Intel® Simics® Simulator New User Training

Lab 002 – The Command-Line Interface & Scripting

Lab 002 – Intel® Simics® Simulator Command-Line Interface & Scripting

When a simulation is running, you can use the command-line interface (CLI) to interact with the simulation. Graphical User Interfaces (GUI) for the Intel® Simics® Simulator provide some features for ease-of-access, but if you want all the simulator features, you will need to use the CLI. In addition, simulations are usually started using CLI scripts. It is therefore important to be able to understand and use the CLI.

A. Start an empty session

We will start working with CLI in an empty session – i.e., a session with no target machine loaded.

1. Open a terminal in your project directory. If in doubt, remember that ISPM can help you.
2. Start an empty session by not passing any arguments to the **`simics[.bat]`** script.

   In a Windows CMD environment:

   ```
   C:\...> simics.bat
   ```

   In a Linux environment:

   ```
   $ ./simics
   ```

   As before, the terminal will turn into the simulator command line, and commands will be entered at the blue **`simics>`** prompt.

   ```
   Intel® Simics® Simulator 7 (build 7059 linux64) © 2024 Intel Corporation

   Use of this software is subject to appropriate license.
   Type 'copyright' for details on copyright and 'help' for on-line documentation.

   simics>
   ```

B. Basic Command-Line Interface (CLI) operation

Tab-completion at the CLI

1. To make life easier, the CLI performs command auto-completion when you press the **TAB** key after having typed some initial letters in a command or object name.

   In the CLI, type the beginning of a command as shown below:

   ```
   simics> list-m <TAB>
   ```

   Which expands to

   ```
   simics> list-modules
   ```

   Then press **<ENTER>** to run the command.

2. If more than one possible auto-completion option exists, a single **TAB** keystroke will not complete the command. However, if you press the **TAB** key twice, the CLI displays a list of all possible matches. Try this with **`list-`**:

   ```
   simics> list- <TAB><TAB>
   ```

   The CLI displays a list of commands beginning with **`list-`**:

   ```
   simics> list-
   list-attributes                list-instrumentation-groups  list-processors
   list-bookmarks                 list-instrumentation-tools   list-processors-summary
   list-checkpoints               list-interfaces              list-script-branches
   list-classes                   list-length                  list-sections
   list-commands                  list-modules                 list-segments
   list-components                list-notifier-subscribers    list-session-comments
   list-debug-contexts            list-notifiers               list-simics-search-paths
   list-device-regs               list-object-references       list-targets
   list-directories               list-objects                 list-telemetry-classes
   list-failed-modules            list-packages                list-thread-domains
   list-hap-callbacks             list-persistent-images       list-transactions
   list-haps                      list-port-forwarding-setup   list-variables
   list-hypersim-patterns         list-preferences             list-vars
   list-instrumentation           list-presets
   list-instrumentation-callbacks list-prioritized-packages
   simics> list-
   ```

   a. Tab out to form the command **`list-attributes`**. Once a command is on the command line, tab-completion will show the available arguments and flags:

   ```
   simics> list-attributes <TAB><TAB>
   ```

   Results in:

   ```
   simics> list-attributes
   -i  -show-description  attribute-name =  class =  object =  substr =
   ```

3. For some arguments, tab-completion will also show available values when they can be enumerated. Try this with the **`object=`** argument of **`list-attributes`**:

   ```
   simics> list-attributes ob <TAB>
   ```

4. This will expand to "**`object =`**". Use tab-completion to show available values:

```
simics> list-attributes object = <TAB><TAB>
```

Listing all the objects in the simulation (not many since we started a session without a target machine):

```
simics> list-attributes object =
bp                    bp.exception  bp.notifier         bp.time      sim.cmdline
bp.bank               bp.gfx        bp.os_awareness     breakpoints  sim.gui
bp.console_string     bp.hap        bp.source_line      params       sim.rexec
bp.control_register   bp.log        bp.source_location  prefs        sim.tlmtry
bp.cycle              bp.magic      bp.step             sim          sim.transactions
bp.cycle_event        bp.memory     bp.step_event       sim.cli
```

5. Use tab-completion to form the name of the **`prefs`** object:

```
simics> list-attributes object = pr <TAB>
```

Then press **<ENTER>** to run the command.

You should see a set of attributes starting like this:

```
simics> list-attributes object = prefs
┌──────────────────────┬──────┬────────┬─────┬────────────┐
│      Attribute       │ Type │  Attr  │Flags│Value       │
├──────────────────────┼──────┼────────┼─────┼────────────┤
│cli_table_border_style│s     │Optional│     │"thin"      │
│default_log_endianness│s     │Optional│     │"little"    │
│enable_multithreading │b     │Optional│     │TRUE        │
│force_ipv4            │b     │Optional│     │FALSE       │
...
```

Help at the command line

6. The Intel Simics simulator also contains a rich **help system** that documents many aspects of the tool, including command-line commands.

In the CLI, execute the following command to display the various help categories.

```
simics> help
```

The CLI will print something like the following. The precise set of categories vary with the Intel Simics simulator products, packages, and target systems installed. The help system on the command-line is dynamically extended with any new command or modules that gets loaded or defined at run time.

```
simics> help
The help command shows information on any topic, like a command, a class, an object, an interface, a
hap, a module, an attribute or a function or type from the Simics API.

To get more information about the help command, type help help.

Type help category to list the commands for a specific category. Here is a list of command
categories:

  Breakpoints    Debugging  Help            Logging  Networking   Probes      Recording
  CLI            Disks      Image           Matic    Notifiers    Processors  Registers
  Components     Execution  Inspection      Memory   Parameters   Profiling   Snapshots
  Configuration  Files      Instrumentation Modules  Performance  Python      Tracing

The complete documentation may also be read in a web browser by running the documentation script
found in the Simics project folder.
```

7. Type **help** followed by a category to display a list of available commands for the category you specified. For example, look at the **Logging** category:

```
simics> help Logging
```

Requesting help on the "Logging" category will result in output similar to this (the details will vary with Intel® Simics® simulator version and installed packages, as always):

```
simics> help Logging
Commands available in the "Logging" category:

<conf_object>.log-group   enable/disable a log group
<conf_object>.log-level   set or get the log level
log                       print log entries for all objects
log-filter                suppress log messages for the object
log-group                 enable/disable a log group
log-level                 set or get the log level
log-setup                 configure log behavior
log-size                  set log buffer size
log-type                  set or get the current log types
```

8. The most common use of the help system is probably to look for help on specific commands or objects. To make this easy, the help system supports tab completion on everything it can provide help on.

Test this for commands.

Type `help` with a partial command name, then press the **TAB** key to complete the command. If more than one possibility exists, press the **TAB** key twice to display all possible matches. Just like with normal CLI tab completion for commands.

```
simics> help load- <TAB><TAB>
```

Which should result in something like this:

```
simics> help load-
load-binary    load-file       load-intel-obj  load-persistent-state
load-coverage  load-intel-hex  load-module     load-target
```

9. Tab-complete out to help on **`load-binary`**:

```
simics> help load-binary
```

Help for a command lists the arguments and flags that the command takes. Often, the same command is available in both name-spaced and global variants, and sometimes it is present on multiple different classes or interfaces.

```
simics> help load-binary
Command load-binary

   Synopsis
      load-binary filename [offset] [-v] [-pa] [-l] [-n]
      <memory_space>.load-binary filename [offset] [-v] [-pa] [-n]
      <processor_info>.load-binary filename [offset] [-v] [-pa] [-n]

   Description
      Load a binary (executable) file, filename, into the given physical or virtual
      memory space. The supported formats are ELF, Motorola S-Record, PE32 and PE32+.
      For ELF, all segments with a PT_LOAD program header are loaded.

      [...]

      load-binary uses Simics's Search Path and path markers (%simics%, %script%) to
      find the file to load. Refer to The Command Line Interface chapter of the Simics
      User's Guide manual for more information on how Simics's Search Path is used to
      locate files.

   Provided By
      Simics Core

   See Also
      load-file, add-directory
```

10. If you do not know the name of a command but have an idea for what it does, the help system provides a way to search the content of the help text for text matches. In the next section, you will execute a command file. Execute the **help-search** command to find the command that will run a command file.

    `Help-search` (also known as apropos) searches in the help text of the commands, and thus broad searches often return very many matches.

    Try this for a general term like "**`load`**":

```
simics> help-search load
```

    The resulting list of hits is long, as would be expected.

11. To be more specific, you can try searching for regular expressions. For example, find all commands where the help text refer to "**`load`**" followed by "**`file`**":

```
simics> help-search -r ".*load.*file.*"
```

This provides a list of hits that might be more appropriate. In general, searching might be easier to do in the help system in the browser.

12. Another way to find commands is to search on their name (only). The **`list-commands`** command lists all commands. Use its **`substr`** argument to only search for command names that contain the given string. Try it with "**`run`**":

```
simics> list-commands substr = run
```

This lists the matching commands, along with a short description string for each.

```
┌───────────────────────────────────────────────┬───────────────────────────────────────────┐
│                    Command                    │             Short Description             │
├───────────────────────────────────────────────┼───────────────────────────────────────────┤
│<bank_instrumentation_subscribe>.bp-run-       │run until specified device access occurs   │
│until-bank                                     │                                           │
├───────────────────────────────────────────────┼───────────────────────────────────────────┤
│<bp-manager.bank>.run-until                    │run until specified device access occurs   │
├───────────────────────────────────────────────┼───────────────────────────────────────────┤
│<bp-manager.con-gfx>.run-until                 │run until graphical break matches          │
├───────────────────────────────────────────────┼───────────────────────────────────────────┤
...
```

Capturing command-line output

13. You can capture all the commands entered and all the output printed on the command-line to a file, which is handy if you want to review what happened in a session later. Test this functionality by saving the rest of this simulation session to a file.

