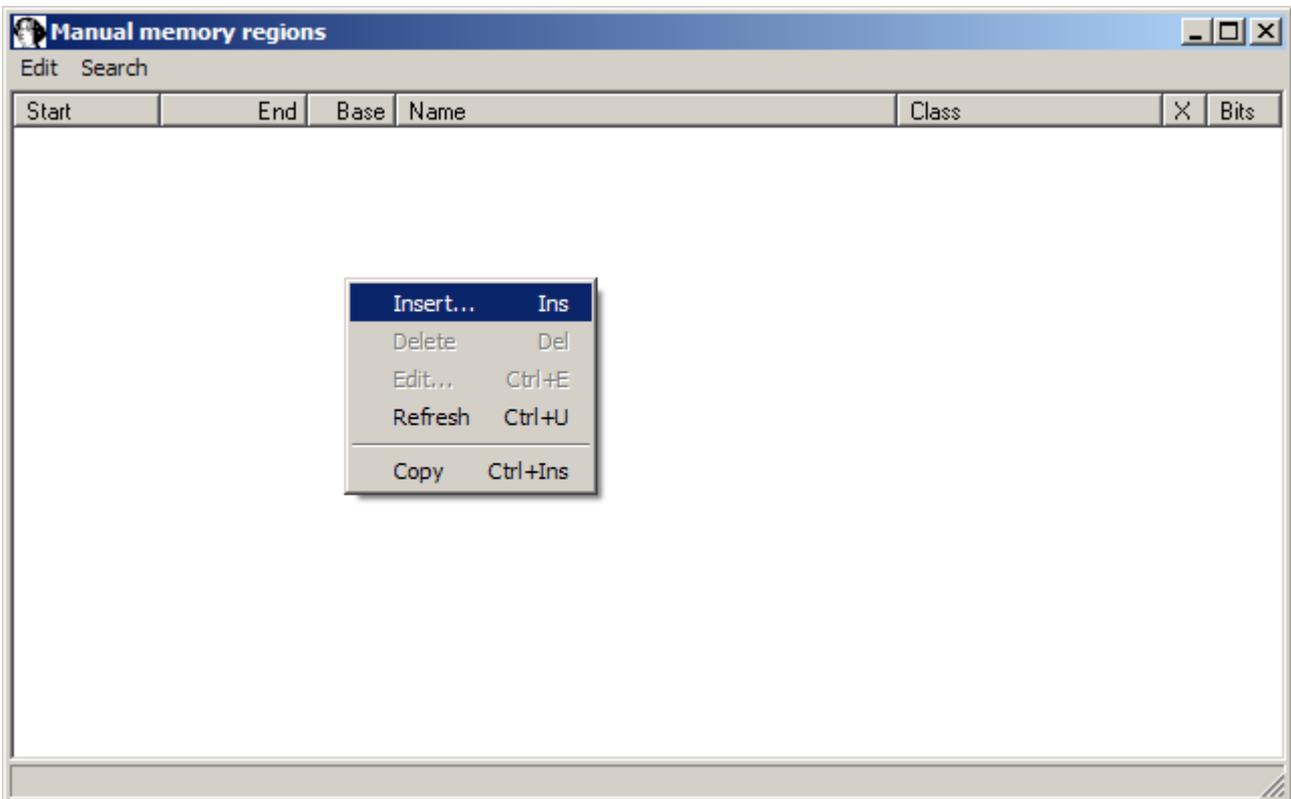
The BIOS entrypoint is displayed but the disassembly is wrong since the default MEMORY

segment is 32-bit.

