virtutech

Simics Programming Guide

Simics Version	3.0
Revision	1376
Date	2007-01-24

Contents

Chapter 1

About Simics Documentation

1.1 Conventions

Let us take a quick look at the conventions used throughout the Simics documentation. Scripts, screen dumps and code fragments are presented in a `monospace` font. In screen dumps, user input is always presented in bold font, as in:

```
Welcome to the Simics prompt
simics> this is something that you should type
```

Sometimes, artificial line breaks may be introduced to prevent the text from being too wide. When such a break occurs, it is indicated by a small arrow pointing down, showing that the interrupted text continues on the next line:

```
This is an artificial ↵
line break that shouldn't be there.
```

The directory where Simics is installed is referred to as `[simics]`, for example when mentioning the `[simics]/README` file. In the same way, the shortcut `[workspace]` is used to point at the user's workspace directory.

1.2 Simics Guides and Manuals

Simics comes with several guides and manuals, which will be briefly described here. All documentation can be found in `[simics]/doc` as Windows Help files (on Windows), HTML files (on Unix) and PDF files (on both platforms). The new Eclipse-based interface also includes Simics documentation in its own help system.

Simics Installation Guide for Unix and for Windows

These guides describe how to install Simics and provide a short description of an installed Simics package. They also cover the additional steps needed for certain features of Simics to work (connection to real network, building new Simics modules, . . .).

Simics User Guide for Unix and for Windows

These guides focus on getting a new user up to speed with Simics, providing information on Simics features such as debugging, profiling, networks, machine configuration and scripting.

Simics Eclipse User Guide

This is an alternative User Guide describing Simics and its new Eclipse-based graphical user interface.

Simics Target Guides

These guides provide more specific information on the different architectures simulated by Simics and the example machines that are provided. They explain how the machine configurations are built and how they can be changed, as well as how to install new operating systems. They also list potential limitations of the models.

Simics Programming Guide

This guide explains how to extend Simics by creating new devices and new commands. It gives a broad overview of how to work with modules and how to develop new classes and objects that fit in the Simics environment. It is only available when the DML add-on package has been installed.

DML Tutorial

This tutorial will give you a gentle and practical introduction to the Device Modeling Language (DML), guiding you through the creation of a simple device. It is only available when the DML add-on package has been installed.

DML Reference Manual

This manual provides a complete reference of DML used for developing new devices with Simics. It is only available when the DML add-on package has been installed.

Simics Reference Manual

This manual provides complete information on all commands, modules, classes and haps implemented by Simics as well as the functions and data types defined in the Simics API.

Simics Micro-Architectural Interface

This guide describes the cycle-accurate extensions of Simics (Micro-Architecture Interface or MAI) and provides information on how to write your own processor timing models. It is only available when the DML add-on package has been installed.

RELEASENOTES and LIMITATIONS files

These files are located in Simics's main directory (i.e., `[simics]`). They list limitations, changes and improvements on a per-version basis. They are the best source of information on new functionalities and specific bug fixes.

Simics Technical FAQ

This document is available on the Virtutech website at `http://www.simics.net/support`. It answers many questions that come up regularly on the support forums.

Simics Support Forum

The Simics Support Forum is the main support tool for Simics. You can access it at `http://www.simics.net`.

Other Interesting Documents

Simics uses Python as its main script language. A Python tutorial is available at `http://www.python.org/doc/2.4/tut/tut.html`. The complete Python documentation is located at `http://www.python.org/doc/2.4/`.

Chapter 2

Tutorial

The *DML Tutorial* provided as a separate document is a good introduction to creating new Simics modules. Much of what it presents can subsequently be explored more in-depth in this Guide.

Chapter 3

Build Environment

The functionality of Simics can be extended by user-written modules. Modules can, among other things, contain definitions of new device models. After a successful build, these devices can be included and used in a machine setup.

This chapter will describe how to proceed in order to add new modules to a Simics installation. A separate workspace is used for the user-written modules, which permits many users to share a system-wide (read-only) Simics installation.

As device modeling is the most common task of extending Simics, a new programming language, DML, have been developed to simplify that task. The workspace environment supports modules written in C and DML.

3.1 Notes for Windows Users

To develop new modules for Simics on the Windows platform, you need to work in the Cygwin environment with the MinGW compiler suite installed. See the *Installation Guide* for installation instructions. You also need to ensure that the path to Simics does not contain any spaces, since this confuses the build scripts. Cygwin provides a Unix-like environment, so all the explanations below are written with a Unix environment in mind, including Unix-style pathnames.

3.2 Setting up a Workspace

Note: The workspace makefiles require GNU make (a.k.a. `gmake`), version 3.77 or later, which is available from `ftp://ftp.gnu.org/gnu/make`. In the following text, when you're asked to run `gmake`, this refers to running the GNU make binary, which may be called `gmake` or `make`, depending on your installation.

A workspace is a directory which contains all necessary user-specific files needed to run Simics and develop modules. Setting up a workspace is done using the `workspace-setup` scripts, like this:

```
$ [simics]/bin/workspace-setup.bat $HOME/my-simics-workspace
```

where *[simics]* is the location of the Simics-installation. The script can also be invoked in a `cmd.exe` command prompt window. The `.bat` extension is necessary if the script is invoked in a Cygwin shell; Cygwin does not automatically append `.bat` as `cmd.exe` does.

Note: The workspace setup script does **not** rely on the `configure` being run in the installed Simics.

The script will create a workspace directory with the following contents:

```
simics        simics-eclipse  GNUmakefile
compiler.mk   config.mk       .workspace-properties
modules/      targets/        host/
```

`simics`
Starts Simics in command-line mode.

`simics-eclipse`
Starts Simics with the Eclipse frontend.

`GNUmakefile`
Makefile to build all modules under the `modules` directory. The file is called `GNUmakefile` to signify that it requires GNU make. Do not edit this file, instead you may create a file called `config-user.mk`, where you can override settings in `config.mk`.

`compiler.mk`
Make file that selects the C compiler to use by setting the *CC* variable. A matching C++ compiler will be searched for by `config.mk` in the same path as *CC* if *CXX* is not set. The `compiler.mk` file will not be overwritten when the workspace is updated.

`config.mk`
Includes `[simics]/config/config.mk` that contains default definition of make flags for different compilers, such as *CFLAGS*. Do not edit this file; override variables in `config-user.mk` instead.

`config-user.mk`
Optional file that may contain user defined make variables overriding the ones in `[simics]/config/config.mk`.

`module-user.mk`
Optional file that may contain user defined make targets and variables overriding the ones in `[simics]/config/module.mk`.

`modules/`
Contains user-developed modules. The default target in `GNUmakefile` builds all modules in the `modules` directory.

targets/
Contains some pre-configured machines, to be used as examples.

<**host**>**/**
The build working directory, which is named after the host type, for example `x86-linux`, `v9-sol8-64`, `amd64-linux` or `x86-win32`. When a module is compiled, any intermediate build files, like dependency and object files (`.d .o`) are generated in the `<host>/obj/modules/<module>/` directory. The resulting module file is placed in `<host>/lib/`, and the Python command file for the module is copied to the `<host/lib/python/` directory.

.workspace-properties
For internal use.

When the workspace has been created, you may type **make** (or possibly **gmake**) to build all the modules, or **./simics** to start Simics.

In order to rebuild all modules, type **make clean**, followed by **make**. In order to rebuild just a single module, type **make clean-**<**modulename**>, for example:

```
$ make # builds all modules
$ make clean-mymodule # removes all objectfiles for "mymodule"
$ make mymodule # builds "mymodule"
```

The `clean` targets only remove object files and similar intermediates for the module not needed when running. To remove the actual module files as well, use `make clobber` or `make clobber<modulename>`.

3.2.1 Workspace setup script invocation

The workspace setup script is used to *create* and *upgrade* workspaces. It can also create module skeletons to start with when writing new devices.

The setup script is located in the `bin` subdirectory of the Simics installation, and is invoked like this:

```
$ [simics]/bin/workspace-setup [options] [workspace]
```

You may put the `[simics]/bin` directory in your *PATH* variable, and invoke the script directly.

```
$ workspace-setup [options] [workspace]
```

Option summary (can also be shown using the *--help* option):

-v, --version
Prints information about Simics (version, installation directory).

-n, --dry-run
Execute normally, but do change or create any files.

-q, --quiet
Do not print any info about the actions taken by the script.

--force
Force using a non-empty directory as workspace. Note: even with this option, modules in the module sub-directory will **not** be overwritten.

--device=MODULE_NAME
Generate skeleton code for a device, suitable to use as a starting point for implementing your own device. The default implementation language is DML. See the *--c-device* and *--py-device* options for creating devices using other languages. If the device already exists, this option is ignored. To recreate the skeleton, remove the device directory.

--c-device=MODULE_NAME
Similar to *--device*, but creates a device using C instead of DML.

--py-device=MODULE_NAME
Similar to *--device*, but creates a device using Python instead of DML.

--copy-device=MODULE_NAME
Copies the source for a sample device/module into the workspace. The files will be copied from the Simics installation. If the device already exist in your workspace, you must manually delete it first.

3.2.2 Updating workspaces

To upgrade your workspace to a new Simics version, run the script again with no arguments.

```
$ cd $HOME/my-simics-workspace
$ [path-to-new-simics]/bin/workspace-setup
```

It will do the necessary updates in the workspace, but leave the user-modifiable files intact. (Modified files that need to be overwritten are saved in backup versions with the extension "`.~N~`" (Unix) or "`~N~.backup`" (Windows), where N is the first free number.)

3.3 Adding modules to a workspace

The `modules` subdirectory contains source code for modules, one module per directory entry.

3.3.1 Creating a new DML module

To add a DML module to a workspace, specify the *--device* option.

```
$ cd $HOME/my-simics-workspace
```

```
$ [simics]/bin/workspace-setup --device mydevice
```

This will create some skeleton code under the `modules/` directory.
After adding a module, you can build it using the top-level makefile:

```
$ gmake
```

To emphasize that the makefile require GNU Make, it is called `GNUmakefile`. The sub-makefiles in the module directories are named `Makefile`.

When running make, command lines will not be printed by default. To see the commands, pass `VERBOSE=yes` to make:

```
$ gmake VERBOSE=yes
```

3.3.2 Creating a new C module

An example module written in C can be added in the same way as DML modules, but using the *--c-device* option.

```
$ cd $HOME/my-simics-workspace
$ [simics]/bin/workspace-setup --c-device my_c_device
```

3.3.3 Adding an Existing Module

The top-level makefile will automatically attempt to build all modules under the `modules/` directory. If you have a module somewhere else, you may put a symlink there:

```
$ ln -s $HOME/mydevice ./my-workspace/modules/
```

You may need to adapt the Makefile for the workspace-based build environment. Use a generated skeleton Makefile as a template for your rewrite.

A module to which the source is distributed with Simics, can be copied into the workspace by using *--copy-module*.

```
$ cd $HOME/my-simics-workspace
$ [simics]/bin/workspace-setup --copy-module gdb-remote
```

3.4 Advanced Tweaks

3.4.1 Platform specific notes

The workspace setup script has a default set of make-variables (*CC*, *CFLAGS*, etc.), which are setup in the `compiler.mk` and `[simics]/config/config.mk` files. If you need to

change any of the variables in `config.mk`, you may override them in `config-user.mk`. The `compiler.mk` may be edited by the user.

The makefile with the actual build rules for all modules is `[simics]/config/module.mk`. This file is included at the end of each module `Makefile`. To override rules or variables in this `module.mk`, add a `module-user.mk` file in the workspace, similar to the `config-user.mk` file described above.

Windows

There is a special wrapper utility called `cygwrap` which is used for launching native windows binaries, which converts the Cygwin-style paths used by GNU make to native paths. It also does some argument translation, for example, to allow the *LIBS* variable to contain Unix-style `-l` library options.

The default set of variables assumes that you will be using GCC. If you want to use a different compiler, you need to change the *CC* variable in `compiler.mk`. The flags for the compiler are setup in `[simics]/config/config.mk`.

If your compiler is not supported by `config.mk`, please report to Virtutech.

3.4.2 Module Makefiles

In order to make the build environment in Simics recognize a module as a build target, there must be a makefile called `Makefile` in its source directory.

A module makefile must set up a number of make variables and then include the generic makefile for Simics modules. The following is an example of a module's `Makefile`, for a module written in C:

```
MODULE_CLASSES=FAS366U

SRC_FILES=esp.c

MODULE_CFLAGS=-DFAS

SIMICS_API = 3.0

include $(MODULE_MAKEFILE)
```

- `MODULE_CLASSES` (optional) is a space-separated list of configuration classes that this module registers. This information is used by ***SIM_get_class()*** to determine that this module should be automatically loaded when one of the listed classes is requested.
- `SRC_FILES` is a list of all source files for the module. You should only list C or C++ files here. Note that C source files must have a `.c` suffix, and that C++ source files must have a `.cc` suffix. For modules written in python a `.py` file can be listed instead.
- `MODULE_DIR` (optional) is the name of the module source directory; this may be different from the module name. Defaults to `[workspace]/modules/<module>`. This

primarily affects how the module makefile sets up VPATH, but do not use this just to allow source files to be found in another directory, use EXTRA_VPATH for that.

- MODULE_CFLAGS (optional) are module specific flags for the compiler.
- MODULE_LDFLAGS (optional) are module specific flags for the linker.
- SIMICS_API (optional) specifies the version of the Simics API that the module is written for. See section 9.3 for a description on how to compile old modules with a new Simics version.
- MODULE_EXPORT_SYMS (optional) is a space separated list of file names. These files should contain symbol names, each on a separate line, to be exported globally from the module. This information is used when linking the module. Setting this variable could be useful if your module wants to load other shared object files or DLLs. The symbols init_local and _module_capabilities_ are always exported.
- EXTRA_VPATH (optional) can be set to instruct the generic module makefile to look for source files in more directories than in MODULE_DIR.
- EXTRA_OBJ_FILES (optional) may contain names of object files already compiled, that should be linked into the final module.

The user can also add new rules to the makefile, either before or after the inclusion of the generic $(MODULE_MAKEFILE). This is usually not needed.

The following variables can be used in the module's makefile (i.e. [workspace]/modules/<module>/Makefile). They should be considered *read-only*, i.e. they should not be changed.

- SIMICS_WORKSPACE is the full path to the workspace directory.
- TARGET_DIR is the directory in which compiled modules are placed ([workspace]/[host-type]/lib).
- SRC_BASE is the full path to the workspace modules directory ([workspace]/modules).
- HOST_TYPE is the Simics host architecture, i.e. what OS/hardware Simics has been compiled for, such as x86-win32 or v9-sol8-64.
- TARGET is the name of the module being compiled.

3.4.3 Standard Module Host Defines

There are a number of defines that are set depending on the host that the module is being compiled on. They are usually not needed, but useful in some special cases. Examples: HOST_64_BIT, HOST_32_BIT, HOST_BIG_ENDIAN, and HOST_LITTLE_ENDIAN. There are also defines specifying the host architecture and host operating system. All these defines are set in the Simics include file global.h.

3.4.4 User Defined Module Version Numbers

It is possible to set a user defined version number (or string) in loadable modules. This is done by setting the `USER_VERSION` define in `MODULE_CFLAGS` in the module's `Makefile`. The version number will be printed by the **list-modules** and **list-failed-modules** commands.

3.4.5 Module Loading Support

When Simics starts, it will check all modules for their supported architecture and word size. Only modules that match the running Simics binary will be available for loading into Simics. While scanning the modules, Simics will also check what classes the module will register when it is loaded. This way modules can be loaded automatically when classes they define are used in a configuration.

If a module does not match the current Simics, it will be added to the list of failed modules. This list can be displayed with **list-failed-modules**, that takes an optional parameter `-v` for more verbose output.

```
simics> list-failed-modules

Current module version number: 1050  Lowest version number supported: 1050

MODULE           DUP  VERSION USR_VERS  LINK
-----------------------------------------------
8042                  1040
image                 1040
spitfire-mmu.so                          X
```

The columns after the module name (or file name in the case of a link error) indicate different kinds of errors. An `X` in the `DUP` column means that this module could not be loaded because this module has the same name as another module found in the Simics module search path, and that this one was overridden. An `X` in the `VERSION` column means that the module was created for another, non-compatible, version of Simics. `LINK` means that this module cannot be loaded into Simics because of unresolved symbols. Use **list-failed-modules -v** to see the actual error message from the run-time module loader.

Chapter 4

Programming Concepts

A Simics module is just some executable code loaded dynamically into the simulator, so it could virtually perform any operation. To be of any practical use however, a module has to interact with Simics, other modules, or the user. Simics provides a set of functions (the Simics API) for the modules to work within the simulator framework. The API defines a number of concepts like classes, objects, interfaces, haps and events, which will be presented below. The *Simics Reference Manual* provides the complete API documentation.

Simics modules can be written using different programming languages:

DML (Device Modeling Language)
: DML has been designed to make writing new device model as simple as possible. It integrates in the language many of the concepts defined by the Simics API so that the user can focus on the model rather than on the interaction with Simics. *This is by far the best language to write a device model*. Note that although it was designed with device models in mind, DML can be used to develop other types of modules.

Python
: Python can be used to quickly write a new Simics module. It is well suited for Simics extensions that are not called very often, like some types of statistics gathering, or BIOS emulation. It can also be used for fast prototyping of more complex extensions that later on will be rewritten in a compiled language like C or DML. Note that in all cases, DML is easier to use than Python to write a device model.

C/C++
: C/C++ can be used for writing any type of Simics modules. Writing in C/C++ exposes all interactions with Simics, which can be sometimes cumbersome. C/C++ development is only recommended if the functionality being developed could seriously impact Simics performance (by being called very often during execution, by manipulating large quantities of data, ...). A few specialized functions in the API (like the Micro-Architectural Interface) are only available to C/C++ modules.

