



Intel® Simics® Simulator New User Training

Lab 006 – Advanced debugging

Lab 006 – Advanced debugging

This lab continues the debugging theme, showing the low-level and advanced debug capabilities of the Intel® Simics® Simulator. It uses the Linux device driver for the training device to demonstrate how to debug software that is close to the hardware.

A. Extract the Linux kernel

To debug the code of the Linux kernel, you need the kernel **vmlinux** file for the target system and make it available on your host where the debugger can find it. Sometimes, the **vmlinux** file is available outside of the target system as the result of a cross-compilation process (if you built your own Linux kernel and filesystem with Buildroot or Yocto, for example). However, in the case of the Intel Simics Quick Start Platform (QSP) with Clear Linux setup that is used for the training, the kernel must be extracted from the target system. The same applies to other packaged distributions like Ubuntu and Fedora.

For Clear Linux, the kernel debug information **vmlinux** file is not included in the standard installation. To get it, you need to install the **linux-dev** bundle, which takes up too much disk to be part of the standard disk images in the Intel Simics packages in its entirety. However, the disk image you are using includes the relevant part of the **linux-dev** bundle, so that you can extract the **vmlinux** file for debugging.

You will use the Intel Simics Agent system to copy the **vmlinux** file from the target system to the host.

1. Start a new simulation session using the first checkpoint from lab 001 (called **MyFirstCheckpoint.ckpt**).

```
[${C:>} simics[.bat] MyFirstCheckpoint.ckpt
```

2. Run the simulation forward:

```
simics> r
```

3. Go to the target **Serial Console** window, and check the location of the **vmlinux** file:

```
# cd /lib64/modules/6.6.21-yocto-standard/build
```

4. List the files:

```
# ls
```

The directory contains several files, among them the **vmlinux** file:

```
root@machine:/lib64/modules/6.6.21-yocto-standard/build# ls
Documentation          ipc
Kbuild                 kernel
Kconfig                lib
[...]
include                virt
init                   vmlinux
io_uring
```

5. Go to the command-line, and start the agent manager:

```
running> start-agent-manager
```

6. Go back to the target **Serial Console** window and use the **simics-agent** command *on the target system* to copy the **vmlinux** file to the host. From the perspective of the target, this is an “upload” operation since it moves things off of the system. Use <TAB> to have the target Linux auto-complete the name of the file.

```
# simics_agent_x86_linux64 --upload vmlinux
```

7. Wait for the operation to complete. Both the target system and the simulator command-line will indicate that the operation finished.

In the target serial console, it should look something like this:

```
root@machine:/lib64/modules/6.6.21-yocto-standard/build# simics_agent_x86_linux64 --  
upload vmlinux  
machine, v1.5, Feb 21 2023 14:13:26  
machine connected (1b90f02e886b3cbb)  
simics_agent_x86_linux64: Success (0)
```

On the simulator command-line, you will see a couple of printouts indicating that the operation finished:

```
[matic0 info] The Simics agent has terminated.  
[matic0 info] disconnected from machine1 (0x1b90f02e10d5a84c)
```

8. From the simulator command line, you can use **ls** to check the contents of the project directory on the host. You should see the **vmlinux** file there.

```
running> ls
```

9. **Quit** this simulation session.

```
running> quit
```

B. Debug driver operations

This part will show how to debug the operations of a Linux kernel driver, using the driver for the training card. The driver is a dynamically loaded kernel module, which makes debugging it a bit more complicated than if it had been built into the kernel as a static module.

Finding the load address

To debug the driver, it is necessary to figure out where it is loaded. The Linux Operating System Awareness (OSA) system does not provide this information (currently, it could in principle be developed), so you have to use a trick involving checkpoints. This trick can be applied to any software stack with similar behavior.

10. Start a new simulation session using the first checkpoint (**MyFirstCheckpoint**) from Lab 001 (the system is booted but the driver not yet loaded).

```
[${C:>}] simics[.bat] MyFirstCheckpoint.ckpt
```

11. Run the simulation forward.

```
simics> r
```

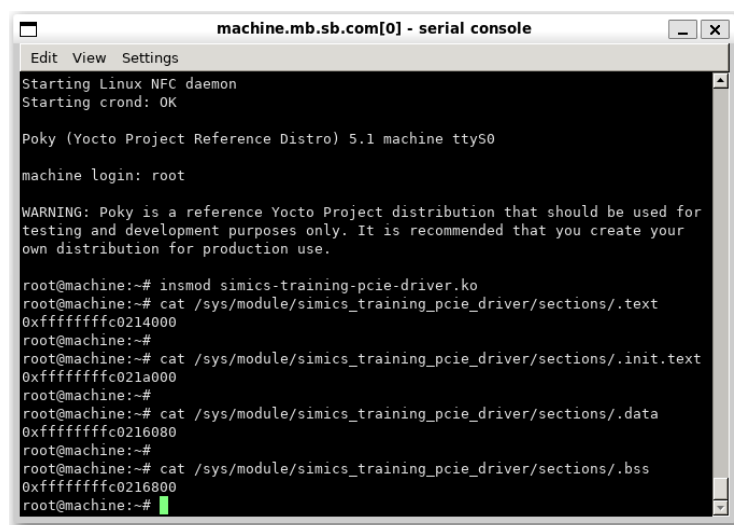
12. Go to the target **Serial Console** window, and enter the following command to load the driver:

```
# insmod simics-training-pcie-driver.ko
```

13. Next, find out where the code for the driver was loaded. This is done by reading from the **/sys** file system in Linux. There are quite a few sections that need to be found, due to how Linux kernel modules are compiled.

```
# cat /sys/module/simics_training_pcie_driver/sections/.text
# cat /sys/module/simics_training_pcie_driver/sections/.init.text
# cat /sys/module/simics_training_pcie_driver/sections/.data
# cat /sys/module/simics_training_pcie_driver/sections/.bss
```

14. This will show you one address for each section of the driver.



```
machine.mb.sb.com[0] - serial console
Edit View Settings
Starting Linux NFC daemon
Starting crond: OK

Poky (Yocto Project Reference Distro) 5.1 machine ttyS0
machine login: root

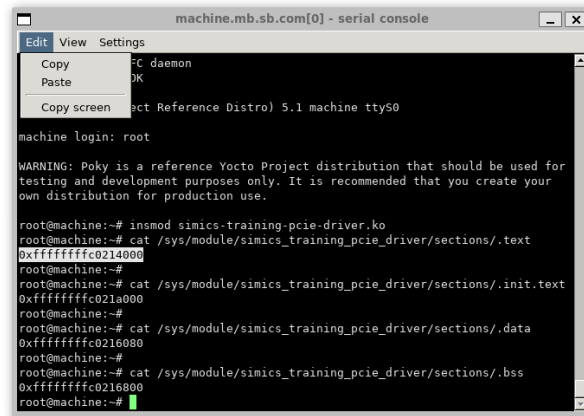
WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.

root@machine:~# insmod simics-training-pcie-driver.ko
root@machine:~# cat /sys/module/simics_training_pcie_driver/sections/.text
0xffffffffc0214000
root@machine:~#
root@machine:~# cat /sys/module/simics_training_pcie_driver/sections/.init.text
0xffffffffc021a000
root@machine:~#
root@machine:~# cat /sys/module/simics_training_pcie_driver/sections/.data
0xffffffffc0216080
root@machine:~#
root@machine:~# cat /sys/module/simics_training_pcie_driver/sections/.bss
0xffffffffc0216800
root@machine:~#
```

