



# **Intel<sup>®</sup> Simics<sup>®</sup> Simulator New User Training**

## **Lab 005 – Basic debugging with the Intel<sup>®</sup> Simics<sup>®</sup> simulator**

# Lab 005 – Basic debugging with the Intel® Simics® Simulator

This is the first lab on software debugging in the Intel® Simics® Simulator. It covers basic debug operations through the CLI. There is also a second lab, which covers advanced breakpoints and more command line work. In addition to typical debugger operations, this lab also covers additional concepts like scripting the debugger and using the Intel Simics Agent to control and copy files to the target system.

## A. Prepare for debugging

The code used in the debug exercises is found in the training package. That puts it in the product installation tree, which is a rather inconvenient location to access it. To make the debug labs easy to perform, you should copy the target software code from the installation into your project.

There is a script available that does this for you.

1. Launch a session using the **900-copy-debug-files-to-project.simics** script. We use batch mode for this session so that the simulator quits automatically when the script has finished.

```
[C:\>] simics[.bat] --batch-mode targets/simics-user-training/900-copy-debug-files-to-project.simics
```

2. The script should run for a very short amount of time. The directory contents of **targets/simics-user-training** should have changed. Confirm the existence of the **files**, **target-code** and **target-source** subdirectories. In a host shell in your project do:

In a Windows CMD environment:

```
dir /B /OGN targets\simics-user-training
```

In a Linux environment (assuming GNU **ls**). Note that the first argument is the number “one” NOT a lowercase L:

```
ls -l --group-directories-first targets/simics-user-training
```

The result should look like below (depending on which previous lab you did). You want to make sure that you see the first three entries:

```
files
target-code
target-source
900-copy-debug-files-to-project.simics
lab-002-custom-command.py
lab-002-script-branch.simics
...
```

## B. Get the software into the target system

In this section, you will load a user-level program to the target system, start it, and manually set up for debugging. You will use the Intel Simics Agent to get the software on to the target.

1. Open the second checkpoint from lab 001, with the kernel driver for the training device loaded (called **AfterDriver.ckpt** if you followed the instructions verbatim).

```
[$|C:>] simics[.bat] AfterDriver.ckpt
```

2. Once the checkpoint has been opened, **run** the simulation forward.

```
simics> r
```

3. Go to the target **Serial Console** window, and make sure you are back at target prompt.

Check that there is a **simics-agent** running on the target system (it is started automatically at boot on the QSP Clear Linux setup that we are basing the training on):

```
# ps -Af | grep simics
```

If there is an agent running, it should look like this:

root	263	1	0	15:14	?	00:00:00	/usr/bin/simics-agent
root	323	276	0	16:33	ttyS0	00:00:00	grep simics

4. If no process was found, start the **simics-agent** client program on the target:

```
# simics_agent_x86_linux64 &
```

Note that this is often necessary when using software stacks that do not start the agent automatically. You might also need to copy the agent binaries into the target from a USB drive image (or similar) to make them available.

5. Go to the simulator command line, and start the agent manager on the side:

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

6. Connect to the agent running on the target. Since there is only one target machine, it will automatically find it and connect to it.

```
running> agent_manager.connect-to-agent
```

If everything works correctly, you should get a message like:

```
matic0 connected to cl-qsp0 (0x1b90f02e886b3cbb)
```

Which means that you have an agent handle called **matic0** connected to the target. This handle is used to interact with the target system.

7. Quickly check that **matic0** is functioning and connected:

```
running> matic0.info
```

Which should say that the handle is connected to something:

```
Information about matic0 [class agent_handle]
=====

    Connected to : machine0
...
```

8. Next, upload the target program. It is available in the project thanks to the script you ran in Section A above. Locate it using **lookup-file** and the **%simics%** file marker:

```
running> lookup-file "%simics%/targets/simics-user-training/target-
code/demo-one"
```

This should find the file *inside your project*, and not inside the installation.

9. Use the Intel Simics agent to upload the file. The following command uploads the file as an executable and puts it in **/root/**, which is the home directory on the target system where you are currently running. Note that the parentheses around the **lookup-file** command are needed in order to evaluate the command and use the result as an argument to the **upload** command.

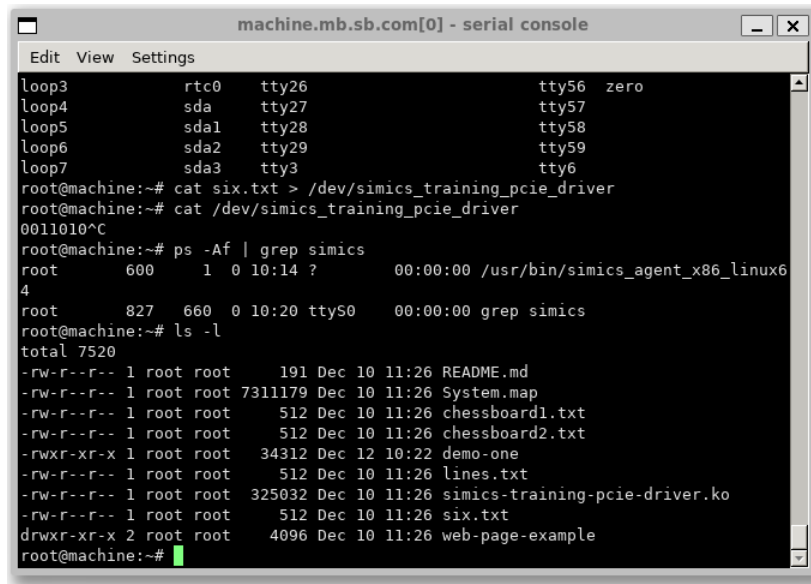
```
running> matic0.upload -executable (lookup-file
"%simics%/targets/simics-user-training/target-code/demo-one")
"/home/root/"
```