This chapter will cover the concepts that the Simics API defines. It will also present how to use them, first in DML, then in Python and C/C++.

4.1 Classes and Objects

The *Simics User Guide* explains how a Simics configuration is described in terms of objects and attributes . Each device is represented by an object whose attributes correspond to the state of the device.

Most Simics modules work the same way: they define *classes* describing how an object looks like and registering its attributes and interfaces. Objects can be instantiated from these classes and connected to the rest of the configuration.

Note that the *module* and *class* concepts are not linked, so a module does not necessarily define any class and it may also define more than one.

To learn more about classes and objects in DML, refer to section 5.1. For C and Python, refer to sections 6.1 to 6.3.

4.2 Attributes

A Simics class can register *attributes* that will act as data members for all objects instantiated from this class. For Simics, an attribute is an abstraction, defined by its type and a pair of ***get()/set()*** functions. When an attribute is read (i.e., when the ***SIM_get_attribute()*** function is used on the object), the corresponding ***get()*** function is called. Likewise, when an attribute is written to, the ***set()*** function is executed. These functions can perform any kind of operation provided they return a value (for ***get()***) or accept a value to be written (for ***set()***).

Attributes have different types and properties. The *Simics User Guide* presents a high level description of attributes. Let us here have a closer look at their definition.

Definition

The C and DML definition of the structures representing an attribute are the following:

```
typedef enum {
        Sim_Val_Invalid  = 0,
        Sim_Val_String   = 1,
        Sim_Val_Integer  = 2,
        Sim_Val_Floating = 3,
        Sim_Val_List     = 4,
        Sim_Val_Data     = 5,
        Sim_Val_Nil      = 6,
        Sim_Val_Object   = 7,
        Sim_Val_Dict     = 8,
        Sim_Val_Boolean  = 9
} attr_kind_t;

typedef struct {
        integer_t          size;
        struct attr_value *vector;
```

```
} attr_list_t;

typedef struct {
        integer_t          size;
        uint8             *data;
} attr_data_t;

typedef struct {
        struct attr_value key;
        struct attr_value value;
} attr_dict_pair_t;

typedef struct {
        integer_t         size;
        attr_dict_pair_t *vector;
} attr_dict_t;

typedef struct attr_value {
        attr_kind_t kind;
        union {
                const char    *string;
                integer_t      integer;
                integer_t      boolean;
                double         floating;
                attr_list_t    list;
                attr_dict_t    data;
                attr_data_t    data;
                conf_object_t *object;
        } u;
} attr_value_t;
```

When using an attribute in C or DML, one should take care of only using the union member that corresponds to the attribute type in the `attr_value_t` structure. Here are some example of attribute manipulation in C and DML:

```
// create an integer attribute
attr_value_t a;
a.kind = Sim_Val_Integer;
a.u.integer = 4711;

// or better, using help functions
attr_value_t a;
a = SIM_make_attr_integer(4711);

// create a list attribute
```

```
attr_value_t l;
l = SIM_alloc_attr_list(2);

// writing to the first element
l.u.list.vector[0] = SIM_make_attr_integer(1)

// reading the second element of the list
integer_t other = l.u.list.vector[1].u.integer;
```

A complete documentation of attributes related functions is provided in the *Simics Reference Manual*.

In Python, attributes are automatically converted to or from the corresponding Python type, so manipulating attributes is completely transparent:

Attribute Type	Python Equivalent
Invalid	Cast a `SimExc_Attribute` exception.
String	`str` (a Python string)
Integer	`int` or `long`
Floating	`float`
List	`list`
Data	`tuple`
Nil	`None`
Object	An object from the **conf** namespace.
Dict	`dict`
Boolean	`bool`

Type

When registering an attribute, a type definition should be provided for Simics to check that the attribute is always set properly. This type definition is a string defined by to the following rules:

- A simple type is represented by a single letter: `i` (integer), `f` (floating), `s` (string), `b` (boolean), `d` (data), `o` (object), `D` (dictionary), `n` (nil) or `a` (all).
- You can match several types by using the operator | like in `s|o` (string OR object). Note that if you provide a string when an object is expected, Simics will call the ***SIM_get_object()*** function to try to translate the string into a valid object.
- Lists are defined by square brackets: `[` and `]`. Lists can be specified in three ways:
  - Lists with a fixed amount of elements: `[iffsf]` matches (integer, floating, floating, string, floating).
  - lists of an arbitrary number of elements: `[i*]` and `[i+]` both match any list containing only integers, with the difference that `[i*]` will match an empty list, while `[i+]` requires at least one list element.

  - lists with a size specifier: `[i{1:4}]` matches lists of integers with 1 to 4 elements. `[i{4}]` is the same as `[i{4:4}]`, i.e. a list of four integers.

  When you specify a size, the list definition can have only one element.

- `|` has higher priority then concatenation in list description, so `[i|si|s]` should be read as matching any two-elements list with integer or string elements. For clarity you can use a comma anywhere in the type definition, it will be ignored. The example could be written as `[i|s,i|s]`

As an example, the type for `sim.simics_path` is `"s|[s*]"`, which means that it can either match a string or a (possibly empty) list of strings.

When writing complex attributes, you may notice that the type description strings don't cover all the possibilities offered by the attribute structure definition. If you need to register attributes that can not be described in a type string (like complex variable size lists), you will need to use the `a` type and perform the type checking by yourself in the ***set()*** function. You may also want to review your attribute and change its definition to match a possible type description, or divide your attribute into several simpler attributes.

For example, an attribute accepting a list composed of one object and one or more integers can not be described in a type string (list definition with variable size can only have one element). You may want to rewrite your attribute to use a sub-list for the integers: `[o[i+]]`, or you can perform type checking yourself.

To learn more about attributes in DML, refer to section 5.2. For C and Python, refer to section 6.4.

Attribute Data Ownership

Attributes of the `Sim_Val_String` and `Sim_Val_Data` types contain references to data that does not fit in the attribute structure itself. In addition, the types `Sim_Val_List` and `Sim_Val_Dict` have an extended attribute structure allocated. To avoid memory leaks and corruption, it is important to keep track of ownership of this data.

When the `attr_value_t` type is used for function arguments, it is the caller that owns the data, and that is responsible for freeing it after the call. If the called function wants to keep a reference to the attribute value, it should make a copy of it.

When a function returns an `attr_value_t`, it may return a reference to a static data area, that will be reused in future calls. The caller has to use the returned data, or create a copy of it, before calling any other functions. If an attribute get function allocates new memory every time the attribute is read, it will leak memory.

When an `attr_value_t` is returned, the attribute structure used for lists and dictionaries should be freed by the caller using ***SIM_free_attribute()***. This function will only free the attribute structure, not any string or data memory.

Note: Code written in Python does not have to handle attribute data ownership as described in this section.

4.3 Interfaces

Interfaces allows objects to interact with each other. A interface defines a number of functions. Every class that implements an interface (by registering it) must define those functions. An object A may then ask for an interface registered by an object B, and thus use the interface's functions to communicate with B. Simics comes with many predefined interfaces but you can freely add your owns.

One of the most important interfaces for a device class is probably the `io-memory` interface. Implementing the `io-memory` interface enables a class to be mapped into a memory space and be accessed by the simulated machine. In C or DML, it is defined as:

```
typedef int (*map_func_t)(conf_object_t *obj,
                          addr_space_t memory_or_io,
                          map_info_t map_info);
typedef exception_type_t (*operation_func_t)(conf_object_t *obj,
                                             generic_transaction_t *mem_op,
                                             map_info_t map_info);

typedef struct io_memory_interface {
        map_func_t map;
        operation_func_t operation;
} io_memory_interface_t;
```

Another important interface is the `event-poster` interface, which should be implemented by all classes that post events. Other interfaces can be used for things like raising processor interrupts or implementing PCI device functionality. A complete list of all interfaces and functions for handling interfaces can be found in the *Simics Reference Manual*.

To learn more about interfaces in DML, refer to section 5.3. For C and Python, refer to section 6.5.

4.4 Logging

It is often a good idea for a module to be able to log what is happening. The Simics API provides logging facilities based on the following criteria:

level
The verbosity level ranging from 1 through 4 in decreasing importance order.

type
Each log message belongs to a specific category:

`info`
Info or debug message without any consequence on the simulation.

`error`
An error occurred that prevents the simulation (or part of the simulation) from

running properly. Note that error messages do not have any logging level and are always printed out.

`unimplemented`
A model does not implement a specific functionality, bit or register.

`spec_violation`
A model received commands from the target program that violates the device specification.

`target_error`
An error occurred in the target machine (not in the simulator)..

`undefined`
The simulation has been put in a state where the device behavior is undefined.

group
A bit-field, defined and registered by the module, used to separate different part of the device, for example to separate between the configuration of a device and using the device's services, or the PCI interface and the Ethernet interface, etc.

To learn more about logging in DML, refer to section 5.4. For C and Python, refer to section 6.6.

4.5 Events

Sometimes a module needs to execute some code a fixed amount of simulated time into the future. This can be achieved by posting an event, which will cause a callback function to be called when a specified amount of time has passed. There are two event queues in Simics, the time queue which measures simulated time, and the step queue that measures simulated steps. Devices should generally use the time queue, unless they are interested in the exact number of CPU steps. A device that wants to raise a timer interrupt every 10 ms is an example of a module that should use the time queue. A profiling module that wants to sample the program counter every 10th step should on the other hand use the step queue. You can print out the current event queue by using the **print-event-queue** or **piq** command.

The *Simics User Guide* describes event scheduling and timing in more detail. For more information about event handling in DML, refer to section 5.5. For C and Python, refer to section 6.7.

4.6 Haps

A module may also want to react to certain Simics events. This may both be events related to the simulated machine, like processor exceptions or control register writes, and other kinds of events, like Simics stopping the simulation and returning to the command line. In Simics such events are referred to as *haps*. To react to a hap a module can register a callback function to the hap. This callback function will then be called every time the hap occurs. Again, a complete list of all haps, descriptions of what parameters the callback functions

should take and all functions used to register callbacks and manipulate haps can be found in the *Simics Reference Manual*.

DML has no built-in features for hap, so you should refer to the C/C++ and Python explanation in section 6.8.

4.7 Commands

A module that interacts with the user usually defines a number of commands, which can be invoked by the user from the command line. Commands can be bound to a specific namespace (i.e., a class or an interface) or to the global namespace.

- Commands bound to classes are used to manipulate individual instances of that class. For example, most devices have a **status** command that prints the state of the device.
- Commands bound to interfaces can be used on all objects implementing that interface. For example, the **log-level** command is available to all objects implementing the `log-object` interface.
- Global commands can be used for things that do not pertain to a specific object. This is the case of commands like **output-radix** or **print**.

For more information about writing new commands, refer to chapter 7.

Chapter 5

Programming with DML

5.1 Modules, Classes, and Objects

DML hides much of Simics concepts to make the development of device models easier. For example, DML takes care of all module initialization, so nothing needs to be performed when a DML module is loaded or unloaded in Simics.

Each DML file mentioned in the module's `Makefile` defines a Simics class automatically. The class name is provide by the `device` statement at the beginning of the DML file:

```
device my-device;
```

Unless they are `static`, all data declared in the DML file are automatically defined as object-scope data, which means that DML defines automatically the class structure from which objects will be instantiated. For example:

```
data int link_id;
```

defines a *link_id* variable in the object structure.

5.2 Attributes

Registers defined in the DML files are automatically registered as both object structure variables and attributes. The line:

```
register aprom_0 size 1 @ 0x00 "Address PROM (MAC address)";
```

will define a variable in the object structure that contains the value of the register `aprom_0`. It will also define a corresponding attribute so that the state of the register can be saved and restored during checkpointing.

You can also manually add attributes in DML. All that is required is an attribute declaration, including name, type and configuration type. If the type of the attribute is simple, then the declaration itself is sufficient and the default ***set*/*get*** methods will be used.

5.2.1 A Simple Example

The simplest possible attribute holds the value of a simple data type and allows the attribute to be read and written without any side effects. Let us take the example of a counter attribute:

```
attribute counter {
        parameter documentation = "A sample counter attribute";
        parameter type          = "i";
        parameter allocate_type = "int64";
        parameter configuration = "required";
}
```

The two type declarations serve different purposes. The *"i"* tells Simics type system to check the value in set and get operations for an integer type. *allocate_type* defines the internal representation of the attribute value. It is perfectly possible to let those differ to be incompatible, but if they do, you must provide ***get***/***set*** methods that do the correct conversions. See the *DML Reference Manual* for details.

5.2.2 A Pseudo Attribute

When the attribute is more complex, for example if writing an attribute can have side effects, or if it contains by a complex data type, the ***set***/***get*** methods must be provided. They are defined as:

```
method set(attr_value_t val) {
        ...
}
method get() -> (attr_value_t val) {
        ...
}
```

A slightly more complicated example is a pseudo attribute which, when setting values, will add to the value of the counter, and for which getting is an error.

```
attribute add_counter {
        parameter documentation = "A sample pseudo attribute";
        parameter type          = "i";
        parameter configuration = "pseudo";

        method set(val) {
                $counter += val.u.integer;
        }
        method get() -> (val) {
                log "error", 1, 1 : "Get is not allowed";
                throw;
        }
}
```

Note that no type check is required in the ***set*** method, since the type *"i"* is unambiguously checked by Simics. Note also that there is no *allocate_type* parameter for the attribute. Since it is a pseudo attribute that does not store any value, there is no type to allocate.

5.3 Interfaces

5.3.1 Using Interfaces

Using an interface in a module implemented in DML, is done by *connecting* an object to the device model you are developing, specifying which interfaces you are planning to use.

The `connect` statement actually perform two steps at the same time: it defines an attribute that can take an object as a value, and it tells the DML compiler that a number of interfaces belonging to this object can be used in the current device model.

The following code will create an *irq_dev* attribute that accepts only objects implementing the `simics_interrupt` interface as a value.

```
connect irq_dev {
    parameter documentation = "The device that interrupts are sent to.";
    parameter configuration = "required";

    interface simple_interrupt;
}
```

Once an object has been connected, using the interfaces that were specified is simple:

```
    ...
    if ($irq_raised == 0 && irq == 1) {
        log "info", 3: "Raising interrupt.";
        $irq_dev.simple_interrupt.interrupt($irq_dev, $irq_level);
    }
    ...
```

5.3.2 Implementing an Existing Interface

Implementing an existing interface in DML is done with the *implement* statement, which contains the implementation of all the functions listed in the interface. The interface is automatically registered by the DML compiler so that other objects can use it on the current device model:

```
implement ethernet_device {
    // Called when a frame is received from the network.
    method receive_frame(conf_object_t *link, dbuffer_t *frame) {
        inline $receive_packet(frame);
    }
```

```
    // Link speed negotiation functions. Unimplemented, we accept any speed.
    method auto_neg_request(phy_speed_t req_speed)->(phy_speed_t speed) {
        log "unimplemented", 2:
            "Link auto negotiation request (0x%llx)", cast(req_speed, uint64),
        speed = req_speed;
    }
    method auto_neg_reply(phy_speed_t speed) {
        log "unimplemented", 2:
            "Auto negotiation reply (0x%llx)", cast(speed, uint64);
    }
}
```

5.3.3 Implementing an New Interface

Implementing a new interface is pretty much the same as implementing an existing interface, except that you must declare the structure defining the interface first:

```
struct myifc_interface_t {
    int (*my_function)(conf_object_t *o, int i);
}

implement myifc {
    method my_function(int i) -> (int j) {
        j = i + 1;
    }
}
```

There are a few things to remember:

- The `implement` statement assumes that an interface called `foo` is defined in a structure called `foo_interface_t`. In our case we define a structure called `myifc_interface_t` to implement the interface `myifc`. All the predefined Simics interfaces already follow to that rule.
- The `implement` statement is done using DML *methods*, but the interface is declared in a C-like structure with C function pointers. This implies that all functions in the interface must take a `conf_object_t *` pointer as first argument, since that's the standard declaration implied by the keyword `method`. This argument corresponds to the object the interface is applied upon.
- To share an interface definition between DML and C code, the structure definition must be moved in an external `.h` that will be included in both the C code and the `%header` statement of the DML code. Refer to the *DML Reference Manual* for more information.

5.4 Logging

Logging support is built into the language. Log groups are defined with the global declaration `loggroup`. For example:

```
loggroup config;
loggroup request;
loggroup response;
```

Log outputs are made with the *log* statement as follows:
`log` *type*`,` *level*`,` *groups*`,` *string*`,` `[`*value*`]`
where the parameters mean:

type
One of the strings:

- *"error"*
- *"info"*
- *"undefined"*
- *"spec_violation"*
- *"target_error"*
- *"unimplemented"*

level
An integer from 1 through 4, determining the verbosity level at which the message will be logged.

groups
One or several log groups, as defined by the *loggroup* statement, combined with the bitwise or operator "|". If no log groups are defined, the integer 1 makes sense to use.

string, values
A formatting string, as for the C function ***printf()***, optionally followed by a comma separated list of values to be printed.

A small example:

```
loggroup example;

method m(uint32 val) {
        log "info", 4, example : "val=%u", val;
}
```

5.5 Events

The callbacks and posting is handled by declared *event objects*. The declaration typically looks as follows:

```
event future {
        parameter timebase = "seconds";

        method event(void *data) {
                log "info", 1, 1 : "The future is here";
        }
}
```

The *timebase* parameter selects which queue to post on:

"seconds"
use the time queue with seconds as units;

"cycles"
use the time queue with cycles as units;

"steps"
use the step queue.

The method ***event()*** is called when the event is picked from the queue. It takes an optional *data* argument that is set when the event is posted.