    Enter the command (note that calling logs `*`**`.log`** is a good idea as that makes them easy to find and most text editors consider such files as text files they can open):

```
simics> start-command-line-capture filename = lab002.log
```

Controlling the command-line number format

14. The **`output-radix`** command is used to specify the base radix that used for display as well as controlling the grouping of digitals for numbers.

To set the output to hexadecimal with 4-digit grouping:

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

15. Test it by entering some numbers:

```
simics> 0xdeadbeef
```

Which should result in an output with digit grouping applied:

```
0xdead_beef
```

16. Try a decimal number:

    ```
    simics> 100000
    ```

    Which should be converted to a hexadecimal number:

    ```
    0x0001_86a0
    ```

17. Not all commands use the default number format for their output. For example, commands tend to print addresses in hexadecimal and time in decimal. The simulator maintains a digit grouping setting for each base to more nicely format such output.

    Check the digit grouping settings:

    ```
    simics> digit-grouping
    ```

    The output is something like this (your settings might vary):

    ```
    ┌─────┬──────┐
    │Radix│Digits│
    ├─────┼──────┤
    │    2│     0│
    │    8│     0│
    │   10│     0│
    │   16│     4│
    └─────┴──────┘
    ```

18. Set the digit grouping for decimal numbers to 3, without changing the default output radix:

    ```
    simics> digit-grouping 10 3
    ```

19. Check the digit grouping settings again:

    ```
    simics> digit-grouping
    ```

    Decimal number should now have "3" as its digit grouping.

20. Test the format of decimal numbers using the **dec** command (which prints its argument in decimal):

    ```
    simics> dec 1000000
    ```

    This should look like:

    ```
    "1_000_000"
    ```

21. Set base 10 as the default format:

    ```
    simics> output-radix 10
    ```

22. Test it:

```
simics> 0xdeadbeef
```

Which should result in:

```
3_735_928_559
```

Using the **`output-radix`** command without a grouping means the current grouping for the specific radix is used.

23. To use the same current output base and digit groupings in your next session, use the command **`save-preferences`**:

```
simics> save-preferences
```

24. Check the preference values:

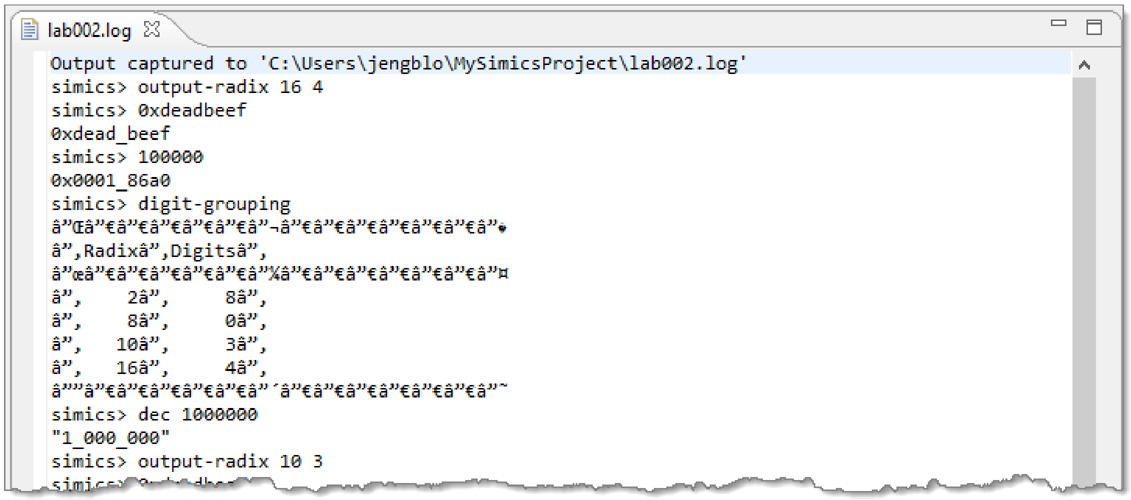
```
simics> list-preferences
```

Resulting in:

```
[...]
         output_grouping: [0, 0, 3, 4]
            output_radix: 10
[...]
```

Command-line output text encoding and log files in editors

Many command-line commands use UTF-8 text output to provide more easily read output. This includes commands that print tables (like **`digit-grouping`** shown above) and commands that present hierarchical structures (like **`list-objects -tree`**). When viewing a log file saved from a simulation session in an editor that defaults to another encoding, this can result output that looks like garbage. For example:



```
Output captured to 'C:\Users\jengblo\MySimicsProject\lab002.log'
simics> output-radix 16 4
simics> 0xdeadbeef
0xdead_beef
simics> 100000
0x0001_86a0
simics> digit-grouping
â”Œâ”€â”€â”€â”€â”€â”¬â”€â”€â”€â”€â”€â”€â”•
â”‚Radixâ”‚Digitsâ”‚
â”œâ”€â”€â”€â”€â”€â”¼â”€â”€â”€â”€â”€â”€â”¤
â”‚    2â”‚     8â”‚
â”‚    8â”‚     0â”‚
â”‚   10â”‚     3â”‚
â”‚   16â”‚     4â”‚
â””â”€â”€â”€â”€â”€â”´â”€â”€â”€â”€â”€â”€â”˜
simics> dec 1000000
"1_000_000"
simics> output-radix 10 3
```

The solution is to **set the editor to interpret the files as UTF-8**. How this is done varies between editors. Most modern editors will default to UTF-8.

Table formatting

Tables are very common in the command-line. You can control the default formatting tables using the "**`table-border-style`**" command. The current border style is a save-able preference.

25. Set the style to **`thick`**, and test it using digit-grouping:

```
simics> table-border-style thick
simics> digit-grouping
```

The output has a thick border on the outside, and between the header and the table:

```
simics> digit-grouping
┏━━━━━━┯━━━━━━┓
┃Radix │Digits┃
┣━━━━━━┿━━━━━━┫
┃     2│     0┃
┃     8│     0┃
┃    10│     3┃
┃    16│     4┃
┗━━━━━━┷━━━━━━┛
```

26. Set the style to **`borderless`** to not show any borders at all. This is very compact, but it might also make the tables harder to read. Try it:

```
simics> table-border-style borderless
simics> digit-grouping
```

Showing:

```
Radix  Digits
    2       0
    8       0
   10       3
   16       4
```

To use the same style in future simulation runs, use the **`save-preferences`** command.

27. In case some output system or editor just cannot handle UTF-8 but you still want to have borders in tables, use the **`ascii`** format.

```
simics> table-border-style ascii
simics> digit-grouping
```

28. The rest of the labs are performed using the **`thin`** borderstyle, which is also default. Reset the setting to this standard:

```
simics> table-border-style thin
simics> save-preferences
```

Paths in the Intel Simics Simulator

To run script files and locate other files from inside of simulation sessions, it is necessary to have an understanding for how the CLI handles paths.

29. A simulation session has a current working directory, which you can check by using the command **pwd**:

```
simics> pwd
```

The output should match the location of your project.

30. The command **`lookup-file`** can be used to convert a relative path, or a path beginning with **`%simics%`** or **`%script%`**, to an absolute path. This will tell you where the simulator finds a certain file (or give an error in case it cannot be found). Try this first with a relative path from the current directory, looking for the target YML file you (indirectly) used in the first lab when you started the target:

```
simics> lookup-file targets/simics-user-training/001-qsp-
training.target.yml
Not in search path: 'targets/simics-user-training/001-qsp-
training.target.yml'
```

You notice that the file cannot be found, because in the absence of **`%simics%`** or **`%script%`** the simulator only searched in your current working directory, which is your project right now.

`lookup-file` does not support tab completion for getting to files, since it does not make sense given that it can also handle non-existing files and paths.

Note that you should always use **forward-slash** (`/`) as a directory separator when writing paths by hand. On Windows host, the simulator will handle the conversion to Windows path format. This makes your scripts portable across host types.

31. To avoid scripts depending on their location on disk, the recommendation is to use the **`%simics%`** file prefix to locate script files from your simulator installation and project. In this way, even if the project location changes or the installation is updated, the scripts will still work.

Try this method of finding the file **`001-qsp-training.target.yml`** and assign the location to a CLI variable. Note that the parentheses **`()`** around **lookup-file** are necessary to make the CLI evaluate the command and not just consider the input as a string.

```
simics> $x = (lookup-file "%simics%/targets/simics-user-training/001-
qsp-training.target.yml")
```

32. Check the path returned:

```
simics> $x
```

This should be the path to the script file in your installed training package, since the installed packages are searched after the project is searched.

33. To find out where **`%simics%`** searches, and in which order, use the command **`list-simics-search-paths`**:

```
simics> list-simics-search-paths
```

This will print the location of the project you are using, followed by the installed packages. The simulator will search the given directories in order and pick the file from the first matching location.

It is possible to control the search order by using the "**`set-prioritized-package`**" command – but that should only be needed if multiple packages define the same scripts, image files, or binary models. This is often used in target scripts to avoid problems with possibly conflicting packages.

