Simics cheatsheet

1 Simics Usage

Glossary & Documentation

Simics *Base*	simics executable + the set of supporting shared libraries (DLLs) + tools
simulation	running of code (target) on a model/platform (target) with advancement of time
host	a computer where simulation runs
target	a simulated code running in its isolated memory region, e.g. Linux or Windows guest
platform	a complete runnable model: full set-up of devices with CPU or with at least a clock provider
package	a set of devices, oftentimes constituting 1 platform, distributed as a whole in an archive and unpacked into 1 directory
*[**Simics**] Script*	simple Unix-shell-like language (a wrapper over Python) used for connecting devices and command line automation
Simics User's Guide	documentation on **Simics** usage — both command line and Eclipse

Below $ stands for Unix shell prompt, > for **Simics** prompt.

Set up a workspace with platform package

```
$ path/to/simics-base/bin/project-setup .
$ bin/addon-manager -c # remove default package associations
# allow scripts & shared libraries be found:
$ bin/addon-manager -s /path/to/platform-package
$ bin/addon-manager -s /path/to/additional/package
```

Start up

```
$ ./simics targets/platf/platf.simics
```

Shell command line arguments ⟶ **Simics** command line correspondence:

```
$ ./simics start.simics
```

⟶

```
$ ./simics
> run-command-file start.simics
```

```
$ ./simics script.py
```

⟶

```
$ ./simics
> run-python-file script.py
```

```
$ ./simics -e '$config_variable1=value; $config_var2=value' start.simics
# but NOT ./simics start.simics -e $varibables ...
```

⟶

```
$ ./simics
> $config_variable=value
> $config_var2=value
> run-command-file start.simics
```

Environment & Packages

```
> version # list of installed packages
$ ./simics -v # the same
> pwd # current directory where simics is running
> list-directories # where Simics searches files
```

To debug chain of called auxiliary scripts include s:

```
$ ./simics -script-trace targets/platf/platf.simics
```

Commands for running simulation

```
> c[ontinue]
> r[un] 100 cycles    # or 10 steps or 0.1 seconds
> ptime [-all]    # to show target's time
```

Printing device structure

To find devices by name/class or interface name:

```
> list-objects -all name    # searches also for the class
> list-objects -all substr = mem # object names containing mem
> list-objects -all iface = my_interface # by interface name
```

To examine device structure (**Simics** *components* and *devices*) of a given component *myPlatform*:

```
# 1-level representation: immediate children myPlatform:
> list-objects namespace = myPlatform
# multi-level one: all children of myPlatform with all hierarchy:
> list-objects namespace = myPlatform -tree
```

To find all objects with the same class as given device:

```
> platf.myDevice->classname
my_class
> list-objects -all my_class
> list-objects -all class = my_class
```

Registers

```
# print all registers:
> print-device-regs platf.myDevice
# print fields of register myRegister:
> print-device-reg-info platf.myDevice.myBank.myRegister
```

Writing/reading with side effects

```
> write-device-reg
      platf.myDev.bank.myBank.myGrp.myReg 0x1
# (Register groups like myGrp may be omitted in devices)
> read-device-reg platf.myDev.bank.myBank.myGrp.myReg
```

Writing/reading **without** side effects (aka set/get)

It's done through attributes:

```
# To set to value 0x1
platf.myDev->myBank_myGrp_myReg = 0x1
# To get the value:
platf.myDev->myBank_myGrp_myReg
```

Device information

Model information

```
# To find info about class/module/package for your device:
> help platf.myDev
  Class myClass
  Provided By
    myModule (from myPackage)
  # ...then documentation about the device is printed.
# To list all classes provided by a module:
> list-classes -m myModule
# Configuration information
> platf.myDev.info
```

Runtime information

```
# Runtime information:
> platf.myDev.status
# pretty-print device attributes with values:
> list-attributes platf.myDev
# to search all attributes/registers containing mem:
list-attributes platf.myDevice substr = mem
```

Debugger commands

To use **Simics** to debug target:

```
dis[assemble] # show assembler commands at current address
# Break points:
break address
break-hap Core-Magic-Instruction
```

Examining current state

pselect	show currently running CPU
memory-map	print all mapped devices
pregs	print all CPU registers
probe-address *addr*	path to target *addr*

x86-specific and platform-specific commands

memory-trace *addr*	path to target *addr*
pregs	to know x86 mode (16/32/64-bit), 1st line
reset-button-press	to reboot the target
power-button-press	to press power button