To post an event, you can use the ***post()*** method, or the ***post_on_queue()*** method on the event object.

```
method post(when, data)
method post_on_queue(queue, when, data)

# post an event 100s in the future with the optional argument some_data
inline $future.post(100, some_data);

# post an event 10s in the future on cpu2
inline $future.post(SIM_get_object("cpu2), 10, NULL);
```

If you changed your mind and a posted, but not yet handled, event is no longer valid, it can be canceled by calling the ***remove()*** method on the event object.

```
method remove(data)

inline $future.remove(some_data);
```

To find out if there is an event posted but not yet handled, the method ***posted()*** can be called, and to get the time remaining until the event will be handled, the method ***next()*** will return the time as specified by `timebase`.

```
method posted(data) -> (truth)
method next(data) -> (when)

local bool is_this_event_posted;
local double when_is_this_event_posted;
inline $future.posted(some_data) -> (is_this_event_posted)
inline $future.next(some_data) -> (when_is_this_event_posted)
```

DML also provides a convenient shortcut with the `after` statement. An `after` statement is used to call a DML method some time (always in simulated seconds) in the future. It is not possible to cancel pending calls from *after* statements.

```
# call my_method() after 10.5s
after (10.5) call my_method();
```

Refer to the *DML Reference Manual* for more information.

5.6 Haps

DML has no built-in features for hap, so you should refer to the C/C++ and Python explanation in section 6.8.

Chapter 6

Programming with C and Python

6.1 Module Loading and Unloading

Most modules need to do some work when initially loaded into Simics. Typically this work includes registering the classes implemented by the module, and their attributes.

C/C++

A module written in C/C++ must implement the functions ***init_local()*** and ***fini_local()***. They must exist, even if they are empty. The functions are defined as:

```
void
init_local(void)
{
}

void
fini_local(void)
{
}
```

If the module is written in C++, these functions must be declared *extern "C"* for C linkage. Simics does currently not support unloading of modules, so ***fini_local()*** is usually empty.

Python

Before the first statement, a structured comment is required. It must be formatted as follows:

```
# MODULE: module-name
# CLASS: class-name
```

The Python file is executed as a program at load time, so all Python statements in global scope will be executed. Normally there will be some statements that registers classes and attributes with Simics.

6.2 Classes

Each Simics class implemented by a module must be registered with Simics. Remember that classes registered in a module should be listed in the `MODULE_CLASSES` variable in the module's `Makefile`. This allows Simics to automatically load the required modules when reading a configuration file.

Registering a class is done by creating and filling a `class_data_t` structure, and then call the function ***SIM_register_class*** with the new class name and the `class_data_t` structure that describes it. The members in the `class_data_t` structure are:

new_instance
A function called when creating an instance of the class. This function will be described in objects below.

finalize_instance
This function is called once ***new_instance*** has returned, and all attributes in a configuration have been set.

description
A string that should contain a description of the class.

kind
The class kind tells Simics whether objects of this class should be saved when a checkpoint is created. Valid values are:

`Sim_Class_Kind_Vanilla`
class instances will be saved as part of checkpoints (this is the default if *kind* is not given any value.)

`Sim_Class_Kind_Pseudo`
class instances will never be saved.

`Sim_Class_Kind_Session`
is not used for the time being, and thus has the same meaning as `Sim_Class_Kind_Pseudo`.

C/C++

In C/C++, registration of classes is usually done from within the mandatory *init_local()* function. The C definition of `class_data_t` and ***SIM_register_class()*** is the following:

```
typedef struct class_data {
        conf_object_t *(*new_instance)(parse_object_t *parse_obj);
        int            (*delete_instance)(conf_object_t *obj);
```

```
        void            (*finalize_instance)(conf_object_t *obj);
        const char      *description;
        class_kind_t     kind;
} class_data_t;

conf_class_t *
SIM_register_class(const char *name, class_data_t *class_data);
```

SIM_register_class() returns a pointer to a `conf_class_t` structure which is used internally by Simics to keep track of the class information. This pointer can be used when referring to the class in calls to other functions.

A simple ***init_local()*** initialization function could look like this:

```
void
init_local(void)
{
        class_data_t cdata;
        conf_class_t *my_class;

        memset(&cdata, 0, sizeof(cdata));
        cdata.new_instance = my_new_instance;
        cdata.description = "This is my class";
        cdata.kind = Sim_Class_Kind_Session;

        my_class = SIM_register_class("my-class", &cdata);

        // Other initializations...
}
```

Python

In Python, the registration is done when the global statements of the module are executed (at load time):

```
def new_instance(parse_obj):
    ...

def finalize_instance(obj):
    ...

class_data = class_data_t()
class_data.new_instance = new_instance
class_data.finalize_instance = finalize_instance
class_data.description = """
    A sample class used for documentation purposes only.""";
```

```
SIM_register_class("sample-doc-class", class_data)
```

6.3 Objects

C/C++

In C/C++ one must manually maintain a corresponding object structure that is used to hold data about individual objects for each Simics class. Simics's class system supports a very limited form of inheritance, where all classes must inherit the class `conf_object_t`, the basic configuration object class, either directly or indirectly. When declaring an object structure the first element must be the object structure of the class it inherits. An object structure for a class that directly inherits `conf_object_t` is declared as:

```
typedef struct my_object {
        conf_object_t obj;
        // Other variables...
} my_object_t;
```

A Simics object contains (in the `conf_object_t` field) all the information related to its class and its attributes. When an object is created, the ***new_instance()*** function declared in the class definition is called. The ***new_instance()*** function is responsible for allocating an object structure for the new object and initializing the all fields except the `conf_object_t` part. A typical function would look like this:

```
static conf_object_t *
my_new_instance(parse_object_t *parse_obj)
{
        my_object_t *mo = MM_ZALLOC(1, my_object_t);
        // Initializations...
        return &mo->obj;
}
```

When inheriting some other class the object structure should instead include the object structure of that class as its first field. The ***new_instance()*** function should also call some function that initializes the object structure of the inherited class. If the object inherits directly from `conf_object_t`, the `conf_object_t` part of the structure can be initialized by calling the ***SIM_object_constructor()*** function.

A very common class to inherit is `log_object_t`, which automatically provides log facilities to the inheriting classes. The `log_object_t` part of the object structure can be initialized by calling the ***SIM_log_constructor()*** function. The object structure and ***new_instance()*** function would then look like this:

```
typedef struct my_object {
        log_object_t log;
```

```
        // Other variables...
} my_object_t;

static conf_object_t *
my_new_instance(parse_object_t *parse_obj)
{
        my_object_t *mo = MM_ZALLOC(1, my_object_t);
        SIM_log_constructor(&mo->log, parse_obj);
        // Initializations...
        return &mo->log.obj;
}
```

Python

In Python, an object is an instance of a Python class, created by the ***new_instance()*** function referred to when registering the class. The python class is usually written as:

```
class sample_python_class:
    def __init__(self, obj):
        obj.object_data = self
        self.obj = obj

        # other inits

    # misc methods
```

A typical ***new_instance()*** function looks like:

```
def new_instance(parse_obj):
    obj = VT_alloc_log_object(parse_obj)
    sample_python_class(obj)
    return obj
```

All objects must hold a log object, created with a call to ***VT_alloc_log_object()***, and the log object must refer to the instance of the python class.

6.4 Attributes

Attributes are linked to the class definition, usually just after the class has been declared, with the ***SIM_register_typed_attribute()*** function. It has the following declaration in C, and Python:

```
// in C
int SIM_register_typed_attribute(
```

```
            conf_class_t *cls, const char *name,
            get_attr_t get_attr, lang_void *user_data_get,
            set_attr_t set_attr, lang_void *user_data_set,
            attr_attr_t attr,
            const char *type, const char *idx_type,
            const char *doc);

# in Python
SIM_register_typed_attribute(cls,
                             name,
                             get_attr, user_data_get,
                             set_attr, user_data_set,
                             attr,
                             type, idx_type,
                             doc)
```

The parameters of ***SIM_register_typed_attribute()*** are:

cls
The name of the class (previously registered with ***SIM_register_class()***.)

name
The name of the attribute to register.

get_attr, ***set_attr***
The ***get()*** and ***set()*** functions for the attribute. If one of these operations is not supported, `NULL` (or `None` in Python) can be used.

user_data_get, ***user_data_set***
User defined parameters that are passed to the ***get*** and ***set*** functions as the *arg* parameters. This allows passing one extra argument, for example when the same ***get()*** or ***set()*** function is shared between a number of attributes.

attr
The properties of the attribute, a combination of the configuration type, an optional index type and initialization order. It tells Simics how the attribute will be saved and addressed, and is specified using the constants described below.

The *configuration type* of an attribute must be selected from one of the following values:

`Sim_Attr_Required`
The attribute has to be set when creating the object. It will also be saved during checkpointing.

`Sim_Attr_Optional`
If a value isn't specified, the attribute will keep its default value when creating an object. It will be saved during checkpointing.

Sim_Attr_Session
The attribute is only used in the current session and won't be saved during checkpointing.

Sim_Attr_Pseudo
The attribute does not really represent any internal state. It may work instead as a command in disguise, or as a redundant way of accessing internal state. It won't be saved during checkpointing.

Attributes may also have the following additional kinds added (using a bitwise *or* operation).

Sim_Attr_Persistent
Attribute represents a persistent value and is included in persistent files, created with the **save-persistent-state** command. Persistent attributes are used for data that survives power-cycling.

Sim_Attr_Internal
Indicates that the attribute is internal to the object and should not be accessed directly by other users.

The *index type*, if any, is added (using a bitwise *or* operation) with one or several of the following values:

Sim_Attr_Integer_Indexed
Attribute can be indexed with an integer value.

Sim_Attr_String_Indexed
Attribute can be indexed with a string.

Sim_Attr_List_Indexed
Attribute can be indexed with a list.

In addition the order in which the attribute will be initialized can be defined by adding (also using a bitwise *or* operation) with one of the following values:

Sim_Init_Phase_0
Early initialization (default)

Sim_Init_Phase_1
Late initialization

Attributes with `Sim_Init_Phase_1` will be initialized after attributes with `Sim_Init_Phase_0`, but no other order is guaranteed.

type*, *idx_type
Strings describing the data type of the attribute. *type* describes the type when not indexed, and *type_idx* when indexed.

desc
A documentation string describing the attribute.

6.4.1 A Simple Example

Let us use a simple counter attribute as an example.

In C, we'll have an object declared as:

```
typedef struct my_object {
        conf_object_t obj;
        int foo;
} my_object_t;
```

In Python, we'll use a data member of the class defining the object (**sample-python-class**) that we will call *foo*.

We want to implement an attribute called *counter*, thus we need a pair of ***set*/*get*** functions. *counter* will internally use *foo* to keep its value. The pair of ***get*/*set*** functions could be defined as:

```
// In C
static attr_value_t
get_counter(void *arg, conf_object_t *obj, attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        return SIM_make_attr_integer(mo->foo);
}

static set_error_t
set_counter(void *arg, conf_object_t *obj, attr_value_t *val,
            attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        mo->foo = val->u.integer;
        return Sim_Set_Ok;
}
```

In the ***get_counter()*** function, *obj* is the object that owns the attribute and *arg* is the user information that was registered along with the attribute. Note that *obj* can be safely cast to `my_object_t` (`conf_object_t` is used as a "base type" here). The function creates an `attr_value_t` variable that will be of type `Sim_Val_Integer` and contain the value *foo*. It then returns this attribute value.

The ***set_counter()*** function on the other hand takes a *val* argument which contains the value to be written. The return value is of type `set_error_t`, which is defined as below. Descriptions of the values can be found in the *Simics Reference Manual*.

```
typedef enum {
        Sim_Set_Ok,
```

```
        Sim_Set_Need_Integer,
        Sim_Set_Need_Floating,
        Sim_Set_Need_String,
        Sim_Set_Need_List,
        Sim_Set_Need_Dict,
        Sim_Set_Need_Boolean,
        Sim_Set_Need_Data,
        Sim_Set_Need_Object,
        Sim_Set_Object_Not_Found,
        Sim_Set_Interface_Not_Found,
        Sim_Set_Illegal_Value,
        Sim_Set_Illegal_Type,
        Sim_Set_Attribute_Not_Found,
        Sim_Set_Not_Writable,
        Sim_Set_Ignored
} set_error_t;
```

In Python, the value used in the ***get()/set()*** functions is represented in Python's native types:

```
# In Python
def get_counter(arg, obj, idx):
    return obj.object_data.counter

def set_counter(arg, obj, val, idx):
    obj.object_data.counter = val
    return Sim_Set_Ok
```

Note, however, that the object is referred to by the *log object*, and not the object created from the ***new_instance()*** function that creates object instances. The data of the object is thus referred to via the *object_data* member of the *obj* parameter.

Registering the *counter* attribute is just a matter of calling ***SIM_register_typed_attribute()***:

```
// in C
SIM_register_typed_attribute(my_class,
                             "counter",
                             get_counter, NULL,
                             set_counter, NULL,
                             Sim_Attr_Required,
                             "i", NULL,
                             "A counter");

# and in Python
SIM_register_typed_attribute(my_class,
                             "counter",
```

```
                                get_counter, None,
                                set_counter, None,
                                Sim_Attr_Required,
                                "i", None,
                                "A counter");
```

6.4.2 A Pseudo Attribute

In the previous example, the attribute *counter* was a direct representation of the value *foo* inside the object. Now let us add an attribute called *add_counter* that will increase *foo* by a given value when the attribute is set, and do nothing when the attribute is read. This would give us the following code:

```
// In C
static set_error_t
set_add_counter(void *arg, conf_object_t *obj, attr_value_t *val,
                attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        mo->foo += val->u.integer;
        return Sim_Set_Ok;
}

# and in Python
def add_counter(arg, obj, val, idx):
    obj.object_data.counter += val
    return Sim_Set_Ok
```

There is no need for a get function since this attribute only can be written. The semantics of ***set_add_counter()*** are also slightly different, since the function actually *adds* a value to *foo*.

It is thus possible to create *real* attributes whose value corresponds to a real variable in an object, and *pseudo* attributes which are only used as object "methods".

Registering the *add_counter* attribute is straightforward:

```
// In C
SIM_register_typed_attribute(class_name, "add_counter",
                             NULL, NULL,
                             set_add_counter, NULL,
                             Sim_Attr_Pseudo,
                             "i", NULL,
                             "A sample pseudo attribute.")

# and in Python
```

```
SIM_register_typed_attribute(class_name, "add_counter",
                             None, None,
                             add_counter, None,
                             Sim_Attr_Pseudo,
                             "i", None,
                             "A sample pseudo attribute.")
```

6.4.3 An indexed attribute

If an attribute advertises the possibility of being indexed, Simics will allow access using an index of a type specified when registering the attribute: an integer, a string and/or a list. In Python, a *slice* is also accepted and will be converted to a list as described later in this section. Using indexed attributes in Python would correspond to the following code:

```
object.attr[i] = a
b = object.attr['a string']
c = object.attr[i:j]
d = object.attr[[i,j]]
```

When an indexed access is performed, the *idx* parameter of the ***get()***/***set()*** functions contains the index value. In C this value will be coded as an `attr_value_t` representing an integer, a string or a list. In Python, the index will be the corresponding native python type.

Let us add an array of counters to our previous example:

```
// In C
typedef struct my_object {
        conf_object_t obj;
        int foo[10];
} my_object_t;
```

The ***get()*** and ***set()*** function will be:

```
// In C
static attr_value_t
get_counter_array(void *arg, conf_object_t *obj, attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        if (idx->kind != Sim_Val_Nil) {
                if (idx->u.integer < 0 || idx->u.integer >= 10)
                        return SIM_make_attr_invalid();

                return SIM_make_attr_integer(mo->foo[idx->u.integer]);
        } else {
```

```
                attr_value_t ret = SIM_alloc_attr_list(10);
                int i;
                for (i = 0; i < 10; i++) {
                        ret.u.list.vector[i] =
                                SIM_make_attr_integer(mo->foo[i]);
                }
                return ret;
        }
}

static set_error_t
set_counter_array(void *arg, conf_object_t *obj, attr_value_t *val,
                  attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        if (idx->kind != Sim_Val_Nil) {
                if (idx->u.integer < 0 || idx->u.integer >= 10)
                        return Sim_Set_Illegal_Value;

                mo->foo[idx->u.integer] = val->u.integer;
        } else {
                int i;
                for (i = 0; i < 10; i++)
                        mo->foo[i] = ret.u.list.vector[i].u.integer;
        }

        return Sim_Set_Ok;
}
```

 # And in Python:

```
def get_counter_array(arg, obj, idx):
    if idx != None:
        if idx < 0 or idx >= 10:
            return None
        return obj.object_data.vcounter[idx]
    else:
        return obj.object_data.vcounter

def set_counter_array(arg, obj, val, idx):
    if idx != None:
        if idx < 0 or idx >= 10:
            return Sim_Set_Illegal_Value
        obj.object_data.vcounter[idx] = val
    else:
```

```
        obj.object_data.vcounter = val
    return Sim_Set_Ok

SIM_register_typed_attribute(class_name, "counter_array",
                             get_counter_array, None,
                             set_counter_array, None,
                             Sim_Attr_Optional | Sim_Attr_Integer_Indexed,
                             "[i{10}]", "i",
                             "Sample array of counters")
```

These functions handle both indexed and non-indexed access to the attribute. This is important for checkpointing, as all attributes are saved and restored using *non-indexed access*. Thus, if you wish your attribute to be checkpointed, you will always have to handle non-indexed accesses.

