Intel® Simics® Simulator Internals Training

Lab 01-03

Packages and Modules

1 Introduction

This is about how classes, packages and modules work in the Intel® Simics® Simulator.

2 Module Loading

2.1 Start a New Simulation Session

1. Start the simulator with a script from the internals training.

   On Linux:

   ```
   ./simics simics-internals-training/03-modules
   ```

   On Windows:

   ```
   simics.bat simics-internals-training/03-modules
   ```

   This just simply loads a default internals setup with a hello-world application. There is no point in running it.

2.2 Trace Classes to Their Packages

Understanding the package that a class comes from helps resolve module collisions. It also helps understanding who is responsible for a certain model, by tracing it back to the package.

2. Let's start by listing all classes that the simulator has loaded currently, and the modules they are loaded from:

   ```
   simics> list-classes -l -m -show-port-classes
   ```

   You will see many classes from different modules. Note that there are some classes that are listed as coming from **<no module>**. These are classes that belong to the simulator core and thus are not loaded from any module.

Example output:

The following classes have been registered:

Class	Module	Short description
NS16450	NS16x50	model of NS 16450 UART
NS16450.HRESET	NS16x50	simulates resetting by power loss
NS16450.Reset	NS16x50	legacy support, use SRESET instead
NS16450.SRESET	NS16x50	simulates resetting, raising the MR input signal
NS16450.regs	NS16x50	
NS16550	NS16x50	model of NS 16550 UART
NS16550.HRESET	NS16x50	simulates resetting by power loss
NS16550.Reset	NS16x50	legacy support, use SRESET instead
NS16550.SRESET	NS16x50	simulates resetting, raising the MR input signal
NS16550.regs	NS16x50	
bp-manager	bp-manager	manages the set of breakpoints in Simics
bp-manager.bank	bp-manager	device access

…

3. The previous step traced from classes to modules. Next, let's see which package each module belongs to. This is done with a verbose listing of the loaded modules:

```
simics> list-modules -l -v
```

The column **Package** shows you the package that a module was loaded from.

Note the **Shadowing** column that indicates that the same-name module was present in multiple packages.

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package	Path	Shadowing
NS16x50	Loaded	7033	7033	6250	7	Yes	03e7b846c9f4237f2b7f378d079d5dcd44e7abef	Simics Base	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\win64\lib\NS16x50.dll	
bp-manager	Loaded	7033	7033	6250	7	Yes	03e7b846c9f4237f2b7f378d079d5dcd44e7abef	Simics Base	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\win64\lib\bp-manager.dll	
clock	Loaded	7033	7033	6250	7	Yes	03e7b846c9f4237f2b7f378d079d5dcd44e7abef	Simics Base	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\win64\lib\clock.dll	

[…]

The current setup is a mix of modules from three different packages: The Simics base package, the RISC-V CPU package, and the Training package. The build IDs and paths are going to be different in your current setup.

4. Next, let's investigate the packages themselves. **List** all packages that are configured in the current Intel Simics simulator project:

```
simics> list-packages
```

The output shows quite a few packages, likely including:

Pkg	Name	Version	Description	Build ID	Path
1000	Simics Base	7.18.0	The Simics simulator framework and core models	simics:7033	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0
1030	Crypto Engine	7.7.0	Simics crypto-engine and OpenSSL library	simics:7031	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\..\simics-crypto-engine-7.7.0
1031	Simics GDB	7.3.0	Simics GNU Debugger (GDB)	simics:7029	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\..\simics-gdb-7.3.0
1033	Simics Python	7.2.0	Python interpreter for Simics	simics:7023	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\..\simics-python-7.2.0
2050	Platform: RISC-V CPU	7.7.0	Generic RISC-V CPU	simics:7028	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\..\simics-risc-v-cpu-7.7.0
2053	Platform: RISC-V Simple	7.5.0	RISC-V Simple platform	simics:7027	C:\Users\jengblo\AppData\Local\Programs\Simics\simics-7.18.0\..\simics-risc-v-simple-7.5.0

…

Note that there are packages from which no modules are currently loaded. Modules are only loaded on demand.