*Note that this lab uses "root" since that is the user that is logged in in the checkpoint. It could just as well be done using the "simics" user on the target. However, using root makes the insertion of drivers and access to device nodes a lot easier. So, root is being used even though really you should never run Linux software as root.*

10. Check that the program is indeed copied to the target system: go to the target **Serial Console** window, and do:

```
# ls -l
```

You should see the demo-one executable being executable, alongside the files used to test the driver previously.



```
machine.mb.sb.com[0] - serial console
Edit View Settings
loop3      rtc0    tty26          tty56 zero
loop4      sda     tty27          tty57
loop5      sda1    tty28          tty58
loop6      sda2    tty29          tty59
loop7      sda3    tty3           tty6
root@machine:~# cat six.txt > /dev/simics_training_pcie_driver
root@machine:~# cat /dev/simics_training_pcie_driver
0011010^C
root@machine:~# ps -Af | grep simics
root      600    1  0 10:14 ?        00:00:00 /usr/bin/simics_agent_x86_linux64
root      827    660 0 10:20 ttyS0    00:00:00 grep simics
root@machine:~# ls -l
total 7520
-rw-r--r-- 1 root root    191 Dec 10 11:26 README.md
-rw-r--r-- 1 root root 7311179 Dec 10 11:26 System.map
-rw-r--r-- 1 root root   512 Dec 10 11:26 chessboard1.txt
-rw-r--r-- 1 root root   512 Dec 10 11:26 chessboard2.txt
-rwxr-xr-x 1 root root  34312 Dec 12 10:22 demo-one
-rw-r--r-- 1 root root   512 Dec 10 11:26 lines.txt
-rw-r--r-- 1 root root 325032 Dec 10 11:26 simics-training-pcie-driver.ko
-rw-r--r-- 1 root root   512 Dec 10 11:26 six.txt
drwxr-xr-x 2 root root   4096 Dec 10 11:26 web-page-example
root@machine:~#
```

11. Test the **demo-one** program to see what it does. The program takes two arguments:
- A string to scroll across the LED display. This can contain some punctuation marks, numbers, and capital letters. Other characters will be rendered as... well, other.
  - The path to the PCI mappings of the training device, in the Linux **/sys** filesystem. You can build the path by tab completion, at least towards the end. The **0000:02:00.0** address indicates that this is function **0**, on device **00**, on PCI(e) bus **02**, on controller **0000**.

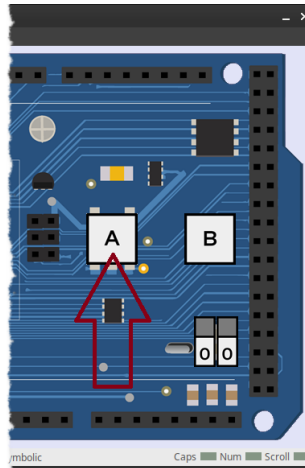
```
# ./demo-one "0123-C...TEST STRING!" /sys/bus/pci/devices/0000\:02\:00.0
```

Note that for the debug exercise, it is important to start the string to be displayed with a few numbers.

The program will manipulate the hardware directly, using **mmap** to make the device memory accessible in user space. If the start is successful, it should print the addresses at which it mapped the local memory and control register bank of the training device.

```
root@machine:~# ./demo-one "0123-C...TEST STRING!"
/sys/bus/pci/devices/0000\:02\:00.0
Device control registers mapped to address 0x7fcf60c2d000, for 0x1000
bytes.
Device local memory mapped to address 0x7fcf5fa29000, for 0x1000000
bytes.
Press button A on panel to start - scrolling text '0123-C...TEST
STRING!'!
```

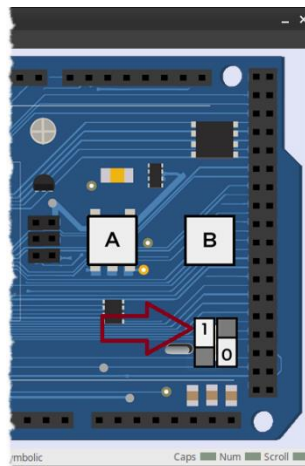
12. The program is now running but waiting for user input to continue. Press the button A on the System Panel to get it into the main loop:



13. The text will scroll past and flicker a lot since the simulation is running much faster than real time on average. To calm things down and get a steadier pace of output, enable real-time-mode:

```
running> enable-real-time-mode
```

14. In the **LED Panel**, change the **left-most toggle** to “On” and see what happens.



15. Once you are bored with watching the display, quit the target software program by pressing **button B** in the **System Panel**.
16. Do *not* quit the simulation session, just leave it running and move on the next section.

## C. Get into debug

The first step of doing debug is getting to the start of the code. The Intel Simics Simulator is a system-level debugger, and you are debugging a user-level program. That means that you can't just stop the simulation and expect to get into the target program – more often than not, something else than the program will be running. Thus, the debugger needs to know when the program is actively running in order to stop at breakpoints and only step within the bounds of the program. This is achieved using OS awareness in the Intel Simics simulator.

### Enable operating system awareness

OS awareness could be used to detect when the program is started, but in this case, a *magic instruction* is used instead. A magic instruction is a NOP on the hardware that the simulator detects, and it is common to insert magic instructions into code that is known to run on the Intel Simics simulator. A typical use case for magic instructions is to indicate the start and end of interesting pieces of code. Magic instructions are independent of OS awareness and are often used in benchmarks for simplicity and robustness.

