

Simics Cheatsheet

1 Simics Usage (command line)

1.1 Glossary & Documentation

Simics Base	<code>simics</code> executable + the set of supporting shared libraries (DLLs) + tools
<i>simulation</i>	running of code (target) on a model/platform (target) with advancement of time
<i>host</i>	a computer where simulation runs
<i>target</i>	a simulated code running in its isolated memory region, e.g. Linux or Windows guest
<i>platform</i>	a complete runnable model: full set-up of devices with CPU or with at least a clock provider
<i>package</i>	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
<i>Simics User's Guide</i>	documentation on Simics usage — both command line and Eclipse

Below **\$** stands for Unix shell prompt, **>** for **Simics** prompt. *Big_italic* text means user-supplied arguments for commands/functions.

1.2 Getting help

```
# Get help on some command:
> help instantiate-components
# Finding sections of documentation mentioning it:
> apropos instantiate-components
```

1.3 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
```

1.4 Start up

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

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

	<code>\$./simics start.simics</code>
→	<code>\$./simics</code> <code>> run-command-file start.simics</code>
	<code>\$./simics script.py</code>
→	<code>\$./simics</code> <code>> run-python-file script.py</code>
	<code>\$./simics -e '\$config_variable1=value; \$config_var2=value' start.simics</code> # ! NOT ./simics start.simics -e \$variables ...
→	<code>\$./simics</code> <code>> \$config_variable=value</code> <code>> \$config_var2=value</code> <code>> run-command-file start.simics</code>

1.5 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
```

1.6 Commands for running simulation

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

1.7 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
```

1.8 Registers

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

1.8.1 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
```

1.8.2 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
```

1.9 Connects — attributes that point to other devices

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

1.10 Device information: static

```
# 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
```

1.11 Device information: dynamic

```
# 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
```

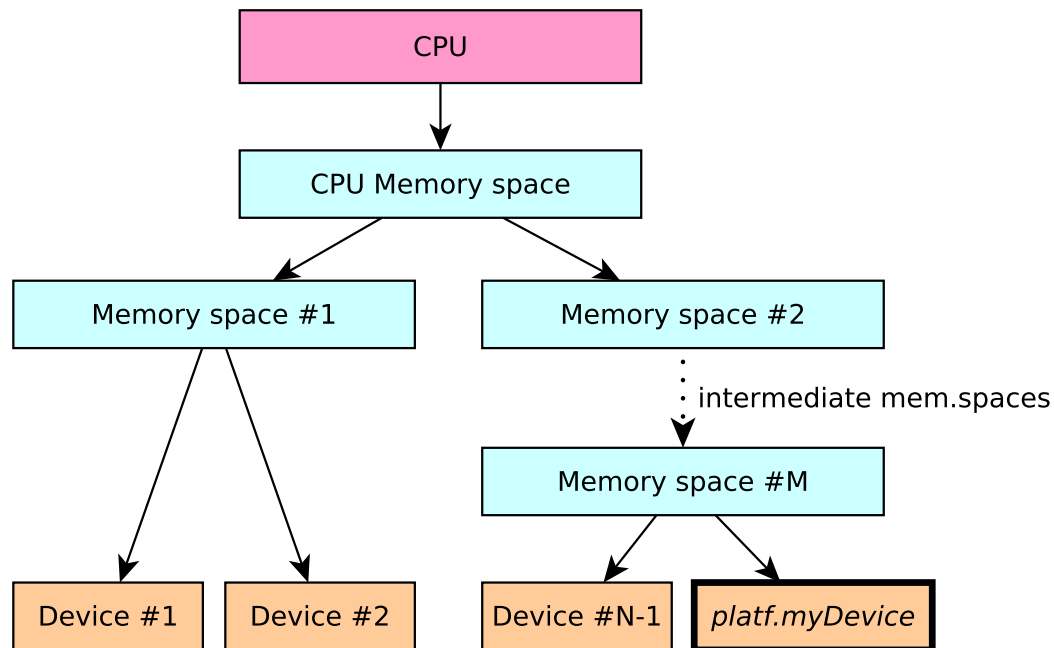
1.12 Debugger commands

To use **Simics** to debug target:

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

1.13 Memory spaces

All target software runs on CPU and can access devices only through hierarchy of memory spaces:



```
# To find memory spaces the device is mapped in:
> devs platf.myDevice
```

1.14 Examining current state

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

1.15 x86-specific and platform-specific commands

memory-trace <i>addr</i>	path to target <i>addr</i>
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

1.16 Moving files target ↔ host

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

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

1.17 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
```