As shown in the example usages above, it is also possible to use strings, lists and Python slices as index. An index corresponding to a slice is provided as a list of two elements, the first and last element that should be included. Note that this means a slight change, since the Python slice refers to the first element included, and the one after the last included. This means that: `object.attr[i:j]` will be translated to an index parameter with the value `[i, j - 1]`, or in C, an attribute of type `Sim_Val_List` with two `Sim_Val_Integer` elements i and $j - 1$.

Note: When accessing attributes from Python, you may not be able to access an indexed attribute without specifying an index even if your ***get*** function supports a non-indexed access. The reason is that the short access syntax is ambiguous: it can not differentiate between an indexed access to a Simics attribute and a non-indexed access later indexed by Python itself. You will have to explicitly use ***SIM_get_attribute()*** to access the attribute's non-indexed value.
As an example, let's consider two attributes, *foo* and *bar*. *foo* is an array of 5 elements that do not support indexing, while *bar* is a table of 10 elements supporting both indexed and non-indexed accesses:

```
# non-indexed access to 'foo' attribute
simics> @conf.object.foo
[0,1,2,3,4]

# non-indexed access to 'foo' attribute, later indexed by python
simics> @conf.object.foo[2]
2

# indexed access to 'bar' attribute
simics> @conf.object.bar[12]
12

# no access performed to 'bar' attribute
simics> @conf.object.bar
<object bar>

# non-indexed access to 'bar' attribute
simics> @SIM_get_attribute(conf.object, "bar")
[0,1,2,3,4,5,6,7,8,9]
```

6.5 Interfaces

6.5.1 Using Interfaces

An object that wants to interacts with another through an interface uses the ***SIM_get_interface()*** functions to retrieve the interface structure. It can then call the other object using the functions defined in the structure.

```
# in Python
ifc = SIM_get_interface(conf.phys_mem, "memory_space")
val = ifc.read(conf.phys_mem, conf.cpu0, 0x1234, 4, 0)
```

```
// In C
conf_object_t *obj = SIM_get_object("phys_mem");
memory_space_interface_t *ifc;
attr_value_t val;

ifc = (memory_space_interface_t *) SIM_get_interface(obj, "memory_space");
val = ifc->read(obj, SIM_get_object("cpu0"), 0x1234, 4, 0);
```

In Python, you can use the *iface* attribute of the object to access the interface directly:

```
# In Python
val = conf.phys_mem.iface.memory_space.read(conf.phys_mem, conf.cpu0, 0x1234,
```

When you are using interfaces inside an object, you will often need to define which object you want to talk to via an attribute. The classic way of doing that is to define an attribute with type `o|n`, and check if the object passed as argument implements the necessary interface:

```
// In C
static attr_value_t
get_an_object(void *arg, conf_object_t *obj, attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;
        return SIM_make_attr_object_or_nil(obj->an_object);
}

static set_error_t
set_an_object(void *arg, conf_object_t *obj, attr_value_t *val,
              attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        if (val->u.kind == Sim_Val_Nil) {
                mo->an_object = NULL;
        } else {
                foo_interface_t *foo_ifc =
                    (foo_interface_t *)SIM_get_interface(val->u.object, "foo")
                if (SIM_clear_exception() != SimExc_No_Exception)
                        return Sim_Set_Interface_Not_Found;
                mo->an_object = val->u.object;
        }

        return Sim_Set_Ok;
}
```

```
[...]

SIM_register_typed_attribute(class_name, "an_object",
                             get_an_object, NUL,
                             set_an_object, NULL,
                             Sim_Attr_Optional,
                             "o|n", NULL,
                             "An object implementing the 'foo' interface");
```

6.5.2 Implementing an Existing Interface

Implementing an existing interface consists in filling in the functions pointers that are listed in the interface definition with the functions that should be called. The interface should then be registered using the ***SIM_register_interface()*** function:

```
SIM_register_interface(conf_class_t *class, char *name, void *iface)
```

Where the parameters are:

class
: The class that will advertise that it implements the interface

name
: The name of the interface

iface
: The filled interface structure.

This gives us the following code:

```
# In Python
def my_operate(memhier, space, map, memop):
    print "my_operate() has been called!"

ifc = timing_interface_t()
ifc.operate = my_operate
SIM_register_interface(my_class, "timing-model", ifc)
```

```
// And in C
static cycles_t
my_operate(conf_object_t *mem_hier, conf_object_t *space,
           map_list_t *map, generic_transaction_t *mem_op)
{
        // do something
}
```

```
static conf_class_t *my_class;
static timing_model_interface_t ifc;

void init_loca(void)
{
        ...
        ifc.operate = my_operate;
        SIM_register_interface(my_class, "timing-model", (void *) &ifc);
        ...
}
```

6.5.3 Implementing an New Interface

It is unfortunately not possible to implement new interfaces in Python since it would require additional type wrapping to exchange data between Python and the C layer that implements interfaces.

Implementing a new interface in C is pretty much the same as implementing an existing interface, except that you must declare the structure defining the interface first:

```
typedef struct {
        int (*my_function)(conf_object_t *o, int i);
} myifc_interface_t;

int my_function(conf_object_t *o, int i)
{
       return i + 1;
}

...
```

Note that:

- It is very common to have as a first argument of an interface function the object on which the function is applied.
- The structure defining the interface `foo` should be called `foo_interface_t`.

If you plan to use the interface you are declaring in DML code, these two rules are mandatory. Refer to the section 5.3 to get more explanation about interfaces in DML.

6.6 Logging

Logging in C or Python is handled by the ***SIM_log_register_group()*** and ***SIM_log_message()*** functions.

A single call to ***SIM_log_register_groups()*** registers all groups for the class. The function is used as:

```
# In Python
SIM_log_register_groups(class, gnames)

// In C
SIM_log_register_groups(conf_class_t *cls, const char **gnames)
```

where the parameters are:

classname
The name of the Simics class in Python, and the class structure in C.

groupnames
A tuple of strings in Python, and a NULL-terminated array of strings in C.

An example:

```
# In Python
SIM_log_register_groups("sample-device", ("config", "request", "response"))

// In C
static char *groupnames[] = { "config", "request", "response", NULL };
SIM_log_register_groups(my_class, &groupnames);
```

The log group values will be defined by the order of the strings in the tuple as a power of 2 series, so in the example above “config” corresponds to 1, “request” corresponds to 2 and “response” corresponds to 4.

Log outputs is handled with the ***SIM_log_message()*** function. It takes the following parameters:

```
# In Python
SIM_log_message(obj, level, groups, type, msg)

// In C
void
SIM_log_message(conf_object_t *obj,
                int level,
                int groups,
                log_type_t type,
                const char *msg);
```

with the parameters meaning:

obj
The log object instance.

level
An integer from 1 through 4, determining the lowest verbosity level at which the message will be logged.

groups
The bitwise or:ed values of one or several log-groups. If no log groups have been registered, 1 is a good value to use.

type
One of:

- `Sim_Log_Error`
- `Sim_Log_Info`
- `Sim_Log_Spec_Violation`
- `Sim_Log_Target_Error`
- `Sim_Log_Undefined`
- `Sim_Log_Unimplemented`

msg
The string to log.

Python

A ***SIM_log_message()*** example:

```
def get_counter_array(arg, obj, idx):
    SIM_log_message(obj, 4, 1, Sim_Log_Info, "get_counter_array")
    if idx != None:
        if isinstance(idx, (int, long)):
            SIM_log_message(obj, 1, 1, Sim_Log_Error,
                            "index must be integer")
            return None
        return obj.object_data.vcounter[idx]
    else:
        return obj.object_data.vcounter
```

C/C++

Logging from a Simics module written in C/C++ is made easier and clearer by the macros ***SIM_log_info()***, ***SIM_log_error()***, ***SIM_log_undefined()***, ***SIM_log_spec_violation()***, ***SIM_log_target_error()*** and ***SIM_log_unimplemented()***. These macros use internally the function ***SIM_log_message()***, and should always be used instead.

The usage is identical for all, except that ***SIM_log_error()*** does not have a level parameter. The prototypes are:

```
static void SIM_log_info(int lvl, log_object_t *dev, int grp,
                         const char *str, ...);

static void
SIM_log_undefined(int lvl, log_object_t *dev, int grp,
                  const char *str, ...);

static void
SIM_log_spec_violation(int lvl, log_object_t *dev, int grp,
                       const char *str, ...);

static void
SIM_log_target_error(int lvl, log_object_t *dev, int grp,
                     const char *str, ...);

static void
SIM_log_unimplemented(int lvl, log_object_t *dev, int grp,
                      const char *str, ...);

static void
SIM_log_error(log_object_t *dev, int grp,
              const char *str, ...);
```

The parameters are identical to the ***SIM_log_message()*** function described above. Note that the macros take a variable number of arguments to allow you to write ***printf()***-like strings.

A small example:

```
static attr_value_t
get_counter_array(void *arg, conf_object_t *obj, attr_value_t *idx)
{
        my_object_t *mo = (my_object_t *)obj;

        SIM_log_info(4, obj, 1, "get_counter_array");
        if (idx->kind != Sim_Val_Nil) {
                if (idx->kind != Sim_Val_Integer)
                        SIM_log_error(obj, 1,
                                      "Index must be integer");
                        return SIM_make_attr_invalid();

                return SIM_make_attr_integer(mo->foo[idx->u.integer]);
        }
        else {
                attr_value_t ret = SIM_alloc_attr_list(10);
```

```
                int i;
                for (i = 0; i < 10; i++) {
                        ret.u.list.vector[i] =
                                SIM_make_attr_integer(mo->foo[i]);
                }
                return ret;
        }
}
```

6.7 Events

For events, the API is identical for Python and C/C++. To post an event in the future, based on time, the ***SIM_time_post()*** and ***SIM_time_post_cycles()*** function is used. It has the following declaration:

```
void
SIM_time_post(conf_object_t *obj,
              double seconds,
              sync_t sync,
              event_handler_t func,
              lang_void *user_data);

void
SIM_time_post_cycle(conf_object_t *obj,
                    cycles_t cycles,
                    sync_t sync,
                    event_handler_t func,
                    lang_void *user_data);
```

They take five arguments specifying the object whose time queue should be used (*obj*), the amount of time until the event occurs—in seconds or in cycles—(*seconds*), whether just the processor or the whole machine should be synchronized when the event occurs (*sync*), the callback function that should be called (*func*) and the parameter for the callback function (*user_data*).

If for some reason you do want to remove a posted but not yet handled event, you can cancel it with a call to ***SIM_time_clean()***, specifying the object, callback function and parameter.

You can also check how long time is left until an event occurs using ***SIM_time_next_occurrence()***, again specifying the object, callback function and parameter. The time left to the event is returned in cycles.

If you want to post an event a number of simulated steps into the future it should instead post to the step queue. Posting to the step queue is very similar to posting to the time queue, but the functions ***SIM_step_post()***, ***SIM_step_clean()*** and ***SIM_step_next_occurrence()*** should be used instead.

Refer to the *Simics Reference Manual* for more information on the functions available and their parameters.

6.8 Haps

6.8.1 Reacting to Haps

When you want your module to record or react to a hap, you can register a callback function with the specific hap. The signature for the callback function differs between the different haps.

Registering a callback is made with any of the functions ***SIM_hap_add_callback()***, ***SIM_hap_add_callback_index()***, ***SIM_hap_add_callback_range()***, ***SIM_hap_add_callback_obj()***, ***SIM_hap_add_callback_obj_index()*** or ***SIM_hap_add_callback_obj_range()***. See the *Reference Manual* for information about the differences.

```
# In Python
SIM_hap_add_callback(hap, func, user_data)

// In C
hap_handle_t
SIM_hap_add_callback_obj(const char *hap,
                         obj_hap_func_t func,
                         typed_lang_void *user_data);
```

The parameters of ***SIM_hap_add_callback_obj()*** are:

hap
: The name of the hap (string);

func
: The callback function;

user_data
: Any user data you require for your callback function. The value is passed as-is by simics.

The function returns a handle which can be used to manipulate the hap callback later on.

A list of all haps can be found in the *Simics Reference Manual*, including the signatures for the callback functions.

A small example:

```
# In Python
def stopped(user_data, obj, exception, errstr):
    print "Stopped"
```

```
def started(user_data, obj):
    print "Started"


handle1 = SIM_hap_add_callback("Core_Continuation",
                               started,
                               None);

handle2 = SIM_hap_add_callback("Core_Simulation_Stopped",
                               stopped,
                               (2, "str"))
...
```

```
// In C
static void
stopped(void *user_data, conf_object_t *obj,
        integer_t exception, char *errstr)
{
        // do something
}


static void
started(void *user_data, conf_object_t *obj)
{
        // do something
}

static hap_handle_t h1, h2;

void init_local()
{
        ...
        h1 = SIM_hap_add_callback("Core_Continuation",
                                  started,
                                  NULL);

        h2 = SIM_hap_add_callback("Core_Simulation_Stopped",
                                  stopped,
                                  NULL);
        ...
}
```

Should you no longer want to subscribe to a hap, the notification callbacks can be canceled using one of the functions ***SIM_hap_delete_callback()***, ***SIM_hap_delete_callback_obj()***, ***SIM_hap_delete_callback_id()*** and ***SIM_hap_delete_callback_obj_id()***. See the *Reference Manual* for the differences. The signature for ***SIM_hap_delete_callback()*** is:

```
SIM_hap_delete_callback(hap, func, user_data)
```

The parameters are:

hap
The name of the hap

func
The callback function registered

user_data
User provided data

A callback must have been registered with the exact same parameter values. To continue the above example:

```
# In Python
def at_end():
    # using the callback parameters
    SIM_hap_delete_callback("Core_Simulation_Stopped", stopped, NULL);
    # using the hap handle
    SIM_hap_delete_callback_id(handle2)
```

```
// In C
fini_local()
{
        // Using the callback parameters
        SIM_hap_delete_callback("Core_Simulation_Stopped", stopped, NULL);
        // Using the hap handle
        SIM_hap_delete_callback_id(h2);
}
```

6.8.2 Providing Haps

It might be that some things happening in your module are worth notifying others of, for example for automation with scripts. To do this, you can provide your own haps to which interested parties can subscribe.

Adding a new type

Before subscribers can be notified of a new hap, the hap must be known. A new hap type is made known through registration. Registering a new hap type is done with the function ***SIM_hap_add_type()***. The signature is:

```
# In Python
SIM_hap_add_type(hap, params, param_desc, index, desc, old_hap_obj)

// In C
hap_type_t
SIM_hap_add_type(const char *hap,
                 const char *params,
                 const char *param_desc,
                 const char *index,
                 const char *desc,
                 int old_hap_obj);
```

where the parameters are:

hap
: The name of the hap, which must be unique

params
: A string specifying the number of parameters of the hap and their types. The return value and the first two parameters of the callback function are always the same and are not included in the list. A valid parameter description string contains only the following type description characters:

  - `i` – int
  - `I` – integer_t (64 bit integer)
  - `e` – exception_type_t
  - `o` – configuration object (i.e. void* in C and Python object in Python)
  - `s` – string
  - `m` – memory transaction (`generic_transaction_t *` in C)
  - `c` – configuration object (`conf_object_t *` in C)
  - `v` – `void *`

param_dessc
: space separated list of descriptive parameter names (in the same order as *params*, so that the first word is the name of the first parameter. If *param* is the empty string, *param_desc* may be None.

index
: A string describing the index value for the hap, or None if there is no index value. The meaning of indexes is up to you to define.

desc
A human readable description of the hap.

old_hap_obj
Always 0.

The return value is a handle that must be saved for operations on the hap. Example:

```
# In Python
hap_handle = SIM_hap_add_type("My_Special_Hap",
                              "ii",
                              "val1 val2",
                              None,
                              "Called when something special "
                              " happens in my module.",
                              0);
if hap_handle <= 0:
    # error handling
    ...

static hap_type_t hap_handle;
```

```
// In C
void
init_local()
{
        ...
        hap_handle = SIM_hap_add_type("My_Special_Hap",
                                      "ii",
                                      "val1 val2",
                                      NULL,
                                      "Called when something special "
                                      " happens in my module.",
                                      0);
        if (hap_handle <= 0) {
                /* error handling */
                ...
        }
}
```

Notifying subscribers

Whenever the special condition for the hap is met, the subscribers to the hap should be notified. Since the call itself incurs some overhead, it is good practice to use ***SIM_hap_is_active()*** to check if there are any subscribers before calling the notification function.

```
# In Python
SIM_hap_is_active(hap)
```

```
// In C
int
SIM_hap_is_active(hap_type_t hap)
```

where the parameter *hap* is the value returned from ***SIM_hap_add_type()***. ***SIM_hap_is_active()*** returns 1 if there are callbacks registered for the hap type, and 0 otherwise.

The notification of subscribers is normally done with one of ***SIM_hap_occurred()***, ***SIM_hap_occurred_vararg()***, ***SIM_hap_occurred_always()***, and ***SIM_hap_occurred_always_vararg()***. See the *Reference Manual* for information about the differences.

```
# In Python
SIM_hap_occurred(hap, obj, value, list)
```

```
// In C
int
SIM_c_hap_occurred(hap_type_t     hap,
                   conf_object_t *obj,
                   integer_t      value,
                   ...);
```

The parameters are:

hap
: The handle to the hap type, as returned from ***SIM_hap_add_type()***.

obj
: The object for which the condition is met.

value
: Only meaningful if the hap is indexed. The meaning is defined by you.

list
: In Python, the parameters to the hap callback as a Python list.

In C, hap parameters will be provided as additional parameters to the function. A short example:

```
# In Python
```

```
def some_func(obj, v1, v2):
    if some_condition:
        if SIM_hap_is_active(hap_handle):
            SIM_hap_occurred(hap_handle, obj, 0, [v1, v2])
```

```
// In C
static void
some_func(conf_object_t *obj, int v1, int v2)
{
        if (some_condition) {
                if (SIM_hap_is_active(hap_handle))
                        SIM_c_hap_occurred(hap_handle,
                                           obj, 0,
                                           v1, v2)
        }
}
```

Chapter 7

Adding New Commands

7.1 Introduction

Commands in Simics are written in Python, and they access the internal state of Simics by using the Simics API, and by accessing configuration objects and their attributes. Since modules cannot add functions to the Simics API, all commands for devices and extensions use the configuration system.