Moving files target ⟷ host

```
> start-simicsfs-server
```

```
target$ mkdir a
target$ simicsfs-client a
```

Saving info

Save logs from current point

```
> start-command-line-capture filename
```

Save simics variables to file

```
> start-command-line-capture filename
> list-variables
> stop-command-line-capture
```

Check points

write-configuration "*checkpoint_name*"	— save checkpoint
read-configuration "*checkpoint_name*"	— restore it back

Connects — attributes that point to other devices

```
# to set connect attribute:
> platf.myDevice->myConnect = "platf.anotherDevice"
# to zero connect attribute:
> platf.myDevice->myConnect = FALSE
```

Miscellaneous

```
# To dump network packages (for analysis with Wireshark):
> pcap-dump link=ethernet_switch0 filename=myFile
```

2 Simics DML 1.4 Programming & C/DML API

Glossary & Documentation

DML	Device Modelling Language is a wrapper over C for writing (fast) devices. Its compiler is included into **Simics** Base.
device	any **Simics** class for modelling real devices. Different **Simics** devices communicate via *interfaces*.
Model Builder User's Guide	overview of **Simics**/DML programming
DML 1.4 Reference Manual	language specification
API Reference Manual	API function list for DML & C, describes ownership rules

Create stub device

```
$ bin/project-setup --device example-dev
$ ls modules/example-dev
test/ example-dev.dml Makefile module_load.py
```

The header of *example-dev*.dml is then:

```
dml 1.4;                 // Obligatory .dml header
device example_dev;      // Class name.
                         // Note: - changed to underscore _
```

Create stub interface with Python wrapper

```
$ bin/project-setup --interface sample
$ ls modules/sample-interface
Makefile sample-interface.dml sample-interface.h
```

Python support is enabled by `IFACE_FILES` in the Makefile. The generated C **struct** name is `sample_interface_t`.

Arithmetics

RHS is int64, LHS truncates

Assignments are equivalent to casts and hence can truncate:

```
local uint8 x = 0xffff // results in x == 0xff
```

All aritmetic operators like +, ∗ convert its operands to int64:

```
local uint16 i = 0x7fff; local int8 j = 2; // let us sum them:
                      calculates as int64 → 0x8001
local int16 x =           i + j;              // finally results in x == 1
                                    truncates to int16
```

Comparisons act as uint64 or int64: ==, <, <=

Comparisons on only uint64 act as proper uint64, however in comparisons int64 vs. uint64 the operands are converted to int64!

```
local uint64 u;
u = -1;    // equivalent to u = cast(-1, uint64) = 2^64 − 1, all ones
u == -1    // FALSE! Equivalent to int64(u) == int64(-1),
           // where upper bit is cropped: int64(u) == (2^63 − 1).
u == cast(-1, uint64) // true. As comparison is b/w two uint64
u > -1     // true, but unlike C! it's int64(u) > int64(-1): 2^63 − 1 > 1
```

Syntax

Statements

```
// Printing through log statements:
log log-type, level, groups: "format-string", arg_1, ..., arg_N;
              default 1  default 0 (no group)
// (The "format-string" is the same as in C, see 'man 3 printf')
```

log-level	usage rule
1	messages to be read by all, esp. errors
2	crucial events for boards/devices, e.g. their resets
3	any other messages to be read by device driver writers or validators
4	internal device debug messages

```
// Dynamic allocation (like malloc() in C):
local type * x = new type;
// e.g. for int array:
local int * x = new int[100];
delete x; // Deallocation like free() in C
// Raising/catching exceptions:
try {
    throw; // YES, no data can be carried by exception
} catch {
    ...
}
```

Expressions

```
sizeof value // : int — get byte size of the value
sizeoftype type // : int — get byte size of the type
x[10:8] // Get bits 10—8 of integer x:
x[8] // Get bit 8 of integer x:
```

Types

uint1...**uint64** and **int1**...**int64**.

