DEVMEM – Device Memory Read Write – Opensource Project HOWTO
Author: Kaiwan N Billimoria

URL: https://github.com/kaiwan/device-memory-readwrite

Last Updated: 12 Aug 2016.

Introduction

The purpose of the project is to enable developers/testers/anybody (with sufficient know-how), to peek and poke memory locations on a device, or even a PC!

The memory locations in question could be in: * user-space * kernel-space * pure kernel * device driver * hardware memory (memory-mapped into the kernel virtual address space)

Caveats / Important to Understand

1. The Linux OS (any modern VM-based OS), will check for and disallow invalid memory region accesses (reads or writes). This "memory protection" is an important feature of a modern VM-based OS.

This kernel module/utilities let you as the end-use specify any address to be read/written; please do realize, though, that attempting to read/write an invalid (or unmapped) memory location(s) would cause the kernel to **fault**, and thus, the process context (which in this case will be the utility process) to be killed.

In other words, please specify only memory locations that are known to you to be mapped and valid, not just any random address :)

2. For the same reason as point 1 above, if you specify a **user-space** address, you will typically be specifying a virtual address (VA). Every Linux process has it's own page maps, which really means that VA 'x' for process A is probably very different from VA 'x' for process B! By definition, therefore, the VA's you're trying to see (or write to) are the VA's of the current process context, i.e., the utility programs, rdmem and wrmem only...

By contrast, kernel virtual addresses are the same for all processes, so looking up/modifying a valid kernel VA from these utils is fine!

3. In a similar vein, especially when working with (memory-mapped) hardware registers, the rdmem/wrmem utils will do their job, but that does not mean all's okay! Accessing certain registers **might require a certain protocol** to be followed; this is often the case on sophisticated hardware. For example, I tried accessing some arbitrary "face-recognition-mode" registers on the Pandaboard; it "works" but causes a crash in the kernel. This is as am not accessing them as they are supposed to be accessed.. (So you ask: **how exactly** are they supposed to be accessed?? Well, it's completely board/platform-dependant; you will really have to dig into the TRM's (Tech Ref Manuals) for that board. As examples, one can find the TRM etc for the Pandaboard here:http://pandaboard.org/content/resources/omap-device .)

4. As of now, this project supports and has been tested on:

    - Intel x86 32-bit Linux PC, running Ubuntu 10.04, plus VmWare guest Ubuntu 10.10
    - Intel x86_64 (Core i7) 64-bit laptop, running Ubuntu 15.04 [earlier 11.04]
    - ARM Cortex0-A8 (32-bit) on the Beagle-Board, running generic Linux kernel ver 2.6.33.x
    - ARM Cortex-A9 (dual-core 32-bit) on the TI Pandaboard, running generic Linux kernel ver 3.1.5 .

(No reason it shouldn't work on other 32-bit ARM/MIPS/etc platforms. If you try it, do drop me a line).

[There currently seems to be an issue with the underlying ioctl() in the driver on 64-bit Linux; I'll try & sort it out. Update: Realized that it's – probably - not a 64-bit issue at all, it's a kernel version issue; the ioctl() from 2.6.36 is different; working on fixing it.

Yup, it's now fixed; please grab latest rel (ver 0.1.3 as of this writing).]

Details

The project consists of two main parts: * the user-space read and write utility programs (xrdmem, xwrmem) * the kernel-space device driver (rwmem)

The kernel device driver lets us specify (optional) an IO region to map into the kernel virtual address space, by specifying the physical address and length as module parameters (module params are always optional).

```
$ modinfo ./rwmem.ko
filename: rwmem.ko
license: GPL
description: Char driver to implement read/write memory support.
author: (c) Kaiwan NB
srcversion: ADC0280A4B68D115C9FA691
depends:
vermagic: 2.6.38-10-generic SMP mod_unload modversions
parm: iobase_start:Start (physical) address of IO base memory (typically h/w registers mapped here by the processor) (ulong)
parm: iobase_len:Length (in bytes) of IO base memory (typically h/w registers mapped here by the processor) (uint)
parm: force_rel:Set to 1 to Force releasing the IO base memory region, even if (esp if) already mapped. Could be dangerous! (uint)
parm: reg_name:Set to a string describing the IO base memory region being mapped by this driver. (charp)
$
```

Note that the above output was for the driver built for the x86 PC.