15. Go to the simulator command line and use the information collected above to configure the debugger using **add-symbol-file** with relocation information. To avoid repeating the path to the **simics-training-pcie-driver.ko** in each command, assign it to a CLI variable called **\$ko**:

```
running> $ko = (lookup-file "%simics%/targets/simics-user-  
training/target-code/simics-training-pcie-driver.ko")
```

16. Add a relocation for the **.text** section. Find the address on the target serial console, select it, and then copy it using the **Edit** menu:



17. Use the copied value when creating the **add-symbol-file** command

```
running> add-symbol-file $ko 0xffffffffc0072000_or_something ".text"
```

18. For the **.text.unlikely** section:

```
running> add-symbol-file $ko 0xffffffffc0072035_or_some  
".text.unlikely"
```

19. For the **.init.text** section:

```
running> add-symbol-file $ko 0xffffffffc0077000_or_some ".init.text"
```

20. For the **.data** section:

```
running> add-symbol-file $ko 0xffffffffc0074000_or_something ".data"
```

21. For the **.bss** section:

```
running> add-symbol-file $ko 0xffffffffc0074600_or_something ".bss"
```

Note: Performing this sequence of actions manually is cumbersome. It will be automated using simulator scripting in a later section of this lab. At this point, the point is to understand the relationship between the target state and debug setup.

22. Check the results:

```
running> show-memorymap
```

The result should show all sections you added above, available for all contexts. The filename in the text below is abbreviated for clarity; the details of your output might

vary with where the module got loaded, and the path will vary depending on the location of your project:

```
Contexts matching *
=====

ID          Address      Size  Flags      Offset File
1 0xffffffffc0214000 0x35   wx       .text /home/.../simics-training-pcie-driver.ko
3 0xffffffffc0216080 0x2b8   rw       .data /home/.../simics-training-pcie-driver.ko
4 0xffffffffc0216800 0x68    rw       .bss  /home/.../simics-training-pcie-driver.ko
2 0xffffffffc021a000 0x200   wx       .init.text /home/.../simics-training-pcie-driver.ko
```

Note that the debug information is attached to all debugger contexts – when debugging low-level code like in this lab, this method works fine. If you start to mix debugging code in different contexts, you will want to restrict the kernel symbols to only the kernel context.

23. Check that the symbols have been loaded to reasonable addresses. First, enable the debugger:

```
running> enable-debugger
```

24. Check the list of functions in the debug information:

```
running> sym-list -functions
```

This will show the list of functions defined in the kernel driver, with addresses that are close to the load locations provided above:

Symbol	Kind	Type	Address	Size
[...]				
chari_ioctl	function	long int (*)(struct file *, unsigned int, long unsigned int)	0xffff_ffff_c021_4100	33
chari_mmap	function	int (*)(struct file *, struct vm_area_struct *)	0xffff_ffff_c021_4880	118
chari_open	function	int (*)(struct inode *, struct file *)	0xffff_ffff_c021_40c0	40
chari_read	function	ssize_t (*)(struct file *, char *, size_t, loff_t *)	0xffff_ffff_c021_4590	398
chari_release	function	int (*)(struct inode *, struct file *)	0xffff_ffff_c021_4080	33
chari_write	function	ssize_t (*)(struct file *, const char *, size_t, loff_t *)	0xffff_ffff_c021_4730	318
init_module	function	int (*)()	0xffff_ffff_c021_a010	532
irq_handler_button	function	enum irqreturn_t (*)(int, void *)	0xffff_ffff_c021_41a0	58
simics_pcie_module_init	function	int (*)()	0xffff_ffff_c021_a010	532
simics_pcie_probe	function	int (*)(struct pci_dev *, const struct pci_device_id *)	0xffff_ffff_c021_41f0	897
simics_pcie_remove	function	void (*)(struct pci_dev *)	0xffff_ffff_c021_4140	79
vm_close	function	void (*)(struct vm_area_struct *)	0xffff_ffff_c021_4050	31
vm_open	function	void (*)(struct vm_area_struct *)	0xffff_ffff_c021_4010	48

25. Set up the source code path map. Since the paths used are rather long, you can break it up by using CLI variables with intermediate results.

First, the code build location (which you cannot know without looking into the debug information):

```
running> $s = "/home/root/simics-training-pcie-driver"
```

26. The local location for the source code:

```
running> $d = (lookup-file "%simics%/targets/simics-user-
training/target-source/simics-training-pcie-driver")
```

27. Clear the path map and add a mapping from \$s to \$d:

```
running> clear-pathmap
running> add-pathmap-entry $s $d
running> show-pathmap
```

Please note that this operation is rather finicky when mapping from Linux to Windows paths – do not include a final slash in the “source” path. The debugger might be utterly confused. Essentially, the path map operation matches the “source” string in the path to a file and replaces the matching segment with the “destination” string. Thus, even a single-character difference at the end of a string can affect what happens.

28. You should also add the debug information for the Linux kernel. The source code for the kernel is not available in the lab package, but at least you will see the function names in the stack trace. Use the kernel **vmlinux** image you retrieved in Section A.

```
running> add-symbol-file vmlinux
```

Note: this way of adding the debug information for the Linux kernel only works since kernel address-space layout randomization (KASLR) has been turned off for the Clear Linux training setup. In a standard deployment, the kernel would have been relocated, and you would have needed to figure out the relocation and add a corresponding offset to the mapping.

29. Check the memory map:

```
running> show-memorymap
```

You should see several new mapping entries corresponding to the debug information for the kernel.

With this, it is time to debug some kernel code!

Note that for drivers that are statically built into the Linux kernel, setting up debug is much easier. Just provide the debugger with the **vmlinux** file, and all static modules will resolve. The same holds for typical embedded operating systems used in firmware and classic real-time operating systems: a firmware image with debug information is sufficient since no dynamic loading and relocation is being used.

Debug the driver char write function

To debug a write operation to the driver, set a breakpoint on the write function registered with the character interface in the device driver.

30. Enable the command-line based debugger – this should already have been done above, to inspect the symbols, but just in case.

```
running> enable-debugger
```

31. Set up hexadecimal output:

```
running> output-radix 16 4
```

32. Find the address of **chari_write()**, the write function:

```
running> sym-address chari_write
```

33. Set a breakpoint on the entry point of **chari_write()**, using the address returned:


```
running> bp.memory.break (sym-address chari_write)
```

This was an exercise in how to set arbitrary breakpoints on memory; this particular breakpoint on a function can just as well be set using source breakpoints:

```
running> bp.source_location.break chari_write
```

34. Check the breakpoints set (from the command line):

```
running> bp.list
```

There should be a breakpoint set, something like:

```
running> bp.list
```

ID	Description	Enabled	Oneshot	Ignore count	Hit count
0x0001	machine.cell_context break matching (addr=0xfffffffffc0194606, len=0x0001, access=x)	true	false	0x0000	

The address of the function might vary.

