# Debugging code snippets in IDA Pro 5.6 using QEMU emulator



## Introduction

IDA Pro 5.6 has a new feature: automatic running of the QEMU emulator. It can be used to debug small code snippets directly from the database. In this tutorial we will show how to dynamically run code that can be difficult to analyze statically.

## Target

Many firmwares for ARM processors that are built with ARM's RealView Compiler (RVCT), utilize so-called "scatter-loading" to save flash space. The linker concatenates all code and initialized data regions together, and then adds a small stub which copies the data where necessary at the very start of execution. This means that if you only have a flash ROM dump, you will need to figure out where the data is copied to continue analysis. The situation can be even more complicated if the linker used the optional compression. Reconstructing decompression algorithm from dead listing can be quite a time-consuming process. That's why QEMU emulation can be very helpful in such cases.

Here's how a typical scatter-loading stub looks like:

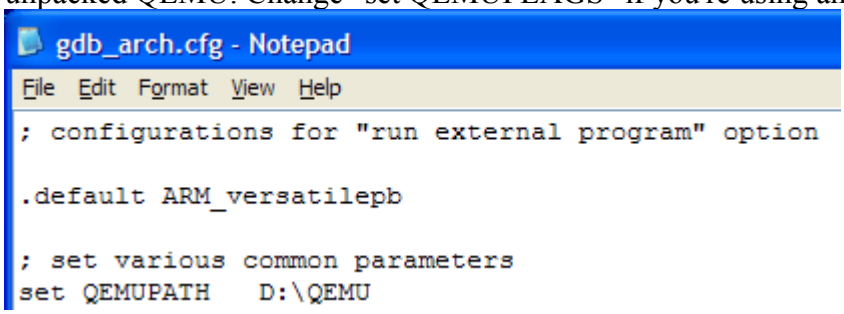
```
sub_86210                               ; CODE XREF: ROM:00086044↑j
                B       loc_86218
; ---------------------------------------------------------------------------
                B       sub_8DF8C
; ---------------------------------------------------------------------------

loc_86218                               ; CODE XREF: sub_86210↑j
                ADR     R0, _pointers   ; take address of table pointers
                LDMIA   R0, {R10,R11}   ; load relative start (R10) and end (R11)
                ADD     R10, R10, R0    ; calculate absolute start pointer
                SUB     R7, R10, #1
                ADD     R11, R11, R0    ; calculate absolute end pointer

loc_8622C                               ; DATA XREF: sub_86210+28↓o
                CMP     R10, R11        ; is is the end of the table?
                BEQ     sub_8DF8C       ; if yes, jump to main entrypoint
                LDMIA   R10!, {R0-R3}   ; load four registers
                ADR     LR, loc_8622C   ; set return address
                TST     R3, #1          ; is low bit set?
                SUBNE   PC, R7, R3      ; if yes, use relative address
                MOV     PC, R3          ; otherwise it's ablsolute
; End of function sub_86210

; ---------------------------------------------------------------------------
_pointers       DCD region_table - 0x86248 ; DATA XREF: sub_86210:loc_86218↑o
                DCD region_table_end - 0x86248
```

As you can see, it loads four registers (R0 to R3) at at a time from the table and then calls a function at the address in R3. We can analyze what the called functions do from dead listing, but just running the code is much easier.