5. To check the modules that a certain package offers, use **`list-modules`** command with the **`package-number`** argument:

```
simics> list-modules package-number = 2050
```

This shows the modules being provided by the package.

2.3 Automatic Module Loading

You rarely need to load modules manually, since the simulator auto-loads a module when objects of a class provided by the module are created. Let's verify that.

6. Find a class in a module that has not yet been loaded. For example, one of the sample devices shipping in the Simics Base package. Note that you can list classes that have not yet been loaded, since the simulator tracks which classes are available before they are loaded:

```
simics> list-classes substr=sample_device -m
```

This should show something like the following:

```
The following classes are available:
```

Class	Module	Short description
sample-device-c	sample-device-c	N/A (module is not loaded yet)
sample_device_cxx_after	sample-device-c++	N/A (module is not loaded yet)
sample_device_cxx_after_bank	sample-device-c++	N/A (module is not loaded yet)
...		
sample_device_pkg_prio	sample-device-pkg-prio	N/A (module is not loaded yet)
...		

The **`Short description`** column indirectly indicates which modules are loaded.

7. Double-check the status of the **`sample-device-dml`** module:

```
simics> list-modules substr=sample-device-dml
```

This should show the module as not loaded, something like this:

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package
sample-device-dml		7033	7033	6250	7	Yes	03e7b846c9f4237f2b7f378d079d5dcd44e7abef	Simics Base

The "**`User Version`**" of a package is a string that is set at compilation time. For many modules, it is used to store a hash identifying the precise git commit that was used to compile the module, aiding in debug.

8. List the classes available in the module:

```
simics> list-classes module = sample-device-dml
```

This shows the module containing a single class, **`sample_device_dml`**.

9. Create an object of the class **`sample_device_dml`**.

The following command uses Python to call the simulator API call to create a new object:

```
simics> @SIM_create_object('sample_device_dml','a_sample',[])
```

The response should be like the below output. The reported package name might be different, but it should correspond to the package you saw when listing the modules verbosely.

```
<the sample_device_pkg_prio 'a_sample'>
```

10. Check again if the module is loaded:

```
simics> list-modules substr=sample-device-dml
```

This time, you see the module being reported as **`Loaded`** in the **`Status`** column:

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package
sample-device-dml	Loaded	7033	7033	6250	7	Yes	03e7b846c9f4237f2b7f378d079d5dcd44e7abef	Simics Base

2.4 Port Classes

When loading the **sample-device-dml** module, the module set up a port class for the register bank of the **sample_device_dml** class.

11. List the classes containing the string "**sample_device**" again.

```
simics> list-classes substr=sample_device -m -show-port-classes
```

Resulting in something like this:

```
The following classes are available:
```

Class	Module	Short description
sample-device-c	sample-device-c	N/A (module is not loaded yet)
…		
sample_device_dml	sample-device-dml	sample DML device
sample_device_dml.regs	sample-device-dml	sample DML devicecustom desc
…		

The **sample_device_dml** class now has documentation, since the module has been loaded. There is also a new class, **sample_device_dml.regs**. This is a port class for the register bank of the sample device, which is not visible until the module is loaded.

12. Check the object(s) created by the **SIM_create_object** call:

```
simics> list-objects substr=a_sample -all -class-desc
```

This should show two objects, like this:

Object	Class	Class description
a_sample	<sample_device_dml>	sample DML device
a_sample.bank.regs	<sample_device_dml.regs>	sample DML devicecustom desc

Note that the register bank object **a_sample.bank.regs** was created automatically when you created the **a_sample** object, since it is a port object.

Also, note that the port object *class* is called **sample_device_dml.regs**. There is no "bank" in the name of the class, only in the name of the object.

2.5 Multiple Classes in One Module

Intel Simics simulator modules usually contain multiple classes, as they provide a subsystem, a discrete chip, or similar aggregation of functionality.

13. To find the classes in a module, you can use the **list-classes** command:

```
simics> list-classes module = virtio
```

This module contains a number of different devices in the **virtio** family.