34. Do NOT quit the session, it will be used in the next lab section.

C. Python code on the simulator command line

Continue in the same session as before and explore a bit of inline Python.

1. Check the current Python version that is being used inside the simulator:

```
simics> @sys.version
```

Interfacing with the command-line environment

2. Check the value of the variable **`$x`** from inside of Python:

```
simics> @simenv.x
```

3. Call a function on the string, inside of Python:

```
simics> @simenv.x.swapcase()
```

4. Create a string containing the current date and time, using the **`time.strftime()`** function. Long year-month-day date format, and then the current hour, minutes, and seconds in 24-hour format:

```
simics> @time.strftime("%Y-%m-%d [%H:%M:%S]")
```

5. Assign the date string to a Python variable:

```
simics> @t = time.strftime("%Y-%m-%d [%H:%M:%S]")
```

6. Check the value of the Python variable:

```
simics> @t
```

7. Check if there is a CLI variable called **`t`**:

```
simics> $t
```

   This should print:

```
No CLI variable "t"
```

8. Assign the date string in the Python variable **`t`** to the CLI variable **`t`** using **`simenv`**. Check the value of **`$t`** after the assignment:

```
simics> @simenv.t = t
simics> $t
```

9. Use a date string to set the **`start-command-line-capture`** filename, using the inline Python expression evaluation facility in CLI (i.e., put the Python expression inside "backticks", ``). Note that the date string format you used previously is not usable in a file name (colons are not really appreciated by the host operating system).

   Before starting the new capture stop the capture you started earlier (without a file argument, stop-command-line-capture will stop all current captures):

```
simics> stop-command-line-capture
```

10. Start a new log file using Python to generate a date-stamped log:

```
simics> start-command-line-capture ("lab002-" +
`time.strftime("day_%Y-%m-%d_time_%H-%M-%S")` + ".log")
```

Note that this will create a new output file right next to the previous one.

The name of the file is printed by the start-command-line-capture command:

```
simics> start-command-line-capture ("lab002-" +
`time.strftime("day_%Y-%m-%d_time_%H-%M-%S")` + ".log")
Output captured to 'C:\Users\jengblo\MySimicsProject\lab002-day_2023-04-19_time_14-58-
44.log'
```

If you want to stop capture only to this file in the future, make a note of the name. Or save it in a CLI variable since the **`start-command-line-capture`** command returns the name of the file it captures to.

11. Look at your project in a file browser to see the new file. The CLI also features a built-in file listing function:

```
simics> ls
```

Which will show the files and directories in the current working directory, including the log files. Something like the below, note the two log files:

```
simics> ls
.cproject           bin                                     modules
.modcache           compiler.mk                             simics-eclipse.bat
.package-list       config.mk                               simics-gui.bat
.project            doc                                     simics.bat
.project-properties documentation.bat                       targets
.settings           lab002-day_2023-04-19_time_14-58-44.log
GNUmakefile         lab002.log
```

12. Note that you can also generate a timestamp for a file using the CLI **`date`** command instead of inline Python. It uses the same formatting strings as the Python **`time.strftime()`** function, but it also has some easy defaults. To get a string suitable for time-stamping files, use the predefined "**`file`**" format:

```
simics> start-command-line-capture ("lab002-" + (date format=file) +
".log")
```

Note that this means that you have two simultaneous CLI captures running.

Accessing configuration objects

The objects in the Intel Simics Simulator configuration are mapped to the **`conf`** namespace in Python. Even an empty session contains some basic infrastructure objects that can be used to practice.

13. To see the objects in the simulation, use **`list-objects`** (this command will be revisited in more details in later labs):

```
simics> list-objects
```

14. Use the **sim** object as an example. It stores simulation-global information, such as the current project directory. Read the current project from the **project** attribute of the object:

```
simics> sim->project
```

15. In Python, the sim object is found as conf.sim:

```
simics> @conf.sim
```

The return value just shows that this is a simulation object called "**sim**", of class "**sim**":

```
simics> @conf.sim
<the sim 'sim'>
```

16. Access the **project** attribute, as a property of the **conf.sim** object:

```
simics> @conf.sim.project
```

17. Alternatively and more precisely, the attr namespace inside the object holds all attributes – even for the case where they collide with the names of sub-objects in the simulator object hierarchy.

```
simics> @conf.sim.attr
```

18. Tab completion works in Python too:

```
simics> @conf.sim.attr.p <TAB>
```

This should show all attributes beginning with **p** (the precise set might well change between Intel Simics Simulator base package releases):

```
simics> @conf.sim.attr.p
@conf.sim.attr.package_info            @conf.sim.attr.page_sharing_savings
@conf.sim.attr.package_list            @conf.sim.attr.page_sharing_statistics
@conf.sim.attr.package_path            @conf.sim.attr.page_stats
@conf.sim.attr.page_sharing            @conf.sim.attr.ports
@conf.sim.attr.page_sharing_consistent @conf.sim.attr.prioritized_packages
@conf.sim.attr.page_sharing_page_size  @conf.sim.attr.project
@conf.sim.attr.page_sharing_pages      @conf.sim.attr.prompt
```

19. Read the project attribute using this namespace:

```
simics> @conf.sim.attr.project
```

Running CLI commands from Python

When using Python for scripting, it is not uncommon to want to run some CLI commands from inside of Python. Global CLI commands are found in the Python namespace **cli.global_cmds**. Each command is a Python function with one keyword argument for each command-line argument. All dashes are mapped to underscores.

20. Try a simple CLI command:

```
simics> hex 1000000 -p
```

21. To call the command from Python, use Python keyword arguments corresponding to the CLI command arguments. It is not possible to use positional arguments – all arguments have to be named. To find the names of the arguments to command, you can use command-line **`help`** or tab completion, or the Python **`help()`** function. In general, tab completion will always show the correct names of the arguments, while **`help`** can sometimes provide abbreviated information.

    Check help on the hex command:

    ```
    simics> help hex
    ```

    The output shows that the name of the main argument is "**`value`**":

    ```
    Command hex

       Synopsis
          hex value [-u] [-p]
    ...
    ```

    The flag **`-p`** gets translated to an argument called **`_p`**.

22. To see the Python-specific wrapping, use the Python **`help()`** function on the Python-wrapped function:

    ```
    simics> @help(cli.global_cmds.hex)
    ```

    This shows the details on the Python wrapping, including the precise names of all keyword arguments:

    ```
    Help on function hex in module cli_impl:

    hex(*args, **kw)
        Function to run the 'hex' command.
        No positional parameters are accepted.
        The following keyword-only parameter(s) are accepted:
        - value: required, integer
        - _u: optional, flag, default value - False
        - _p: optional, flag, default value - False
    ```

23. With this information you can construct a call to **`cli.global_cmds.hex()`**:

    ```
    simics> @cli.global_cmds.hex(value=1000000, _p=True)
    ```

    This will return the same string as printed by the command.

24. The return value from this command (and most others) can be used in a Python expression. Capture it into a Python variable called **a**:

    ```
    simics> @a=cli.global_cmds.hex(value=1000000, _p=True)
    ```

    No output is shown, since the command value has been consumed.

25. Read the value of the Python variable **a**, to see how the return value is represented:

    ```
    simics> @a
    ```

26. Commands local to an object can be called using the **`cli_cmds`** namespace on the object. To try this in an empty session, you can use the **`sim`** object and its **`info`** command.

    Test the command on the command line:

    ```
    simics> sim.info
    ```

27. In Python, this translates to a Python function:

    ```
    simics> @conf.sim.cli_cmds.info
    ```

    This will return a reference to the function.

28. To actually call the command function, parentheses are needed:

    ```
    simics> @conf.sim.cli_cmds.info()
    ```

    Note that this command (and a few others) do not actually return any value that can be retrieved into Python. Most commands do, but not all.

29. Do NOT quit the session, it will be used in the next lab section.

D. Start a target from the simulator command line

Pretty much everything in the Intel Simics simulator can be done from the command line. This includes setting up new simulator targets. In this lab section, you will start a target machine from the running simulation session, instead of launching a new session and providing a script file at startup.

1. Continuing from the CLI prompt from the previous session…

   At the CLI prompt, build up the following command to launch the same script that you started in the previous lab. Remember to use tab completion to build up the command! ***Do NOT press <ENTER> after the command has been built***!

```
simics> load-t<TAB>
…
simics> load-target s<TAB>
…
simics> load-target "simics-user-training/001-qsp-training" <TAB><TAB>
```

2. The final press of **<TAB>** will show you a list of all the parameters that the target accepts. This is a longer set than you saw when you used **--help** on the target. The reason is that by default, the help only shows parameters with and **advanced** level of 1. Tab-completion shows all, regardless of their **advanced** level.

```
simics> load-target "simics-user-training/001-qsp-training"
namespace =
output:eth_link =
output:service_node =
output:system =
platform:machine:hardware:apic_bus:class =
[…]
platform:persistent_state:writable_state_dir =
preset =
preset_yml =
presets =
simulation:real_time_mode =
training_card:bb_pixel_update_time =
training_card:bb_toggle_check_interval =
training_card:clock_frequency =
training_card:name =
training_card:output:training_card =
training_card:owner =
training_card:pcie_slot =
training_card:use_behavioral_box =
```

3. Set the memory size to a value that you have not used so far. Remember that tab completion is your friend. Use it to complete the somewhat lengthy parameter name. Note that we cannot use unit suffixes in CLI, so you have to use compute the numeric values of “kilo” etc. (this is a known limitation of this way of providing script parameters):

```
simics> load-target "simics-user-training/001-qsp-training"
platform:machine:hardware:memory_megs = 24*1024
```

Press **<ENTER>** to send this command to the simulator.