The output of /proc/iomem shows reserved IO memory region areas (the addresses are physical/bus addresses).

If you want to release an existing IO region (one that's already taken up by another driver (or the kernel)), use

the 'force_rel=1' module parameter above. Note that you'll have to specify the exact start address and length in order for it to work. **Also realize that doing this could adversely affect the system!**

Usage: Read Utiltiy

$ ./rdmem

```
Usage: ./x86_rdmem [-o] <address/offset> [len]
[-o]: optional parameter:
 : '-o' present implies the next parameter is an OFFSET and NOT an absolute
address [HEX]
 (this is the typical usage for looking at hardware registers that are offset
from an IO base..)
 : absence of '-o' implies that the next parameter is an ADDRESS [HEX]
offset -or- address : required parameter:
 start offset or address to read memory from (HEX).

len: optional parameter:
 length : number of items to read. Default = 4 bytes (HEX)
 Restrictions: length must be in the range [4-131072] and
 a power of 2 (if not, it will be auto rounded-up to the next ^2).
$
```

Firstly, notice that both the rdmem & wrmem utilities take an optional '-o' , for 'offset', parameter.

a) If NOT used, it implies that the address being passed is absolute (a virtual address, actually). Examples 1 and 2 below demonstrates this feature.

b) If '-o num' is specified, it implies that the number passed (num) is not an absolute offset but rather an **offset** from the IO base memory address passed to the device driver 'rwmem' as a module parameter (which itself is optional).

The driver will remap this IO base location to a kernel virtual address. So, if you want to read/write offsets from a physical HW location, you are expected to pass the IO base location (typically a physical address) and then use an offset when using the rdmem/wrmem utilities. Example 3 below demonstrates this feature.

Usage Examples

In order to test, a simple kernel module (i call it vm_img) was used to poke (kernel) memory with a pattern.

Output from vm_img.ko :

```
--snip--
[ 1083.335722] vm_img_init:58 : 32-bit architecture:
--snip--
[ 1083.335789] PAGE_OFFSET = 0xc0000000
[ 1083.335790] VMALLOC_START = 0xe0800000
VMALLOC_END=0xff7fe000
--snip--
[ 1083.335801] eg. kernel vaddr: &statgul = 0xe084c8e0, &jiffies_64 = 0xc07c7a40, &loc =
0xd7697f4c
[ 1083.335804] kptr = 0xc0e0c000 vptr = 0xe0acb000
--snip--
```

Now, the location 'kptr' has some RAM (allocated with kmalloc) poked with the pattern '0xdeadface....', and the location 'vptr' (allocated with vmalloc) has RAM poked with the pattern '0xbadabada....'.

Eg. 1: Simple read/write of memory (RAM) on an x86 PC-based system

```
$ ./rdmem 0xc0e0c000
```

device file open failed. Driver 'rwmem' not loaded?: No such file or directory

```
# insmod ../drv_rwmem/rwmem.ko   << Notice no phy addr passed; thus we don't do
                                    any ioremap() internally .. >>
# ./rdmem 0xc0e0c000
      +0          +4          +8          +c            0   4   8   c
+0000 de           ad          fa          ce            .   .   .   .

#
```

Yes, it shows up.

Now lets view 32 bytes of kernel RAM from 'kptr': ```

```
$ ./rdmem 0xc0e0c000 20
     +0          +4          +8          +c            0   4   8   c
+0000 de ad fa ce de ad fa ce de ad fa ce de ad fa ce ................
+0010 de ad fa ce de ad fa ce de ad fa ce de ad fa ce ................
```

(Notice how the 2nd parameter value is '20' - it's always hex (0x20 = 32))

Now lets view 32 bytes of kernel RAM from 'vptr': ```

```
$ ./rdmem 0xe0acb000 20
      +0          +4          +8          +c            0   4   8   c
+0000 ba da ba da ba da ba da ba da ba da ba da ba da ................
+0010 ba da ba da ba da ba da ba da ba da ba da ba da ................
```

(Notice how the 2nd parameter value is '20' - it's always hex (0x20 = 32))
Again, it's clear.