35. Go to the target **Serial Console** window and send some characters (16 to be precise, alternating **ff** and **00**) to the device. For example:

```
# echo ff00ff0000ff00ff > /dev/simics_training_pcie_driver
```

36. The simulation should stop on the execution breakpoint.

If the simulation does not stop, the most likely explanation is that you got the wrong address in the **add-symbol-file** command for the **".text"** section. Double-check that the value is correct. If wrong, do **remove-symbol-file** using the ID for the mapping returned by **show-memorymap**. Then, redo the **add-symbol-file** command for **".text"**.

37. Find out which processor caught the breakpoint:

```
simics> pselect
```

38. Check the stack trace:

```
simics> stack-trace
```

It should show that the code is stopped in the function **chari_write()**. Something like this:

```
#0 0xfffffffffc0214730 in chari_write(file=(struct file *) 0xfffff88810b167100, buffer=(const char *) 0x55b8189e0ab0 "ff00ff0000ff00ff\n...", len=0x0011, offset=(loff_t *) 0xfffff900007cfe0) at /home/root/simics-training-pcie-driver/simics-training-pcie-driver.c:323
#1 0xfffffffff8132ac51 in vfs_write()
#2 0xfffffffff8132c8c5 in ksys_write()
#3 0xfffffffff8132c96d in __x64_sys_write()
#4 0xfffffffff81fad127 in do_syscall_64()
#5 0xfffffffff820012a in entry_SYSCALL_64_after_hwframe()
```

39. Use the **list** command to see the current location in the code:

```
simics> list
```

Investigate the state

40. Check the contents of the **buffer** variable. It points at the string you sent to the driver from the command-line.

```
simics> sym-value buffer
```

It should print out the value you sent in from the target Linux command-line:

```
(const char *) 0x984e80 "ff00ff0000ff00ff\n..."
```

It is worth noting that the string is *not necessarily* zero terminated. To know the size, look at the **len** parameter. Depending on the current system state when you sent the string to the driver, there might be garbage characters after the end of the input (the last character of the input string is **\n**) – or there might be zeroes, that makes it look like the string is properly terminated when inspected.

41. Check the value of the **len** variable:

```
simics> sym-value len
```

It should be 17 (16 characters plus the newline), or 0x11:

```
simics> sym-value buffer
(const char *) 0x82e890 "ff00ff0000ff00ff\n..."
simics> sym-value len
0x0011
```

42. Use **examine-memory** (another name for **x**) to read the memory contents. Read one extra byte to check what the memory looks like beyond the end of the string:

```
simics> examine-memory (sym-value buffer) 18
```

Hardware access breakpoints

Use breakpoints to observe how the driver interfaces with the hardware.

43. The driver will write to the framebuffer in the local memory of the training device. To break when such writes happen, put a breakpoint on the entire memory of the training device.

Investigate the memory map of the training device subsystem:

```
simics> machine.training_card.local_memory.map
```

The result should look like this:

Base	Object	Fn	Offset	Length	Target	Prio	Align	Swap
0x0000	machine.training_card.ram		0x0000	0x0010_0000		0		
0x1000_0000	machine.training_card.master_bb.bank.regs		0x0000	0x1000		0	8	

44. The local ram is mapped from address zero, for one megabyte. Set a breakpoint on reads and writes on this memory range, using the **bp.memory.break** command on the memory space:

```
simics> bp.memory.break object = machine.training_card.local_memory  
0x0 0x10_0000 -r -w
```

45. Run the simulation forward:

```
simics> r
```

46. The simulation will stop when the code writes the memory. The simulator will show the stop location on the command line:

```
[machine.training_card.local_memory] Breakpoint 3: machine.training_card.local_memory 'w' access  
to p:0x1000 val=0xff  
chari_write(filep, buffer=(const char *) 0x984e80 "ff00ff0000ff00ff\n", len=0x0011, offset) at  
/root/swbuild/simics-training-pcie-driver/simics-training-pcie-driver.c:353  
353          i+=2;
```

Note that due to compiler optimizations, the debugger stops on what looks like the wrong line.

47. Check the current stack trace from the command line:

```
simics> stack-trace
```

48. Look at the current location from the command line, showing a bit more context that is default by providing a **maxlines** argument:

```
simics> list maxlines = 15
```

This shows we are in the code writing to memory.

Watch the target memory change

The training device contains a local memory separate from the system's main RAM. This memory is used to hold the current image displayed, and this is the memory that the device driver is currently writing to.

49. The memory breakpoint message told you that the simulation stopped with an access to offset **0x1000** in the local memory map.
50. The address as seen from the driver code is held in the variable **destptr**. Check out the value:

```
simics> sym-value destptr
```

Note that the value is the address as seen from the processor, which is a virtual address:

```
(volatile uint8_t *) 0xfffffc90003001000
```

51. Convert the virtual address to the physical address that the processor is accessing – which is not the address inside the local memory map:

```
simics> l2p (sym-value destptr)
```

This should provide a value like **0xf100_1000**, indicating the place in the physical memory map as seen from the processor where the local RAM of the **training_card** is mapped.

52. Check the value that was written, by using the x command to look at the memory contents at offset 0x1000 in the local memory:

```
simics> x machine.training_card.local_memory 0x1000
```

53. To see the progress of writing memory from the code, ignore the breakpoint for next five hits. This is done using the **bp.ignore-count** command. First, find the ID of the memory breakpoint:

```
simics> bp.list
```

Pick up the ID from the breakpoint set on the local memory:

ID	Description	Enabled	Oneshot	Ignore count	Hit count
...					
<ID>	machine.training_card.local_memory break matching (addr=0x0, len=0x0010_0000, access=rw)	true	false	0x0000	0x0001

54. Tell the simulator to ignore the next five hits of the breakpoint, using the ID from above:

```
simics> bp.ignore-count ID 5
```

55. Run the simulation forward.

```
simics> r
```

56. When the simulation stops, check the “Hit count” on the breakpoint.

```
simics> bp.list
```

It should have increased to seven (five ignored hits plus the first hit, plus the current hit).