1. Enable OS awareness so that the simulator will track the executing user-level processes in Linux:

```
running> machine.software.enable-tracker
```

### Break at program start using magic instructions

2. Tell simulator to break (stop) when hitting a magic instruction:

```
running> bp.magic.break
```

3. Check the breakpoints:

```
running> bp.list
```

There should be a single breakpoint that stops on magic zero on any processor:

ID	Description	Enabled	Oneshot	Ignore count	Hit count
1	break on magic instruction (0) on any processor	true	false	0	0

4. Go to the **Serial Console** window and do <UP-ARROW> and <ENTER> to start the program again.

```
# ./demo-one "0123-C...TEST STRING!" /sys/bus/pci/devices/0000\:02\:00.0
```



5. The simulation should stop almost immediately. Printing a message about hitting a magic instruction breakpoint:

```
Breakpoint 1: magic instruction (0)
```

6. When the simulation stops on a breakpoint, the debugger sets the *current processor* to the one that hit the breakpoint. The current processor is used as the implicit target for global debugger commands like stepping and inspecting memory.

Check the current processor using the **pselect** command.

```
simics> pselect
```

This will return "**machine.mb.cpu0.core[N][0]**" (*N* can be either 0 or 1, which one depends on the how the Linux operating system schedules the program for this particular run).

## Activate the debugger

7. Activate the built-in debugger, using the CLI.

```
simics> enable-debugger
```

## Set up symbols and source code

Next, you need to set up symbols and source code. When the magic breakpoint was hit, the simulator switched the current front-end processor (the default processor for many commands, specifically analysis commands) to the one where the magic instruction executed on. However, to debug the program, you want to use the debugger context that corresponds to the Linux process of the program. That context is managed by operating system awareness (OSA) and is found by listing the OSA node tree.

- List the OSA node tree on the CLI.

```
simics> machine.software.node-tree
```

The response from the CLI will list all currently known debug contexts:

```
[Linux]--*[Kernel]--*[Other]--*[name='machine.mb.cpu0.core[0][0]']
                                     |
                                     |--[name='machine.mb.cpu0.core[1][0]']
                                     |
                                     |--[Idle]--[name='machine.mb.cpu0.core[0][0]']
                                     |
                                     |--[name='machine.mb.cpu0.core[1][0]']
                                     |
                                     |--[Threads]--[name='kworker/0:1H',tid=175]
                                                         |
                                                         |--[name='ext4-rsv-conver',tid=177]
                                                         |
                                                         |--[name='kworker/0:5',tid=201]
                                                         |
                                                         |--[name=kauditd,tid=24]
                                                         |
                                                         |--*[Userspace]--[name=python3,pid=264]--[name=python3,tid=264]
                                                         |
                                                         |--[name=pacdiscovery,pid=282]--[name=pacdiscovery,tid=282]
                                                         |
                                                         |--[name='systemd-resolve',pid=257]--[name='systemd-resolve',tid=257]
                                                         |
                                                         |--[name='systemd-udev',pid=196]--[name='systemd-udev',tid=196]
                                                         |
                                                         |--[name=mcelog,pid=244]--[name=mcelog,tid=244]
                                                         |
                                                         |--[name=systemd,pid=1]--[name=systemd,tid=1]
                                                         |
                                                         |--*[name='demo-one',pid=306]--[name='demo-one',tid=306]
```

Note that the **demo-one** process is shown as an active process by the asterisk (\*) next to it.

There are two contexts called **demo-one** in the debug context tree. The outer one is the process that is running, and it has a Linux **pid** (process ID) associated with it. The inner one is executing thread, which has a **tid** (thread ID). This reflects how Linux handles its processes – OSA has to follow the structure of the operating system that is being tracked in order to make sense.

- Check the current debug context.

```
simics> debug-context
```

You should see that the debugger has already picked up the context of the **demo-one** application (**pid** and **tid** should match what you saw in the OSA node tree). Note that the debugger has no idea (yet) about function names and related symbol information, because no symbol data is loaded yet. Like this:

```
dbg0 (/Linux/Userspace/pid=306/tid=306 on machine.software)
Now debugging /Linux/Userspace/pid=306/tid=306 on machine.software
??()
Cpuid
```

- If the debug context is not pointing to **demo-one**, you can switch it as follows:

```
simics> debug-context context-query = "name='demo-one'"
```

11. Now that you have found the debug context, it is time to add the symbols.

```
simics> add-symbol-file symbolfile = targets/simics-user-  
training/target-code/demo-one context-query = "name='demo-one'"
```

The output should indicate that the query matches some contexts. If the number is zero, it means that the query was malformed somehow.

```
Context query name="demo-one" currently matches 2 contexts.  
Symbol file added with id '1'.
```

12. Now that we have debug symbols loaded, try inspecting the back trace.

```
simics> bt
```

Showing:

```
#0 0x4015cf in main(argc, argv) at /root/swbuild/demo-one/demo-one.c:326
```

13. Now try to list the source code you are currently debugging.

```
simics> list
```

Which should show:

```
Unable to locate file: /root/swbuild/demo-one/demo-one.c
```

Since the SW was built on a different machine in directories that might not even exist on your machine, the debugger cannot find the sources. That is a quite common scenario where you different build machines and test machines.

14. Set up a path map that tells the debugger how to translate build paths into debugger host paths.

```
simics> add-pathmap-entry context-query = "name='demo-one'" source =  
"/root/swbuild/demo-one" destination = targets/simics-user-  
training/target-source/demo-one
```

This tells the debugger, whenever it finds a source location starting with `/root/swbuild/demo-one` it will look into `targets/simics-user-training/target-source/demo-one` instead. This way `/root/swbuild/demo-one/demo-one.c` will be found as your project's `targets/simics-user-training/target-source/demo-one/demo-one.c`.