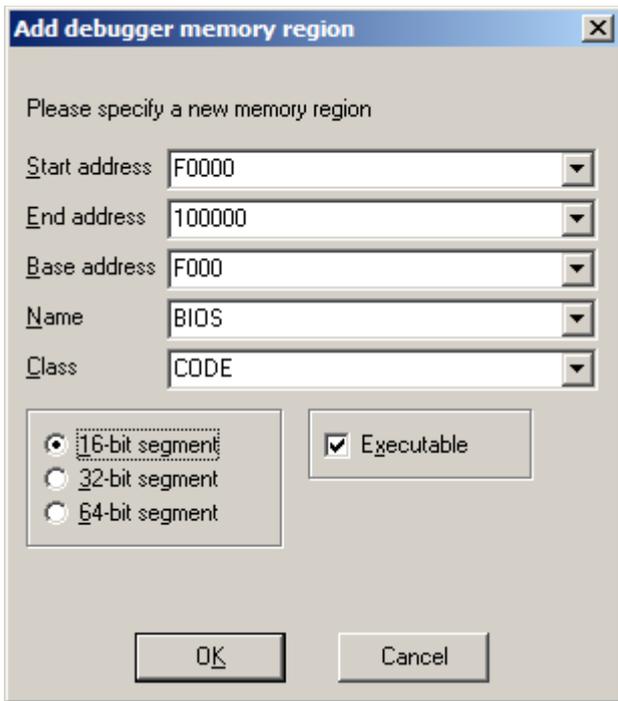
Let's create some manual memory regions to reflect the real memory map.



Go to Debugger | Manual memory regions.



Right-click and choose "Insert..." (or press Ins).



Enter the following details:

Start address: F0000

End address: 100000

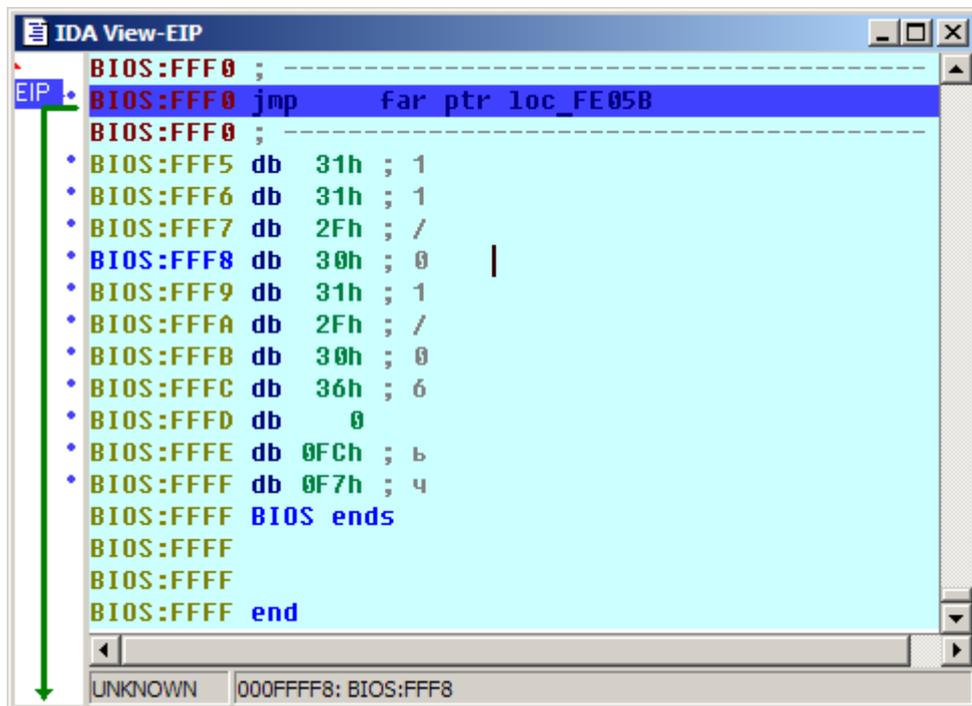
Base address: F000

Name: BIOS

Class: CODE

select "16-bit segment"

Click OK.