57. Check the contents of the memory being written:

```
simics> x machine.training_card.local_memory 0x1000
```

Note that several additional bytes of the pattern have been written to memory:

```
simics> x machine.training_card.local_memory 0x1000
p:0x00001000  ff00 ff00 00ff 0000 0000 0000 0000 0000 .....
```

58. Once the driver has set the bytes in memory it will trigger a display update in the hardware device. Put a breakpoint on hardware register access to stop when this happens.

```
simics> bp.bank.break -w register =
machine.training_card.master_bb.bank.regs.update_display_request
```

59. To focus on the accesses to the hardware registers, disable the breakpoint on accesses to local memory. The ID of the breakpoint was already retrieved previously.

```
simics> bp.delete ID
```

60. Check that the memory breakpoint is removed, and that the code breakpoint and new bank breakpoint are present:

```
simics> bp.list
```

61. Run forward.

```
simics> r
```

62. The simulation will stop outside of the driver, inside the Linux kernel `iowrite32()` function, for which you do not have source code information available (only symbols):

```
[machine.training_card.master_bb.bank.regs] Breakpoint 3:
machine.training_card.master_bb.bank.regs write at offset=0x10 size=0x4 value=0x1
ini=machine.mb.cpu0.core[0][0]
iowrite32()
0xffffffff817484ff (iowrite32 + 0xf)          ret
```

63. Check the stack trace:

```
simics> stack-trace
```

The result should be similar to this:

```
#0 0xffffffff817484ff in iowrite32()
#1 0xffffffffc021483a in chari_write(file, buffer, len, offset) at /home/root/simics-training-
pcie-driver/simics-training-pcie-driver.c:357
#2 0xffffffff8132ac51 in vfs_write()
#3 0xffffffff8132c8c5 in ksys_write()
#4 0xffffffff8132c96d in __x64_sys_write()
#5 0xffffffff81fad127 in do_syscall_64()
#6 0xffffffff8200012a in entry_SYSCALL_64_after_hwframe()
```

64. Step through the `ret` instruction and back into the driver:

```
simics> si
```

The simulator command line should show something like this once you get back into the driver:

```
simics> si
[machine.mb.cpu0.core[0][0]] cs:0xffffffff817484ff p:0x0009484ff ret
chari_write(file, buffer=(const char *) 0x55b8189e0ab0 "ff00ff0000ff00ff\n", len=0x0011,
offset) at /home/root/simics-training-pcie-driver/simics-training-pcie-driver.c:360
360          return len;
```

Trace device DMA

After the driver writes the command register in the device, the device will take over and read pixel values from memory and update the LEDs (via the I2C bus). In lab 004 you traced memory accesses from the device driver as well as the I2C bus activities. In this lab, you will break on DMA accesses from the device itself.

65. Create a new memory breakpoint, focusing on the buffer that the software has written. The buffer starts at 0x1000, and the software wrote 16 bytes.

```
simics> bp.memory.break object = machine.training_card.local_memory  
0x1000 16 -r -w
```

66. Check that the breakpoint has been set, and note its breakpoint ID for later removal:

```
simics> bp.list
```

67. Run the simulation forward.

```
simics> r
```

The simulator should stop with a message about an access device:

```
[machine.training_card.local_memory] Breakpoint 6: machine.training_card.local_memory 'r' access  
to p:0x1000 len=4
```

Since this is a breakpoint, the simulation will also stop the execution of the processor, and indicate the current code location, resulting in a printout like this:

```
??()  
mov rbp,qword ptr [rbx+0xd8]
```

The code location reported has *nothing* to do with the access, it is just wherever the processor happened to be running at the time that the DMA operation was performed. It will vary depending on OS scheduling etc.

68. To see the time of the operation, use the **ptime** command. **ptime** shows the time for the current processor. With the **-all** option, it shows the time for all processors in the system. In this case, the **master_bb** device operates within the time frame of **core[0][0]**. Check the time between pixel updates:

```
simics> ptime -all
```

69. Run the simulation forward, to get to the next DMA read:

```
simics> r
```

70. Look at the time again to figure out the time for each pixel update:

```
simics> ptime -all
```

71. Compute the difference in **cycles** on **core[0][0]** between the two stops to see the time for one pixel update. Copy the two values and compute it on the simulator command line. Use the **dec** command to make sure the output is in decimal. For example:

```
simics> dec (52_391_587_144_290 - 52_391_587_143_750)
```

72. Look at the **LED Panel** to see the pixels change color one at a time. Each stop should correspond to one more pixel being painted.

```
simics> r
```

73. Remove all breakpoints set so far.

```
simics> bp.delete -all
```

74. Run the simulation to make sure the whole display updates.

```
simics> r
```

Break on Exception

Next, you will use an advanced breakpoint to break when the device issues an interrupt. The interrupts in the training system come from button presses in the panel.

75. Make sure the simulation is running if it is not already.
76. Go to the target **serial console**. Do a read from the device to make the driver wait for an interrupt caused by the button being pressed.

```
# cat /dev/simics_training_pcie_driver
```

77. Pause the simulation:

```
running> stop
```

78. Set a breakpoint on all exceptions. This is overly broad, but will help find the actual interrupt number used by the device (note that you did the same thing in lab 004):

```
simics> bp.exception.break -all object = "machine.mb.cpu0" -recursive
```

79. Run forward to catch the next regular exception (timer tick) so that you know the next exception is the one from the next button press.

```
simics> r
```

80. Fake a button press to the button i2c device using the simulator CLI. This shows how you can script target inputs to make sure they are controlled, and without depending on clicking in a GUI. This was also already done in lab 004:

```
simics>
@conf.machine.training_card.button_a.port.button_in.iface.signal.signal_raise()
```

81. To follow the signal interface protocol, make sure to also lower the signal. It is OK to do this immediately in this case, since the button will already have registered the press.

```
simics>
@conf.machine.training_card.button_a.port.button_in.iface.signal.signal_lower()
```

82. Run the simulation:

```
simics> r
```

83. The simulation should stop almost immediately, printing something like this:

```
[machine.mb.cpu0.core[1][0]] Breakpoint 5: machine.mb.cpu0.core[1][0] Interrupt_38(550)
exception triggered
```

84. Check that the debugger is currently debugging on the core that took the exception:

```
simics> pselect
```

If not, make sure to **pselect** it so that you work with the right core in the following.

85. When a PCIe MSI-X interrupt happens, it is first handled by the Linux kernel before being dispatched to the device driver. Step one instruction to see where the first-line interrupt handler is located in memory:

```
simics> si -r
```

This should get into the interrupt handler code in the Linux kernel:

```
rsp <- 0xffff_c900_0192_be48
eflags <- 0x0002
irq_entries_start()
0xfffffffff81e00248 (irq_entries_start + 0x38)    push 88
```

86. Three more steps should get you into the next level interrupt handling function:

```
simics> si -r
simics> si -r
simics> si -r
```

It should look like this:

```
simics> si -r
[machine.mb.cpu0.core[1][0]] cs:0xfffffffff81e00248 p:0x002e00248  push 88
rsp <- 0xffff_c900_0192_be40
0xfffffffff81e0024a (irq_entries_start + 0x3a)    jmp 0xfffffffff81e00940
simics> si -r
[machine.mb.cpu0.core[1][0]] cs:0xfffffffff81e0024a p:0x002e0024a  jmp 0xfffffffff81e00940
common_interrupt()
0xfffffffff81e00940 (common_interrupt + 0x0)      add qword ptr [rsp],0xffffffff80
```

87. To avoid noise in the rest of the exercise, remove the exception breakpoint. Use **bp.list** to list the breakpoints and find the ID, and then **bp.delete ID** to remove the exception breakpoint:

```
simics> bp.list
simics> bp.delete ID
```

88. To find out where the interrupt handling enters the device driver code, put a breakpoint on all code in the driver. Use **show-memorymap** to figure out where the code is located:

```
simics> show-memorymap
```

The driver code is in the **.text** section.