Now lets attempt to change (poke) some locations here: ```

```
$ ./wrmem
Usage: ./wrmem [-o] <address/offset> <value>
[-o]: optional parameter:
 : '-o' present implies the next parameter is an OFFSET and NOT an absolute address
[HEX]
 (this is the typical usage for writing to hardware registers that are offset from an
IO base..)
 : absence of '-o' implies that the next parameter is an ADDRESS [HEX]
offset -or- address : required parameter:
 memory offset or address to write to (HEX).
```

```
value: required parameter:
 data to write to above address/offset (4 bytes) (HEX).
$

$ ./wrmem 0xc0e0c008 aabbccdd
$ ./rdmem 0xc0e0c000 10
      +0          +4          +8          +c            0   4   8   c
+0000 de ad fa ce de ad fa ce aa bb cc dd de ad fa ce ................
```

Clearly, above, we can see that the pattern has changed to what was written at the appropriate location!

Eg. 2: Reading the value of 'jiffies' on an x86 PC

The 'vm_img' kernel module mentioned earlier, also shows us the location of the 'jiffies_64' kernel global variable (which holds the current jiffies value); in the above run, it's kernel location turns out to be 0xc07c7a40. So:

```
$ ./rdmem 0xc07c7a40 ; sleep 1; ./rdmem 0xc07c7a40 ; sleep 1; ./rdmem 0xc07c7a40
+0        +4        +8          +c            0   4   8   c
+00 00 00 1a bc 20 ...
+00 00 00 1a bd 23 ...
+00 00 00 1a be 1f ....
```

We can see how it's updated...(in fact, CONFIG_HZ=250 on this system, and some + factor is expected as well...).

Eg. 3: Reading a hardware register on the Beagle Board

Lets use this project to peek at the registers holding identity information on the http://www.beagleboard.org'>Beagle Board.

(Of course, for this purpose, the utils and driver were cross-compiled using an x86-to-ARM toolchain: Code Sourcery G++ Lite 2009q3-67).

I'm using the Rev C3 board; the relevant manual is http://www.ti.com/lit/pdf/spruf98 '>“OMAP35x Application Processor : Technical Reference Manual” Literature Number : SPRUF98T (Rev July 2011)..

(If you wish to follow this example(s) in complete detail, please download the manual and view the pages referred to here..)

Reading the DEVICE CONTROL PRODUCTION ID register (a 128-bit register):

From the Tech Ref manual, page 189, 190:

--snip--

Table 1-4. CONTROL_PRODUCTION_ID Address Offset Physical Address Type

Instance 0x0000 0000 0x4830 A210 R GENERAL

Description This register shows the device type and some options availability.
--snip--

```
root@beagleboard:/media/mmcblk0p2# insmod rwmem.ko iobase_start=0x4830a210 iobase_len=0x10
rwmem: registered with major number 253 rwmem: cdev rwmem.0 added
rwmem: Device node /dev/rwmem.0 created.
Little-endian.
root@beagleboard:/media/mmcblk0p2# ./xrdmem -o 0 10
      +0          +4          +8          +c          0   4   8   c
+0000 00 00 00 f0 ca fe b7 ae 0f 01 40 17 04 03 23 09 ..........@...#.
root@beagleboard:/media/mmcblk0p2#
```

All params are in HEX. Passing an offset of 0x0 (as seen in the manual), we read in 0x10 (=16 decimal =128 bits) bytes. Using the manual, the relevant bits can then be interpreted.

Similarly, by carefully looking them up in the manual, hardware registers/memory regions can be peeked and poked at!

End document.