4. Once the command has completed, check the target setup using the **`print-target-info`** command. Check the size of the memory, to see that the parameter did indeed have effect:

```
simics> print-target-info
```

This should show a 24GiB memory size:

```
QSP x86 with Linux - Simics Training Setup
    ┌──────────┬─────────────────────────┐
    │System    │a QSP x86 chassis        │
    │Processors│2 QSP X86-64, 2000.0 MHz │
    │Memory    │24 GiB                   │
    │Ethernet  │1 of 1 connected         │
    │Storage   │2 disks (208 GiB)        │
    └──────────┴─────────────────────────┘
```

Controlling the run from the command line

5. Run the simulation for a short while using the **`run`** command. The **run** command as you have used it before just starts the simulation running until it is stopped. It can also run the simulation for a specific amount of target time. For the second case, it takes a number and a unit to make it easy to express both large and small time spans.

   To run through the first 4 cycles of the boot:

```
simics> run 4 cycles
```

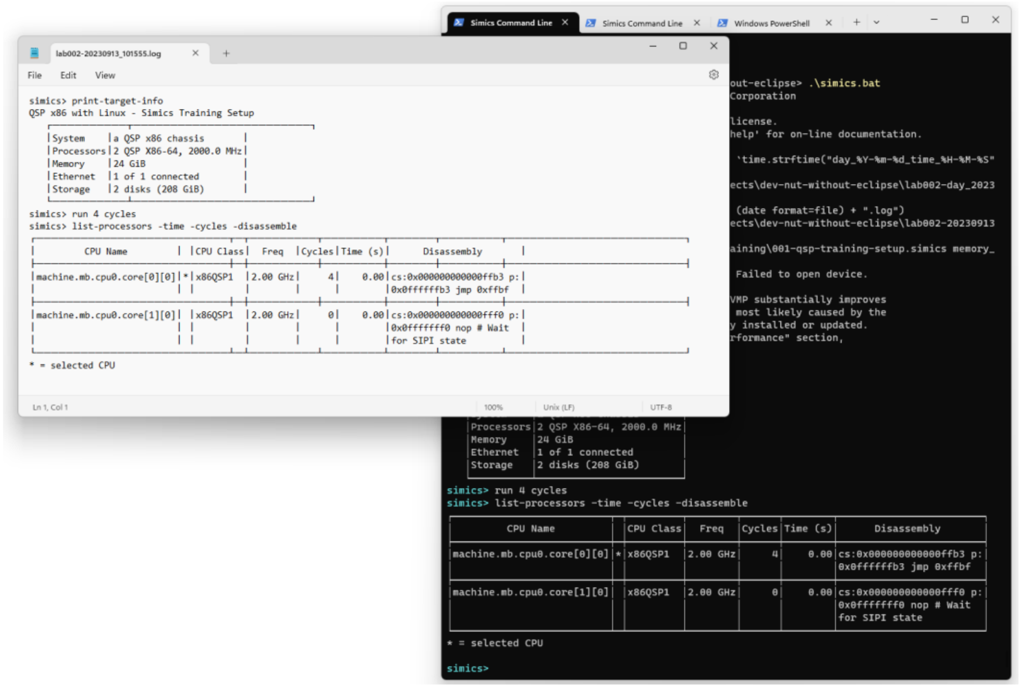
6. When the simulation stops, check the time:

```
simics> list-processors -time -cycles -disassemble
```

This shows the time, cycle count, and current instruction on all processors in the system (as you will learn later, time in the simulation is local to each processor, but the time difference between them is kept within one time quantum).

7. (Re)Open the command-line session log file that you set up in the previous session (`lab002-….log`) in an editor. It should show a mirror of the current command-line session:

   For example:



```
simics> print-target-info
QSP x86 with Linux - Simics Training Setup

    |System    |a QSP x86 chassis     |
    |Processors|2 QSP X86-64, 2000.0 MHz|
    |Memory    |24 GiB                |
    |Ethernet  |1 of 1 connected      |
    |Storage   |2 disks (208 GiB)     |

simics> run 4 cycles
simics> list-processors -time -cycles -disassemble

|         CPU Name         | |CPU Class|  Freq   |Cycles|Time (s)|      Disassembly      |
|machine.mb.cpu0.core[0][0]|*|x86QSP1  |2.00 GHz|     4|    0.00|cs:0x000000000000ffb3 p:|
|                          | |         |         |      |        |0x0ffffffb3 jmp 0xffbf  |
|machine.mb.cpu0.core[1][0]| |x86QSP1  |2.00 GHz|     0|    0.00|cs:0x000000000000fff0 p:|
|                          | |         |         |      |        |0x0fffffff0 nop # Wait  |
|                          | |         |         |      |        |for SIPI state          |

* = selected CPU
```

```
out-eclipse> .\simics.bat
Corporation

license.
help' for on-line documentation.

 'time.strftime("day_%Y-%m-%d_time_%H-%M-%S"

ects\dev-nut-without-eclipse\lab002-day_2023

 (date format=file) + ".log")
ects\dev-nut-without-eclipse\lab002-20230913

aining\001-qsp-training-setup.simics memory_

 Failed to open device.

VMP substantially improves
 most likely caused by the
y installed or updated.
rformance" section,

    |Processors|2 QSP X86-64, 2000.0 MHz|
    |Memory    |24 GiB                  |
    |Ethernet  |1 of 1 connected        |
    |Storage   |2 disks (208 GiB)       |

simics> run 4 cycles
simics> list-processors -time -cycles -disassemble

|         CPU Name         | |CPU Class|  Freq   |Cycles|Time (s)|      Disassembly      |
|machine.mb.cpu0.core[0][0]|*|x86QSP1  |2.00 GHz|     4|    0.00|cs:0x000000000000ffb3 p:|
|                          | |         |         |      |        |0x0ffffffb3 jmp 0xffbf  |
|machine.mb.cpu0.core[1][0]| |x86QSP1  |2.00 GHz|     0|    0.00|cs:0x000000000000fff0 p:|
|                          | |         |         |      |        |0x0fffffff0 nop # Wait  |
|                          | |         |         |      |        |for SIPI state          |

* = selected CPU

simics>
```

8. Run the simulation for a few seconds:

```
simics> run 4 s
```

9. When the simulation stops, check the time again:

```
simics> list-processors -time -cycles -disassemble
```

10. To see the number of instructions executed, add the **`-steps`** argument.

```
simics> list-processors -time -cycles -steps
```

    Note that the steps are far fewer than the cycles, since many instructions take multiple cycles – like waiting for interrupts to happen before continuing and similar. The second core in the system has also been disabled for most of the run, and therefore shows only a few instructions.

11. Let the simulation run until you stop it by using the **`run`** command without any arguments.

```
simics> run
```

    Note that the CLI prompt reads "`running>`" while the simulation is running. This indicates the current state of the simulation.

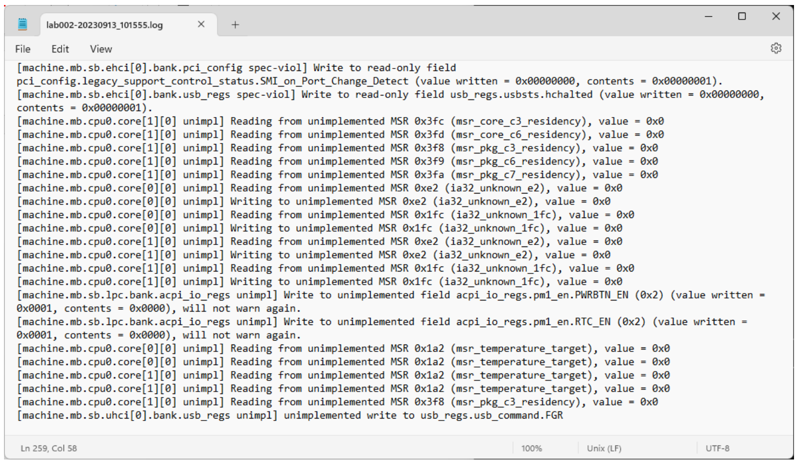
12. Once the target system has booted, stop the simulation by using the **`stop`** command:

```
running> stop
```