1.18 Check points

```
write-configuration "checkpoint_name" — save checkpoint
```

```
read-configuration "checkpoint_name" — restore it back
```

1.19 Miscellaneous

```
# To dump network packages (for analysis with Wireshark):
```

```
> pcap-dump link=ethernet_switch0 filename=myFile
```

```
# Treat all input as Python code
```

```
> python-mode
```

```
> > > Enter your code
```

2 Simics DML 1.4 Programming & C/DML API

2.1 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 <i>interfaces</i> .
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
model writers	those who write Simics devices
users	those who use Simics devices: device driver writers, firmware/UEFI writers, validators/testers, etc

2.2 Getting help

```
# Print explanation of C/Python/DML API functions:
```

```
> api-help SIM_add_configuration
```

2.3 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 _
```

2.4 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`.

2.5 Arithmetics

2.5.1 RHS is int64, LHS truncates

Assignments are equivalent to casts and hence can truncate:

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

All arithmetic 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;
// truncates to int16
// finally results in x == 1
```

2.5.2 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) = 264 - 1, all ones
u == -1 // FALSE! Equivalent to int64(u) == int64(-1),
// where upper bit is cropped: int64(u) == (263 - 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): 263 - 1 > 1
```

2.6 Syntax

2.6.1 Statements

```
// Printing through log statements:
log log-type, level, groups: "format-string", arg1, ..., argN;
// default default 0 (no group)
// (The "format-string" is the same as in C, see 'man 3 printf')
```

<i>log-level</i>	usage rule
1	most important messages (for both users and model writers), typically error
2	crucial events for boards/devices, e.g. their resets
3	any other messages (for users)
4	internal device debug messages (for model writers)
<i>log-type</i>	usage rule
info	informational message
spec_viol	specification violation by target software (for users)
unimpl	attempt to use not implemented functionality (for users)
error	internal device error (for model writers).  There is a limit (default 10_000) after which simulation stops!
critical	like spec_viol or error, but  stops simulation

```
// 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 {
    ...
}
```

2.6.2 Expressions

```
sizeof value // : int — get byte size of the value
sizeof 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:
```

2.6.3 Types

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

2.6.4 Methods

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

```
method name(inputType1 arg1) -> (outputType1, outputType2) {
    ...
    local outputType1 var1 = ...; local outputType2 var2 = ...;
    return (var1, var2);
}
// calling this method:
local inputType1 val = ...;
local outputType1 x; local outputType2 y;
(x, y) = name(val);
```

2.6.5 Bitfields

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

2.6.6 Object declarations

```
objectType objectName {  
    method methodName {  
        ...  
    }  
}  
register regName @ offset is (template1, ...);  
// @ offset is a syntax for "param offset = offset;"
```

2.6.7 Module variables and other data objects

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

2.6.8 Interfaces

Definition in `.dml`:

```
extern typedef struct {  
    method name(conf_object_t *obj, type1 value1)  
        -> out_type;  
} sample_interface_t;
```

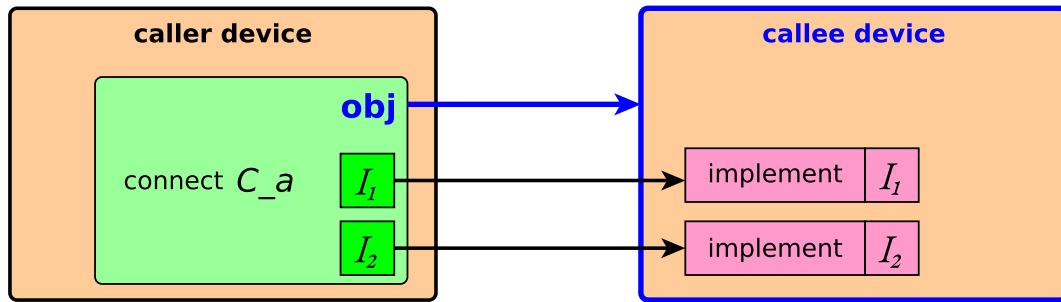
Obligatory definition in C `.h` file **for using the interface from Python**:

```
SIM_INTERFACE(sample) { // essentially 'typedef struct' also  
    out_type (*name)(conf_object_t *obj, type1 value1);  
}  
#define SAMPLE_INTERFACE "sample" // necessary!
```

Then it is called **without** `obj`: just `name(value1)`.

2.6.9 Explaining `connect`s and `implement`s

- `connect`s are for **outbound** calls
- `implement`s are for **inbound** calls



unnamed (whole-device)	named (wrapped in attribute)
<div>connect</div> <div>-</div>	<pre>connect C_a { ❶ interface I1 { ❷ ... } interface I2 { ❷ ... } } connect C_b { ❶ interface I2 { ❷ ... } }</pre>
<div>implement</div> <div> <pre>implement I1 { ... } implement I2 { ... }</pre> (as in picture above) </div>	<pre>port P_A { implement I1 { ... } implement I2 { ... variant 1 ... } } port P_B { implement I2 { ... variant 2 ... } }</pre>

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;`

⚠ `implement {}` does NOT check that all (or any!) methods of an interface are really provided, defaulting to **NULL**.

2.7 Calling device code from Python

If device `implement` interface `iface1` then its methods can be invoked from **Simics** command line via Python:

```
@conf.platf.device.iface.iface1.method1(argument1, ...)
```

2.8 Debugging with Gdb

To debug simics and its modules itself:

Terminal #1:

Recompile your module with debugging support:

\$ 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

2.8.1 Using gdb for debugging target

load-module gdb-remote

new-gdb-remote 50000 **# open port 50000**

2.9 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:

<i>T</i>	type spec	<i>cType</i> — DML/C type
uint64, int64	i	uint64, int64
boolean	b	bool
floating	f	double
string	s	char*
object	o	conf_object_t
list	$[x_1 \dots x_n]$	fixed-width tuple with n elements of types x_1, \dots, 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 $m \leq \text{size} \leq n$
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**.

2.10 Attribute initialization

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

2.11 Standard register templates

2.12 Compile-time statements & conditional compilation

```
param p1 = 10; // Non-overrideable parameter
param p2 default "value"; // Overrideable parameter
template myTemplate {
    param p3; // undefined value — must be given on myTemplate in-
stantiation
    ...
}
// 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"
```

2.13 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.

2.14 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.

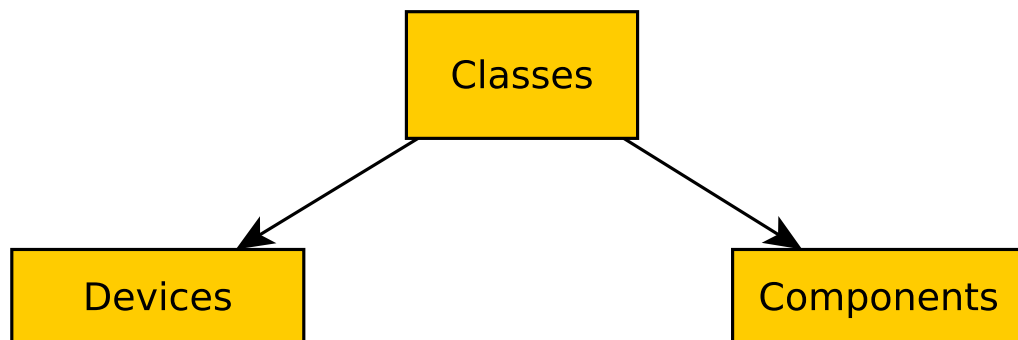
2.15 C API

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

3.1 Glossary & Documentation

<i>Python</i>	this language is used for connecting devices together (writing components), for writing some (slow) devices, for unit-testing
<i>Component</i>	a special Simics class that forms a namespace tree and (typically) in its nodes contains instances of device classes. Components implement required <code>component</code> interface and optional <code>component_connector</code> interface.



3.2 Creating devices dynamically

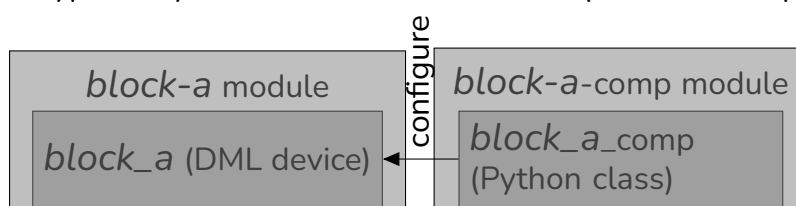
```
> create-myDevice-comp "system.mydev"  
> connect system.mydev system.other.connect  
> instantiate-components
```

3.3 Modules/Components/Classes

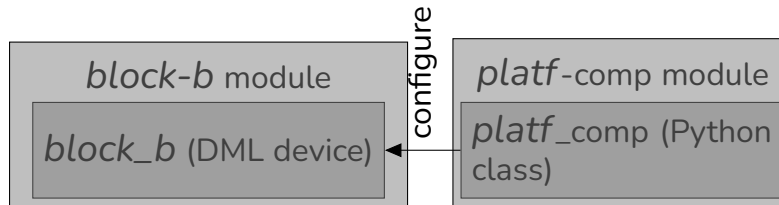
A **Simics** module includes classes, there are 2 types of them:

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

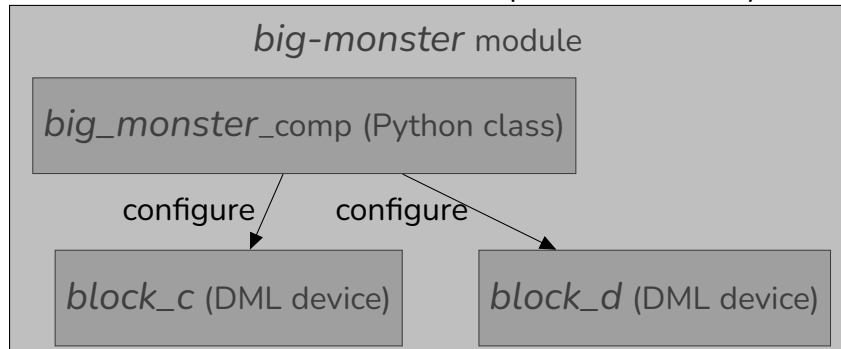
A typical layout with 1-to-1 device-component correspondence:



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:



3.4 Connecting devices

- from Script:

```
platf.device1->connect1 = platf.device2
```


- from Python:

```
conf.platf.device1.connect1 = conf.platf.device2
```

3.5 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

 Do not confuse: *Connectors* connect *component* objects, while *connects* (+implements) connect *device* objects. `connect` command acts on connectors only!

3.6 Structure and initialization order

A typical structure of component class code:

```

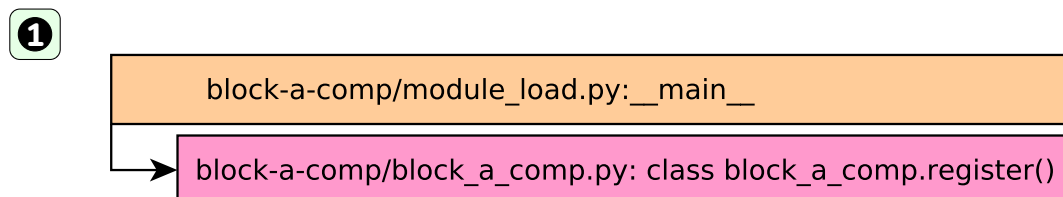
class block_a_comp(StandardComponent):
    @classmethod
    def register(cls):
        ...
    def __init__(self):
        ...
    def setup(self):
        ...
    def add_objects(self):
        ...
    class attribute_name(ConfigAttribute):
        def _initialize(self):
            ...
        def _finalize(self):
            ...
    class component_connector(Interface):
        def get_connect_data(self, block_a_connector):
            ...
        def connect(self, block_a_connector, data_from_b):
            ...
    ...
    class component(StandardComponent.component):
        ...

```

Let us examine initialization of modules → components → devices for this sample program:

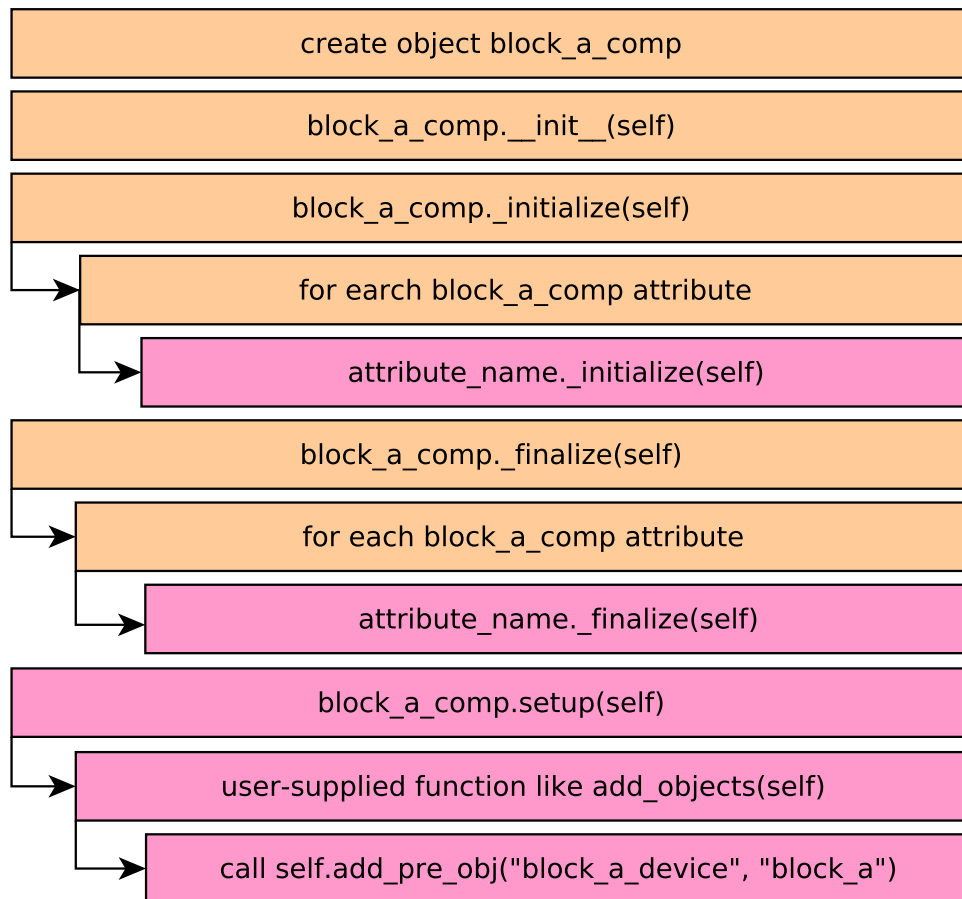
- ❶ load-module block-a-comp
- ❷ load-module block-b-comp
- ❸ create-block-a-comp "block_a_component"
- ❹ create-block-b-comp "block_b_component"
- ❺ connect block_a_component.eth0 block_b_component.eth0