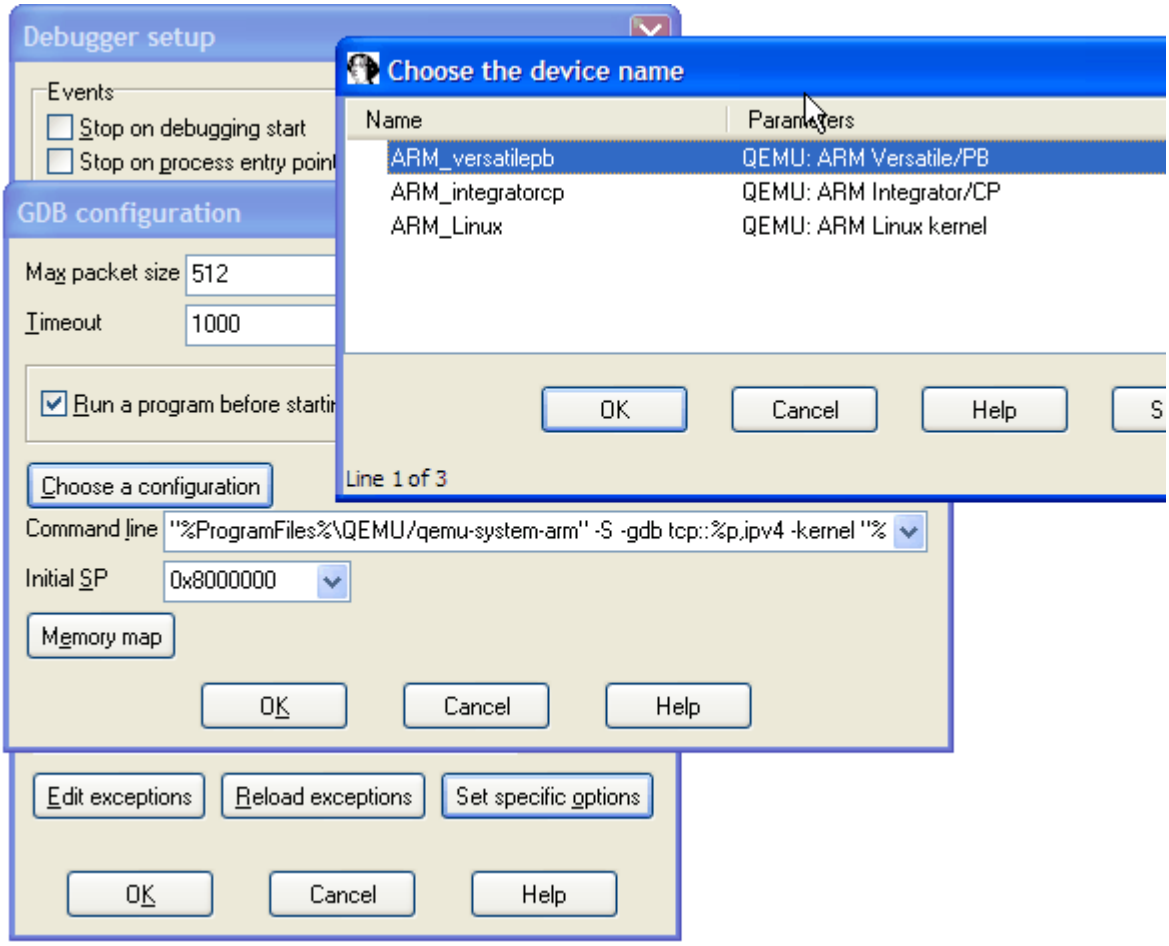
# Configuring QEMU

First we need to set up automatic running of QEMU.

1. Download a recent version of QEMU with ARM support (e.g. from http://homepage3.nifty.com/takeda-toshiya/qemu/index.html). If qemu-system-arm.exe is in a subdirectory, move it next to qemu.exe and all DLLs.
   **Note**: if you're running Windows 7 or Vista, it's recommended to use QEMU 0.11 or 0.10.50 ("Snapshot" on Takeda Toshiya's page), as the older versions listen for GDB connections only over IPv6 and IDA can't connect to it.
2. Edit cfg/gdb_arch.cfg and change "set QEMUPATH" line to point to the directory where you unpacked QEMU. Change "set QEMUFLAGS" if you're using an older version.




3. In IDA, go to Debug-Debugger options..., Set specific options.
4. Enable "Run a program before starting debugging".
5. Click "Choose a configuration". Choose Versatile or Integrator board. The command line and Initial SP fields will be filled in.