13. And quit this session by using the **`quit`** command:

```
simics> quit
```

14. Check the command-line capture files.



```
[machine.mb.sb.ehci[0].bank.pci_config spec-viol] Write to read-only field
pci_config.legacy_support_control_status.SMI_on_Port_Change_Detect (value written = 0x00000000, contents = 0x00000001).
[machine.mb.sb.ehci[0].bank.usb_regs spec-viol] Write to read-only field usb_regs.usbsts.hchalted (value written = 0x00000000,
contents = 0x00000001).
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x3fc (msr_core_c3_residency), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x3fd (msr_core_c6_residency), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x3f8 (msr_pkg_c3_residency), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x3f9 (msr_pkg_c6_residency), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x3fa (msr_pkg_c7_residency), value = 0x0
[machine.mb.cpu0.core[0][0] unimpl] Reading from unimplemented MSR 0xe2 (ia32_unknown_e2), value = 0x0
[machine.mb.cpu0.core[0][0] unimpl] Writing to unimplemented MSR 0xe2 (ia32_unknown_e2), value = 0x0
[machine.mb.cpu0.core[0][0] unimpl] Reading from unimplemented MSR 0x1fc (ia32_unknown_1fc), value = 0x0
[machine.mb.cpu0.core[0][0] unimpl] Writing to unimplemented MSR 0x1fc (ia32_unknown_1fc), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0xe2 (ia32_unknown_e2), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Writing to unimplemented MSR 0xe2 (ia32_unknown_e2), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x1fc (ia32_unknown_1fc), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Writing to unimplemented MSR 0x1fc (ia32_unknown_1fc), value = 0x0
[machine.mb.sb.lpc.bank.acpi_io_regs unimpl] Write to unimplemented field acpi_io_regs.pm1_en.PWRBTN_EN (0x2) (value written =
0x0001, contents = 0x0000), will not warn again.
[machine.mb.sb.lpc.bank.acpi_io_regs unimpl] Write to unimplemented field acpi_io_regs.pm1_en.RTC_EN (0x2) (value written =
0x0001, contents = 0x0000), will not warn again.
[machine.mb.cpu0.core[0][0] unimpl] Reading from unimplemented MSR 0x1a2 (msr_temperature_target), value = 0x0
[machine.mb.cpu0.core[0][0] unimpl] Reading from unimplemented MSR 0x1a2 (msr_temperature_target), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x1a2 (msr_temperature_target), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x1a2 (msr_temperature_target), value = 0x0
[machine.mb.cpu0.core[1][0] unimpl] Reading from unimplemented MSR 0x3f8 (msr_pkg_c3_residency), value = 0x0
[machine.mb.sb.uhci[0].bank.usb_regs unimpl] unimplemented write to usb_regs.usb_command.FGR
```

They should contain all the various log messages printed during the target system boot, as that is also part of the command-line output. Log messages can also be handled separately and directed to output files independent of the general command-line capture. This will be covered in later labs.

E. Script target system actions

Create a custom script with a script branch

Script branches are crucial to automating actions in the simulation. In this exercise, you will build a simple script branch in a custom script. The script will open the first checkpoint you saved, and then automate the actions you took to test the training device.

1. Create a new file named **`lab-002-script-branch.simics`** in **`<your project>/targets/simics-user-training`** and open it in a text editor of your choice.
2. Add the following code to the script:

```
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, then test the driver
    bp.console_string.wait-then-write $con "# " "cat lines.txt > /dev/simics_training_pcie_driver\n"
    # run for one/tenth second to give the hardware time to update
    bp.time.wait-for $cpu 0.1
    # Stop execution
    stop
}
```

   Note that due to how Word prints and exports to PDF, the text from the box above will likely be missing all the spaces used for indentation when pasted into an editor. That does not matter for CLI code, but it does not look pretty. You have to manually fix the indentation to make it look like the example code above.

3. Save the file.
4. Run your script file.

   In a Windows CMD environment:

```
C:\...> simics.bat
targets\simics-user-training\lab-002-script-branch.simics
```

   In a Linux environment:

```
$ ./simics targets/simics-user-training/lab-002-script-branch.simics
```

5. Before you run the simulation, check that the script branch has been created and is active. The **list-script-branches** command will show all branches, what they are waiting on, and the name they were given by the user.

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

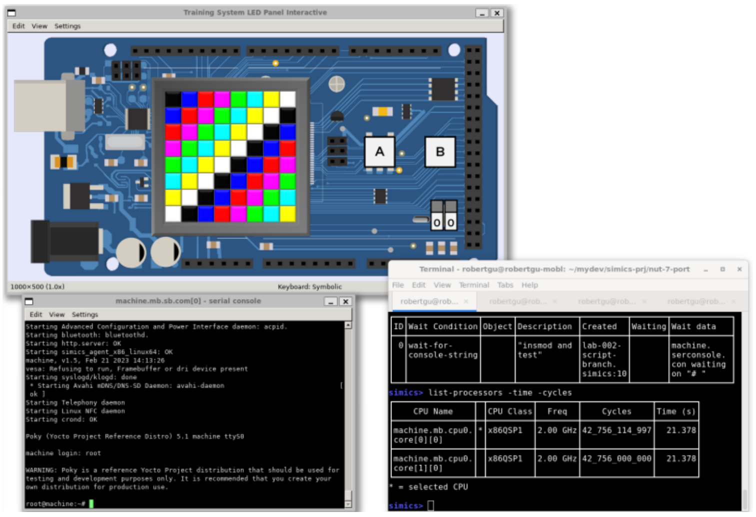
Which should show the script branch from your script.

```
simics> list-script-branches
┌──┬─────────────────┬──────┬───────────────────┬───────────────┬───────┬──────────────────┐
│ID│Wait Condition   │Object│Description        │Created        │Waiting│"Wait data"       │
├──┼─────────────────┼──────┼───────────────────┼───────────────┼───────┼──────────────────┤
│ 0│wait-for-console-│      │"insmod and test"  │lab-002-script-│       │machine.          │
│  │string           │      │                   │branch.simics:10│      │serconsole.con    │
│  │                 │      │                   │               │       │waiting on "# "   │
└──┴─────────────────┴──────┴───────────────────┴───────────────┴───────┴──────────────────┘
```

6. Check the current time of the processors, and see that it matches the state after you took the first checkpoint:

```
simics> list-processors -time -cycles
```

7. Rearrange the windows and so that you can see the Serial Console window as well as the LED Panel window and the command line. For example:


Training System LED Panel Interactive
Edit View Settings
A
B
1000×500 (1.0x)
Keyboard: Symbolic
machine.mb.sb.com[0] - serial console
Edit View Settings
Terminal - robertgu@robertgu-mobl: ~/mydev/simics-prj/nut-7-port
File Edit View Terminal Tabs Help

```
Starting Advanced Configuration and Power Interface daemon: acpid.
Starting bluetooth: bluetoothd.
Starting http.server: OK
Starting simics_agent_x86_linux64: OK
machine, v1.5, Feb 21 2023 14:13:26
vesa: Refusing to run, Framebuffer or dri device present
Starting syslogd/klogd: done
 * Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon                    [
 ok ]
Starting Telephony daemon
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:~#
```

```
┌──┬──────────────┬──────┬────────────┬──────────┬───────┬────────────┐
│ID│Wait Condition│Object│Description │Created   │Waiting│Wait data   │
├──┼──────────────┼──────┼────────────┼──────────┼───────┼────────────┤
│ 0│wait-for-     │      │"insmod and │lab-002-  │       │machine.    │
│  │console-string│      │test"       │script-   │       │serconsole. │
│  │              │      │            │branch.   │       │con waiting │
│  │              │      │            │simics:10 │       │on "# "     │
└──┴──────────────┴──────┴────────────┴──────────┴───────┴────────────┘
simics> list-processors -time -cycles
┌───────────────┬─┬─────────┬────────┬──────────────┬────────┐
│   CPU Name    │ │CPU Class│  Freq  │    Cycles    │Time (s)│
├───────────────┼─┼─────────┼────────┼──────────────┼────────┤
│machine.mb.cpu0.│*│x86QSP1 │2.00 GHz│42_756_114_997│  21.378│
│core[0][0]     │ │         │        │              │        │
├───────────────┼─┼─────────┼────────┼──────────────┼────────┤
│machine.mb.cpu0.│ │x86QSP1 │2.00 GHz│42_756_000_000│  21.378│
│core[1][0]     │ │         │        │              │        │
└───────────────┴─┴─────────┴────────┴──────────────┴────────┘
* = selected CPU

simics>
```

8. Run the simulation:

```
simics> run
```

9. Watch the script type the commands into the target system, in the Serial Console. As a result, the LED Panel should update, and the simulation eventually stop automatically.

10. Check that the script branch finished and disappeared:

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

The result should be an empty list.

Interactive automation

Moving on to some interactive use of scripting features.

11. Stopping the simulation when a certain string appears in the output is useful when scripting. Test this, by setting a breakpoint on the appearance of "**`root@machine`**". Console string breakpoints are managed using the breakpoint manager, found in the **bp** namespace. Each type of breakpoint has its own namespace inside of **bp**.