There are several benefits of using Python to define commands: The code is forced to make its internal state visible, allowing other script users to access it. The module can be distributed in binary form, while the Python commands are still available with source, allowing a user to change them in a way he or she prefers.

The command can be declared in two files that should reside in the same directory as the source code of the module:

`commands.py`
: Contains commands that will be declared when the module is loaded. These are usually commands related to a specific class or interface. For example, ⟨**my_class**⟩**.info** should be declared in this file.

`gcommands.py`
: Contains global commands that will be declared when Simics is started, whether or not the module has been loaded. Commands placed here are often used to create new instances of a class. For example, **new-tracer** is declared in `gcommands.py` in the `trace/` module source directory.

When the module is compiled, these two files (if they exist) are copied to the `simics/`*host*`/lib/python` directory; e.g., `simics/x86-linux/lib/python`. The resulting file will get the name `mod_`*module-name*`_commands.py` (respectively `_gcommands.py`) with any dash (`-`) characters replaced with underscore (_).

7.2 Example of a new command

This is an example on how to add a command in Python, as it would appear in the `commands.py` file of a module's source code subdirectory:

```
# this line imports definitions needed below
from cli import *

# this is my Python function that will be called when the
# command is invoked from the Simics prompt.
def my_command_fun(int_arg, str_arg, flag_arg):
    print "My integer: %d" % int_arg
    print "My string: %s" % str_arg
    print "Flag is",
    if flag_arg:
        print "given"
    else:
        print "not given"
    return int_arg

# register our new command
new_command("my-command", my_command_fun,
            args  = [arg(int_t, "arg", "?", 10), arg(str_t, "name"),
                     arg(flag_t, "-f")],
            alias = "mc",
            type  = "my-module-commands",
            short = "my command does it",
            doc_items = [("NOTE", "This command is best")],
            see_also = ["my_other_command"],
            doc = """
<b>my-command</b> is best.
This is its documentation. <i>arg</i>
is the first argument...""")
```

The code above will define a command called **my-command**. When the command is invoked from the Simics command line interface, the function ***my_command_fun()*** will be called. The function must take exactly as many arguments as the command; in this case 3: an integer, a string, and a flag. The first command argument is called "*arg*" and is optional, indicated by `"?"`. If omitted by the user, the command function will be called with the default value, 10. The second argument is called "name" and is required (since there is no `"?"`, there is no default value). The last argument is a flag and will have a value of 1 if the flag is given, otherwise 0. Flags are thus always optional.

If the command function returns a value (a string or an integer) Simics will print this value on the terminal or pass it on to other commands as input arguments, e.g., **print -x (my-command 15 foo)**.

Writing **help my-command** at the Simics prompt will display:

```
NAME
   my-command - my command does it
```

```
SYNOPSIS
   my-command [arg] name [-f]
ALIAS
   mc
DESCRIPTION
   my-command is best. This is its documentation. arg is the first argument.
NOTE
   This command is best
SEE ALSO
   my_other_command
```

The command can be invoked in different ways, here are some examples:

```
simics> my-command 2 foo -f
My integer: 2
My string: foo
Flag is given
2
simics> my-command bar
My integer: 10
My string: bar
Flag is not given
10
simics> my-command 20 -f
Argument error: argument number 2 is missing in 'my-command', string expected.
SYNOPSIS: my-command [arg] name [-f]
simics> print -x (mc -f name="hello there!" arg = 23)
My integer: 23
My string: hello there!
Flag is given
0x17
```

In the last case the alias is used and the command is passed to the print command that outputs the value in hexadecimal notation.

All the parameters of the ***new_command()*** function will be explained bellow. After the parameter name follows the type of the parameter and if the argument is required, recommended, or optional:

name — string (required)
First argument (no need to write name =) and the name of the command. May include digits and underscores as well as dashes. Must begin with a letter.

fun — function (required)
The command handler function that will be called when the command is executed. The number of arguments must match the **`args`**-list(see below). Since new_command

is executed when the `commands.py` file is loaded into Python the function must be defined before the new_command call, as in the example.

`args` — list of argument specifiers (required)

This is a list of the arguments given to the command, and must match the arguments of the function described above. An argument specifier is created by calling the function ***arg()***. This function takes as arguments a type (this is in fact also a function), that is called to handle the argument. Examples of available types are `int_t`, `str_t`, `addr_t`, `filename_t()`, and `flag_t`. See the next section for a discussion of these.

To create an argument list of an integer and a string, use:

```
..., args = [arg(int_t), arg(str_t)], ...
```

It is, however, recommended that names for the parameters are specified. This is done as a second argument for ***arg()***:

```
..., args = [arg(int_t, "value"), arg(str_t, "file")], ...
```

This way the documentation of the argument list (**help** command) will use these names and also makes it possible to enter the argument in any order at the command line, e.g.,

```
command file = "/tmp/bar" value = 17
```

The **flag** type requires the name to be specified and the name must begin with a hyphen, e.g., "–all". The corresponding value passed to the command handler function will be 1 if the flag is given on the command line or 0 otherwise.

The **addr** type can be used for addresses. It understands argument of the form `p:0xcf00` (physical address), `v:0xff00` (virtual address), or `0xffdc` at the command line. The command handler function will receive a tuple of the prefix and the address, e.g., `("v", 0xcff0)`. If only the address is given, `"v"` will be used.

Sometimes it is convenient to have other arguments than flags optional. To indicate this, add `"?"` as the third argument to **arg()**, and the default value as the fourth; e.g.,

```
..., args = [arg(int_t, "value", "?", 1),
             arg(str_t, "file")], ...
```

makes `value` an optional argument with 1 as its default value.

The ***arg*** function takes more parameters than those already mentioned:

```
..., args = [arg(str_t, "cpu",
```

```
                expander = exp_fun,
                is_a = test_fun)], ...
```

will connect an argument completion (expander) function for the *cpu* arg, i.e., the function ***exp_fun()*** will be called when the user presses the TAB key while entering the argument value. The expander function takes an argument representing the text the user has written for the argument. For example, if the user presses TAB after typing **command cpu = ultr**, the ***exp_fun()*** will be passed `"ultr"` and should return a list of strings completing `"ultr"`. Here is an example of an expander function:

```
def exp_fun(comp):
    return get_completions(comp, ["ultraI", "ultraII",
                                  "ultraIII", "pentium"])
```

When called with `"ultr"`, it will return `["ultraI", "ultraII", "ultraIII"]`. The get_completions function filters the list and keep elements with prefix comp. The expander functions only works for the string type for the moment.

The *is_a* argument takes a function that tests if the argument is valid. It should return 1 for valid values and 0 otherwise. For example the read-reg command has as first argument a CPU (optional) and as second argument a register. If a user types **read-reg g7**, CLI will incorrectly interpret g7 as a CPU since g7 is a valid string and the command will fail. The ***is_a*** function could therefore return 0 for non-CPUs and pass the string to the next argument, which will match since `g7` is a register.

Polyvalue

A command argument can be of multiple types as well (*polyvalues*). For example,

```
...,
args = [ arg((str_t, int_t, flag_t), ("cpu","value","-all"), "?",
             (int_t, 0, "value"), expander = (exp1, exp2, None)) ],
...
```

will create an argument that is either a string, an integer, or a flag. The argument passed to the command handler function is a tuple specifying the arg-type, the value, and the name of the argument. E.g., **command foo** will pass `(str_t, "foo", "cpu")` to the command handler function. This is why the default value looks the way it does. The corresponding expander function will also be used. **command cpu = abc<tab>** will use the `exp1` expander.

doc — string (required if not doc_with is used)

This is the documentation of the command. Some simple, HTML-like formatting markup can be used, such as `<i>`, `<b>` and `<br/>` for italic, bold and line-break. A blank line separates paragraphs. Italic does not usually work in terminals so underlining will be used instead. Use italics when referring to arguments and bold for command names.

type — string (default is "misc-commands")
This is the command category that the command belongs to. All categories will be listed when the **help** command is used. **help** *category* will list all command in that category. Any whitespace within this string will be replaced with dashes.

short — string (recommended)
A short description of the command used when listing commands (e.g. **help** –all).

repeat — function (optional)
If such a function is supplied, it will be called when the user enters an empty line (i.e., just presses enter) after this command has been run. The arguments passed to the **repeat** function will be the same as those of **fun** (see above). This is used to implement the behavior of commands like **disassemble**, where pressing enter after having run one **disassemble** command disassembles the instructions following the last one of the previous command.

namespace — string (optional)
Makes the command a *namespace* command. Such commands are invoked as a method call of the form *object*`.`*command*, e.g., `rec0.playback-start`. This means that the command handler function will get the namespace object as first argument (a recorder object) and then the rest of the arguments. The namespace string is either the class name of the object, e.g., `"recorder"`, or the name of a "superclass" of the object, or an interface that the object implements. For instance, the class name "processor" is a superclass of any processor, e.g., "ultrasparc-ii", and "breakpoint" is an interface that a memory-space object implements.

The full name of a namespace command will be <class_or_interface>.command (angle brackets included) and this is what is printed when command are listed, but as stated above the command is invoked with **object.command**.

Device commands are typically implemented using namespace commands. This is because they usually operate on a single configuration object, e.g., a MMU or cache object.

doc_with — string (optional)
This argument can be specified if a command should be documented together with another one. For example the **disable** command is documented with the **enable** command since they are tightly coupled together. So the doc argument is missing for the disable command and all documentation is written in the enable command. Note: **doc_with** together with a namespace command must be given as "<class_or_interface>.command"

alias — string or list of strings (optional)
Specify aliases for this command. Does not work with namespace commands.

infix — 0,1 (default 0)
Indicates that the command should be an infix command. For example the arithmetic command sets this argument to 1.

pri — integer (default is 0)
Sets the priority for the commands, typically only relevant to infix commands. For example, * has higher priority than + (200 and 100, respectively).

left — 0,1 (default 0)
If the command is left associative or not (right associative).

7.3 Argument Types

Following is a list of the available argument types. Note that some of them are "real" arguments, while some are generator functions which, when called with appropriate parameters, return an actual argument type.

addr_t

Accepts a target machine address, optionally with an address space prefix, such as `v:` for virtual addresses or `p:` for physical.

bool_t(true_str, false_str)

Accepts only the two strings *true_str* and *false_str*, the strings "TRUE" and "FALSE", and boolean values (that is, the strings "0" and "1"). It passes True or False to the command function depending on which string (or value) was given.

Both arguments are optional; if not given, TRUE, FALSE, 1, and 0 will still be recognized.

filename_t(dirs = 0, exist = 0, simpath = 0)

Generator function for filename arguments. If the *dirs* argument is zero (which is default), no directories will be accepted. The *exist* flag, when set, forces the file to actually exist. If *simpath* is true, files will be checked for existence using `SIM_lookup_file()`, searching the Simics search path. *simpath* implies *exist*. On Windows, if Cygwin path conversion is performed (see `SIM_native_path()` for details), the filename will be converted to host native format.

float_t

Accepts floating-point numbers.

int32_t

Accepts any integer that fits in 32 bits (signed or unsigned). The value passed to the command function is cast into an unsigned value.

int64_t

Accepts any integer that fits in 64 bits (signed or unsigned). The value passed to the command function is the value cast to unsigned.

int_t

Accepts any integer (regardless of size).

integer_t

Accepts any integer that fits in 64 bits (signed or unsigned). Corresponds to the Simics API's integer_t.

obj_t(desc, kind = None)

Returns an argument which accepts any object.

desc is the string returned as a description of the argument. *kind* can be used to limit the accepted objects to only allow objects of a certain kind. This parameter can either be a class name that the object should be an instance of, or the name of an interface that the object must implement.

range_t(min, max, desc, positive = 0)

Returns an argument which accepts any integers x, such that $min \leqslant x \leqslant max$. *desc* is the string returned as a description of the argument.

If *positive* is true, any negative values will be "cast" into positive ones using the formula $max + v + 1$, where v is the negative value.

sint32_t

Accepts any signed integer that fits in 32 bits.

sint64_t

Accepts any signed integer that fits in 64 bits.

str_t

Accepts any one word or quoted string.

string_set_t(strings)

Accepts only strings from the given set. *strings* can be any iterable, such as a tuple, list, or set, in which case the return value is the exact string the user gave; or a dictionary mapping acceptable user input strings to return values.

The optional parameter *visible* is a list of strings. If given, only strings in this list will be suggested by the expander.

uint32_t

Accepts any unsigned integer that fits in 32 bits.

uint64_t

Accepts any unsigned integer that fits in 64 bits.

7.4 Info And Status Commands

To simplify the creation of **info** and **status** commands, there are a couple of helper functions that make it easy to add these commands and have the output formatted in a standard fashion. Instead of calling ***new_command*** directly, you call ***new_info_command*** and ***new_status_command***. The functions you provide to these functions should not print anything directly, instead they should return the information to be printed.

7.4.1 Example

```
import sim_commands

def get_info(obj):
    return [("Connections",
             [("Up",   obj.up),
              ("Down", obj.down)]),
            ("Sizes",
             [("Width",  obj.width),
              ("Height", obj.height),
              ("Area",   obj.width * obj.height)])]

sim_commands.new_info_command('sample-device', get_info)

def get_sample_status(obj):
    return [(None,
             [("Attribute 'value'", obj.value)])]

sim_commands.new_status_command('sample-device', get_status)
```

7.4.2 Information Structure

The data returned from the functions should be a list of *sections*, where each section is a tuple of a section title and a list of entries. The section title should be a string or `None`. An entry is a tuple of a name and a value. The name is a string, and the value can be just about anything.

Chapter 8

Miscellaneous Information

8.1 System Calls and Signals

For Unix-like operating systems, Simics will register its built-in signal handlers to make system calls restartable after the signal has been handled (cf. the `SA_RESTART` flag in the `sigaction(2)` man page).

However, only some system calls are restartable, so when writing modules for Simics, you have to make sure that you restart the other system calls yourself:

```
do {
        res = accept(sock, &saddr, &slen);
} while (res == -1 && errno == EINTR);
```

8.2 Text Output

Since Simics has its own handling of text output, which allows users to redirect output to several recipients, the standard C library output routines should not be used from within a Simics module. However, by including the `simics/api.h` header file, some C library function names are redefined as preprocessor macros, which call the Simics versions of these functions instead. E.g., ***vprintf*** is redefined as:

```
#undef vprintf
#define vprintf(str, ap) SIM_printf_vararg(str, ap)
```

This should work just fine for most applications, but if you, for some reason, need to call the actual C library functions, you can either `#undef` the replacement definition of that function, or insert `#define SIM_BC_NO_STDOUT_REDEFINE` before including `simics/api.h`. See the section about `SIM_BC_NO_STDOUT_REDEFINE` in the Simics *Reference Manual* for details about these macro definitions.

Whenever Simics is requested to write output using the ***SIM_write()*** family of functions (see the Simics *Reference Manual* description of ***SIM_write*** for a list of these), that text is

written to *stdout*, and a call to each previously registered output handler is made. You can register new output handlers using the ***SIM_add_output_handler()*** function.

Here is a pseudo-code example of how a module could get Simics to write a copy of all its text output to a log file:

```
static void
output_handler(void *file, const void *buf, size_t count)
{
    fwrite(buf, 1, count, (FILE *)file);
}

static void
init_local(void)
{
    SIM_add_output_handler(output_handler,
                           (void *)fopen("my-log.txt", "w+"));
}
```

8.3 Using Threads in Simics Modules

It is possible to write modules for Simics that use POSIX threads, though the Simics API functions can only be called from the main Simics thread. For another thread to get control of the main thread, it can call the only thread-safe Simics API function ***SIM_thread_safe_callback()***. This function will install a callback that Simics main thread will call as soon as possible, even if Simics is at prompt waiting for a command input.

When the user-installed callback is run, it is executed in the main Simics thread, and it is safe to call API functions from there. Another thread in the module may at this time also call API functions, if it synchronizes correctly with the callback function.

Some support functions for C/C++ programmers, that are not really part of the Simics API, are also safe to call from a thread. This includes the ***vtprintf()*** family of functions, such as ***pr()***, and also the VTMEM malloc routines.

8.4 Header Inclusion Order

For modules written in C or C++, the general order of header file inclusion should be from most to least general:

Language	eg. <stdio.h> (C), <map> (C++)
Operating system	eg. <unistd.h> (Unix), <windows.h> (Windows)
Simics	eg. <simics/api.h>
Application	specific to your module

Observing this order is likely to minimise any problems of interference between include files. In particular, Simics may redefine some standard functions with preprocessor macros and this can cause problems unless the standard headers are included first.

Chapter 9

Compatibility

Starting with Simics 3.0, Simics has a backward compatible ABI for minor versions (third digit change). Any exceptions are noted in the release notes. As a result, user written modules can be moved to a newer minor Simics release without recompilation. The Simics API is backward compatible. If compiling towards a older major version API, the backward compatibility support must be explicitly requested in the module makefile, as described later in this chapter.

9.1 Simics API

The functions and types defined in the Simics API are supported between Simics versions. These functions all have the `SIM_` prefix. There are also some internal functions prefixed `VT_` and `DBG_` that are not supported and usually not documented. The Simics API can be accessed from C/C++ by including the `simics/api.h` header file. The Simics specific memory allocation functions from `simics/alloc.h` are also supported.

Simics also defines some utility functions in `simics/utils.h`, that can be useful when programming in C/C++. These functions are currently *not* part of the supported Simics API.