6. Memory map will be filled from the config file too. You can edit it by clicking the "Memory map" button, or from the Debugger-Manual memory regions menu item. See below for more details

Now on every start of debugging session QEMU will be started automatically.

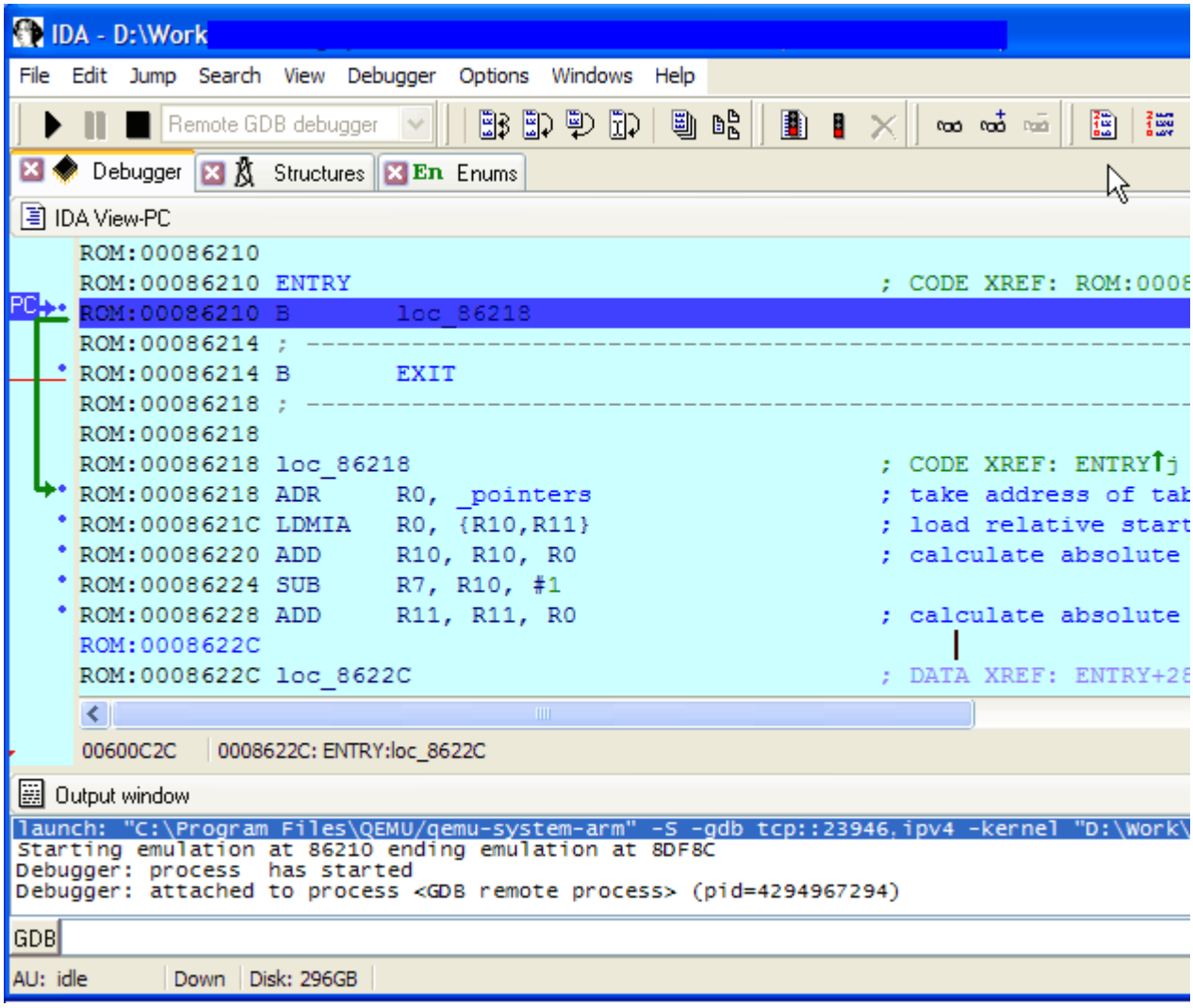
# Executing the code

By default, initial execution point is the entry point of the database. In our case we want to execute only the function we're interested in. There are two ways of doing it:

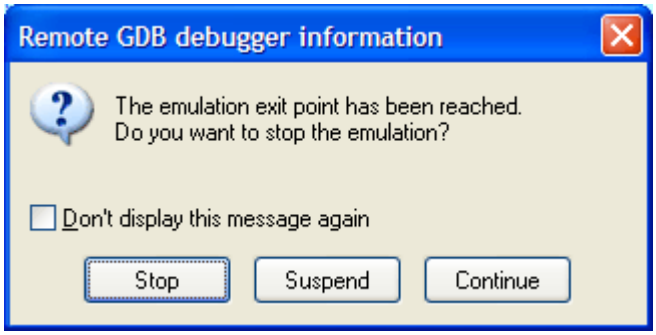
1. Select the code range that you want to execute, or
2. Rename starting point **ENTRY** and ending point **EXIT** (convention similar to Bochs debugger)

We'll use the second option and rename sub_86210 to ENTRY and sub_8DF8C to EXIT.

If you press F9 now, IDA will write the database contents to an ELF file (**database.elfimg**) and start QEMU, passing the ELF file name as the "kernel" parameter. QEMU will load it, and stop at the initial point.



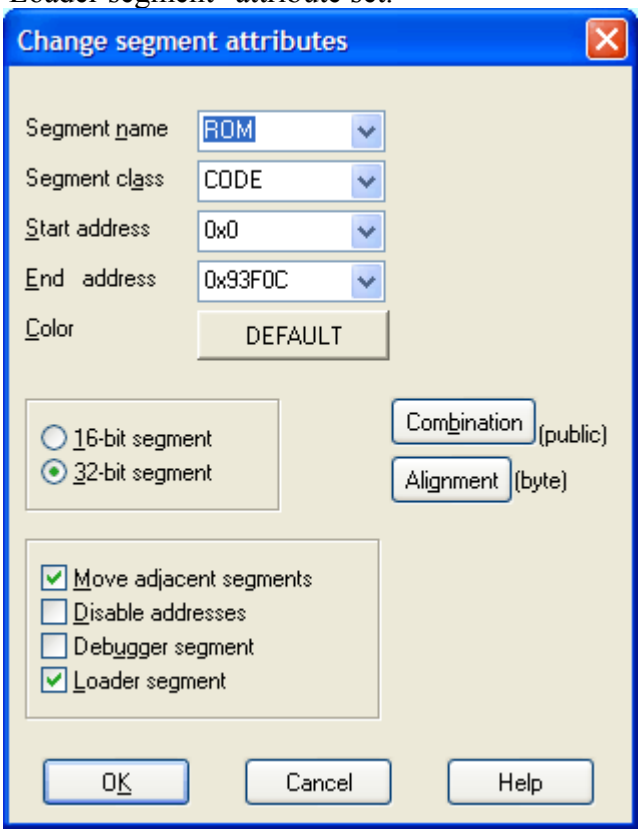

Now you can step through the code and inspect what it does. Or just press F9 again and let it run until it reaches the EXIT address. In that case, IDA will display this prompt:



# Saving results to database

If you want to keep the moved/unpacked data for later analysis, you'll need to copy it to the database. For that:

1. answer "Suspend".
2. edit segment attributes (Alt-S) and make sure that segments with the data you need have the "Loader segment" attribute set.




3. choose Debugger-Take memory snapshot and answer "Loader segments".
   **Note**: if you answer "All segments", IDA will try to read the whole RAM segment (usually 128M) which can take a VERY long time.
4. Now you can stop the debugging and inspect the new data.
   **Note**: this will update your database with the new data and discard the old. It will not be possible to execute the scatter-loading code again.