89. Copy the address and size from the **.text** section, and paste them into a break command. Note the breakpoint number that breakpoint manager gives the new breakpoint.

```
simics> bp.memory.break -x 0xfffffffffc0072000_or_some 0x35_or_some
```


90. Check the current set of breakpoints:

```
simics> bp.list
```

It should look something like this, the breakpoint IDs are likely to vary:

ID	Description	Enabled	Oneshot	Ignore count	Hit count
0x0006	machine.cell_context break matching (addr=0xfffffffffc0214000, len=0x08f6, access=x)	true	false	0x0000	

91. Continue the simulation.

```
simics> r
```

The simulation should stop in the driver code, at the top of the function **irq_handler_button()**.

```
[machine.cell_context] Breakpoint 6: machine.cell_context 'x' access to v:0xfffffffffc02141a0  
len=4  
irq_handler_button(irq=0x0022, pdev=(void *) 0x0) at /home/root/simics-training-pcie-  
driver/simics-training-pcie-driver.c:201  
201      static irqreturn_t irq_handler_button(int irq, void *pdev) {
```

92. Delete all execution breakpoints using the **-all** option to **bp.delete**. Alternatively, you could delete all the breakpoints set above using their breakpoint IDs.

```
simics> bp.delete -all
```

93. Check the hardware status of the **button_a_status** register:

```
simics> print-device-reg-info  
machine.training_card.master_bb.bank.regs.button_a_status
```

The register should have the value 1.

```
Status of button A [machine.training_card.master_bb.bank.regs:button_a_status]  
  
      Bits : 32  
      Offset : 0x204  
      Value : 1  
  
Bit Fields:  
  button_a_status[31..0] : 00000000000000000000000000000001
```

When receiving the button press, the device will first set the status register for the button affected to 1. It will then trigger an interrupt alerting the processor about the button press. This interrupt is the exception that the execution was stopped by above.

94. The next step of the driver is to find out which button was pressed. This should result in accesses to the device. Set a breakpoint on accesses to the control registers (both read and write):

```
simics> bp.bank.break machine.training_card.master_bb.bank.regs
```

95. Continue the execution to see if the software does access the device:

```
simics> r
```

96. This should hit a hardware access breakpoint, printing something like this – hitting the kernel function **ioread32()** which is used to read from devices.

```
[machine.training_card.master_bb.bank.regs] Breakpoint 9:
machine.training_card.master_bb.bank.regs read at offset=0x204 size=0x4 value=0x1
ini=machine.mb.cpu0.core[1][0]
ioread32()
0xffffffff8162444d (ioread32 + 0x3d)          ret
```

You should recognize the offset from the register information for **button_a_status**.

97. Do a **stack-trace** to see the current code location:

```
simics> stack-trace
```

The trace indicates that the kernel **ioread32** function was called from the device driver **chari_read** function.

98. Get out of the read function using the debugger **step-out** command:

```
simics> step-out
```

99. You should see the driver code appear:

```
chari_read(file, buffer=(char *) 0x7fad66e44000 <cannot read memory>, len, offset) at
/root/swbuild/simics-training-pcie-driver/simics-training-pcie-driver.c:260
260          if(btnstatus != 0) {
```

100. Use the **list** command on the command line:

```
simics> list
```

101. Check the value of the variable controlling the upcoming **if** statement:

```
simics> sym-value btnstatus
```

It should be 1, and the code should thus go into the body of the **if** and clear the button status.

102. Run forward again.

```
simics> r
```

The simulation stops on a device access breakpoint. The device sees a write with the value **0x0001** to offset **0x204**, i.e., the **button_a_status** register.

```
[machine.training_card.master_bb.bank.regs] Breakpoint 9:
machine.training_card.master_bb.bank.regs write at offset=0x204 size=0x4 value=0x1
ini=machine.mb.cpu0.core[1][0]
iowrite32()
0xffffffff8162450f (iowrite32 + 0xf)          ret
```

103. Check the value of the **button_a_status** register after this write, using the same command as above:

```
simics> print-device-reg-info
machine.training_card.master_bb.bank.regs.button_a_status
```

The value is zero. Which might seem funny since the code wrote **0x0001**, but this is a “write one clears” register, where the status is cleared by writing a one to the bits in the register that should be cleared. A device model is free to do anything it wants

with the values it receives – just like real hardware, it does not necessarily just store the value written.

104. This concludes the lab on how to debug a device driver in the Intel Simics Simulator.

Quit this simulation session.

C. Debugging PCIe driver initialization and probe

The previous section showed how to debug the code of the device driver after it had been loaded. It is also possible to debug the initialization phase of the driver. In order to do this, you need to know where a driver will be loaded before it is loaded. Thanks to the simulation's determinism, this can actually be achieved.

If you start the simulation from a checkpoint with a start script, each run should be exactly identical. Thus, starting the simulation with a script that loads the driver immediately should result in the same load address each time. This can be leveraged into a solid initialization debug solution.

Set up script to load driver and print .ko load addresses

The first step is to build a script that will load the kernel module automatically, ensuring determinism in the operation and the same load address each time. You already did this back in Lab 002, and you can use that script as the starting point for this script. In addition, the script has to retrieve the section mappings. For this, you should use the Simics Agent to retrieve the mappings— that is a lot easier than trying to parse the output from the target serial console.

1. Locate the script file **lab-002-script-branch.simics** that was created in lab 002. It should be located in **<project>/targets/simics-user-training/**.
2. Copy it to a new file called **lab-006-debug-insmod.simics**.
3. **Open** the new script for editing in your favorite editor.
4. Delete everything in the script branch after the line:

```
bp.console_string.wait-then-write $con "# " "insmod simics-training-pcie-driver.ko \n"
```

Like this:

```
read-configuration "MyFirstCheckpoint.ckpt"

# set up variables, since the checkpoint does not preserve them
$system = "machine"

# Fix the console dimming
$system.training_card.panel.con.dimming FALSE

# script-branch to automatically do the insmod
script-branch "insmod and test" {
    # Pick up the name of the serial console
    local $con = $system.serconsole.con
    local $cpu = $system.mb.cpu0.core[0][0]

    # Signal that we start
    echo "Start simulation to insmod driver"

    # Send in a newline to trigger a prompt
    $con.input "\n"
    # input insmod
    bp.console_string.wait-then-write $con "# " "insmod simics-training-pcie-driver.ko \n"
```