9.2 Supported Python API

The `cli`, `sim_core` and `sim_commands` Python modules export several functions that may be used by Simics module commands. The following functions and symbols are supported.

Python module `cli`:

- ***number_str()***
- ***str_t(), int_t(), range_t(), float_t(), flag_t(), addr_t(), filename_t(), obj_t(), int64_t(), sint64_t(), uint64_t(), int32_t(), sint32_t(), uint32_t(), integer_t()*** (used by ***new_command()***)
- ***object_expander()***

- ***new_command()***
- ***arg()***
- ***eval_cli_line()***
- ***run_command(), quiet_run_command()***
- ***get_available_object_name()***
- `simenv`
- `CliError`

Python module `sim_commands`:

- ***new_info_command()***
- ***new_status_command()***

Python module `sim_core`:

- `pre_conf_object`

9.3 Migrating Modules from Older Versions

There are only minor changes in the API between Simics version 3.0 and 2.2. Some functions that have been marked deprecated for a long time were removed.

The Simics API was changed for the 2.0 release to allow modeling of heterogeneous target systems, i.e., systems that have processors of different architectures all running in the same Simics session.

Modules written for the 1.4 and 1.6 versions of Simics can usually be compiled for Simics 2.0, using the *obsolete API* backward compatibility layer. Modules for version 1.8 should compile without any major changes, but in all those cases, there may be some minor updates that have to be done manually.

Modules from older versions has to specify what API that are written for by setting the make variable SIMICS_API. This variable can have one of the following values:

`3.0`
Compile module for the 3.0 API.

`2.2`
Compile module for the 2.2 API.

`2.0`
Compile module for the 2.0 API.

1.8
Compile module for the 1.8 API.

1.6
Compile module for the 1.6 API.

1.4
Compile module for the 1.4 API.

9.4 Migrating Python Code from Older Versions

To run a Python module, or some other Python Code, in Simics that was written for an older version, the old API define has to be included early in the Python source file. For example, to run Python code written for Simics 2.0, add the following line:

```
from simics_2_0_api import *
```

9.5 Build Environment Compatibility

Only the make variables, and makefiles described in the Programming Guide are supported. User written scripts and makefiles that rely on the internals of the Simics makefiles and other parts of the build environment may break in future Simics versions.

Chapter 10

More Complex Examples

Three example device models are included in the DML Toolkit package; an AM79C960 (ISA) Ethernet controller, a DS12887 real-time clock and a DEC21140A (PCI) Ethernet controller.

10.1 AM79C960-dml

AM79C960 was a rather common ISA Ethernet card, used mostly in PCs. It may be a little out-dated by now, but still serves as a good example.

Normally the module and device class would both have been named **AM79C960**, but because Virtutech already had a AM79C960 device model written in C, the sample DML module and the device class are named **AM79C960-dml**.

It may be good to have the documentation for the AM79C960 chip when looking at the sample code, so that you can compare the code to the specification. The documentation can be found on the Internet, search for `am79c960.pdf` on `www.google.com` and you will get lots of links to it.

The source code for the sample device can be found in the directory `[simics]/src/devices/AM79C960-dml`. If you want to try modifying the **AM79C960-dml** module yourself, we recommend that you set up a user workspace and copy the source code there, as described in section 3.

If you compile the **AM79C960-dml** module yourself, you will see some compilation warnings like this:

```
/home/mve/simics/src/devices/AM79C960-dml/AM79C960.dml:567: ➥
In AM79C960_dml.mac_address.set
/home/mve/simics/src/devices/AM79C960-dml/AM79C960.dml:567: ➥
warning: not a function: sscanf
```

They do not indicate any error. They are caused by a limitation in the current DML compiler. There is currently no support for proper declarations of C functions with variable numbers of arguments, which causes the warnings.

10.1.1 Running the AM79C960 model

If your distribution contains the simulated machine *enterprise*, you can find the Simics script `enterprise-AM79C960-dml.simics` in the directory `[simics]/targets/x86-440bx`. This file creates an enterprise machine using the **AM79C960-dml** module instead of the default **AM79C960 module**. The AM79C960 object is called **lance0**.

To do something interesting with the AM79C960 model we need to connect it to something that it can talk to. We can use a **service-node** for this. These commands create an **ethernet-link**, a **service-node** and connect the AM79C960 device to the **ethernet-link**:

```
simics> new-ethernet-link
Created ethernet-link ethlink0
simics> new-service-node link = ethlink0 ip = 10.10.0.1
netmask = 255.255.255.0
Created service-node ethlink0_sn0
Connecting ethlink0_sn0 to ethlink0
Setting IP address of ethlink0_sn0 on network ethlink0 to 10.10.0.1
ethlink0_sn0
simics> lance0.connect ethlink0
```

Start the simulation and let the machine boot and login as the user `root`. No password is required. Stop the simulation and set the log level of the **lance0** object to 2:

```
simics> lance0.log-level 2
[lance0] Changing log level: 1 -> 2
```

You can now start the simulation again and send a ping packet to the **service-node** by entering `ping -c 1 10.10.0.1` in the console on the simulated machine. The **lance0** object will log what happens:

```
simics> c
[lance0 info] Packet sent, dst ff:ff:ff:ff:ff:ff,
src 10:10:10:10:10:30, length 64 bytes.
[lance0 info] Packet received, dst 10:10:10:10:10:30,
src 20:20:20:20:20:00, length 64 bytes.
[lance0 info] MAC address matches, packet accepted.
[lance0 info] Packet sent, dst 20:20:20:20:20:00,
src 10:10:10:10:10:30, length 102 bytes.
[lance0 info] Packet received, dst 10:10:10:10:10:30,
src 20:20:20:20:20:00, length 102 bytes.
[lance0 info] MAC address matches, packet accepted.
```

We can see that the enterprise machine first sends a 64-byte packet to the broadcast address ff:ff:ff:ff:ff:ff and then receives a 64-byte reply from 20:20:20:20:20:00, the MAC address of the **service-node**. These are an ARP request and an ARP reply to get the MAC address of the **service-node**.

After this the enterprise machines sends a 102-byte packet to the **service-node** and receives a 102-byte reply. These are the actual ping request and ping reply.

If you try to ping again you may not see the ARP packets, since the simulated machine may already have the MAC address of the **service-node**.

If you want more detailed logs you can change the log level to 3 or 4. At log level 3 a lot more information about what is going on in the device will be logged. The device polls for packets to transmit regularly, so this will cause a lot of output. At log level 4 all accesses that the processor does to the device will also be logged.

10.1.2 Comments to the Code

The source code of the AM79C960 module is quite richly commented, so if you have the documentation for the AM79C960 chip you should hopefully be able to understand most of the code without too much problem.

The AM79C960 model is far from complete, it implements just enough functionality that the device model can be used with Linux 2.4. A list of known limitations can be found at the top of the source file.

The implementation of the model is quite straight forward. There is one unusual design, though. The AM79C960 device has many 32-bit and 64-bit values that are split into multiple 16-bit register. The model stores many of these values in a separate attributes, and makes the registers reflect the attribute by using the *data_accessor* template.

For example, the 64-bit *Logical Address Filter* that resides in CSR8 - CSR11 is stored in the *logical_address_filter* attribute. Some of the registers, for example, CSR24 and CSR 25 that contain *Base Address of Receive Ring*, are not even implemented and the values are only stored in separate attributes.

10.2 DS12887-dml

DS12887 is a very common real-time clock device. It is used, among other places, in PC computers. There are also many other devices that are extensions of the DS12887, for example, DS17485 and M5823.

Normally the module and device class would both have been named **DS12887**, but because Virtutech already had a DS12887 device model written in C, the sample DML module and the device class are named **DS12887-dml**.

It may be good to have the documentation for the DS12887 chip when looking at the sample code, so that you can compare the code to the specification. The documentation can be found on the Internet, search for `ds12887.pdf` on `www.google.com` and you will get lots of links to it.

The source code for the sample device can be found in the directory `[simics]/src/devices/DS12887-dml`. If you want to try modifying the DS12887 yourself, we recommend that you set up a user workspace and copy the source code there, as described in section 3.

If you compile the DS12887-dml module yourself, you will see some compilation warnings like this:

```
/home/mve/simics/src/devices/DS12887-dml/DS12887.dml:291: ?
In DS12887_dml.time.set
/home/mve/simics/src/devices/DS12887-dml/DS12887.dml:291: ?
warning: not a function: sscanf
```

They do not indicate any error. They are caused by a limitation in the current DML compiler. There is currently no support for proper declarations of C functions with variable numbers of arguments, which causes the warnings.

10.2.1 Running the DS12887 model

If your distribution contains the simulated machine *enterprise*, you can find the Simics script `enterprise-DS12887-dml.simics` in the directory `[simics]/targets/x86-440bx`. This file creates an enterprise machine using the **DS12887-dml** module instead of the default **DS12887** module. The DS12887 object is called **rtc0**.

You can, for example, log what is happening to the device during the boot by setting the log level of the **rtc0** object to 3:

```
simics> rtc0.log-level 3
[rtc0] Changing log level: 1 -> 3
simics> c
[rtc0 info] Update-ended interrupt triggered, raising UF.
Pressing return
[rtc0 info] Periodic interrupt frequency set to 1024.000000 Hz.
[rtc0 info] Periodic interrupt triggered, raising PF.
[rtc0 info] UF lowered.
[rtc0 info] PF lowered.
[rtc0 info] UIE set.
[rtc0 info] Periodic interrupt triggered, raising PF.
[rtc0 info] Update-ended interrupt triggered, raising UF.
[rtc0 info] Raising interrupt.
[rtc0 info] UF lowered.
[rtc0 info] PF lowered.
[rtc0 info] Lowering interrupt.
[rtc0 info] UIE cleared.
[rtc0 info] Periodic interrupt triggered, raising PF.
[rtc0 info] Update-ended interrupt triggered, raising UF.
```

If you raise the log level to 4 all access the processor does to the device will be logged. The **rtc0** object is accessed a lot during the boot, so you probably don't want to run the entire boot with log level 4.

Note that Linux only uses the real-time clock while booting and shutting down. Once it has booted it uses other timers to keep the time, so to get Linux to access the D12887 again once it has booted, you have to reboot the system.

10.2.2 Comments to the Code

The source code of the DS12887 module is quite richly commented, so if you have the documentation for the DS12887 chip you should hopefully be able to understand most of the code without too much problem.

The DS12887 model is quite complete. The few limitations are listed at the top of the source file.

A difference between the documentation of the DS12887 and the model is that the model has two register banks, while the documentation only describes one. This is because of the way the device is used in PC computers. The registers described in the documentation correspond to the *registers* bank. When the device is used in a PC a small translation device with two registers that forwards accesses to the registers of the DS12887 is mapped in the port space. This translation device corresponds to the *port_registers* bank. If you want to use the model as a pure DS12887, just ignore the port_registers bank.

The model handles the updating of the time registers in a more complicated way than absolutely necessary. A simple implementation could post an event that raises the UIP flag and then an event that lowers the UIP flag, updates the time registers and compares them to the alarm registers each simulated second. To avoid having to post these events, the model instead saves the simulated time that the real-time clock time was last set, and the time it was set to. From this information the the current real-time clock time can be calculated at any time, and the time registers are only updated when they are read. There is a comment above the *base_time* attribute in the source code that describes the details of the time representation.

Similarly, events for the periodic interrupt, alarm interrupt and update-ended interrupt are only posted if the corresponding interrupt flag is not already raised. This implementation means that, since Linux does not use the device after the boot, the model does not need to post any more events once Linux has booted.

The device model uses the time conversion functions ***os_gmtime()*** and ***os_timegm()***, available in the header file `simics/utils.h`. These are not part of the official Simics API, but they are very handy when you need host-independent time conversion functions. They work like the ***gmtime()*** and ***timegm()*** functions available on Unix hosts, but are host independent. They also have the associated types `os_time_t` and `os_tm_t`, which correspond to `time_t` and `struct tm` on Unix hosts.

10.3 DEC21140A-dml

The DEC21140A is a PCI Ethernet card. As the AM79C960, it is obsolete today but it provides a good example of a PCI device written in DML.

Normally the module and device class would both have been named **DEC21140A**, but because Virtutech already had a DEC21140A device model written in C, the sample DML module and the device class are named **DEC21140A-dml**.

It may be good to have the documentation for the DEC21140A chip when looking at the sample code, so that you can compare the code to the specification. The documentation can be found on the Internet, for example by looking for `ec-qn7nc-te.ps.gz` with Google.

The source code for the sample device can be found in the directory `[simics]/src/devices/DEC21140A-dml`. If you want to modify the **DEC21140A-dml** module yourself, we recommend that you set up a user workspace and copy the source code there, as described in section 3.

10.3.1 Running the DEC21140A model with Enterprise

If your distribution contains the simulated machine *enterprise*, you can find the Simics script `enterprise-DEC21140A-dml.simics` in the directory `[simics]/targets/x86-440bx`. This file creates an enterprise machine using the **DEC21140A-dml** module instead of the default **AM79C360** module. The DEC21140A object is called **dec0**.

The script `enterprise-DEC21140A-dml.simics` configures automatically a network link (connected to the DEC device) and a service-node on the link.

Start the simulation, let the machine boot, and log in as the user `root`. No password is required. You need to load the right kernel module for the DEC device to be recognized:

```
[root@enterprise root]# modprobe tulip
PCI: Setting latency timer of device 00:02.0 to 64
[root@enterprise root]#
```

Once the driver is loaded, the card will be automatically configured by the system. Wait a few seconds and issue an `ifconfig` command to check that the device `eth0` is configured:

```
[root@enterprise root]# ifconfig
eth0      Link encap:Ethernet  HWaddr 10:10:10:10:10:30
          inet addr:10.10.0.15  Bcast:10.10.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:78 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:168 (168.0 b)
          Interrupt:10 Base address:0xd080

lo        Link encap:Local Loopback
[...]
[root@enterprise root]#
```

You can try to ping the service-node to check that networking is working:

```
[root@enterprise root]# ping 10.10.0.1
PING 10.10.0.1 (10.10.0.1) from 10.10.0.15 : 56(84) bytes of data.
64 bytes from 10.10.0.1: icmp_seq=1 ttl=31 time=0.286 ms
64 bytes from 10.10.0.1: icmp_seq=2 ttl=31 time=0.095 ms
64 bytes from 10.10.0.1: icmp_seq=3 ttl=31 time=0.095 ms
64 bytes from 10.10.0.1: icmp_seq=4 ttl=31 time=0.095 ms
```

```
64 bytes from 10.10.0.1: icmp_seq=5 ttl=31 time=0.094 ms
64 bytes from 10.10.0.1: icmp_seq=6 ttl=31 time=0.095 ms
[...]
```

10.3.2 Running the DEC21140A model with Ebony

If your distribution contains the simulated machine *ebony*, you can find the Simics script `ebony-linux-dec21140a-dml.simics` in the directory `[simics]/targets/ebony`. This file creates an ebony machine using the **DEC21140A-dml** module. The DEC21140A object is called **dec0**.

The script `ebony-DEC21140A-dml.simics` configures automatically a network link (connected to the DEC device) and a service-node on the link.

Start the simulation and let the machine boot. The network card will be configured automatically at the prompt. You can try to ping the service-node to check that networking is working:

```
# ping 10.10.0.1
PING 10.10.0.1 (10.10.0.1): 56 data bytes
64 bytes from 10.10.0.1: icmp_seq=0 ttl=31 time=0.0 ms
64 bytes from 10.10.0.1: icmp_seq=1 ttl=31 time=0.0 ms
64 bytes from 10.10.0.1: icmp_seq=2 ttl=31 time=0.0 ms
64 bytes from 10.10.0.1: icmp_seq=3 ttl=31 time=0.0 ms
```

10.3.3 Comments to the Code

The source code of the DEC21140A module is quite heavily commented, so if you have the documentation for the DEC21140A chip you should hopefully be able to understand most of the code without too much problem.

The DEC21140A model is far from complete, it implements just enough functionality that the device model can be used with Linux.

Note that the source code has been divided into two files:

`DEC21140A.dml`
: This file contains the register bank definitions for the PCI configuration and the CSR registers.

`DEC21140A-eth.dml`
: This file contains code to handle the network and Ethernet frames..

The DEC21140A model is built around other models: the SROM is configured as a Microwire EEPROM; the MAC layer and communication with the Ethernet link are handled by the generic **mii-transceiver** and **mii-bus-management** modules. In each case, the DEC device communicates via the interfaces that these devices export.

Chapter 11

Writing PCI Devices

11.1 Introduction

Simics provides support for writing your own PCI devices, both in DML and in C/C++.

- In DML, *templates* are provided to make it easy to generate a PCI device and re-use the existing code to handle the configuration registers, generate PCI interrupts, etc. Section 11.2 present these templates in more details. An example of a DML PCI device is the DEC21140A model described in section 10.3.
- In C or C++, Simics provides a *PCI Device API* defined in `pci-device.h` and implemented in `pci-device.c`. The `pci-device.c` provides several generic PCI device functions and interfaces to communicate with the **pci-bus**. There is a sample PCI device included with Simics called **sample-pci-device** that shows how the PCI Device API can be used. Section 11.3 provides an overview of the API. Function and type specific information can be found in the *Simics Reference Manual*.

11.2 Writing a PCI Device in DML

11.2.1 Overview

Note: This chapter assumes a basic knowledge of the PCI standard, how PCI is modelled in Simics and how to program in DML. Refer to the *Simics User Guide* for Simics PCI modelling information.

Creating a minimal PCI device is very easy: you just need to import the file `pci-device.dml` in your device. It will define the following:

- A PCI configuration register bank called `pci_config`, populated with the registers defined by the PCI standard.
- A `pci_bus` *connect* object to let the PCI device be plugged into a PCI bus.
- Several functions to handle DMA and interrupts.