    To set a breakpoint on a certain string appearing on a certain console, provide the name of the console object, as well as the string to break on:

```
simics> bp.console_string.break object = machine.serconsole.con string =
"root@machine"
```

12. Check that the breakpoint is set:

```
simics> bp.list
```

13. By the way, to check what other types of breakpoints are available, use the bp.list-types command:

```
simics> bp.list-types
```

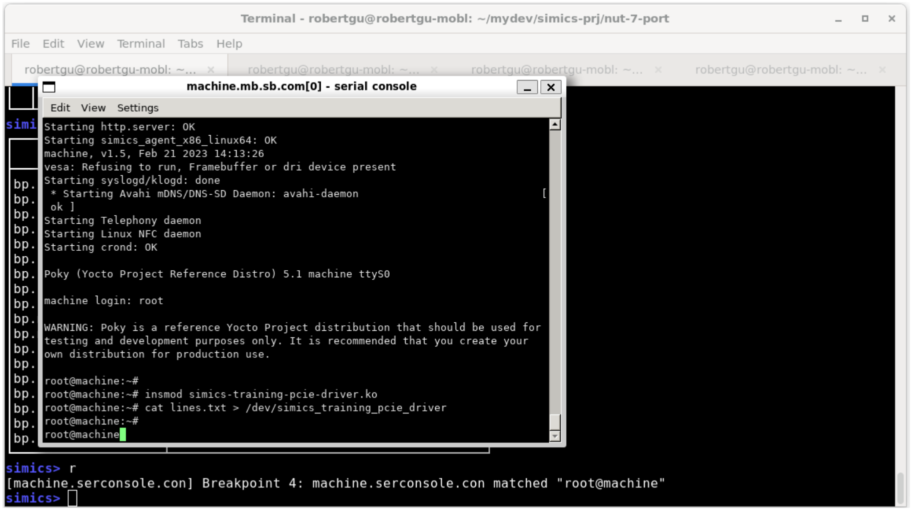
14. Run the simulation forward from the command line. "**`r`**" is the same as "**`run`**".

```
simics> r
```

15. Press **`<ENTER>`** in the **Serial Console** window. You should see the following in the CLI console and the simulation should pause and show the prompt:

```
simics> r
[machine.serconsole.con] Breakpoint 4: machine.serconsole.con matched "root@cl-qsp"
simics>
```

    Note that the breakpoint triggers as soon as the string is seen, in the middle of the output line:


Terminal - robertgu@robertgu-mobl: ~/mydev/simics-prj/nut-7-port
File Edit View Terminal Tabs Help
robertgu@robertgu-mobl: ~...
machine.mb.sb.com[0] - serial console
Edit View Settings

```
Starting http.server: OK
Starting simics_agent_x86_linux64: OK
machine, v1.5, Feb 21 2023 14:13:26
vesa: Refusing to run, Framebuffer or dri device present
Starting syslogd/klogd: done
 * Starting Avahi mDNS/DNS-SD Daemon: avahi-daemon                              [
 ok ]
Starting Telephony daemon
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:~#
root@machine:~# insmod simics-training-pcie-driver.ko
root@machine:~# cat lines.txt > /dev/simics_training_pcie_driver
root@machine:~#
root@machine
```

```
simics> r
[machine.serconsole.con] Breakpoint 4: machine.serconsole.con matched "root@machine"
simics>
```

16. Send some input to the serial console, by using the **`input`** command on the console instead of typing interactively. Note that in order to send a command to the target Linux, it needs to end in a newline. The simulator does not add that automatically, since it cannot assume anything about the current target system context:

    ```
    simics> machine.serconsole.con.input "ls\n"
    ```

    Nothing happens, since the simulation is paused.

17. Run the simulation forward:

    ```
    simics> r
    ```

    The input string will be sent to the target serial console. The command will be executed. And then the simulation will stop again once the target prompt is being printed. This demonstrates how you can prototype scripting actions interactively.

18. Quit this session:

    ```
    simics> quit
    ```

F. Optional. Create a custom command in Python

This section uses Python to create a custom command for the Intel Simics Simulator. It shows how the CLI can be extended, and some techniques for working with Python in simulations including prototyping operations at the command-line and using **`run-python-file`**.

You will create a custom CLI command that will write a pattern to the training device memory and initiate the display update operation – going straight to the hardware using simulator back-doors. The command will have to take three arguments: the memory space to address, the address in that memory space that the frame buffer starts at, and the contents of the framebuffer to set. Instead of the somewhat clumsy byte pattern of the device driver we tried in the previous lab, the command will use a string argument where each character encodes the color for one pixel. Eight times less typing, and much easier to understand.

Get started

1. Start a new simulation session from the second checkpoint you saved in Lab 001, "**`AfterDriver.ckpt`**". Having the driver loaded is necessary, since it initializes the hardware state. Without it, you would have to manually configure the device.

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

Enough of variants for Windows and Linux: from this point on, command-line starts of the Intel Simics Simulator will be written in the above single-line portable style. Paths will be Linux-style as that works correctly on both types of host.

Prototype the memory write operation

When building Python scripts and modules for the Intel Simics simulator, it is very common to prototype the code on the command line. In this example, you will build up a memory writing operation.

2. In the CLI, check where the frame buffer used by the training device is located, by using the **`info`** command on the **`machine.training_card.master_bb`** device:

```
simics> machine.training_card.master_bb.info
```

This should show the frame buffer base as residing in **`machine.training_card.local_memory`**, at offset **`0x1000`**.

3. Check the contents of the display buffer in the local memory using the **x** command, for 64 bytes (**0x40**):

```
simics> machine.training_card.local_memory.x 0x1000 0x40 group-by = 32
```

This should show a pattern of **0xff** and **0x00**, with four bytes for each LED "pixel" in the display.

```
p:0x00001000  00ffff00 00ffff00 00ffff00 00ffff00      ................
p:0x00001010  00ffff00 00ffff00 00ffff00 00ffff00      ................
p:0x00001020  00ffff00 00ffff00 ffffff00 ffffff00      ................
p:0x00001030  ffffff00 ffffff00 00ffff00 00ffff00      ................
```

Note that default is to show the bytes in order of increasing memory addresses.

4. To change the memory contents, you will perform operations on the local memory. To see what interfaces are available for operations on the memory space, use **help**:

```
simics> list-interfaces machine.training_card.local_memory
```

The output contains the interfaces available for interacting directly with the memory space:

```
┌────────────────────────┐
│Interfaces              │
├────────────────────────┤
│breakpoint              │
│breakpoint_query_v2     │
│breakpoint_trigger      │
│conf_object             │
│direct_memory_lookup    │
│direct_memory_lookup_v2 │
│log_object              │
│map_demap               │
│memory_space            │
│transaction             │
│translate               │
│translator              │
└────────────────────────┘
```

You should use the **transaction** interface that was introduced in the Intel Simics Simulator version 6.

5. Check out how to use the transaction interface using API help:

```
simics> api-help transaction_interface_t
```

You can access the functions defined by the interface using Python, using the **conf** namespace and **iface** mechanism (**@conf.*full.object.name*.iface.*interface-name*.*call-name*()**).

To write data to the memory space, the **issue** function should be used:

```
    exception_type_t (*issue)(conf_object_t *NOTNULL obj,
                              transaction_t *NOTNULL t,
                              uint64 addr);
```

The arguments to the call are **obj**, **t**, and **addr**. The **obj** is always implicit when calling model interfaces from Python and does not need to be specified. **t** is a **transaction_t** object, which you will have to create before calling the interface. **addr** is the address to target with the transaction.

6. **transaction_t** can be easily created from Python in a simulation session. For the full details, check the "*Transactions*" chapter of the *Model Builder User's Guide*. The transaction used here should be a *write*, using *inquiry* semantics, and containing a string of bytes valued 1 to 16. Create a transaction with these properties:

```
simics> @t=transaction_t( inquiry=True, write=True,
data=bytes((1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)))
```

This requires some explanation.

inquiry indicates whether the simulator should look at this access as debug (inquiry) access or real access. An inquiry access should not trigger any side-effects – such as device actions triggered by memory accesses. When writing to plain memory from the command line it makes sense to set it to **True**, since the operation is a backdoor operation and not part of the actual execution of the target system.

The command-line **get** and **set** commands use inquiry accesses, while the **read** and **write** commands use real accesses.

The **data** attribute is a Python byte string, also known as **bytes**. Such a string can be created from a tuple or list of values, as shown here. The size of the operation is inferred from the size of the **data** (but it can also be set in case you want to generate a certain size memory operation).

7. Send the transaction to memory, at address **0x1000**:

```
simics>
@conf.machine.training_card.local_memory.iface.transaction.issue(t,
0x1000)
```

Note that the call is done on the memory space object itself – no port object is used for inbound memory transactions to memory spaces since they logically only have a single such entry point.

8. As a result of the call, the CLI prints the return value from the interface call. In this case, it is **<exception_type_t.Sim_PE_No_Exception: 1025>**. To see what other values could have been returned, check the definition of the return type of the **issue** call, **exception_type_t**:

```
simics> api-help exception_type_t
```

The help text is rather long, but the critical bit is the definition of the enum **`exception_type_t`**:

```
typedef enum {
    Sim_PE_Cancelled              = 1022,
    Sim_PE_Async_Required         = 1023,
    Sim_PE_Deferred               = 1024,
    Sim_PE_No_Exception           = 0x401,
    Sim_PE_Silent_Break           = 0x402,
    Sim_PE_Stop_Request           = 0x403,
    Sim_PE_Inquiry_Outside_Memory = 0x404,
    Sim_PE_Inquiry_Unhandled      = 0x405,
    Sim_PE_Execute_Outside_Memory = 0x406,
    Sim_PE_IO_Not_Taken           = 0x407,
    Sim_PE_IO_Error               = 0x408,
    Sim_PE_Stall_Cpu              = 0x409,
    Sim_PE_Instruction_Finished   = 0x40a,
    Sim_PE_Default_Semantics      = 0x40b,
    Sim_PE_Ignore_Semantics       = 0x40c,
    Sim_PE_Last                   = 0x40d
} exception_type_t;
```

`1025` (**`0x401`**) corresponds to **`Sim_PE_No_Exception`**. I.e., the operation succeeded. This will be used in the command code for error checking.