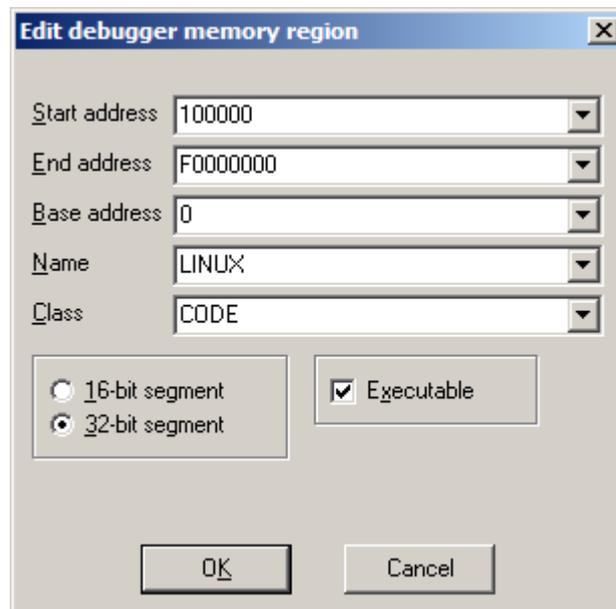


Now the disassembly is correct and you can trace the BIOS code.

## Debugging the kernel

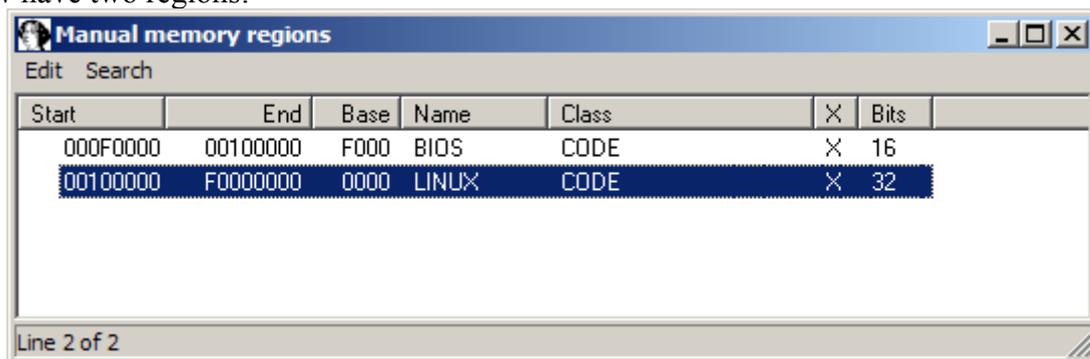
To debug the Linux kernel, we first need to create memory regions where it will execute. The usual kernel entrypoint is at address 100000, and it can use memory almost to the maximum 4G address.

Open memory regions list (Debugger | Manual memory regions) and add a new region:

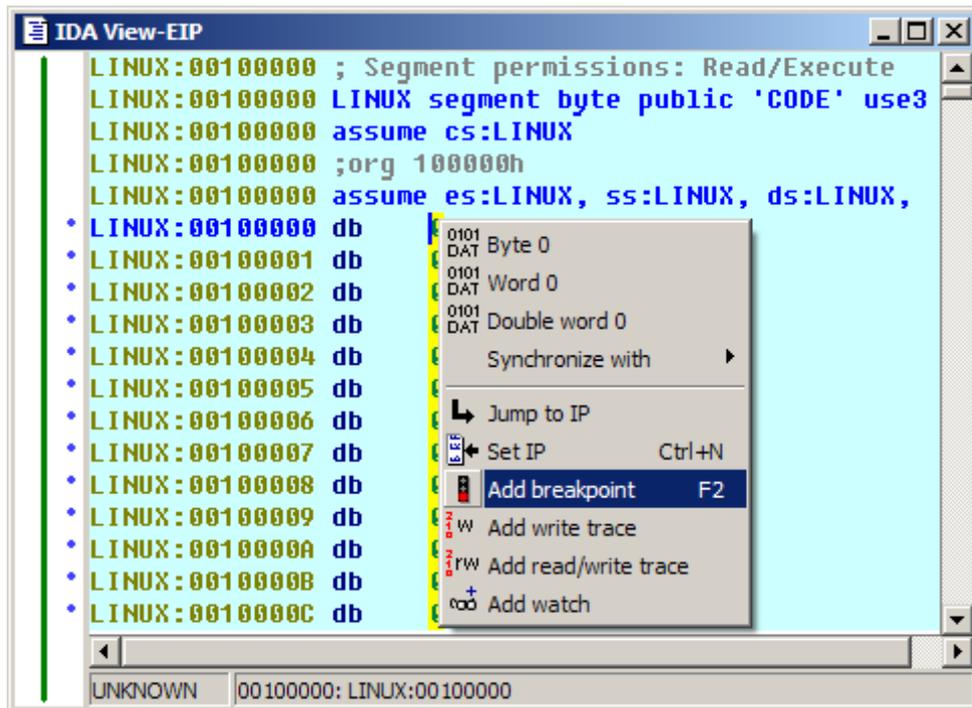


Start address: 100000  
End address: F0000000  
Base address: 0  
Name: LINUX  
Class: CODE  
select "32-bit segment"  
Click OK.