14. If a module is loaded, the **help** command will show the classes in the module as well as the package it belongs to. In case there are ambiguities between class names, object names, and module names, use the **module:** specifier to help:

```
simics> load-module virtio
simics> help module:virtio
```

The output will show all the classes in the module:

```
Module virtio

   Package
      Simics-Base

   Classes
      virtio-mmio-entropy
      virtio_mmio_blk
      virtio_mmio_fs
      virtio_mmio_net
      virtio_pcie_blk
      virtio_pcie_fs
      virtio_pcie_net
```

The **help** output for a module does not show the port classes, only the declared classes of the module (the classes whose creation will cause the module to be loaded).

15. To find the port objects of a class, use **help** on the class:

```
simics> help virtio-mmio-entropy
```

The output of help shows additional information about the class, in particular the module it comes from, the package the module comes from, and the class description. It also lists the *port objects* that will be created for each object of the class. This covers objects created from port classes and port objects created from "regular classes".

```
   Class virtio-mmio-entropy

   Description
      virtio MMIO entropy source

   Interfaces Implemented
      conf_object, log_object

   Port Objects
      bank.features (bank_instrumentation_subscribe, instrumentation_order, register_view,
      register_view_read_only, transaction)
      bank.mmio (bank_instrumentation_subscribe, instrumentation_order, register_view,
      register_view_read_only, transaction)
      port.SRESET (signal)
…
```

16. To find the port *classes* of a class, use **list-classes** and a substring matching the class name, along with the **-show-port-classes** option. For example, for the **virtio-mmio-entropy** class:

```
simics> list-classes substr=virtio-mmio-entropy -show-port-classes
```

This shows all classes that match the substring, including their port classes. In this case, it should only match a single class.

```
The following classes are available:
```

Class	Short description
virtio-mmio-entropy virtio-mmio-entropy.SRESET virtio-mmio-entropy.features virtio-mmio-entropy.mmio	model of virtio MMIO entropy source

17. **Quit** the Simics session:

```
simics> quit
```

3 Module Shadowing

3.1 Identifying a shadowed module

The training contains a module that is shadowed, i.e., the same module is present in two different packages. Apart from this training example, shadowing can happen if two packages contain the same module (since both packages contain models that use the same hardware), and you have both packages associated with your project. Shadowing might be harmless if the modules are identical, but if the modules differ in versions, shadowing can lead to misbehavior of a simulated platform if it is then using a different version than it was designed for.

The intentionally duplicated training module is called **`sample_device_pkg_prio`**.

1. Start a new empty Intel Simics simulator session in your project:

   (On Windows use "**`simics.bat`**", on Linux "**`./simics`**")

2. Find the module of the class **`sample_device_pkg_prio`**:

   ```
   simics> list-classes substr = pkg_prio -m
   ```

   You see that the module for the class is named **`sample-device-pkg-prio`** and it is not yet loaded:

   ```
   The following classes are available:
   ```

Class	Module	Short description
sample_device_pkg_prio	sample-device-pkg-prio	N/A (module is not loaded yet)

3. Let's inspect the module.

   ```
   simics> list-modules -v substr = sample-device-pkg-prio
   ```

   The output will look something like this:

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package	Path	Shadowing
sample-device-pkg-prio		7059	7059	6250	7	Yes		Simics Base	/path/to/your/Simics-Base/linux64/lib/sample-device-pkg-prio.so	Yes

   The output above indicates that the module has been found and will be loaded from the Simics Base package. The output also reports that there are multiple modules with the name, by putting a **`Yes`** into the **`Shadowing`** column. The package shown in the **`Package`** column depends on the build time of your currently installed Simics Base and Simics Training packages. It could either be Simics Base or Training.

4. Let's check which other packages contain the module, but get shadowed:

   ```
   simics> list-failed-modules -v
   ```

You see Simics reporting the module duplication as well as the path to the duplicate module (i.e., the one that is not getting loaded):

```
Current ABI version: 6191  Lowest supported: 5000
```

Name	Error	[…]	Path
sample-device-pkg-prio	Duplicated module		/path/to/your/Simics-Base/.. /Training/linux64/lib/sample-device-pkg-prio.so

The path would point at either the Simics Base package or the Training package, the one that was not listed by the previous step.