15. Check the path mapping that was set up:

```
simics> show-pathmap
```

16. Try listing the source again, and you'll see that it shows an excerpt of the current source file with the currently executed line marked.

```
simics> list
```

Which should show the magic instruction macro as the current source line:

```
324
325 // Allow Simics to catch the start
-> 326 MAGIC(0);
327
328 // Check argument count
```

At this point, you are ready for source-code debugging.

Note that the setup steps for symbols and source code can often be automated using scripts, avoiding these manual steps. You will see in the next lab section how this can be done.

17. Quit the simulation session.

```
simics> quit
```

## D. Build an automatic load-and-debug script

Build a script to automatically perform the loading and starting of the target software, as well as configuring the debug information. This will make it very easy to get back into the demo program.

1. Create a new script file in **target/simics-user-training/** in your project, called something like **lab-005-auto-debug.simics**, and open it in a text editor of your choice.
2. Reproduce the actions taken to start debugging in the script file. All the way from opening the checkpoint, connecting the Intel Simics agent, uploading the code, starting the program, and setting up the magic breakpoint.

The script should also use **add-symbol-file** to configure the symbol files for the debugger from the script and **add-pathmap-entry** to find the source code without user manual intervention. Scripted setup of symbol files/debug information and path mappings is highly recommended for repeated debug tasks and when sharing debug setups with other users.

Here is what such a file can look like:

```
##
## Load the checkpoint - use the name that you saved it under
##
read-configuration "AfterDriver.ckpt"

$system = machine

local $a = (start-agent-manager)

# Enable OS awareness
$system.software.enable-tracker

# Set up magic breakpoints
bp.magic.break

# Find the executable file
local $f = (lookup-file "%simics%/targets/simics-user-training/target-code/demo-one")

# our context query. Make global for later reuse
$dbg_ctx = "name='demo-one'"

# Add the symbol file from script code
add-symbol-file context-query = $dbg_ctx symbolfile = $f

# And set up a path mapping to automatically find the source.
# it specifies where the code was built, and where it is now
(add-pathmap-entry
 source = "/root/swbuild/demo-one"
 destination =
 (lookup-file "%simics%/targets/simics-user-training/target-source/demo-one/"))

## Script branch for actions with the target
script-branch "load and run" {
    local $con = $system.serconsole.con

    ## Connect to the agent - this puts "matic0" into $h
    local $h = ($a.connect-to-agent)
    $h.wait-for-job

    ## Upload the software
    $h.upload -executable $f "/home/root/"
    $h.wait-for-job

    ## Enter the command on the target:
    $con.input "\n\n./demo-one \"0123-C...TEST STRING!\" \"\sys/bus/pci/devices/0000:02:00.0\"\n"

    ## Set up for debug
    enable-debugger
    enable-real-time-mode
    echo "Ready for Debug!"
}
```

## Test the automation

3. Start a new simulation session using this command file.

```
[C:] simics[.bat] targets/simics-user-training/lab-005-auto-
debug.simics
```

4. Run the simulation forward.

```
simics> r
```

5. The simulation should stop when the target program hits the magic breakpoint, and you should see the message “Ready for debugging” if you used the example script.
6. Check the path mapping set up by the script by issueing the command **show-pathmap**:

```
simics> show-pathmap
```

7. List the current source file again to verify all mappings are working okay:

```
simics> list maxlines=30
```

8. Keep this simulation session running, as we will use it in the next section of the lab.

Automation scripts like this are very handy when iterating on debugging software. You would use your build environment to rebuild software such as the **demo-one** program, and just run this script to test the new version, over and over again. Starting from a checkpoint of a booted system makes it very fast to get to a useful state for software testing, and the Intel Simics simulator allows both the upload of the rebuilt software, setting up the debugger, and running the program under debugger control to be fully automated.

## E. Use the debugger

At this point, you should see the source code for the program. Running the script created in the previous section will allow you to (re)start debugging in a fast and convenient way should you need to.

1. You should be right at the start of `main()`, as indicated by arrow (->)highlighting the `"MAGIC(0);"` line.

Understanding the current state of variables is important for debugging. Inspect which symbols are in the local scope.

```
simics> sym-list -locals
```

The output should look like below. Note that exact addresses can vary:

Symbol	Kind	Type	Address	Size
argc	argument	int	0x00007fffd9931d1c	4
argv	argument	char * *	0x00007fffd9931d10	8
filesize	local	size_t	0x00007fffd9932030	8
filestat	local	struct stat	0x00007fffd9931d20	144
l	local	int	0x00007fffd9932040	4
mem_devnode_name	local	char[315]	0x00007fffd9931db0	315
mem_fno	local	int	0x00007fffd993202c	4
reg_devnode_name	local	char[315]	0x00007fffd9931ef0	315
reg_fno	local	int	0x00007fffd993203c	4
scrollstring	local	const char *	0x00007fffd9932048	8
simics_magic_instr_dummy	local	int	0x00007fffd9932044	4

2. Inspect the value of a variable, in this case `argc`.

```
simics> sym-value argc
```

The value should be "3".

3. Inspecting the first argument given to the application. Since it is a **NULL**-terminated string use `sym-string` this time.

```
simics> sym-string "argv[1]"
```

The output should show the string you provided on the command line to the target program:

```
"0123-C...TEST STRING!"
```

Note that you should only use `sym-string` if you know for a fact that the pointer points to a **NULL**-terminated string.