# Memory map

The memory map in gdb_arch.cfg (also accessible via Debugger-Manual memory regions after choosing a configuration) is **only descriptive**. Changing it will not affect the actual memory configuation of the selected board in QEMU. IDA uses this map to check the database segmentation for conflicts and warn the user.

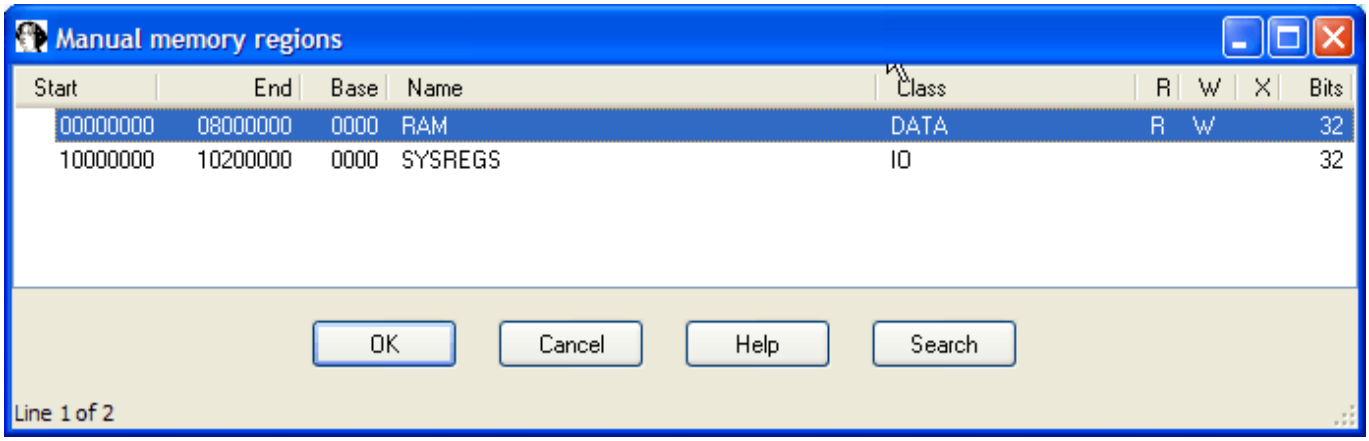
Memory description lines in the config file have the following format:

```
area <class> <name> <start>:<end>
```

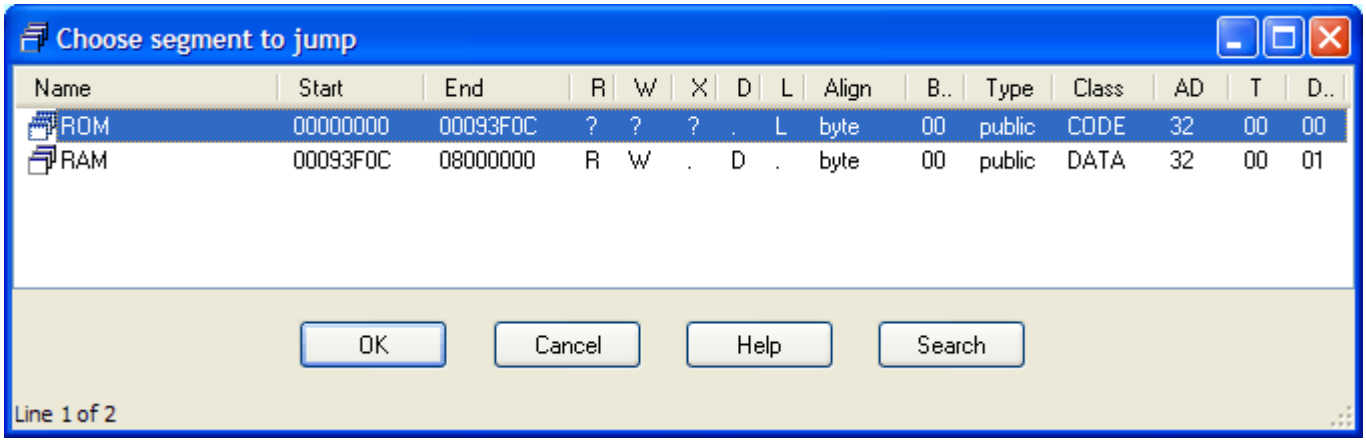
The following segment classes are interpreted by IDA:

- **DATA, RODATA, CODE, ROCODE**: writeable or read-only data or code section.
- **IO**: hardware I/O area. Must not contain any code or data to prevent conflicts with QEMU. If such an area does have code or data in the database, emulation will likely not work properly and IDA will warn the user about that.