Methods

They are called *methods*, not functions, because they accept **implicit** 1st argument — object (current device), like C++ methods.

```
method name(inputType_1 arg_1) -> (outputType_1, outputType_2)
{
    ...
    local outputType_1 var_1 = ...; local outputType_2 var_2 = ...;
    return (var_1,var_2);
}
// calling this method:
local inputType_1 val = ...;
local outputType_1 x; local outputType_2 y;
(x, y) = name(val);
```

Bitfields

```
bitfields 32 {
    uint3 upper_bits @ [31:29];
    ...
}
```

Object declarations

```
objectType objectName {
    method methodName {
        ...
    }
}
register regName @ offset is (template_1, ...);
// @ offset is a syntax for "param offset = offset;"
```

Module variables and other data objects

DML construct	check-pointed	fields	address-mapped	arbitrary data
session	-	-	-	+
saved	+	-	-	-
attribute	+	-	-	+
unmapped register	+	+	-	-
[normal] register	+	+	+	-

Interfaces

Definition in `.dml`:

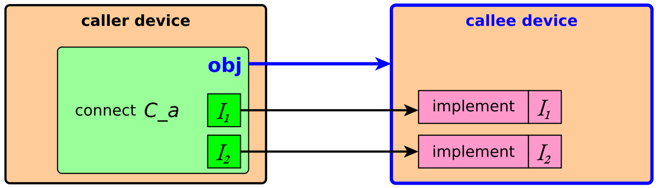
```
extern typedef struct {
    method name(type_1 value_1) -> out_type;
} sample_interface_t;
```

Obligatory definition in C `.h` file:

```
SIM_INTERFACE(sample) { // essentially 'typedef struct' also
    out_type (*name)(conf_object_t * obj, type_1 value_1);
}
```

Explaining `connect`s and `implement`s

- `connect`s are for **out**bound calls
- `implement`s are for **in**bound calls



	unnamed (whole-device)	named (wrapped in attribute)

`connect`	-	connect C_a { ❶
		  interface I_1 { ❷
		    ...
		  }
		  interface I_2 { ❷
		    ...
		  }
		}
		connect C_b { ❶
		  interface I_2 { ❷
		    ...

`implement`	implement I_1 {
	  ...
	}
	implement I_2 {
	  ...
	}
	(as in picture above)
		port P_A {
		  implement I_1 {
		    ...
		  }
		  implement I_2 {
		    ... *variant 1* ...
		  }
		}
		port P_B {
		  implement I_2 {
		    ... *variant 2* ...

To make the whole `connect` required:

❶
```
param configuration = "required";
// other variants: "optional" (default), "pseudo", "none"
```

❷ Individual interfaces are already **required** by default, to make them optional:

```
param required = false;
```

Debugging with Gdb

To debug simics and its modules itself:

```
# Terminal #1:
$ make clobber-my-module
$ make D=1 my-module
$ ./simics
> pid
12345
```

```
# Terminal #2:
$ bin/gdb
>>> attach 12345
>>> br file.dml:100        # set break point on line 100 of file.dml
>>> continue
```

```
# Back to terminal #1:
> run-command-file targets/platf/platf.simics
```

Using gdb for debugging target

```
load-module gdb-remote
new-gdb-remote 50000     # open port 50000
```

Attribute values

`attr_value_t` is a C union that can hold one of a few predefined types. Attributes values are allocated/packed by:

`attr_value_t x = SIM_make_attr_T(cType val)`, and extracted by:

`cType val = SIM_attr_T(x)`, where *T* and *cType* can be:

T	type spec	*cType* — DML/C type
uint64, int64	i	**uint64**, **int64**
boolean	b	**bool**
floating	f	**double**
string	s	**char***
object	o	**conf_object_t**
list	$[x_1 ... x_n]$	fixed-width tuple with n elements of types x_1, ..., x_n
list	$[x*]$	arbitrary-width array of x
list	$[x+]$	non-empty arbitrary-width array of x
list	$[x\{m:n\}]$	array of x with $m \le size \le n$
list	$[x\{n\}]$	fixed-width tuple with n elements of x
dict	D	array of **attr_dict_pair_t**
data	d	**uint8***
nil	n	**void** or x*
invalid		(none, used for indicating errors)

List items are accessed by `SIM_attr_list_item`. Type specs can be OR'ed as $x_1|x_2$. Type spec is used in `param type = "..."`. For first 4 types there are predefined DML templates `uint64_attr`, `int64_attr`, `bool_attr`, `double_attr`.