9. Double-check the **`Sim_PE_No_Exception`** value by checking the value of the constant in the Python **`simics`** namespace:

```
simics> @simics.Sim_PE_No_Exception
```

10. Check the updated memory contents:

```
simics> machine.training_card.local_memory.x 0x1000 0x40 group-by = 32
```

It should show something like this – the first line of memory displayed is modified, while the following lines are just like they were before:

```
p:0x00001000  01020304 05060708 090a0b0c 0d0e0f10     ................
p:0x00001010  00ffff00 00ffff00 00ffff00 00ffff00     ................
p:0x00001020  00ffff00 00ffff00 ffffff00 ffffff00     ................
p:0x00001030  ffffff00 ffffff00 00ffff00 00ffff00     ................
```

Force a display update

Note that the display in the LED Panel console is not changing. To change the colors shown by the LEDs it is necessary to tell the device to do a refresh. After an update is requested, the simulation has to run for the subsystem to do the refresh. It takes many target system cycles to do a refresh since the model is built to read one pixel from memory and then send the value to the corresponding LED over I2C. Each I2C write takes several cycles to complete.

11. To request an LED update, you write a control register in the device. You will look more closely at these kinds of operations later; for now, just use the values provided. Note

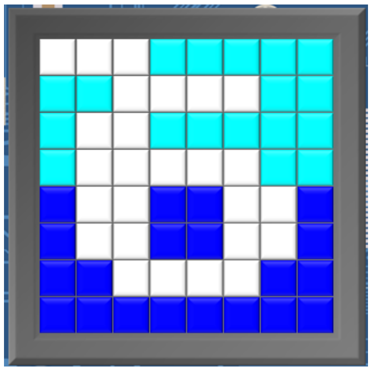
this is using a **`write`** command, in order to make sure that the write has the desired side effect in the device model.

```
simics> machine.training_card.local_memory.write 0x1000_0010 0x1 4
```

12. You then run forward to update the LEDs. Each LED takes about 550 cycles. Try stepping through the display updates by issuing repeated run commands:

```
simics> run 550 cycles
```

Press **<ENTER>** on the command line to repeat this command over and over. There is no need to retype it or even do **<UP-ARROW>** to pull it from history.



As you run each step, you will see the pixels changing color to white. The interpretation of the device is that the first three bytes of the pixel corresponds to the values for Red, Green, and Blue. And the current LEDs only have a single bit of resolution for each color, so anything non-zero is considered on. Thus, a byte value like "**`01020304`**" means that R, G, and B are all on. I.e., the pixel is white.

13. Run for a simulated second to make sure the update process is finished.

```
simics> r 1 s
```

Turn a string into a series of bytes

In order to be able to use a string with color codes as input, you need to build a translator from character codes to color values. This can be worked out on the command line step-by-step, but in the interest of time the full solution is provided below.

14. Create a new empty file called **`lab-002-custom-command.py`** in **`<project>/targets/simics-user-training`** and open it in an editor.

15. Add the following code to the file **`lab-002-custom-command.py`**:

```
## Map function to go from character to list of bytes
## with first byte being red, then green, then blue, then unused
cmap = { 'W' : [0xff, 0xff, 0xff, 0x00],  # white
         '.' : [0xff, 0xff, 0xff, 0x00],  # white
         ' ' : [0x00, 0x00, 0x00, 0x00],  # black
         'K' : [0x00, 0x00, 0x00, 0x00],  # blacK
         'R' : [0xff, 0x00, 0x00, 0x00],  # red
         'G' : [0x00, 0xff, 0x00, 0x00],  # green
         'B' : [0x00, 0x00, 0xff, 0x00],  # blue
         'Y' : [0xff, 0xff, 0x00, 0x00],  # yellow
         'C' : [0x00, 0xff, 0xff, 0x00],  # cyan
         'M' : [0xff, 0x00, 0xff, 0x00]   # magenta
         }
def led_char_to_byte_list(c):
  try:
    # Use the map to retrieve a color
    w=cmap[c]
    return w
  except:
    # Return broken yellow as default color
    return [0x01, 0x01, 0x00, 0x00]

## Convert string to list of bytes
def led_string_to_byte_list(s):
  l = []
  for c in s:
    l += led_char_to_byte_list(c)
  return l
```

Warning! Depending on how copy-paste and Word printing works with PDF files and PDF readers of various types, you will likely have to **manually add indentation** to the code. In Python, white space is significant, and it should be created using spaces.

16. Test the file. Go to the command line and use run-python-file to run the file. Use tab-completion to build the name of the script file:

```
simics> run-script targets/simics-user-training/lab-002-custom-
command.py
```

Note that Linux-style path separators work equally well on Windows hosts.

17. At the command line, call the functions defined in the file. Do a bit of manual unit testing. Start with the core **`led_char_to_byte_list`** function, using both valid and invalid inputs:

```
simics> @led_char_to_byte_list("W")
simics> @led_char_to_byte_list("Y")
simics> @led_char_to_byte_list("R")
simics> @led_char_to_byte_list("Q")
```

18. Next, try the conversion of a string:

```
simics> @led_string_to_byte_list("WRGBYKCM")
```

It seems that the code is working.

Prototype the new CLI command

Next, you will work out how to define a new command, before putting it together with the code you developed above.

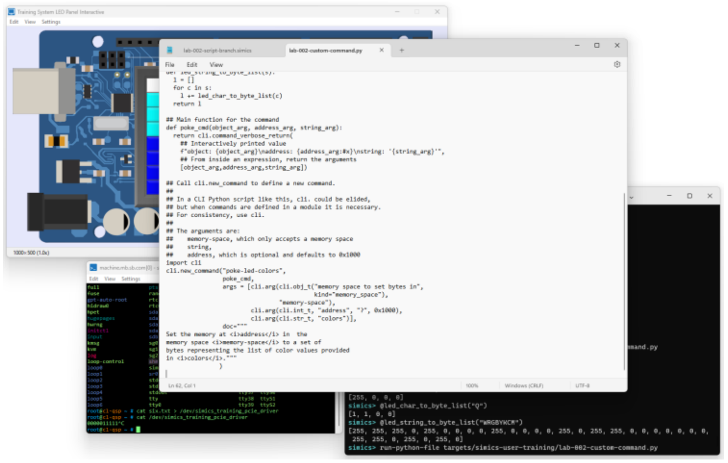
19. Go back to the Python source file. At the end of the file, enter the following code to define a new command. You need to have a function that does the work of the command (**poke_cmd**), and a call to **cli.new_command()** to register the command. Use a placeholder for **poke_cmd** that just confirms that it has received useful arguments, by using the **cli.command_verbose_return** mechanism.

```
## Main function for the command
def poke_cmd(object_arg, address_arg, string_arg):
  return cli.command_verbose_return(
    ## Interactively printed value
    f"object: {object_arg}\naddress: {address_arg:#x}\nstring: '{string_arg}'",
    ## From inside an expression, return the arguments
    [object_arg,address_arg,string_arg])

## Call cli.new_command to define a new command.
##
## In a CLI Python script like this, cli. could be elided,
## but when commands are defined in a module it is necessary.
## For consistency, use cli.
##
## The arguments are:
##    memory-space, which only accepts a memory space
##    string,
##    address, which is optional and defaults to 0x1000
import cli
cli.new_command("poke-led-colors",
                poke_cmd,
                args = [cli.arg(cli.obj_t("memory space to set bytes in",
                                          kind="memory_space"),
                                "memory-space"),
                        cli.arg(cli.int_t, "address", "?", 0x1000),
                        cli.arg(cli.str_t, "colors")],
                doc="""
Set the memory at <i>address</i> in  the
memory space <i>memory-space</i> to a set of
bytes representing the list of color values provided
in <i>colors</i>."""
               )
```

20. Save the file.

    Here is an example of what it might look like, with a simple editor open above the command-line interface and the target consoles:



```
  l = []
  for c in s:
    l += led_char_to_byte_list(c)
  return l

## Main function for the command
def poke_cmd(object_arg, address_arg, string_arg):
  return cli.command_verbose_return(
    ## Interactively printed value
    f"object: {object_arg}\naddress: {address_arg:#x}\nstring: '{string_arg}'",
    ## From inside an expression, return the arguments
    [object_arg,address_arg,string_arg])

## Call cli.new_command to define a new command.
##
## In a CLI Python script like this, cli. could be elided,
## but when commands are defined in a module it is necessary.
## For consistency, use cli.
##
## The arguments are:
##    memory-space, which only accepts a memory space
##    string,
##    address, which is optional and defaults to 0x1000
import cli
cli.new_command("poke-led-colors",
                poke_cmd,
                args = [cli.arg(cli.obj_t("memory space to set bytes in",
                                          kind="memory_space"),
                                "memory-space"),
                        cli.arg(cli.int_t, "address", "?", 0x1000),
                        cli.arg(cli.str_t, "colors")],
                doc="""
Set the memory at <i>address</i> in  the
memory space <i>memory-space</i> to a set of
bytes representing the list of color values provided
in <i>colors</i>."""
               )
```

```
[255, 0, 0, 0]
simics> @led_char_to_byte_list("Q")
[1, 1, 0, 0]
simics> @led_string_to_byte_list("WRGBYKCM")
[255, 255, 255, 0, 255, 0, 0, 0, 0, 255, 0, 0, 0, 0, 255, 0, 255, 255, 0, 0, 0, 0, 0, 0, 0,
 255, 255, 0, 255, 0, 255, 0]
simics> run-python-file targets/simics-user-training/lab-002-custom-command.py
```

21. Go to the simulator command line and run the Python file again – this will create the new command in CLI. If you had changed any of the other functions in the file, they would also have been redefined. Python is totally dynamic, allowing anything to be redefined at any point.

```
simics> run-script targets/simics-user-training/lab-002-custom-
command.py
```