4. Place a breakpoint on the entry to the function `display_demo()`.

```
simics> bp.source_location.break display_demo
```

Which should show that the breakpoint is planted:

```
Breakpoint 2: 0x2 (planted)
```



5. Verify the placed breakpoints.

```
simics> bp.list
```

You see the magic breakpoint placed by your script as well the function breakpoint we just placed:

ID	Description	Enabled	Oneshot	Ignore count	Hit count
1	break on magic instruction (0) on any processor	true	false	0	1
2	execution of display_demo	true	false	0	

6. Save the current step count to understand how many instructions passed between the start of the application and the hit of the breakpoint.

```
simics> $steps_at_start = (ptime -s)
```

7. Run the simulation forward:

```
simics> r
```

8. The code breakpoint should hit:

```
[tcf] Breakpoint 2 on execution in context demo-one (tid=297)
display_demo(displaystring=(const char *) 0x7fffd99328af "0123-C...TEST STRI..") at
/root/swbuild/demo-one/demo-one.c:268
268      printf("Press button A on panel to start - scrolling text '%s'!\n", displaystring);
```

Note that we stopped and that the current source file is called **demo-one.c**.

9. Check how far the program ran (the exact number might differ).

```
simics> (ptime -s) - $steps_at_start
```

It should take a few hundred thousand steps to reach the breakpoint.

10. Check the state when the breakpoint hits.

```
simics> bt
```

This should show that the run stopped in the **display\_demo()** function.

11. Show the new local environment:

```
simics> sym-list -locals
```

Which should look like this:

Symbol	Kind	Type	Address	Size
aplen	local	int	0x00007fffd9931cf0	4
c	local	char	0x00007fffd9931ce7	1
clen	local	int	0x00007fffd9931cf8	4
displaystring	argument	const char *	0x00007fffd9931cc8	8
g	local	const char[8][8] *	0x00007fffd9931cd8	8
i	local	int	0x00007fffd9931cfc	4
plen	local	int	0x00007fffd9931cf4	4
scene	local	uint32_t *	0x00007fffd9931ce8	8

12. List 15 lines of the current source around the currently executed line of code.

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

Which should point at the print statement:

```

261  int      clen; // char len
262  int      plen, aplen; // pixel len, actual pixel len
263  int      i;
264  char     c;
265  glyph_t  *g;
266
267  // Waiting to start...
-> 268  printf("Press button A on panel to start - scrolling text '%s'!\n", displaystring);
269
270  //
271  // Allocate a buffer to hold the whole set of pixels to display
272  // one longer than the length of the string, to support cheating
273  // wrap-around scrolling
274  //
275  clen = strlen(displaystring);
```

13. Press Enter to show the next 15 lines:

```
simics> <ENTER>
```

The output continues:

```

276  plen = clen * GLYPH_WIDTH;
277  aplen = plen + GLYPH_WIDTH;
278  scene = malloc(sizeof(uint32_t) * GLYPH_HEIGHT * aplen );
279
280  // Create the display image in memory
281  for(i=0;i<clen;i++) {
282      c = displaystring[i];
283      g = glyph_lookup(c);
284      add_glyph_to_scene(scene, i*GLYPH_WIDTH, g, aplen);
285  }
286  // Add the wrap-around at the end - i has the value of
287  // one beyond the end of the string
288  c = displaystring[0];
289  g = glyph_lookup(c);
290  add_glyph_to_scene(scene, i*GLYPH_WIDTH, g, aplen);
```

14. Set a new breakpoint on the loop that starts with the comment **“//Create the display image in memory”** (exact line number might differ. Look carefully where that line is in your output).

```
simics> bp.source_line.break filename = demo-one.c line-number = 281
context-query = $dbg_ctx
```

15. Run the simulation forward again.

```
simics> r
```

The simulation should stop on the new breakpoint when execution reaches the top of the loop.

```
[tcf] Breakpoint 3 on execution in context demo-one (tid=297)
display_demo(displaystring=(const char *) 0x7fffd99328af "0123-C...TEST STRI..") at
/root/swbuild/demo-one/demo-one.c:281
281      for(i=0;i<clen;i++) {
```

16. Check the stack backtrace, using an alias for **“bt”**:

```
simics> stack-trace
```

The CLI stack-trace commands show function arguments in addition to the function names.

17. Use the CLI command **step-into** to step through the code – which steps source lines in the current debug context, on the current processor, including into functions called.

Do this until you reach the function **glyph\_lookup()** (five times, most likely). Note: You might hit breakpoint 3 a few more times along the way, because a line breakpoint corresponds to multiple instructions.:

```
simics> step-into
```

Stop when you see code in **glyph\_lookup()**:

```
simics> step-into
[tcf] Breakpoint 3 on execution in context demo-one (tid=307)
simics>
[tcf] Breakpoint 3 on execution in context demo-one (tid=307)
simics>
display_demo(displaystring=(const char *) 0x7fff6cb558af "0123-C...TEST STRI..") at
/root/swbuild/demo-one/demo-one.c:282
282      c = displaystring[i];
simics>
display_demo(displaystring=(const char *) 0x7fff6cb558af "0123-C...TEST STRI..") at
/root/swbuild/demo-one/demo-one.c:283
283      g = glyph_lookup(c);
simics>
glyph_lookup(c=48 '0') at /root/swbuild/demo-one/big-chars.c:491
491      int i=0;
```

18. The program has reached the function **glyph\_lookup()**, in a different source code file called **big-chars.c**. Check the current code location:

```
simics> list
```

19. Step back to the calling function using **finish-function**:

```
simics> finish-function
```

This should get back to **display\_demo()**:

```
display_demo(displaystring=(const char *) 0x7fffd99328af "0123-C...TEST STRI..") at
/root/swbuild/demo-one/demo-one.c:283
283      g = glyph_lookup(c);
```

20. Back in the loop in **display\_demo()**, step over one more line. Use the **step-over** command.

```
simics> step-over
```

This should end here:

```
display_demo(displaystring=(const char *) 0x7fffd99328af "0123-C...TEST STRI..") at
/root/swbuild/demo-one/demo-one.c:284
284      add_glyph_to_scene(scene, i*GLYPH_WIDTH, g, apen);
```

21. Inspect the value of variable **g**. First, check the type:

```
simics> sym-type g
```

Inspecting it will show the data type used to represent the glyphs that are displayed in the string:

```
const char[8][8] *
```

22. Inspect the content of **g**, or more precisely, the array of chars that it points at:

```
simics> sym-value "*g"
```

It looks like this:

```
(const char[8][8]){ ".....", ".BBBBBB.", ".BB..BB.", ".BB.BBB.", ".BBB.BB.", ".BB..BB.",
".BBBBBB.", "....." }
```

This is a graphical representation of the initial zero from the string to display.