- Several templates to add PCI or PCI Express capabilities to the device with minimal effort.

In this section, we will use as an example the model of a DEC21140A network card. The source code of the model is available in the `src/devices/DEC21140A-dml/` directory and section 10.3 contains a more complete description. Let us now create a minimal DEC21140A:

```
dml 1.0;

device DEC21140A_dml;

import "utility.dml";
import "io-memory.dml";
import "pci-device.dml";
```

We now have a working—although not very useful—PCI device. Let us have a closer look at what is provided by `pci-device.dml`.

11.2.2 pci_config Bank

The `pci_config` bank defines the standard PCI configuration registers for a type 0 PCI header. Following the PCI standard, it interprets all accesses as little-endian.

Some of the registers defined by `pci_config` have a **default value**. Others need to be customized before the device can be used. The *vendor_id*, *device_id*, *revision_id*, *class_code*, *subsystem_vendor_id* and *subsystem_id* registers are handled in a special way to cover the different possible implementations. They are declared as `maybe_constant` registers, which gives them the following behavior: if you set the parameter `value` in one of them, this register will act as a constant. If you do not set `value`, it will act as a read-write register, and take `hard_reset_value` as initial value.

In the DEC21140A device, `pci_config` is extended to give some of these registers a constant value:

```
bank pci_config {
    ...
    register revision_id { parameter value = 0x21; }
    register class_code { parameter value = 0x020000; }
    ...
    register subsystem_vendor_id { parameter value = 0x0E11; }
    register subsystem_id { parameter value = 0xB0BB; }
    ...
}
```

Base address registers are not defined by default, but a set of templates is provided for you to customize them. Refer to section 11.2.3 for more information.

The `pci_config` bank defines a number of **functions that can be called** to let the PCI device perform various actions:

`pci_raise_interrupt()` and `pci_lower_interrupt()`
Respectively raise and lower the interrupt pin defined by the value of the register *interrupt_pin*. A call to one of these functions has no effect if the interrupt is already in the desired status.

`pci_system_error()`
Send a SERR# signal to the PCI bus.

`update_all_mappings()`
This function is described in section 11.2.3.

As an example, the DEC21140A model handles interrupt in the following way (from `DEC21140A-eth.dml`):

```
// Update interrupt summary and PCI interrupt pin
// according to current status
method update_interrupts()
{
    // update interrupt summary
    ...

    // set PCI interrupt pin
    if (($csr.csr5.nis | $csr.csr5.ais) != 0)
        call $pci_config.pci_raise_interrupt();
    else
        call $pci_config.pci_lower_interrupt();
}
```

The `pci_config` bank also provides a number of **functions that you can override** with your own implementation to react to some PCI events:

`interrupt_ack()`
Called when an Interrupt Acknowledge cycle has been sent to the bus. The default function only prints a warning. Unless your system uses Interrupt Acknowledge cycle, which is pretty uncommon, you can leave this function as it is.

`bus_reset()`
Called when the bus is reset. The default function performs a soft-reset on the device and remove all BAR mappings. You should override this function if you need to perform some special action at reset that is can not be handled by the ***soft_reset()*** functions of the registers.

The parameter `busmaster` can be overridden in the `pci_config` bank to prevent the device from doing DMA accesses. It defaults to true.

11.2.3 Base Address Registers

Introduction

The base address registers (BARs) are not defined by `pci_config`, in order to leave you complete freedom to organize them. By default, they will map the PCI device itself in either the PCI memory or I/O space, depending on the type of BAR defined. To each BAR is associated a *function number* (controlled by the `map_func` parameter) that will be used to identify which mapping of the PCI device is accessed. To let some automatic functions handle the base address registers, you will need to provide a list of active BAR by overriding the `base_address_registers` parameter.

Let us take an example. The following device defines one I/O BAR and two memory BARs:

```
bank pci_config {
    ...
    parameter base_address_registers = ["base_address_0",
                                        "base_address_1",
                                        "base_address_2"];

    register base_address_0 @ 0x10 is (io_base_address) {
        parameter size_bits = 7;
        parameter map_func = 2;
    }
    register base_address_1 @ 0x14 is (memory_base_address) {
        parameter size_bits = 12;
        parameter map_func = 2;
    }
    register base_address_2 @ 0x18 is (memory_base_address) {
        parameter size_bits = 14;
        parameter map_func = 6;
    }
    // Other base address registers are not used
    register base_address_3 @ 0x1C is (no_base_address);
    register base_address_4 @ 0x20 is (no_base_address);
    register base_address_5 @ 0x24 is (no_base_address);
    ...
}
```

The BAR *base_address_0* is defined as mapping into PCI I/O space. The `map_func` parameter indicates that our PCI device will be mapped with the function number 2. The `size_bits` parameter determines how many address bits will be used in the BAR, and thus how big the mapping is. A `size_bits` of 7 means that the bits [31:7] will be used, thus the mapping will be 128 bytes long ($2 << 7$).

The BAR *base_address_1* is defined as mapping into PCI memory space. The PCI device will be mapped in a 4096 bytes space. The mapping will use the function number 2 as well,

thus the PCI device won't see any difference between an I/O access via the BAR0 space and a memory access via the BAR1 space.

The BAR *base_address_2* is defined as mapping into PCI memory space. The PCI device will be mapped in a 16384 bytes space. The mapping will use the function number 6.

In DML, the function number associated to the mapping is usually used to determine which register bank should be accessed (each register bank has to have a `function` number). This makes it very easy to map a given register bank via the PCI BAR mechanism. If we add the following code to our example:

```
bank first_bank {
    parameter function = 2;
    ...
}

bank second_bank {
    parameter function = 6;
    ...
}
```

The bank `first_bank` will be mapped by BAR0 in I/O space and BAR1 in memory-space. The bank `second_bank` will be accessible via the space define by BAR2. No additional code is necessary for the device to automatically receive accesses to the right register bank.

Note: The `pci_config` bank is defined as having a function number of 255. This is a Simics convention that allows the PCI bus object to automatically map the configuration registers of PCI devices in the configuration space and ensure that the right register bank is handling the access.

Template, Parameters, and Functions

The following templates are provided to define base address registers:

`memory_base_address`
Defines a 32-bit memory base address register.

`memory_base_address_64`
Defines a 64-bit memory base address register. Note that the register will be 8 bytes wide.

`io_base_address`
Defines a 32-bit I/O base address register.

`no_base_address`
Defines a 4-bytes long non-implemented base address register.

The following parameters can be customized in a given base address register:

map_func
Function number associated to the mapping.

size_bits
Least significant bit for the base address, which indicates the size of the mapping.

A base address register is composed of the following fields:

- The field *s* at bit 0. Its default value is automatically defined by the type of mapping chosen (0 for memory and 1 for I/O). However, you can redefine it by setting the parameter `override_map_bit` to either 0 or 1.
- The field *type* (only for memory base address registers) at bits [2:1] is a constant defining the type of the mapping. It defaults to 0.
- The field *p* at bit 3 for memory base address registers and 1 for I/O base address registers. It defines whether the mapping is prefetchable or not. It defaults to 0.
- The field *base* at bits [31:$size_bits] or [63:$size_bits] contains the base address and will automatically remap the space it controls when changed.

Note that the three first fields have **no functional meaning in Simics**, i.e., changing *s* will not change in which space the mapping is done. The only field having side-effects is the *base* field.

A base register address gives you access to the following functions:

update_mapping()
Triggers a de-mapping and remapping of the space controlled by the BAR. This is a useful function if the *base* is changed by some other means than a memory transaction.

pci_mapping_enabled() -> (bool enabled)
This function is called to check whether a mapping is enabled or not. By default, it returns always true, but you can override it to check, for example, another register before enabling the mapping. Note that the standard control provided by the bits `io` and `mem` in the *command* register in `pci_config` are taken into account *independently* of the value returned by ***pci_mapping_enabled()***.

pci_mapping_object() -> (conf_object_t *obj)
This function is called to obtain the object that should be mapped by the BAR. By default, the PCI object itself is returned, but this can be customized to be any object. Note that this function is called **when removing and when adding the mapping** to the corresponding PCI space, so it is responsible for returning **the same value** for the automatic mapping mechanism to work properly.

pci_mapping_target() -> (conf_object_t *target)
This function is called to obtain the target that should be mapped by the BAR. By default, no target is used. This function allows a PCI device to create a translator or bridge mapping for another object. It is only called when mapping in the PCI space.

pci_mapping_customize(map_info_t *info)
This function is called after the map_info_t information for the mapping has been set up but before the mapping is done. It allows you to customize the map_info_t structure (changing the base address, the size, the priority, etc.) in any way you want, except for the function number associated with the mapping, as this would break the automatic mapping mechanism. The device will print out a warning if you attempt to do that and ignore the new value.

pci_bar_size_bits() -> (int bits)
This function is called when reading the BAR value and when creating the mapping. By default, size_bits is returned. By overriding this function it is possible to create dynamically resizable BARs, where the size is configured by e.g. another configuration register.

pci_bar_is_64() -> (bool is_bar_64)
This function is called when reading the BAR value and when creating the mapping, but only for 64-bit memory BARs. By default, true is returned. If false is returned, the base address will be limited to 32 bits. In other words, it allows you to dynamically reconfigure a 64-bit BAR to 32 bits.

The pci_config bank also gives you access to the ***update_all_mappings()*** function that calls ***update_mapping()*** on all active base address registers (defined by the base_address_registers parameter).

Expansion ROM

The *expansion_rom_base* register is an already customized base address register that will map the ROM object provided by the attribute *expansion_rom*. To enable this register, you need to set the enabled parameter to true, and specify the map_func and size_bits parameters.

11.2.4 Functions

A number of functions are provided to help writing PCI devices:

pci_data_from_memory()* and *pci_value_from_memory()

```
pci_data_from_memory(addr_space_t space, void *buffer,
                     uint64 address, uint64 size)
    -> (exception_type_t ex)

pci_value_from_memory(addr_space_t space, uint64 address,
                      uint8 size)
    -> (uint64 value, exception_type_t ex)
```

Perform a DMA read.

pci_data_to_memory()* and *pci_value_to_memory()

```
pci_data_to_memory(addr_space_t space, void *buffer,
                   uint64 address, uint64 size)
    -> (exception_type_t ex)

method pci_value_from_memory(addr_space_t space, uint64 address,
                             uint8 size)
    -> (uint64 value, exception_type_t ex)
```

Perform a DMA write.

For example, the DEC21140A model reads its descriptors with the following function:

```
// Read a descriptor from memory at address 'addr' to fill in 'desc'
method read_descriptor(descriptor_t *desc, uint32 addr)
    -> (exception_type_t ex)
{
    local int i;
    local uint32 *desc_data = cast(desc, uint32 *);

    log "info", 4, 0: "Fetching a descriptor at address 0x%x", addr;
    call $pci_data_from_memory(Sim_Addr_Space_Memory,
                               cast(desc, void *), addr,
                               sizeof(descriptor_t))
        -> (ex);
    ...
}
```

11.2.5 PCI and PCIe Capabilities

The command DML PCI code contains templates for a number of capabilities. These templates do not provide any side-effects. Registers are behaving as they are expected to by software (`read_only`, `read_write`, `write_1_clears`, etc.) but all other side-effects should be customized.

Each template provides two parameters that should be overridden: an offset parameter that defines where the capability begins in the configuration registers space, and a next pointer parameter that defines where the next capability will be.

A device with the Power Management capability defined by the PCI standard could be implemented this way:

```
bank pci_config {
    ...
    // Power Management
    parameter pm_offset = 0x50;
```

```
    parameter pm_next_ptr = 0x58;
    is defining_pci_power_management_capability;

    // Customize the Power Management capability
    register pm_capabilities {
        field version  { parameter hard_reset_value = 0x2; }
        field ds_init  { parameter hard_reset_value = 0x1; }
        field pme_supp { parameter hard_reset_value = 0x19; }
    }
    ...
}
```

These are the available capabilities with their related parameters:

Power Management

```
    template defining_pci_power_management_capability
        parameter pm_offset
        parameter pm_next_ptr
```

MSI (32 bits)

```
    template defining_msi_capability
        parameter msi_offset
        parameter msi_next_ptr
```

MSI (64 bits)

```
    template defining_msi64_capability
        parameter msi_offset
        parameter msi_next_ptr
```

PCI-X

```
    template defining_pcix_capability
        parameter pcix_offset
        parameter pcix_next_ptr
```

PCI Express

```
    template defining_pci_express_capability
        parameter exp_offset
        parameter exp_next_ptr
```

PCI Express Advanced Error Reporting Capability

```
template defining_pcie_advanced_error_reporting_capability
    parameter aer_offset
    parameter aer_next_ptr
```

Device Serial Number Extended Capability

```
template defining_pci_device_serial_number_extended_capability
    parameter dsn_offset
    parameter dsn_next_ptr
```

Device Power Budgeting Extended Capability

```
template defining_pci_device_power_budgeting_extended_capability
    parameter dpb_offset
    parameter dpb_next_ptr
```

Virtual Channel Extended Capability

```
template defining_pci_virtual_channel_extended_capability
    parameter vce_offset
    parameter vce_next_ptr
```

11.2.6 Source Code

For more detailed information, the source code of the PCI templates and functions is available in your installation in the directory `[simics]/`*host*`/lib/dml/1.0/`, in the files `pci-device.dml` and `pci-common.dml`.

11.3 Writing a PCI Device in C/C++

11.3.1 API and Interfaces

The following APIs and interfaces can be used when writing PCI devices in Simics.

PCI Device API
Used for PCI devices, and optionally for host-to-PCI bridges.

Simics API
Since it is used for all Simics modules, it also applies to PCI devices.

PCI_BUS_INTERFACE
Implemented by the **pci-bus** module in Simics. This interface shouldn't be used directly by PCI devices. The PCI Device API should be used instead. PCI bridges can use this interface to configure the **pci-bus**.

PCI_DEVICE_INTERFACE
Simics internal interface that shouldn't be used directly. May change without notice. Developers should use the PCI Device API instead.

PCI_BRIDGE_INTERFACE
This is a dummy interface currently implemented by all PCI bridges. It will probably be removed in a future Simics version.

PCI_INTERRUPT_INTERFACE
This interface should only be implemented by PCI devices that handle interrupts, i.e., when a device other than the bridge handles PCI interrupts on the PCI bus.

11.3.2 PCI Device Setup

When creating a PCI device using the *PCI Device API*, some common initialization must be done. First, typically in the ***init_local*** function of the module, the PCI device-specific attributes must be registered. This is done by calling ***PCI_register_device_attributes***.

Then, when the object is created, the generic part of the PCI device must initialized. The ***PCI_device_init*** function is used for this, and is called from the device's ***new_instance*** function. It takes several function pointers as arguments. These functions are used to override the generic PCI device behavior. Most of them are optional, but some must be implemented by the new device.

Once the generic part is initialized, the configuration registers should be setup. Several configuration registers are device specific and have be assigned their values here. Example:

```
PCI_write_config_register(&bcm_ptr->pci, PCI_VENDOR_ID,   0x12ae);
PCI_write_config_register(&bcm_ptr->pci, PCI_DEVICE_ID,   0x1647);
PCI_write_config_register(&bcm_ptr->pci, PCI_STATUS,      0x02b0);
PCI_write_config_register(&bcm_ptr->pci, PCI_REVISION_ID, 0x00);
PCI_write_config_register(&bcm_ptr->pci, PCI_CLASS_CODE,  0x020000);
PCI_write_config_register(&bcm_ptr->pci, PCI_HEADER_TYPE, 0x00);
...
```

11.3.3 PCI Device Mappings and BARs

Among the configuration registers there are some base address registers (BARs) that specify the memory and I/O mappings that the device supports. The software uses the BARs to figure out how large the mappings are, where they can be placed in memory, and also to actually map the device in the corresponding memory spaces.

The PCI Device API has two functions that can be called from ***new_instance*** to tell the generic PCI code to handle accesses to a BAR. ***PCI_handle_mapping32*** and ***PCI_handle_mapping64*** are used for 32 and 64-bit mappings respectively. Note that 64-bit mappings use two base address registers.

There are also PCI Device API functions to modify the mappings for the device at runtime. This is usually not needed, but there are PCI devices that allow software to modify the mappings in more ways than with BARs, and in this case the functions in question are useful: ***PCI_set_map_base***, ***PCI_get_map_base***, ***PCI_set_map_size***, ***PCI_get_map_base***, ***PCI_set_map_enable***, ***PCI_get_map_enable***, ***PCI_set_map_offset*** and ***PCI_get_map_offset***.

11.3.4 Accessing Memory

There are several functions in the PCI Device API that can be used to access the configuration, memory and I/O spaces. Calling them is equivalent to issuing PCI bus transactions in a real system. This should only be done by master capable devices.

PCI_data_from_memory and ***PCI_data_to_memory*** access raw data from a specified memory space. No endian conversion is performed. These functions are often used for DMA block transactions.

PCI_value_from_memory and ***PCI_value_to_memory*** use a value in host-endian order, and performs byte-swapping if needed. This is for example useful when the device reads data from memory that it should interpret.

11.3.5 Interrupts

The functions ***PCI_raise_interrupt*** and ***PCI_lower_interrupt*** changes the output value of the interrupt pin used by the device. The interrupt pin is specified in the INTERRUPT_PIN configuration register.

If the interrupt pin is already asserted, a call to ***PCI_raise_interrupt*** has no effect, and similar for ***PCI_lower_interrupt*** when the interrupt pin already is lowered. This is important to be able to model level triggered interrupts, where the interrupt target can count the raise and lower calls from different devices to determine the actual interrupt signal level.