5. Insert the following code before the end of the script branch. Make sure the script branch ends with a “}”:

```
# wait to get back to prompt, and then retrieve the section mappings
# use the Simics agent for this, as it makes it a ton easier to retrieve
# the output from the commands
log-level 0
$a = (start-agent-manager)
$h = ($a.connect-to-agent)
$h.wait-for-job ## this waits for the agent to connect to Simics

# retrieve section mappings using agent run command
# with the capture flag, wait-for-job will return the output from the command
# ... which happens to be just the section load address, very convenient
$driver_name = simics_training_pcie_driver
$job = ($h.run -capture ("cat /sys/module/%s/sections/.text" % [$driver_name] ))
$o = ($h.wait-for-job -capture $job)    ## $o holds the result from the cat command
$text_addr = (atoi $o)                ## Simics CLI does have an atoi command!
$s = ("Identified load address of .text segment: 0x%x" % [ $text_addr ])
echo $s

$job = ($h.run -capture ("cat /sys/module/%s/sections/.text.unlikely" % [$driver_name] ))
$o = ($h.wait-for-job -capture $job)
$text_unlikely_addr = (atoi $o)
$s = ("Identified load address of .text.unlikely segment: 0x%x" % [$text_unlikely_addr])
echo $s

$job = ($h.run -capture ("cat /sys/module/%s/sections/.data" % [$driver_name] ))
$o = ($h.wait-for-job -capture $job)
$data_addr = (atoi $o)
$s = ("Identified load address of .data segment: 0x%x" % [ $data_addr ])
echo $s

$job = ($h.run -capture ("cat /sys/module/%s/sections/.bss" % [$driver_name] ))
$o = ($h.wait-for-job -capture $job)
$bss_addr = (atoi $o)
$s = ("Identified load address of .bss segment: 0x%x" % [ $bss_addr ])
echo $s

$job = ($h.run -capture ("cat /sys/module/%s/sections/.init.text" % [$driver_name] ))
$o = ($h.wait-for-job -capture $job)
$init_text_addr = (atoi $o)
$s = ("Identified load address of .init.text segment: 0x%x" % [ $init_text_addr ])
echo $s

## Generate add-symbol-file commands to use in a future script
echo ""
echo "Setup commands for debug in a future script:"
echo ""
echo "$ko = (lookup-file \"%simics%/targets/simics-user-training/target-code/simics-
training-pcie-driver.ko\")"
echo "$text_addr = 0x%x \nadd-symbol-file $ko $text_addr \".text\" \" % [$text_addr]
echo "$text_unlikely_addr = 0x%x \nadd-symbol-file $ko $text_unlikely_addr
\".text.unlikely\" \" % [$text_unlikely_addr]
echo "$init_text_addr = 0x%x \nadd-symbol-file $ko $init_text_addr \".init.text\" \" %
[$init_text_addr]
echo "$bss_addr = 0x%x \nadd-symbol-file $ko $bss_addr \".bss\" \" % [$bss_addr]
echo "$data_addr = 0x%x \nadd-symbol-file $ko $data_addr \".data\" \" % [$data_addr]
## end of script branch:
}
```

A few comments on the CLI code:

- The % operator can be used in CLI to format strings in the same way as in Python.
- When using the Simics Agent to run commands, you do not end command-line input with “\n” – that is only necessary when feeding in commands over the serial console.

- Using **-capture** with the agent is a simpler way to achieve what **record-start** and **record-stop** do with the serial console. While the serial console capture will tend to include line endings and other debris that gets printed, the agent simply returns the output returned by the command, which is usually a bit cleaner.
- The final part prints a set of CLI commands to the simulation command line itself, for use in the next script. This method is a bit clumsy, but it does provide a simple way to move data from one script to another across unrelated simulation sessions.

Run the automation script

6. **Start** a new simulation session using this script.

```
[${C:>} simics[.bat] targets/simics-user-training/lab-006-debug-  
insmod.simics
```

7. **Run** the simulation forward.

```
simics> r
```

It should produce a list of the section load addresses both on the simulator command line and in the output of **list-session-comments**. This can take a short while in case the simulator is running slowly.

8. Once the commands have been printed, **stop** the simulation.

```
running> stop
```

9. **Copy** the set of commands that the script prints to the simulator command line at the end, for use in the next step. They contain the load addresses for all sections.

```
Start simulation to insmod driver  
simics> r  
Identified load address of .text segment: 0xffffffffc0194000  
Identified load address of .text.unlikely segment: 0xffffffffc0194035  
Identified load address of .data segment: 0xffffffffc0196000  
Identified load address of .bss segment: 0xffffffffc0196600  
Identified load address of .init.text segment: 0xffffffffc0199000  
  
Setup commands for debug in a future script:  
  
$ko = (lookup-file "%simics%/targets/simics-user-training/target-code/simics-training-pcie-  
driver.ko")  
$text_addr = 0xffffffffc0194000  
add-symbol-file $ko $text_addr ".text"  
$text_unlikely_addr = 0xffffffffc0194035  
add-symbol-file $ko $text_unlikely_addr ".text.unlikely"  
$init_text_addr = 0xffffffffc0199000  
add-symbol-file $ko $init_text_addr ".init.text"  
$bss_addr = 0xffffffffc0196600  
add-symbol-file $ko $bss_addr ".bss"  
$data_addr = 0xffffffffc0196000  
add-symbol-file $ko $data_addr ".data"
```

10. **Quit** the simulation session.

```
simics> quit
```

Set up automatic script to debug driver load

Given the above starting point, create a script that sets up for device driver debug. This script should open the checkpoint, set up the debug information, and then do the **insmod** operation. It can be built from the pieces you have already created previously.

A key assumption is that when the Intel Simics Simulator opens the checkpoint and runs the scripted **insmod**, the system state is the same each time. Thus, the section load addresses detected in the previous step can be used each time the checkpoint is opened and the scripted **insmod** run. This lets you debug the driver insertion from the very start.

11. Copy the script **lab-006-debug-insmod.simics** to a new script called **lab-006-debug-insmod-2.simics**.
12. Rework it to include the debug setup steps before the script branch, removing the retrieval of the section addresses using the Intel Simics Agent, and adding the debug setup that the script above printed to the start of the file.

This is what the script will look like:

```
read-configuration "MyFirstCheckpoint.ckpt"

# set up variables, since the checkpoint does not preserve them
$system = "machine"

# Fix the console dimming
$system.training_card.panel.con.dimming FALSE

# set up for debug based on information from the previous script
# just copy-paste the commands that the previous script printed.
$ko = (lookup-file "%simics%/targets/simics-user-training/target-code/simics-training-pcie-
driver.ko")
$text_addr = 0xffffffffc0214000
add-symbol-file $ko $text_addr ".text"
$init_text_addr = 0xffffffffc021a000
add-symbol-file $ko $init_text_addr ".init.text"
$bss_addr = 0xffffffffc0216800
add-symbol-file $ko $bss_addr ".bss"
$data_addr = 0xffffffffc0216080
add-symbol-file $ko $data_addr ".data"

# Pathmap, as before
$s = "/home/root/simics-training-pcie-driver"
$d = (lookup-file "%simics%/targets/simics-user-training/target-source/simics-training-pcie-
driver")
add-pathmap-entry $s $d

# Add the Linux kernel symbols as well
add-symbol-file vmlinux

# Automatically enable the debugger
enable-debugger

# Figure out section lengths, using the Simics TCF debugger API
# Requires the filename and sectionname. It basically reads the debug info
# file anew to extract the information
@def find_section_size(filename,sectionname):
  (err, file_id) = conf.tcf.iface.debug_symbol_file.open_symbol_file(filename, 0, True)
  (err, type_and_sect_info) = conf.tcf.iface.debug_symbol_file.sections_info(file_id)
  sect_info = type_and_sect_info[1]
  sect_size = None
  for sect in sect_info:
    if sect.get("name") == sectionname:
      sect_size = sect["size"]
  return sect_size

# Set breakpoint on the code sections
bp.memory.break $text_addr      `find_section_size(simenv.ko,".text")`
bp.memory.break $init_text_addr `find_section_size(simenv.ko,".init.text")`

# script-branch to automatically do the insmod
script-branch "insmod " {
  # Pick up the name of the serial console
  local $con = $system.serconsole.con
  # Send in a newline to trigger a prompt
  $con.input "\n"
  bp.console_string.wait-then-write $con "# " "insmod simics-training-pcie-driver.ko \n"

# Remove all the code to retrieve the segment addresses
}
```

The Python function **find_section_size()** uses the simulator Debugger API to obtain the size of a section given its name and the debug info file it is contained in. It basically reads the debug file and extracts the information.