23. Remember the string to display? Check it again:

```
simics> sym-string displaystring
```

Which should be:

```
"0123-C...TEST STRING!"
```

24. You can “pretty print” the graphical representation with a bit of inline Python. The empty line after the **print** statement means that you just need to press **<Enter>** to get back to the command-line prompt.

```
simics> @for l in SIM_run_command('sym-value "*g"'):
.....     print(''.join([chr(i) for i in l]))
.....
```

The conversion to **chr()** is needed since the returned value is a list of integer ASCII values, not the actual characters.

```

.....
.BBBBBB.
.BB..BB.
.BB.BBB.
.BBB.BB.
.BB..BB.
.BBBBBB.
.....

```

This looks like a zero rendered on an 8x8 matrix display. Note that the program's internal encoding of colors makes it easier to read, as it uses periods (or spaces) for white pixels instead of a letter. You can find the alphabet that is available in the **bigchars.c** file.

Note that your data could be different depending on the string you provided to the program.

## Looking at memory

25. Next, you want to see how the memory pointed to by **scene** is filled in. To do this, inspect the first 128 (0x80) bytes of memory pointed to by **scene**. Note that the exact addresses may vary.

```
simics> x (sym-value scene) 0x80
```

This will show the malloc-ed memory that **scene** points to. Initially, it is a set of zeroes.

```

ds:0x00d82270  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d82280  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d82290  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d822a0  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d822b0  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d822c0  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d822d0  0000 0000 0000 0000 0000 0000 0000 0000 .....
ds:0x00d822e0  0000 0000 0000 0000 0000 0000 0000 0000 .....

```

The code will change this memory, but it might take a few iterations of the loop for a change to be visible. Each pixel in the display is represented by four bytes in memory, as discussed in the Optional.

26. Do **step-over**, which makes the execution call **add\_glyph\_to\_scene()** and then stop at the top of the loop again.

```
simics> step-over
```

27. When it stops check the memory for changed data. By grouping by 32 bits, each pixel is shown as a unit in the output.

```
simics> x (sym-value scene) 0x80 group-by = 32
```

This shows a few pixels having been updated. Note how the white pixels are rendered as **0xffff\_ff00**.

```
ds:0x013e0270  ffffffff00 ffffffff00 ffffffff00 ffffffff00 .....
ds:0x013e0280  ffffffff00 ffffffff00 ffffffff00 ffffffff00 .....
ds:0x013e0290  00000000 00000000 00000000 00000000 .....
ds:0x013e02a0  00000000 00000000 00000000 00000000 .....
ds:0x013e02b0  00000000 00000000 00000000 00000000 .....
ds:0x013e02c0  00000000 00000000 00000000 00000000 .....
ds:0x013e02d0  00000000 00000000 00000000 00000000 .....
ds:0x013e02e0  00000000 00000000 00000000 00000000 .....
```

The program has not yet got to the point where it is showing the text on the LED. It is setting things up by pre-rendering all the pixel values into a memory buffer.

28. Set a breakpoint on the entry to the function called **scroll\_loop()** in order to get to the code that actually draws to the LED display.

```
simics> bp.source_location.break scroll_loop
```

29. Check the list of breakpoints again:

```
simics> bp.list
```

The output should show four active breakpoints:

ID	Description	Enabled	Oneshot	Ignore count	Hit count
1	break on magic instruction (0) on any processor	true	false	0	1
2	execution of display_demo	true	false	0	1
3	execution of demo-one.c:281	true	false	0	4
4	execution of scroll_loop	true	false	0	

30. Disable the breakpoint on line 281 (the loop), in order to not get stopped all the time:

```
simics> bp.disable 3
```

31. Run the simulation:

```
simics> r
```

32. Go to the **LED Panel** and click the button **A** to start the scrolling.

33. The simulation should stop on the breakpoint on **scroll\_loop()**.

```
[tcf] Breakpoint 4 on execution in context demo-one (tid=297)
scroll_loop(scene=(uint32_t *) 0xd82270, ap1en=0xb0) at /root/swbuild/demo-one/demo-one.c:194
194         int x = 0;
```

## Assembly-code debug

At this point, it is time to dive into assembly code and look a bit more at the low-level state of the system.

34. To see the assembly for the current source code from the command line, go to the command line. Use the **-d** flag to the **list** command:

```
simics> list -d
```

35. To single-step instructions from the command line, use the command **si** (short form of **step-instruction**) four times.

```
simics> si
```

36. The disassembly should show that you have progressed 4 instructions (there should not have been a loop). The first instruction shown in the listing now is the same as the last in the listing above.

```
simics> list -d
```

37. To see register changes as instructions are executed, use the **-r** flag to the **si** command. Try this with a few more instructions. Some should show updates to registers.

```
simics> si -r
```

38. Place a breakpoint on the function **requestredraw()**. This function interfaces directly with the hardware.

```
simics> bp.source_location.break requestredraw context-query = $dbg_ctx
Breakpoint 5: 0x5 (planted)
```

39. Run the simulation:

```
simics> r
```

40. Execution should stop at the start of **requestredraw()**.

41. The function will manipulate the values of hardware control registers. Check the current value of the registers dealing with display updates:

```
simics> print-device-regs machine.training_card.master_bb.bank.regs
pattern = "update_display*"
```