3.2 Create an object and load the module

Check that object creation does indeed load the module.

5. Double check that the module **sample-device-pkg-prio** is not yet loaded:

```
simics> list-modules substr = sample-device-pkg-prio
```

This should show something like this:

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package
sample-device-pkg-prio		7059	7059	6250	7	Yes		Simics Base

Note that the **Status** column does not show it as loaded.

6. Create an object of the class **sample_device_pkg_prio**. We know from above that this is provided by the **sample-device-pkg-prio** module.

The following command uses a Simics Python API call to create a new object:

```
simics> @SIM_create_object('sample_device_pkg_prio','our_object',[])
```

The response should be like the below output. The reported package name might be different, but it should correspond to the package you saw when listing the modules verbosely.

```
[sim info] Loading <sample_device_pkg_prio> from Simics-Base
<the sample_device_pkg_prio 'our_object'>
```

Note that the message about where the module is loaded from is specific to the **sample_device_pkg_prio** device, coded into the module's **module_load.py** file. It is not something the Simics simulator itself prints.

7. Check the status of the module again, and that it is indeed loaded from the package indicated:

```
simics> list-modules substr = sample-device-pkg-prio -v
```

Which should result in output similar to the below. Check the **Path** and **Package**:

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package	Path	Shadowing
sample-device-pkg-prio	Loaded	7059	7059	6250	7	Yes		Simics Base	/path/to/your/Simics-Base/linux64/lib/sample-device-pkg-prio.so	Yes

3.3 Control how shadowing is resolved

In this section we will show you how you can instruct the Simics simulator core to load modules from a certain package in preference to other packages, regardless of the module build time and the standard loading order.

8. Restart Simics to have the module unloaded again (meaning, quit the current session and start a new empty session).

9. Check that the **sample-device-pkg-prio** module has not been loaded yet:

```
simics> list-modules -l
```

   The list of loaded modules should only contain a few core modules.

10. Now use **set-prioritized-packages** to give priority to the package that the module was **not** loaded from the last time.

    If you saw "**Loading <sample_device_pkg_prio> from Simics-Base**" in the previous section, then you will prioritize the Training package as shown below. If

```
simics> set-prioritized-package package = Training
```

    OR

    If you saw "**Loading <sample_device_pkg_prio> from Training**", then you will prioritize the base package

```
simics> set-prioritized-package package = Simics-Base
```

    NOTE: The command uses the short name of the package, which is always free of spaces and potentially shorter than the package name shown in some other command output. We recommend that you can use tab-completion on the package name.

11. List the packages, and check that the prioritized package setting is working:

```
simics> list-packages
```

The output will show that the package you selected is now prioritized as indicated by an asterisk next to its package number.

Pkg	Name	Version	Description	Build ID	Path
6010 *	Simics Training	7.0.0-pre.9	Training materials and demos (requires QSP-x86)	simics:7059	/path/to/your/Simics-Base/../Training
[…]					

12. List the module details again.

```
simics> list-modules -v substr = sample-device-pkg-prio
```

You will now see that the module will be loaded from the package you prioritized. Something like this (for the case that you prioritized Training and the module used to load from Simics Base):

Name	Status	Build ID	ABI	Compat	API	Thread safe	User Version	Package	Path	Shadowing
sample-device-pkg-prio		7059	7059	6250	7	Yes		Simics Training	/path/to/your/Simics-Base/../Training/linux64/lib/sample-device-pkg-prio.so	Yes

13. Check the failed module information again:

```
simics> list-failed-modules -v substr = sample-device-pkg-prio
```

You should also see that the originally winning module (before you used **set-prioritized-package**) is now failing. In the output below that would be the code from the Simics Base package, as can be seen from the path:

```
Current ABI version: 6191  Lowest supported: 5000
```

Name	Error	[…]	Path
sample-device-pkg-prio	Duplicated module		/path/to/your/Simics-Base/linux64/lib/sample-device-pkg-prio.so

14. Finally, create an object of the class:

```
simics> @SIM_create_object('sample_device_pkg_prio','our_object',[])
```

This time you should see that the module is loaded from the prioritized package. For example, if the Training package was prioritized:

```
[sim info] Loading <sample_device_pkg_prio> from Training
<the sample_device_pkg_prio 'our_object'>
```

15. Quit the Simics session.

```
simics> exit
```

3.4 (Optional) The Simics project takes precedence

If you have a working module build environment, you can verify that the project local module always takes precedence over any package modules that it shadows.