- ❻ instantiate-components

The corresponding break down of functions that are normally **simics-defined** or **user-defined**.



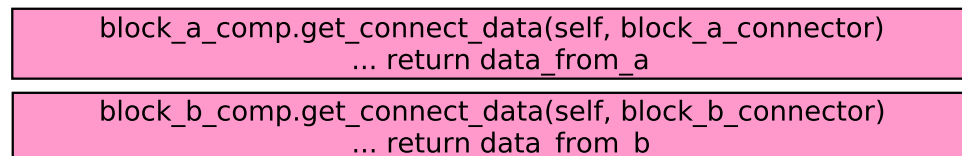
❷ The same for **block_b_comp**

❸ Note that **add_objects** adds normally pre-configured objects:



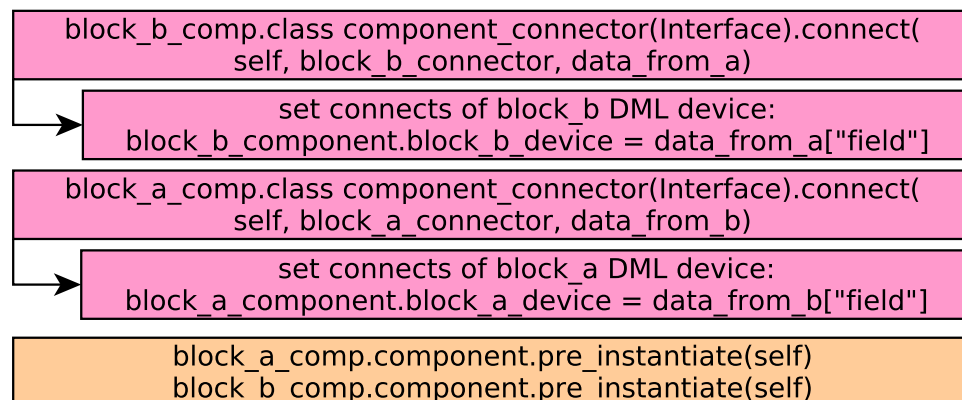
④ The same for `block_b_comp` & `block_b`

⑤



Note `get_connect_data` is basically just a `switch/case` statement that chooses which data (object references, port names, etc) to pass to `connect` function call below.

6a



6b Pre-configuration phase ends and finally `instantiate-components` begins to configure real DML objects:

block_a: call `init()` on all attributes with ``init`` template applied.
block_a: call device method `init()`

block_b: call `init()` on all attributes with ``init`` template applied.
block_b: call device method `init()`

block_a: call `set()` for all connects/attributes

block_b: call `set()` for all connects/attributes

block_a: call device method `post_init()`

block_b: call device method `post_init()`

After that phase `self.get_slot("name")` returns real, already configured, device objects.

6c Rarely some additional tweaks are required on configured objects:

`block_a_comp.component.post_instantiate(self)`

`block_b_comp.component.post_instantiate(self)`

Copyright © 2021—2022 Andrey Makarov
<https://github.com/a-mr/simics-cheatsheet> Version 0.2 (Simics 6.0.116)