22. Check out the new command, using **help**:

```
simics> help poke-led-colors
```

23. Next, run it with arguments. Use tab completion to find the memory space – the command should only show you memory spaces.

```
simics> poke-led-colors memory-space = machine. <TAB>
```

    Using **<TAB>** after **memory-space=** will show quite a few memory spaces. The Intel Simics Simulator uses memory spaces to implement many hardware interfaces.

```
machine.mb.cpu0.mem[0][0]                         machine.mb.port_mem_m
machine.mb.cpu0.mem[1][0]                         machine.mb.sb.ext_conf
machine.mb.dram_space                             machine.mb.sb.ext_io
machine.mb.gpu.dmap_space                         machine.mb.sb.ext_mem
machine.mb.nb.core.remap_space                    machine.mb.sb.lpc_io
machine.mb.nb.pci_bus.cfg_space                   machine.mb.sb.lpc_mem
machine.mb.nb.pci_bus.conf_space                  machine.mb.shadow_mem
...
```

24. Expand the memory-space argument to **machine.training_card.local_memory**.

25. Finish up the command with a string like **WRGBCMYK**, letting the command use the default address (remember that the **address** argument is optional).

```
simics> poke-led-colors machine.training_card.local_memory WRGBCMYK
```

This should print the diagnostic output from the command:

```
simics> poke-led-colors memory-space = machine.training_card.local_memory WRGBCMYK
object: <the memory-space 'machine.training_card.local_memory'>
address: 0x1000
string: 'WRGBCMYK'
simics>
```

Thus, it seems the command declaration is working.

26. Complete the command by connecting the pieces. Change the **poke_cmd** function to include the actual functionality of the command. Add the code shown in bold below to that start of the **poke_cmd**:

```
## Main function for the command
def poke_cmd(object_arg, address_arg, string_arg):
  bytelist = led_string_to_byte_list(string_arg)
  t = transaction_t( inquiry = True, write=True, data = bytes(bytelist))
  e = object_arg.iface.transaction.issue(t, address_arg )
  if e != simics.Sim_PE_No_Exception:
    raise CliError(f"poke_cmd failed to write memory (error {e})")
  return cli.command_verbose_return(
    ## Interactively printed value
    f"object: {object_arg}\naddress: {address_arg:#x}\nstring: '{string_arg}'",
    ## From inside an expression, return the arguments
    [object_arg,address_arg,string_arg])
```

27. Load the updated file:

```
simics> run-script targets\simics-user-training\lab-002-custom-
command.py
```

28. The simulator will print a message warning that the command was redefined:

```
*** redefining the 'poke-led-colors' command
```

It is harmless in *this* context since you are redefining a command that you defined yourself, and doing that in order to experiment. However, in general, this error message might indicate that something is inconsistent.

29. Try the updated command:

```
simics> poke-led-colors machine.training_card.local_memory "WRGBCMYK"
```

30. Check the memory contents, just the 32 bytes that you just changed (0x20 in hex):

```
simics> machine.training_card.local_memory.x 0x1000 0x20 group-by = 32
```

This should result in output like:

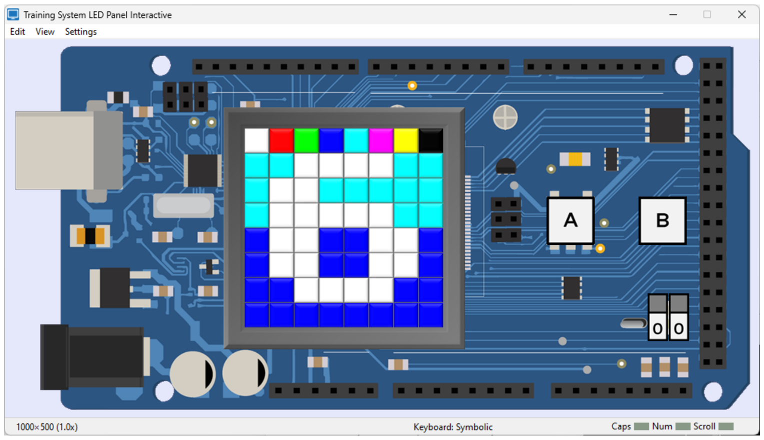
```
p:0x00001000  ffffff00 ff000000 00ff0000 0000ff00      ................
p:0x00001010  00ffff00 ff00ff00 ffff0000 00000000      ................
```

I.e., the memory contents have changed (you set this to **0x01**, **0x02**, etc. above).

31. Update the display:

```
simics> machine.training_card.local_memory.write 0x1000_0010 0x1 4
simics> run 1 s
```

The result should be a "rainbow" in the top row:



Error handling

Dealing with errors is an important part of writing robust user-facing commands. The code in the command checks the return value from the **`transaction.issue()`** call.

32. Test the error handling by addressing non-existent memory:

```
simics> poke-led-colors machine.training_card.local_memory 0x100_0000
WRGBCMYK
```

    Note that there is no error handling for malformed strings – instead the code converts anything unknown into a special byte pattern.

Draw some more pictures

Use the command to draw some more complex patterns.

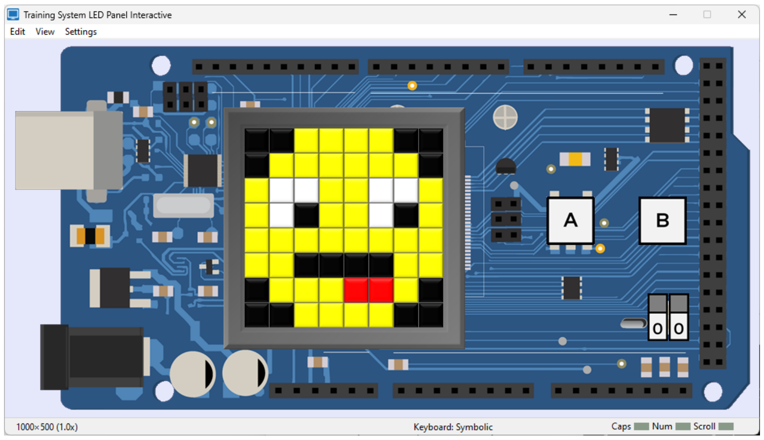
33. Enter a complete image as a string. Here is one example:

```
simics> poke-led-colors machine.training_card.local_memory "  YYYY
YYYYYY  YWWYYWWYYWKYYWKYYYYYYYYYYYKKKKYY YYYRRY   YYYY  "
```

34. Make sure that the display updates:

```
simics> machine.training_card.local_memory.write 0x1000_0010 0x1 4
simics> r
```

The result should look like this:



35. Keep the simulation running, trying different strings. Each time, you need to do the register write to get the display to update.

    For example, use some inline Python to produce 64 black pixels:

```
running> poke-led-colors machine.training_card.local_memory (`" " * 64`)
```

36. Once done, quit the simulation session.

G. Appendix. Complete code for custom command from Section F

```
## Map function to go from character to list of bytes
## with first byte being red, then green, then blue, then unused
cmap = { 'W' : [0xff, 0xff, 0xff, 0x00],  # white
         '.' : [0xff, 0xff, 0xff, 0x00],  # white
         ' ' : [0x00, 0x00, 0x00, 0x00],  # black
         'K' : [0x00, 0x00, 0x00, 0x00],  # blacK
         'R' : [0xff, 0x00, 0x00, 0x00],  # red
         'G' : [0x00, 0xff, 0x00, 0x00],  # green
         'B' : [0x00, 0x00, 0xff, 0x00],  # blue
         'Y' : [0xff, 0xff, 0x00, 0x00],  # yellow
         'C' : [0x00, 0xff, 0xff, 0x00],  # cyan
         'M' : [0xff, 0x00, 0xff, 0x00]   # magenta
         }
def led_char_to_byte_list(c):
  try:
    # Use the map to retrieve a color
    w=cmap[c]
    return w
  except:
    # Return broken yellow as default color
    return [0x01, 0x01, 0x00, 0x00]

## Convert string to list of bytes
def led_string_to_byte_list(s):
  l = []
  for c in s:
    l += led_char_to_byte_list(c)
  return l

## Main function for the command
def poke_cmd(object_arg, address_arg, string_arg):
  bytelist = led_string_to_byte_list(string_arg)
  t = transaction_t( inquiry = True, write=True, data = bytes(bytelist))
  e = object_arg.iface.transaction.issue(t, address_arg )
  if e != simics.Sim_PE_No_Exception:
    raise CliError(f"poke_cmd failed to write memory (error {e})")
  return cli.command_verbose_return(
         ## Interactively printed value
    f"object: {object_arg}\naddress: {address_arg:#x}\nstring: '{string_arg}'",
    ## From inside an expression, return the arguments
    [object_arg,address_arg,string_arg])

## Call cli.new_command to define a new command.
##
## In a CLI Python script like this, cli. could be elided,
## but when commands are defined in a module it is necessary.
## For consistency, use cli.
##
## The arguments are:
##    memory-space, which only accepts a memory space
##    string,
##    address, which is optional and defaults to 0x1000
import cli
cli.new_command("poke-led-colors",
                poke_cmd,
                args = [cli.arg(cli.obj_t("memory space to set bytes in",
                                          kind="memory_space"),
                                "memory-space"),
                        cli.arg(cli.int_t, "address", "?", 0x1000),
                        cli.arg(cli.str_t, "colors")],
                doc="""
Set the memory at <i>address</i> in  the
memory space <i>memory-space</i> to a set of
bytes representing the list of color values provided
in <i>colors</i>."""
               )
```