16. On the OS shell, invoke below command to copy the module sources.

    On Windows:

    ```
    bin\project-setup.bat --copy-device=sample-device-pkg-prio
    ```

    On Linux:

    ```
    bin/project-setup --copy-device=sample-device-pkg-prio
    ```

17. On the OS shell, build the device.

    On Windows:

    ```
    bin\make.bat sample-device-pkg-prio
    ```

    On Linux:

    ```
    make sample-device-pkg-prio
    ```

18. Start a new empty Simics session in your project.

    (On Windows use "**simics.bat**", on Linux "**./simics**")

19. List the failed modules:

    ```
    simics> list-failed-modules -v substr = pkg-prio
    ```

    You will note that both modules delivered by installed packages (Simics Base and Training) are listed as failed. Something like this:

```
simics> list-failed-modules -v substr = pkg-prio
Current ABI version: 6191  Lowest supported: 5000
```

Name	Error		Path
sample-device-pkg-prio	Duplicated module	[…]	/path/to/your/Simics-Base/../Training/linux64/lib/sample-device-pkg-prio.so
sample-device-pkg-prio	Duplicated module		/path/to/your/Simics-Base/linux64/lib/sample-device- pkg-prio.so

20. Set any of the packages as prioritized. Below example uses the training package

    ```
    simics> set-prioritized-package Training
    ```

21. List the failing modules yet again.

    ```
    simics> list-failed-modules -v substr = pkg-prio
    ```

You should see the same list as above, something like this:

```
simics> set-prioritized-package Training
simics> list-failed-modules -v substr = pkg-prio
Current ABI version: 7059  Lowest supported: 6250
```

Name	Error		Path
sample-device-pkg-prio	Duplicated module	[…]	/path/to/your/Simics-Base/../Training/linux64/lib/sample-device-pkg-prio.so
sample-device-pkg-prio	Duplicated module		/path/to/your/Simics-Base/linux64/lib/sample-device- pkg-prio.so

Note that the set of failed modules remain the same. The project takes priority over all packages, including prioritized packages.

22. As a final check that the project always wins, instantiate an object again.

```
simics> @SIM_create_object('sample_device_pkg_prio','our_object',[])
```

The output should show the project name as the load source for the module. For example, if the project is called **`name-of-your-project`**:

```
[sim info] Loading <sample_device_pkg_prio> from name-of-your-project
<the sample_device_pkg_prio 'our_object'>
```

23. Quit the Simics session.

```
simics> exit
```

4 Package Associations with Projects

A running Simics session is always associated with a set of packages known to it. When running from within a Simics project, the list of associated packages is taken from the project, and if the project has no package association list of its own, the list from the Simics Base package will be used. If the Simics Base also does not have a package list, then the session will only know the base package. It is important to note that these lists are not overlayed. If the project has a list, then this is the only list used. The Base Package list is a fallback. The recommendation is to use project-based associations, and this is what we will look at in this lab.

When a project is created via ISPM, you can configure the initial set of associated packages and you can also use ISPM to later change that list. Under the hood, ISPM will use a Simics tool to manipulate the package list and in this training, we will directly use this tool instead of using ISPM.

4.1 Inspecting what packages are associated with the current project

Generally, if things don't work, it is always good to double check what exact packages and versions your project is using. So, let's start with this.