13. Start a new simulation session using the script **lab-006-insmod-debug-2.simics**.

```
[${C:>} simics[.bat] targets/simics-user-training/lab-006-debug-
insmod-2.simics
```


14. Before running, check that there are breakpoints set:

```
simics> bp.list
```

15. Check that there is a script branch waiting to do the **insmod** operation:

```
simics> list-script-branches
```

Note that there is a breakpoint listed that corresponds to the wait-for condition for the script branch.

16. Run the simulation forward.

```
simics> r
```

The execution will stop when the device driver is being initialized. Something like this:

```
[machine.cell_context] Breakpoint 2: machine.cell_context 'x' access to v:0xffffffffc021a010
len=4
Now debugging the x86QSP1 machine.mb.cpu0.core[0][0]
simics_pcie_module_init() at /home/root/simics-training-pcie-driver/simics-training-pcie-
driver.c:705
705      static int __init simics_pcie_module_init(void) {
```

17. Check which processor is active (it will vary depending on the target state):

```
simics> pselect
```

18. Check the stack trace to see the call path inside the kernel to the device driver initialization function:

```
simics> stack-trace
```

19. Check the code at the point where the execution stopped:

```
simics> list maxlines=15
```

The code is right at the start of the Linux device driver module initialization function (**simics_pcie_module_init()**).

20. Check the control registers in the **regs** bank of the **machine.training_card.master_bb** device.

```
simics> print-device-regs machine.training_card.master_bb.bank.regs
```

They are all at their reset values, which are mostly zero (and thus non-functional since the I2C addresses have to be set up).

21. Check where the UEFI and Linux have mapped the registers and memory banks of the device.

```
simics> devs machine.training_card.local_memory
```

This should show a mapping from the **machine.mb.nb.pcie_p2.downstream_port.mem_space** memory space.

22. Check the other mappings in that memory space:

```
simics> machine.mb.nb.pcie_p2.downstream_port.port.mem.map
```

There are four mappings: the local memory, as well as the control registers and MSI-X control banks.

23. Remove the breakpoints that were set up by the start script.

```
simics> bp.delete -all
```

24. Look at the device information structure that gets filled in by the device initialization:

```
simics> sym-value simics_training_pcie_dev
```

Note that the **major** field is zero.

25. Put a breakpoint that stops the execution when the field “major” changes:

```
simics> bp.memory.break -w (sym-address  
simics_training_pcie_dev.major )
```

26. Run the simulation forward.

```
simics> r
```

27. Look at the updated value of **major**:

```
simics> sym-value simics_training_pcie_dev.major
```

28. Remove the breakpoint:

```
simics> bp.delete -all
```

29. Put a breakpoint on the start of the PCIe probe function, which will be called by the kernel when it wants to set up the device as a PCIe device. You could use **bp.source_location.break** here, but to show a different approach, you can also break on a selected line of a source file. This becomes interesting when facing inlined functions that may not be available as symbols.

```
simics> bp.source_line.break filename = (sym-file simics_pcie_probe)  
line-number = 457
```

30. Run the simulation forward.

```
simics> r
```

The simulation should stop on the entry to the **simics_pcie_probe()** function.

31. How did the function get called? Check the stack trace:

```
simics> stack-trace
```

The stack trace shows how the **simics_pcie_probe** function is called via a sequence of kernel functions, which in turn are called by the **simics_pcie_module_init** function:

```
simics> stack-trace
#0 0xffffffffc02141f0 in simics_pcie_probe(pdev=(struct pci_dev *) 0xffff888100b5c000, id=(const struct pci_device_id *) 0xffffffffc0218ce0) at /home/root/simics-training-pcie-driver/simics-training-pcie-driver.c:457
#1 0xffffffff8180f7f0 in pci_device_probe_static__26345()
#2 0xffffffff8198ff60 in really_probe_static__31911()
#3 0xffffffff81990188 in __driver_probe_device_static__31913()
#4 0xffffffff81990264 in driver_probe_device_static__31915()
#5 0xffffffff8199050f in __driver_attach_static__31919()
#6 0xffffffff8198d9e3 in bus_for_each_dev()
#7 0xffffffff8198f882 in driver_attach()
#8 0xffffffff8198ef06 in bus_add_driver()
#9 0xffffffff81991805 in driver_register()
#10 0xffffffff8180de6c in __pci_register_driver()
#11 0xffffffffc021a105 in simics_pcie_module_init() at /home/root/simics-training-pcie-driver/simics-training-pcie-driver.c:763
#12 0xffffffff8100146b in do_one_initcall()
#13 0xffffffff8113ed38 in do_init_module_static__5525()
#14 0xffffffff81140f63 in load_module_static__5529()
#15 0xffffffff81141487 in init_module_from_file_static__5533()
#16 0xfffffc900005b7f00 in ??()
```

32. Check where in the function **simics_pcie_module_init()** the **simics_pcie_probe()** function gets called. Select the frame for **simics_pcie_module_init()** (as printed by the **stack-frame** command):

```
simics> frame frame-number = 11_or_some
```

33. Use **list** to see the current location:

```
simics> list
```

Which should show that the function is calling **pci_register_driver()**:

```
761 pr_info(KBUILD_MODNAME
762          ": PCI device driver registration started\n");
-> 763 err = pci_register_driver(&simics_pcie_driver_pci);
764 if (err) {
765     device_destroy(simics_training_pcie_dev.charClass,
MKDEV(simics_training_pcie_dev.major, 0)); // remove the device
```

It is clear that the PCIe probe function is called immediately upon the PCIe driver being registered with the kernel – it does not wait until the module initialization function has finished.

34. Go back to the current code in **simics_pcie_probe**. The current frame is always frame 0:

```
simics> frame 0
```

35. The driver retrieves the memory mapping for the PCIe main register BAR and stores it in the **bar0_base_addr** field of the device structure. Set a breakpoint on accesses to this field to find out what the driver gets from the kernel.

```
simics> bp.memory.break -w (sym-address
"simics_training_pcie_dev.bar0_base_addr")
```

36. Run the simulation forward.

```
simics> r
```

37. The breakpoint will hit, stopping execution. The current location will be the line after the assignment to the struct field (and thus, the memory write).

```
[machine.cell_context] Breakpoint 5: machine.cell_context 'w' access to v:0xffffffffc0216838
len=8 val=0xfffffc9000002d000
simics_pcie_probe(pdev=(struct pci_dev *) 0xfffff888100b5c000, id) at /home/root/simics-training-
pcie-driver/simics-training-pcie-driver.c:569
569          pr_info(KBUILD_MODNAME
```

Note that this address is a kernel virtual address, not the physical address.