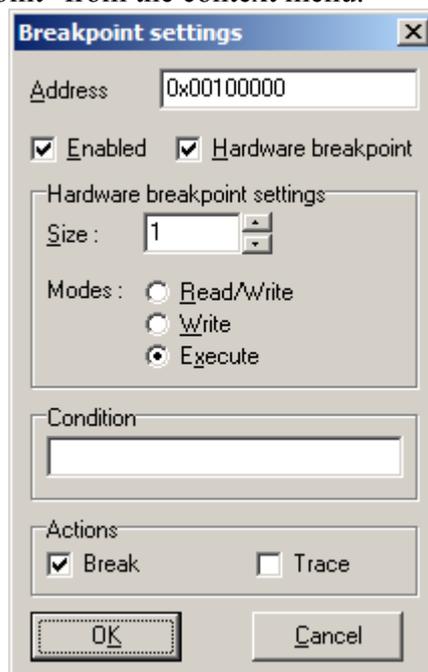
We now have two regions:



Double-click the LINUX region to go to its beginning.



Press F2 or choose "Add breakpoint" from the context menu.



Check "Hardware breakpoint" and select "Execute" in "Modes". Click OK.

```

IDA View-EIP
LINUX:00100000 ; Segment permissions: Read/Execute
LINUX:00100000 LINUX segment byte public 'CODE' use3
LINUX:00100000 assume cs:LINUX
LINUX:00100000 ;org 100000h
LINUX:00100000 assume es:LINUX, ss:LINUX, ds:LINUX,
LINUX:00100000 db 0
LINUX:00100001 db 0
LINUX:00100002 db 0
LINUX:00100003 db 0
LINUX:00100004 db 0
LINUX:00100005 db 0
LINUX:00100006 db 0
LINUX:00100007 db 0
LINUX:00100008 db 0
LINUX:00100009 db 0
LINUX:0010000A db 0
LINUX:0010000B db 0
LINUX:0010000C db 0
UNKNOWN 00100000: LINUX:00100000

```

Now press F9 or choose Debugger | Continue process. You should see BIOS and LILO messages on the screen, and the execution will stop at the "Loading Linux...." message.

```

IDA View-EIP
LINUX:00100000
LINUX:00100000 loc_100000: ; DAT
EIP LINUX:00100000 cld
LINUX:00100001 cli
LINUX:00100002 mov     eax, 18h
LINUX:00100007 mov     ds, ax
LINUX:00100009 assume ds:nothing
LINUX:00100009 mov     es, ax
LINUX:0010000B assume es:nothing
LINUX:0010000B mov     fs, ax
LINUX:0010000D assume fs:nothing
LINUX:0010000D mov     gs, ax
LINUX:0010000F assume gs:nothing
LINUX:0010000F lss     esp, ds:103020h
LINUX:00100016 xor     eax, eax
LINUX:00100018
LINUX:00100018 loc_100018: ; COD
LINUX:00100018 inc     eax
LINUX:00100019 mov     large ds:0, eax
LINUX:0010001E cmp     ds:100000h, eax
LINUX:00100024 jz     short loc_100018
LINUX:00100026 push   0
LINUX:00100028 popf
LINUX:00100029 xor     eax, eax
LINUX:0010002B mov     edi, offset unk_1B69CF
UNKNOWN 00100007: LINUX:00100007