1. In your project, invoke the following command

   On Linux:

   ```
   bin/addon-manager
   ```

   On Windows:

   ```
   bin\addon-manager.bat
   ```

   You see the list of packages currently known to your project. In the usual training setup, you should see the QSP package, the RISC-V-CPU package, and the training package itself. You see the package name, the version and the path to it either as an **absolute path** or a path **relative to your Simics Base package** (NOT relative to the project).

2. Another way is to check the package list from within Simics, so start a new empty Simics session.

   ```
   [./]simics[.bat]
   ```

3. Invoke the list-packages command

   ```
   simics> list-packages
   ```

   Again, you see the packages known to your project, their versions and the absolute paths to their locations. In addition, the command also shows their build IDs and descriptions.

4.2 Remove a package from the package list

In case the project is using a wrong package knowing how to manipulate the package list is important. Let's remove a package association from the project now.

4. At this point you should still have a running, empty Simics session. First, let's fetch our package list content into a variable.

```
simics> $packages=(list-packages)
```

5. Now let's use some Python code to print the full path to package 2050 (the RISC-V CPU package) from the variable.

```
simics> @[p[-1] for p in simenv.packages if p[0]=='2050']
```

   Make sure 2050 is quoted. You'll see the absolute path to package 2050 in the response. Write that down or copy it somewhere. The addon-manager tool identifies packages based on file system paths and later we want to change the association with package 2050, so we need that path.

6. **Quit** Simics.

```
simics> quit
```

7. Tell the **addon-manager** tool to remove package 2050 from your project associations.

   On Linux:

```
bin/addon-manager -d /path/you/saw/in/step/5/to/simics-risc-v-cpu-X.Y.Z
```

   On Windows (do not use quotes around the path!):

```
bin\addon-manager.bat -d X:\path\you\saw\in\step\5\to\simics-risc-v-cpu-X.Y.Z
```

   One you press enter; **addon-manager** will inform you what it will do. You should see a message telling you that the RISC-V-CPU package will be removed, showing the original package list before and after the removal. You are then asked if that is okay.

8. Respond "**y**" to the questions from the script (if you really want to remove the package and if you want to update the project):

```
Do you want to update the package list? (y/n) [y] y
Package list updated
Do you want to update the project? (y/n) [y] y
Project updated successfully
```

   The package is now gone from your project associations (but it has not been deleted from disk).

4.3 Observe the impact of a changed package list

After the removal of the package from the package list, let's see what has changed and what a symptom of a missing package can be.

9. Start an empty simulation session in your project again:

```
[./]simics[.bat]
```

10. List the packages once more:

```
simics> list-packages
```

    You should see that package 2050 is gone from the package list.

11. Now try running the internals training target, using a script from lab 05:

```
simics> load-target simics-internals-training/05-time-quanta
```

    You'll see errors reporting that one or more classes could not be found. The configuration was not completed, and all created objects deleted.

    This is a typical symptom of a missing package: a target tries to use classes that are present in the missing package, and cannot find them. Of course, any simulation setup that does not use any of the classes, images, or scripts that come with package 2050 would still work fine.

4.4 Add a package to the package list

Currently you have a broken package list (at least if you want to continue with the training). You need to repair it by adding the package back into the package list.

12. Tell the **addon-manager** to add package 2050 back into our package list, using the path to the package just like when you removed it. The **-s** option is used to select a package for inclusion.

    On Linux:

```
bin/addon-manager -b -s /path/you/saw/in/step/5/to/simics-risc-v-cpu-X.Y.Z
```

    On Windows:

```
bin\addon-manager.bat -b -s X:\path\you\saw\in\step\5\to\simics-risc-v-cpu-X.Y.Z
```

    When using the "**-b**" option, the **addon-manager** runs in batch mode and hence does not ask any questions and gives way less output. You only see that the package list and the project have been updated.

13. Verify that the package list contains the RISC-V CPU package (package 2050) again by using any of several mechanisms.

    Run the simulator with the "version" switch:

```
[./]simics[.bat] --version
```

    Or starting the simulator and listing packages:

```
[./]simics[.bat]
simics> list-packages
simics> quit
```

    Or the **addon-manager** without any arguments:

```
bin/addon-manager
```