38. Check the translation from the virtual address to a physical address:

```
simics> l2p (sym-value simics_training_pcie_dev.bar0_base_addr)
```

This should show an address like **0xf200_2000**, same as what you find in the system memory map.

```
simics> sym-value simics_training_pcie_dev.bar0_base_addr
(void *) 0xfffffc90001dd3000
simics> l2p 0xfffffc90001dd3000
0xf200_2000
```

39. Check where this address ends up:

```
simics> probe-address p:0xf200_2000_or_something
```

You could also stack the commands:

```
simics> probe-address p:(l2p (sym-value simics_training_pcie_dev.bar0_base_addr))
```

The path through the memory map ends up in **regs** bank of the controller of the training card. Which is indeed what is mapped by **BAR0** of this PCIe device.

40. That concludes the driver initialization debug lab.

Quit this simulation session.

D. Appendix. Complete scripts for Section C.

Lab-006-debug-insmod.simics

```
read-configuration "MyFirstCheckpoint.ckpt"

# set up variables, since the checkpoint does not preserve them
$system = "machine"

# Fix the console dimming
$system.training_card.panel.con.dimming FALSE

# script-branch to automatically do the insmod
script-branch "insmod and test" {
    # Pick up the name of the serial console
    local $con = $system.serconsole.con
    local $cpu = $system.mb.cpu0.core[0][0]

    # Signal that we start
    echo "Start simulation to insmod driver"

    # Send in a newline to trigger a prompt
    $con.input "\n"
    # input insmod
    bp.console_string.wait-then-write $con "# " "insmod simics-training-pcie-driver.ko \n"

    # wait to get back to prompt, and then retrieve the section mappings
    # use the Simics agent for this, as it makes it a ton easier to retrieve
    # the output from the commands
    log-level 0
    $a = (start-agent-manager)
    $h = ($a.connect-to-agent)
    $h.wait-for-job ## this waits for the agent to connect to Simics

    # retrieve section mappings using agent run command
    # with the capture flag, wait-for-job will return the output from the command
    # ... which happens to be just the section load address, very convenient
    $driver_name = simics_training_pcie_driver
    $job = ($h.run -capture ("cat /sys/module/%s/sections/.text" % [$driver_name] ))
    $o = ($h.wait-for-job -capture $job) ## $o holds the result from the cat command
    $text_addr = (atoi $o) ## Simics CLI does have an atoi command!
    $s= ("Identified load address of .text segment: 0x%x" % [ $text_addr ])
    echo $s

    $job = ($h.run -capture ("cat /sys/module/%s/sections/.data" % [$driver_name] ))
    $o = ($h.wait-for-job -capture $job)
    $data_addr = (atoi $o)
    $s= ("Identified load address of .data segment: 0x%x" % [ $data_addr ])
    echo $s

    $job = ($h.run -capture ("cat /sys/module/%s/sections/.bss" % [$driver_name] ))
    $o = ($h.wait-for-job -capture $job)
    $bss_addr = (atoi $o)
    $s= ("Identified load address of .bss segment: 0x%x" % [ $bss_addr ])
    echo $s

    $job = ($h.run -capture ("cat /sys/module/%s/sections/.init.text" % [$driver_name] ))
    $o = ($h.wait-for-job -capture $job)
    $init_text_addr = (atoi $o)
    $s= ("Identified load address of .init.text segment: 0x%x" % [ $init_text_addr ])
    echo $s

    ## Generate add-symbol-file commands to use in a future script
    echo ""
    echo "Setup commands for debug in a future script:"
    echo ""
    echo "$ko = (lookup-file \"%simics%/targets/simics-user-training/target-code/simics-
training-pcie-driver.ko\")"
    echo "$text_addr = 0x%x \nadd-symbol-file $ko $text_addr \".text\" \" % [$text_addr]
    echo "$init_text_addr = 0x%x \nadd-symbol-file $ko $init_text_addr \".init.text\" \" %
[$init_text_addr]
    echo "$bss_addr = 0x%x \nadd-symbol-file $ko $bss_addr \".bss\" \" % [$bss_addr]
```

```

    echo "$data_addr = 0x%x \nadd-symbol-file $ko $data_addr \".data\" \" % [$data_addr]
    ## end of script branch:
}

```

Lab-006-debug-insmod-2.simics

Note that the addresses used in add-symbol-file have to be adjusted to match what you get in your setup.

```

read-configuration "MyFirstCheckpoint.ckpt"

# set up variables, since the checkpoint does not preserve them
$system = "machine"

# Fix the console dimming
$system.training_card.panel.con.dimming FALSE

# set up for debug based on information from the previous script
# just copy-paste the commands that the previous script printed.
$ko = (lookup-file "%simics%/targets/simics-user-training/target-code/simics-training-pcie-
driver.ko")
$text_addr = 0xffffffffc0214000
add-symbol-file $ko $text_addr ".text"
$init_text_addr = 0xffffffffc021a000
add-symbol-file $ko $init_text_addr ".init.text"
$bss_addr = 0xffffffffc0216800
add-symbol-file $ko $bss_addr ".bss"
$data_addr = 0xffffffffc0216080
add-symbol-file $ko $data_addr ".data"

# Pathmap, as before
$s = "/home/root/simics-training-pcie-driver"
$d = (lookup-file "%simics%/targets/simics-user-training/target-source/simics-training-pcie-
driver")
add-pathmap-entry $s $d

# Add the Linux kernel symbols as well
add-symbol-file vmlinux

# Automatically enable the debugger
enable-debugger

# Figure out section lengths, using the Simics TCF debugger API
# Requires the filename and sectionname. It basically reads the debug info
# file anew to extract the information
@def find_section_size(filename,sectionname):
    (err, file_id) = conf.tcf.iface.debug_symbol_file.open_symbol_file(filename, 0, True)
    (err, type_and_sect_info) = conf.tcf.iface.debug_symbol_file.sections_info(file_id)
    sect_info = type_and_sect_info[1]
    sect_size = None
    for sect in sect_info:
        if sect.get("name") == sectionname:
            sect_size = sect["size"]
    return sect_size

# Set breakpoint on the code sections
bp.memory.break $text_addr      `find_section_size(simenv.ko, ".text")`
bp.memory.break $init_text_addr `find_section_size(simenv.ko, ".init.text")`

# script-branch to automatically do the insmod
script-branch "insmod " {
    # Pick up the name of the serial console
    local $con = $system.serconsole.con
    # Send in a newline to trigger a prompt
    $con.input "\n"
    bp.console_string.wait-then-write $con "# " "insmod simics-training-pcie-driver.ko \n"

# Remove all the code to retrieve the segment addresses
}

```