The initial state should be:

Offset	Name	Size	Value
16	update_display_request	4	0
20	update_display_status	4	1
24	update_display_interrupt_control	4	0

42. To get to the point where the code updates the **update\_display\_request** register, set a register bank access breakpoint on the **master\_bb** device. The breakpoint could also

be set on just a single bank, but in this way you do not need to know much about the register banks of the device:

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

43. Run the simulation until the hardware is accessed:

```
simics> r
```

44. When the simulation stops, Simics prints a message that there was a write to the register at offset 0x10 (decimal 16):

```
[machine.training_card.master_bb.bank.regs] Breakpoint 6: machine.training_card.master_bb.bank.regs  
write at offset=0x10 size=0x4 value=0x1 ini=machine.mb.cpu0.core[1][0]
```

45. Check the register values again:

```
simics> print-device-regs machine.training_card.master_bb.bank.regs  
pattern = "update_display*"
```

Note the register value changes. There is change from **0** to **1** for **update\_display\_request**, and from **1** to **0** for **update\_display\_status**. The status register will change back to **0** once the redraw is complete.

Following this, the code will loop, polling **update\_display\_status** until the hardware has redrawn the LED and changed the value of the register back to **1**. The software is not involved in sending values to the LEDs, it just waits for the hardware controller to finish the job.

46. Remove all breakpoints to avoid unnecessary stops in the continued investigation:

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

47. Check the current stack trace again.

```
simics> stack-trace
```

You should be in the **write32** function. The next frame up should be **requestredraw**.

48. Execute **finish-function** twice, to first finish **write32** and then **requestredraw**.

```
simics> finish-function  
simics> finish-function
```

The second invocation should end here:

```
scroll_loop(scene=(uint32_t *) 0xd82270, aplen=0xb0) at /root/swbuild/demo-one/demo-one.c:204  
204         if(gettoggle(0)==0) {
```

49. Check the register values again:

```
simics> print-device-regs machine.training_card.master_bb.bank.regs  
pattern = "update_display*"
```

The register **update\_display\_request** should be **0**, and **update\_display\_status** should be **1**.



50. What does the “1” in the register `update_display_status` mean? Check the detailed register information:

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

It means that the “done” bit is set, i.e., that the hardware control unit has completed a display update operation.

```
Current status of display update operation  
[machine.training_card.master_bb.bank.regs.update_display_status]  
  
      Bits : 32  
      Offset : 0x14  
      Value : 1  
  
Bit Fields:  
  reserved @ [31:1] : 00000000000000000000000000000000  
    done @ [0:0] : 1 "Set when a draw operation has  
                    completed"
```

51. **Quit** the simulation session.

This concludes the basic debug lab – where you have learned how to look at the system state with variables, memory, registers, how to set breakpoints, and how to run and step forward.

## F. Optional. Modify program state & more debugger command line commands

The **demo-one** program hides a small Easter egg. In this lab, you will use the debugger to modify the program state and expose the hidden feature. This lab also shows some additional features, like how to handle global variables.

1. Start a new simulation session with the automatic debug script you created in section D.

```
[C:] simics[.bat] targets/simics-user-training/lab-005-auto-debug.simics
```

2. Run the simulation forward and wait for the magic breakpoint to be hit.

```
simics> r
```

3. Check which file contain the function **main**.

```
simics> sym-file main
"/path/to/project/targets/simics-user-training/target-source/demo-one/demo-one.c"
```

4. Open the file in a text editor of your choice.
5. Find the word **scrollstring**:

```
// string to scroll
const char * scrollstring = NULL;
```

You will find it is the name of a variable in **main()** that holds the string that is sent to the **display\_demo()** function to be displayed.

6. Keep searching to see more places where it is referenced. One of them looks a bit funny:

```
if(eeegg) scrollstring = empty;
```

7. It seems this line has not had any effect previously. To understand why, look at the value of **eeegg**.

```
simics> sym-value eeegg
```

Which indicates that it is false.

### Change the value of a global variable in complicated way

Check what happens if you change the value of the variable. From the command-line, this can be done using the set command to change memory contents. There is also the shortcut "**sym-write**" command, but the purpose of this exercise is to understand how to modify memory based on debug information.

8. The address of the variable can be retrieved using **sym-address**:

```
simics> sym-address eeegg
```

9. Addresses in decimal look silly. Change how numbers are displayed using the **output-radix** command:

```
simics> output-radix 16 4
```

10. Display the address again, this time in hex:

```
simics> sym-address eeegg
```

11. Note that this address is a virtual (logical) address, as defined by the current processor state for the currently running software. Check the current processor, in the same way that you did before:

```
simics> pselect
```

This will return an answer like "machine.mb.cpu0.core[1][0]".

12. Check the size of the variable to know how much memory to look at.

```
simics> sym-list substr = eeegg
```

This shows data for just this variable:

Symbol	Kind	Type	Address	Size
eeegg	global	enum _Bool	0x00000000004050b8	1

13. Use the **x** command again to inspect the memory. It works on the virtual memory of the current processor, unless instructed otherwise.

```
simics> x (sym-address eeegg) 1
```

The output should be a line of zeroes, since this memory is all initialized to zero.

```
ds:0x004050b0      00      .
```

14. Next, you need to modify the variable. This can be done using either physical or logical addresses. In this case, use physical addresses as that shows how to get from software virtual addresses to hardware physical addresses.

The **l2p** (with a lower-case L at the start) command does a logical-to-physical translation using the current processor. Look up the physical address of the variable:

```
simics> l2p address = (sym-address eeegg)
```

15. The **l2p** command can also be used within the namespace of a particular processor, to inspect its mapping – but for the most common debug tasks, the current processor is what you want.