Attribute initialization

Execution stage	SIM_object_is_-configured(obj)	SIM_is_restor-ing_state()
Create object at 1st platform init	-	-
Load checkpoint	-	+
Load micro-checkpoint (reverse execution)	+	+
Manual attribute assignment (hot plug from Simics command line)	+	-

Standard register templates

Compile-time statements & conditional compilation

```
param p1 = 10; # Non-overridable parameter
param p2 default "value"; # Overridable parameter
template myTemplate {
    param p3; # undefined value — must be given on myTemplate instantiation
    ...
}
// Compile-time if
#if (p1 == 20) {
    ...
} #else #if {
    ...
} #else {
    ...
}
// Compile-time ternary operator #? #:
param mode = p1 == 20 #? "equal 20" #: "not equal 20";
// Represent value of parameter as string
param p1_str = stringify(p1); // results in "10"
```

Hash tables

```
import "simics/util/hashtab.dml";
...
local ht_str_table_t tab; // str — string (aka const char*) keys.
ht_init_str_table(&tab, /*keys_owned*/ true);
local double *value = new double; *value = 10.0;
ht_insert_str(&table, "key", cast(value, void *));
local double *get_back = cast(ht_lookup_str(&tab, "key"), double*);
assert *get_back == 10.0;
```

There are also tables for `int` keys or general (`common`) keys.

The secret of DML

Many "internal" features like registers and even connects are actually normal templates for objects (`is object;`) in plain DML defined in `1.4/dml-builtins.dml` and thus they can be expanded.

C API

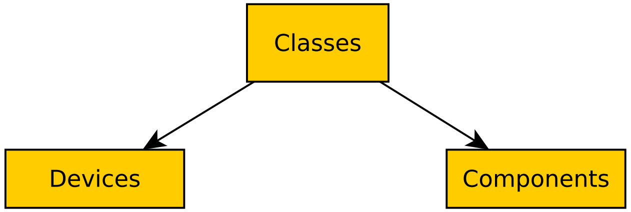
It's possible to do most things in C, e.g. create device by `SIM_create_object`, though normally it's done from Python components.

3 Simics configuration and build system

Glossary & Documentation

Python — this language is used for connecting devices together (writing components), for writing some (slow) devices, for unit-testing

Component — a special **Simics** class that forms a namespace tree and (typically) in its nodes contains instances of device classses. Components implement required `component` interface and optional `component_connector` interface.



Creating devices dynamically

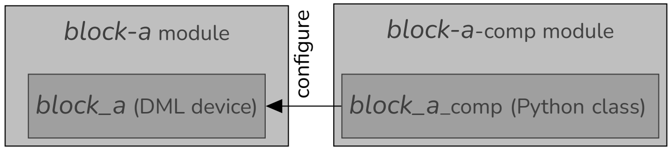
```
> create-myDevice-comp
> connect system.mydev system.some.connect
> instantiate-components
```

Modules/Components/Classes

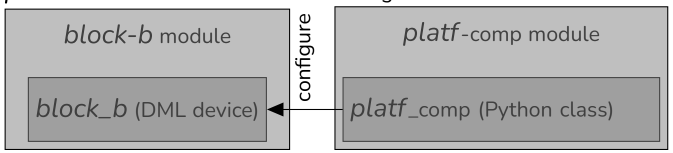
A **Simics** module includes classes, there are 2 types of them:

- devices, typically written in DML
- *components*, typically written in Python

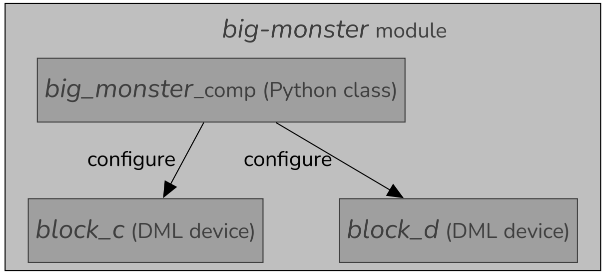
A typical layout with 1-to-1 device-component correspondence:



In simplest case there are no components for a device, so its platform *platf* will have to instantiate and configure the device:



There can be a module with 1 component and many classes:



Connecting devices

- from Script:

platf.$device_1$->$connect_1$ = platf.$device_2$

- from Python:

conf.platf.$device_1$.$connect_1$ = conf.platf.$device_2$

Components vs functions

Components/connectors are used:

- when there is a need to unite big number of devices to prevent pollution of the surrounding namespace
- when this is a separate device that can be used across different packages independently

Structure of components

Calling device code from Python

Then the interface can be invoked from **Simics** command line via Python:

@conf.platf.device.ifaces.$iface_1$.$method_1$(1, None)

https://github.com/a-mr/simics-cheatsheet Version 6.0.100