```

This is the initial loader which decompresses the kernel. If you press F9 once more, you'll see "Uncompressing Linux..." and then "Ok, booting the kernel" messages in QEMU and IDA will stop at the decompressed kernel entrypoint:

```

IDA View-EIP
EIP
LINUX:00100000 assume es:LINUX, ss:LINUX, ds:LINUX, fs:LINUX
LINUX:00100000 cld
LINUX:00100001 mov     eax, 18h
LINUX:00100006 mov     ds, ax
LINUX:00100008 assume ds:nothing
LINUX:00100008 mov     es, ax
LINUX:0010000A assume es:nothing
LINUX:0010000A mov     fs, ax
LINUX:0010000C assume fs:nothing
LINUX:0010000C mov     gs, ax
LINUX:0010000E assume gs:nothing
LINUX:0010000E mov     edi, offset unk_102000
LINUX:00100013 mov     eax, 7
LINUX:00100018
LINUX:00100018 loc_100018:                                ; COD
LINUX:00100018 stosd
LINUX:00100019 add     eax, 1000h
LINUX:0010001E cmp     edi, offset unk_104000
LINUX:00100024 jnz     short loc_100018
LINUX:00100026 mov     eax, offset off_101000
LINUX:0010002B mov     cr3, eax
LINUX:0010002E mov     eax, cr0
LINUX:00100031 or     eax, offset unk_80000000
LINUX:00100036 mov     cr0, eax
LINUX:00100039 jmp     short $+2
LINUX:0010003B mov     eax, offset unk_90100042
LINUX:00100040 jmp     eax
LINUX:00100040 ; -----
LINUX:00100042 db     0Fh

```

This code sets up paging table, enables paging, and then jumps to the "real" kernel entrypoint.

## Adding symbols

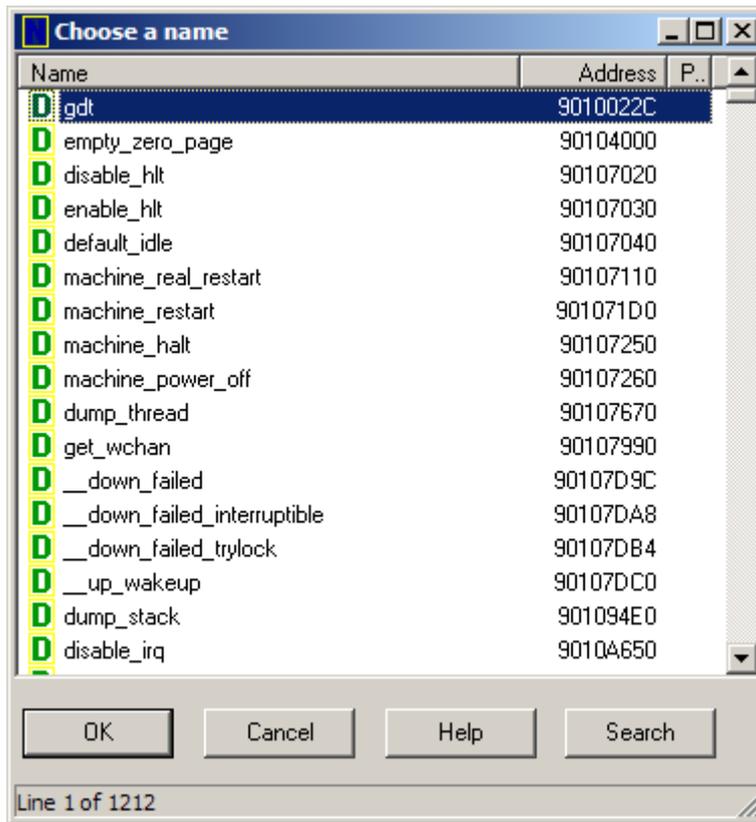
Kernel symbols are available as `/proc/ksyms` or `/proc/kallsyms` pseudo-file after booting. If you get that file from the VM to the host, you can add the symbols to your disassembly. Go to File | Python command... and enter the following short script:

```

ksyms = open(r"D:\ksyms") #path to the ksyms file
for line in ksyms:
    addr = int(line[:8], 16)
    name = line[9:-1] # use line[11:-1] in case of kallsyms
    idaapi.set_debug_name(addr, name)
    MakeNameEx(addr, name, SN_NOWARN)
    Message("%08X: %s\n"%(addr, name))

```

Click OK and wait a bit until it finishes. After that you should see the symbols in the disassembly and name list:



Happy debugging!