Check this by doing l2p on both processor cores in the system:

```
simics> l2p machine.mb.cpu0.core[0][0] (sym-address eeegg)
simics> l2p machine.mb.cpu0.core[1][0] (sym-address eeegg)
```

The core that is not the current processor should return “**No virtual to physical address translation found**”. The other should show the same result as above.

16. Use the physical address returned from **l2p** with the **x** command, using the **p:** prefix. Note that your address will likely be entirely different from what is shown here:

```
simics> x p:0x00_something 1
```

Most commands that deal with addresses accept **p:** and **l:** to mark physical and logical addresses, respectively.

17. You can also stack the CLI expressions:

```
simics> x p:(l2p (sym-address eeegg)) 1
```

18. Change the value of the variable using the physical address and the physical memory of the processor. Find the physical memory by looking at the **physical\_memory** attribute of the current processor:

```
simics> (pselect)->physical_memory
```

This result should be something like “**machine.mb.cpu0.mem[0][0]**”, with the index at the end matching the current processor core. As you saw back in Lab 003, each processor core has its local memory map defined.

19. To change the value, use the **write** command on the memory space. Write needs to know the address to write to, the value to write, and the size of the write. As we saw earlier, the size is one byte.

Use the knowledge collected to create a command like that below. Note that there is no need to mark the address with **p:**, since a memory space only understands physical addresses:

```
simics> machine.mb.cpu0.mem[0?][0].write 0x00_something 1 1
```

Note that your physical address and core number might be different, it depends on the operating system state and precisely when and how you started the **demo-one** program.

Using no hard-coded names (except **eeegg**) you can do it like this – which is more appropriate for automation in a script, but a bit harder to construct interactively:

```
simics> (pselect)->physical_memory.write ((pselect).l2p address = (sym-address eeegg)) 1 1
```

20. Check that this did indeed modify the value:

```
simics> sym-value eeegg
```

21. Using the **x** command to look directly at the memory (use up-arrow in the command line to recall the previous commands you entered):

```
simics> x (sym-address eeegg) 1
```

The value in memory should show as **01**.

22. Clear out any old breakpoints to avoid getting stopped.

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

23. Run the simulation:

```
simics> r
```

24. Press **button A** in the **System Panel** to get the program going.
25. Check that you see the hidden message scroll in the LED display in the System Panel.
26. Press **button B** in the **System Panel** to stop the program.
27. **Quit** the simulation session.

## G. Optional. Break on program start

The approach used in the lab to get into the program and start debugging using a magic instruction can be considered “cheating” to some extent. The magic instruction was used in order to provide an example of how magic instructions can be used, and to let you focus on the essentials of the debugger. In this optional section, you will make the simulation stop when the demo program starts using mechanisms that do not rely on modifying the target software.

1. Start a new simulation session with the script you created in section D (**lab-005-auto-debug.simics**):

```
[${C:>} simics[.bat] targets/simics-user-training/lab-005-auto-debug.simics
```

2. Do NOT run the simulation!
3. Before running the simulation, remove the magic breakpoint. First, check the breakpoints that have been set:

```
simics> bp.list
```

4. Then delete the magic breakpoint listed above, using its ID. Most likely it is 1:

```
simics> bp.delete 1
```

5. Run the simulation forward.

```
simics> r
```

The simulation will *not* stop when the program starts, since the magic breakpoint is disabled.

6. Go to the **LED Panel** and click **button A** to start the demo program.
7. Then click **button B** to quit the demo program.

The simulation should still be running.

8. Go to the command line, and set up an operating system awareness (OSA) breakpoint on the demo-one program being switched in. This means that the simulation will stop when Linux has created the new process and is scheduling it:

```
simics> bp.os_awareness.break machine.software -active node-pattern =  
"name='demo-one'" -once
```

9. Go to the target **Serial Console** window and start **demo-one** again (press <UP-ARROW> to get it via Linux command-line history).

```
# ./demo-one "0123-C...TEST STRING!" /sys/bus/pci/devices/0000\:02\:00.0
```

10. The simulation should stop when the program is starting up. Printing a message like this:

```
Breakpoint 2: Break on activation of nodes that match 'name='demo-one'' in 'machine.software'
Setting new inspection object: machine.mb.cpu0.core[1][0]
[machine.software] Breakpoint 2: machine.software nodes matching 'name='demo-one'' became active.
Now debugging /Linux/Userspace/pid=308/tid=308 on machine.software
??()
<illegal memory mapping>
```

11. List what the processor is about to execute (exact addresses can differ).

```
simics> list -d
```

It looks like this:

```
-> cs:0x00007f5d29361100 <whole instruction not mapped>
    cs:0x00007f5d29361101 <whole instruction not mapped>
    cs:0x00007f5d29361102 <whole instruction not mapped>
    cs:0x00007f5d29361103 <whole instruction not mapped>
    cs:0x00007f5d29361104 <whole instruction not mapped>
```

The reason for this is that the program code has not yet been loaded into memory, and virtual memory pages have not yet been set up for the process. OS awareness has intercepted the very jump to the very first instruction of the program, where Linux has just created the new process. At this point, the Linux kernel will take a series of page faults that will bring in the executable code and data segments into memory.

That process is interesting, but not very useful for debugging software.

12. To skip through the initialization and get to the actual program code, set a breakpoint on the **main()** function. Use the **source\_location** breakpoints of the breakpoint manager:

```
simics> bp.source_location.break "main"
```

13. Run the simulation forward.

```
simics> r
```

The simulation should stop rather soon, inside the **main()** function of the program. The debugger knows about the functions, since the debug information was set up by the script.

14. At this point, you could do the debug exercise again – the program is stopped at almost the same point as when using the magic instruction.

Or quit the simulation.