Here's the map for the Versatile/PB board.



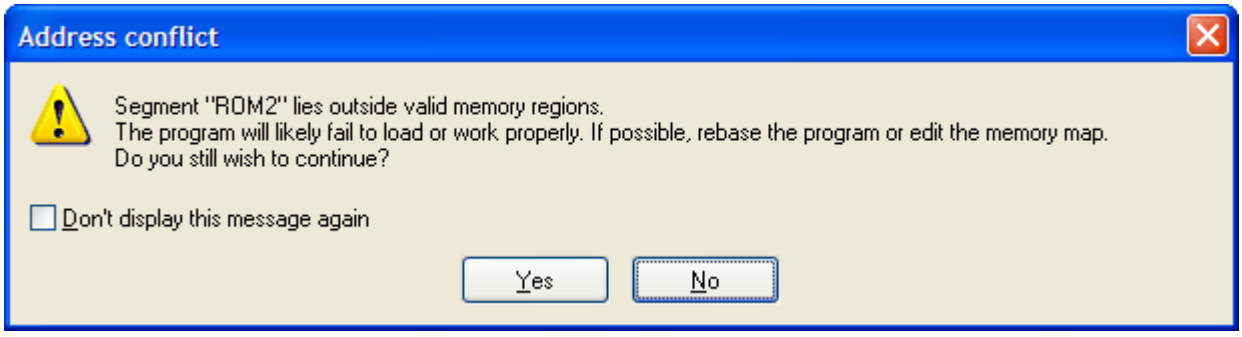

As you can see, there is 128M of RAM at the bottom of the address space and an I/O area. The I/O area doesn't have any access flags set, so IDA will remove it from the segment list when starting the debug session. That will prevent auto-analyzer from trying to read the data in that segment, which could have undesirable side effects. After starting the debugging, the manual memory region list gets merged with the database segments:




"ROM" is the database segment, and "RAM" is added from the memory region list.

# Conflicts

If your database has some segments which lie outiside declared memory map or intersect a reserved (I/O) area, IDA will display the following warning:




You have several options in this case.

1. Ignore and continue, e.g. if the affected segments are not important for the snippet you want to run. Any data which lies outside valid regions will be discared by QEMU.

2. If the code is relocatable, rebase the program so that all segments lie in the valid regions.
3. If the memory map does not reflect the actual board configuration, edit it.
4. Edit the QEMU sources to move RAM and/or I/O addresses so that they don't conflict with the database, compile it, and edit the memory map to reflect the new addresses.

# Technical notes

1. The following special symbols can be used in the command line:

   **%i**: input file name (from Process options dialog). Can be used if QEMU can load the input file directly, e.g. it's a Linux kernel. A few sample configurations in gdb_arch.cfg are provided.
   **%e**: a temporary ELF file created by IDA from the database.
   **%p**: port number from the Process options dialog

2. QEMU emulation is "bare metal", without any OS support. Even MMU is not turned on, so if the code you're debugging relies on that, it might not work.
3. If you need to debug a program inside an OS, it's best to run a full OS inside QEMU, load your program inside it and then attach to it over the GDB connection. You can still use the "Run a program before debugging" option to start QEMU automatically, just don't use the %e option. See the sample "Linux kernel" configurations.
4. The program to be run does not have to be QEMU. The feature can be used, for example, to start a JTAG GDB server such as OpenOCD.

Happy debugging! Please send any comments or questions to support@hex-rays.com