Chapter 12

Writing Components

12.1 Introduction

This chapter describes how to design and write new components. The reader is assumed to be familiar with Simics components and how they are used, as well as configuration classes, objects and attributes. The *Configuration and Checkpointing* chapter in the *Simics User Guide* contains relevant documentation.

12.1.1 Component Basics

Components are implemented in Python and are handled like any other Simics objects. While a component object represents a hardware unit such as a processor board, or PCI card, there are one or more configuration objects associated with it that implement the actual functionality.

Components can be created, deleted, and connected to and disconnected from each other. Each component defines a number of connectors that define how it can be connected to other components. Connectors have direction, up or down. As a result, the connected components form a tree-like hierarchy with a single *top-level* component. There are also special connectors that are neither up nor down: these connectors, called "any", can be used to connect component hierarchies together.

12.1.2 Designing Components

Before writing new components it is important to define the system component hierarchy. This includes deciding: what should the components contain; what connectors are needed between them; if any new connector types are needed; and in a large system, what component is the top-level one. A component should typically have connectors that correspond to real-world connections that are accessible to users. There is no point in having a connector between the processor and the motherboard if, for example, the processor is soldered to it and cannot be replaced. Connections on such a low level are handled by the configuration objects that actually model the hardware.

12.2 Component Module

A component module is a common Simics module containing the implementation of one or more related *component collections*. The `sun-components` module, for example, contains components for several different machine architectures from Sun Microsystems. Each architecture is represented by one component collection.

12.2.1 Component Module Files

A component module source directory may contain several `<name>_components.py` files, each implementing a component collection; one `Makefile`; and one `gcommands.py` file.

The `Makefile` looks the same for all component modules, it simply lists all components that are part of the module.

The `gcommands.py` is loaded when Simics starts. It is responsible for registering all component collections defined by the module using the ***register_components()*** function. A `gcommands.py` file may have the following contents:

```
from components import register_components

register_components('ppc440gp-components')
```

This example registers the **ppc440gp-components** component collection and creates the **import-ppc440gp-components** command that must be run by the user to make any commands related to this collection available in the frontend.

Note: If the component collection does not exist, i.e., there is no .py file with the name given to register_components(), there will be no error message. The reason for this is that a component module may be distributed without some of the collections it define. A successful registration of a collection can be verified by checking that the **import-<collection>** command was created.

12.3 Implementing Components

12.3.1 Component Class

A component file contains Python classes that define all the component classes in a component collection. These classes inherit from the component base class **component_object**, or from one of its sub classes.

Component classes are registered with Simics using the ***register_component_class()*** function. When a component class is registered, a corresponding Simics class is created with the supplied list of attributes. The Simics class uses the name stored in the `classname` data member of the Python class.

```
def register_component_class(python_class,
                             attributes, checkpoint_attributes = [],
                             top_level = False)
```

The first argument, *python_class*, is the Python component class, *attributes* is a list of attributes that can be used to configure the component at creation time. All attributes in this list will be available as command arguments in the **create-**<**component**> and **new-**<**component**> CLI commands. The format of each attribute list entry is [<name>, <kind>, <type>, <description>]. See the documentation of ***SIM_register_typed_attribute()*** for an explanation of these entries.

The *checkpoint_attributes* is a list with the same format as *attributes*. The difference is that *checkpoint_attributes* is only intended for attributes representing internal state that has to be checkpointed. The attributes in this list are not visible as arguments to the creation commands. This means that the attributes can not be of the pseudo or required kinds.

The last argument, *top_level*, should be set to `True` for top level components; the default value is `False`. The register function will add some additional attributes for top level components, and also check that the component class implements all top-level-only methods.

12.3.2 Component Class Data Members

There are a number of data members required in the Python component class that are used by the generic component system.

`classname`
The name of the Simics configuration class representing the component class.

`basename`
Default name for instances of this class if none is given when the user creates a component. The complete default instance name is <basename>_cmp<number>.

`description`
A description of the component class. This will be the help text for the configuration class in Simics.

`connectors`
A list of all connectors for components of this class. The format of the list is described in the next sub-section.

Example of a component class:

```
class ide_disk_component(component_object):
    classname = 'std-ide-disk'
    basename = 'ide_disk'
    description = ('The "std-ide-disk" component represents an IDE disk.')
    connectors = {
        'ide-slot' : {'type' : 'ide-slot', 'direction' : 'up',
                      'empty_ok' : False, 'hotplug' : False, 'multi' : False}}
```

```
    ... <class method code removed> ...

register_component_class(
    ide_disk_component,
    [['size', Sim_Attr_Required, 'i',
      'The size of the IDE disk in bytes.'],
     ['file', Sim_Attr_Optional, 's',
      'File with disk contents for the full disk Either a raw file '
      'or a CRAFF file.']])
```

12.3.3 Defining Connectors

The `connectors` member is a dictionary indexed by connector name. For each connector there is another dictionary describing it with the keywords `type`, `direction`, `empty_ok`, `hotplug` and `multi`:

`type`
The name of the connector type. The connector can only be connected to other connectors of the same type.

`direction`
One of `up`, `down` and `any`. Defines the direction of the connector. A connector can only be connected with connectors of a different direction; this is checked by the generic component connection code.

`empty_ok`
`True` if the connector may be empty after instantiation. The component system will not allow instantiation of components that have empty connectors with `empty_ok` set to `False`.

`hotplug`
`True` if the connector supports connecting and disconnecting after the component has been instantiated. A hotplug connector must have empty_ok set to `True`. A component where all connectors are either `hotplug` or `empty_ok` is called a *standalone* component.

`multi`
`True` if the connector supports multiple connections. A few connectors allow several other connectors to be connected to them. One such example is the ISA bus in legacy PC systems.

12.3.4 Required Class Methods

There are a few class methods that have to be implemented by the component class since they are used by the base class when the component is created:

add_objects(self)
Responsible for adding pre-configuration objects to the component. The pre-configuration objects are Python objects that represent the configuration objects before the component has been instantiated and the actual configuration objects are created. The pre-configuration objects allow attributes to be set without any side-effects and simplifies adding and removing objects before the instantiation. A more detailed description of how to add objects follows in the 12.3.7 section. This method is not called when a checkpoint is loaded, since then the object references will be set with real objects directly.

add_connector_info(self)
Adds dynamic information about all connectors, such as what objects that are involved in the connection and other connector specific data, for example the slot number for PCI buses. This dynamic connector information is sent as arguments to the corresponding connect method (12.3.9) in the component being connected to. The information is stored in the `connector_info` data member that is a dictionary indexed with the connector name, and described in section 12.3.8.

connect_<connector-type>(self, connector, <connector arguments>)
For each connector type that the component implements, there must be a corresponding connect method. The second argument is the name of the actual connector, and any following arguments are connector-type specific and provided by the ***add_connector_info()*** method of the other component in the connection. If the connector has hot-plug support, it must support connections for both the instantiated and non-instantiated component.

disconnect_<connector-type>(self, connector)
For each connector type that has hot-plug support that the component implements, there must be a corresponding disconnect method. The second argument is the name of the actual connector. The method must support disconnections for both the instantiated and non-instantiated component.

12.3.5 Optional Class Methods

There are several methods that have a default implementation in the base component class, but that the component sub class may override.

__init__(self, parse_obj)
The standard Python object constructor. Called when the object is created, it may be used to initialize some state before any other method in the object is called. If this method is implemented, it has to call the ***__init__()*** method of the parent class.

finalize_instance(self)
Called when the component object in Simics has been created. The first thing ***finalize_instance(self)*** in the base class does is to call ***add_objects()*** if the component is not instantiated. If this method is implemented, it has to call the ***finalize_instance()*** method of the parent class.

`instantiation_done(self)`
Called when the component is instantiated. Note that this method is not called when loading a checkpoint with already instantiated components. If this method is implemented, it has to call the ***instantiation_done()*** method of the parent class.

12.3.6 Top-level Class Methods

Top-level components need two additional methods that are used by the component system to handle clock domain configuration and to keep track of all processors in the component hierarchy. There is also one optional method that is used to restore the processor list from a checkpoint.

`get_clock(self)`
Should return the processor object that is the default *clock* in the component hierarchy, that is, in the top-level component and all components below it. All configuration objects within this component hierarchy, also called a *timing domain*, will have the same clock object set as *queue*.

`get_processors(self)`
Similar to ***get_clock()*** but should return a list of all processors in the component hierarchy.

`set_processors(self, cpu_list)`
Optional method that is called when a checkpoint is loaded with a list of all processors previously returned by ***get_processors()***. If this method is not implemented, the component should make sure that that it can recreate the processor list after a checkpoint for use in ***get_processors()***.

12.3.7 Adding Configuration Objects

When the component object has been created, the ***add_objects()*** method is called. Its responsibility is to create pre-configuration objects representing the configuration objects that implement the functionality of the component. The **pre_obj()** Python class is typically used instead of **pre_conf_object()**, since it provides improved name handling, as described in 12.3.11.

The pre-configuration objects should be added to the `o` namespace of the component. When the component later is instantiated, all object references in this namespace are converted to configuration objects.

The `o` namespace may contain simple pre-configuration objects, and also arrays of objects.

Example from a Serengeti processor board component:

```
def add_objects(self):
    for i in range(self.num_cores):
        self.o.mmu[i] = pre_obj('mmu(x + %d)' % i, self.mmuclass)
        self.o.cpu[i] = pre_obj('cpu(x + %d)' % i, self.cpuclass)
        self.o.cpu[i].processor_number = get_next_cpu_number()
```

```
self.o.cpu[i].mmu = self.o.mmu[i]
self.o.cpu[i].control_registers = [['mid', i]]
self.o.cpu[i].freq_mhz = self.freq_mhz
```

After instantiation, the `o` namespace will contain references to the newly created Simics configuration objects instead.

12.3.8 Providing Connection Information

For each connector, some information must be supplied to the other component of the connection. This usually includes a configuration object that implements a Simics interface, which is needed by some configuration object in the other component. The information is stored in the `connector_info` data member of the component, and passed to the other component in the call to its connect method. The `connector_info` dictionary is indexed by connector name. This dictionary should be filled in by the ***add_connector_info()*** method that is called directly after ***add_objects()***, and is also called when restoring an instantiated component from a checkpoint.

The format of the values in the connector_info dictionary is a list in the common case. It may be a tuple as well, described in 12.3.10. The length of the list, and the types of the values, depends on the kind of the connector. The up and the down part of a connector may have different arguments in the list.

At instantiation, any pre-configuration references in `connect_info` will be converted to configuration object references.

A dual-function PCI device with two SCSI controllers may have the following ***add_connector_info()*** method. The first controller is implemented by configuration object `sym0` and the second by `sym1`.

```
def add_connector_info(self):
    self.connector_info['pci-bus'] = [[[0, self.o.sym0], [1, self.o.sym1]]]
    self.connector_info['scsi-bus0'] = [self.o.sym0, 7]
    self.connector_info['scsi-bus1'] = [self.o.sym1, 7]
```

For the `pci-bus` connector type, the argument is a list with PCI function number and object implementing the `PCI_DEVICE_INTERFACE` interface. The `scsi-bus` connector type have two arguments, the object implementing the `SCSI_TARGET_INTERFACE` and the initial SCSI ID of the component on the SCSI bus.

12.3.9 Connection Methods

For each connector type that a component implements, it must also have a corresponding ***connect_<connection-type>()*** method.

Arguments

The first argument (excluding the Python `self` reference) to the connect method is the name of the connector. The following arguments come from the `connector_info` list of the

other component in the connection. The connector method typically sets attributes in the objects of the component that are related to the connection.

In the example in the previous section, the `connector_info` for an up-directed `pci-bus` connector in a PCI device was shown. The corresponding connect method in the component that has a down-directed `pci-bus` connector may look like the following. The component in the example has four `pci-bus` connectors, called `pci-slot0`, etc.

```
def add_connector_info(self):
    for i in range(4):
        self.connector_info['pci-slot%d' % i] = [i, self.o.pcibus]

def connect_pci_bus(self, connector, device_list):
    slot = self.connector_info[connector][0]
    bus = self.connector_info[connector][1]
    for dev in device_list:
        bus.pci_devices += [[slot, dev[0], dev[1]]]
```

This method first reads some data for the components own connector, to know what PCI `slot` and `bus` object it has associated with the connector. It then iterates over `device_list`, that is the list with PCI functions and objects from the previous example. This list, together with the PCI slot number, is used to set the `pci_devices` attribute in the bus object.

A connect method should make as few assumptions as possible about the objects that are passed to it. This makes it easier for developers to add new components that use the same standard connector definitions, so that they can be connected to existing components.

Method Invocation Time

The connect methods are called when the components are instantiated, and not when the CLI command <**component**>**.connect**, or **connect-components**, is invoked. This simplifies coding the connect methods, since they are only called once, and implementing disconnection of components before instantiation is simpler.

Method Invocation Order

The connect methods are called starting from the top-level component and working down the hierarchy. The connect method in the lower component of a connection is called while traversing down the hierarchy downwards. When the leaves have been reached, the connect methods are called in reverse order back up the hierarchy. This way components may pass data down the hierarchy which can be used to by lower components to determine what data to pass up the hierarchy.

Connect Failures

A connect method may not fail. If a connection isn't possible, it should be prevented by the standard connector mechanisms, such as connector type and direction checking. In cases

where this isn't enough, it is also possible to have a connector check method that is called before the connect method. This is described in section 12.3.10.

Hotplug Support

For connectors that support hotplugging, there must also be a disconnect method. It is similar to the connect method, but only takes the connector name as argument. This method should restore all object attributes related to the connection to their initial state. To properly support hotplugging, the connect and disconnect methods also have to handle the case of objects in the `o` namespace being real Simics configuration objects and not pre-configuration ones.

Note: Components may only be connected to other components sharing the same instantiation state, i.e., instantiated components may only be connected to other instantiated components, and non-instantiated components may only be connected to other non-instantiated components.

12.3.10 Disallowing Connections

Sometimes the generic connector mechanism does not provide sufficient support for determining when a connection is allowed or not. For example, it may require device specific knowledge. To solve this, the component may have a check method that, unlike the connect method, is called when the connect command is invoked. The check method is called with the same arguments as the connect method by default. Optionally the other component may supply a tuple of two different argument lists. Then the first list is supplied to the connect method, and the second list to the check method.

Check methods are often used for connectors that support multiple connections, i.e. they have the `multi` attribute set to `True`. One example is the ISA bus component that has a multi-connector where the check method makes sure that ports of the attached devices do not collide.

If a check method determines that a connection is not allowed, it should raise a Python exception with a message saying why the connection failed.

12.3.11 Naming of Configuration Objects

Components set the names of configuration objects when the corresponding **pre_obj** object is created, for example:

```
self.o.cpu = pre_obj('cpu', 'x86-p2')
```

The **x86-p2** processor object will get the name **cpu** unless there already exist another object with the same name. In that case, an underscore and a unique number will be appended to the name, for example **cpu_0** for the first duplicate and **cpu_1** for the second.

Automatic Object Name Numbering

The component can decide where in the name the unique number should appear by including a `$` sign in the name. Underscores are not added in that case.

```
self.o.cpu = pre_obj('cpu$', 'x86-p2')
self.o.tlb = pre_obj('cpu$_tlb', 'x86-tlb')
```

The first set of objects created by this code will be called **cpu0** and **cpu0_tlb**, and the second **cpu1** and **cpu1_tlb**. As with the _<number> suffix, the number for the name is assigned when the **pre_obj** object is created.

User-defined Object Name Numbering

In some cases, a simple automatic numbering of objects is not sufficient, but object names are still unknown at creation time. A typical example is a large server with processor boards and a backplane, where processors should be numbered depending on what slot of the backplane the processor board is connected to. The component system solves this by allowing simple parentheses-enclosed expressions in the object name, where `x` is replaced with a supplied value. This expression can be re-evaluated at anytime until the component is instantiated, by calling the ***rename_component_objects()*** method.

The following example creates four processors, numbered from 4 up to 7. The non-evaluated name is `cpu(x + 0)`, `cpu(x + 1)`, etc. When ***rename_component_objects()*** is called, `x` is replaced by 4, and the final object names will be `cpu4`, `cpu5`, etc.

```
for i in range(4):
    self.o.cpu[i] = pre_obj('cpu(x + %d)' % i, 'x86-p2')

self.rename_component_objects('4')
```

12.4 Component Attributes and Commands

12.4.1 Common Component Attributes

All components have the following configuration attributes defined. Refer to the *Simics Reference Manual* for additional documentation.

`connectors`
Dictionary with all connectors. Same as the `connectors` data member.

`object_list`
List of all configuration objects that are part of the component. This corresponds to all objects in the `o` namespace.

`object_prefix`
Object name prefix, set by the **set-component-prefix** command when the component was created.

connections
List of all connections for the component. For each connection there is a sub-list with the name of the connector, the name of the other component, and the name of the connector on the other component.

top_level
Boolean value that is TRUE if the component is a top-level one.

instantiated
Boolean value that is TRUE if the component has been instantiated.

12.4.2 Top-level Component Attributes

Top-level component also have the following attributes:

components
A list of all components below the top-level one in the component hierarchy.

cpu_list
A list of all processors in this top-level component, and in all components below it in the component hierarchy.

12.4.3 User-defined Attributes

There are two kinds of attributes that can be added to a component using the ***register_component_class()*** function. The first and most common kind is intended for simple parameterizing of the component. These attributes are automatically converted to command arguments for the **create-<component>** and **new-<component>** commands. The other kind is intended for attributes that are only providing checkpoint support of the component-internal state, which users are not expected to access. The format of both attribute lists described in section 12.3.1

For each attribute there has to be a pair of set and get methods defined in the component class. The name of the set method should be ***set_<attribute>()***, and similarly for the get method. See the ***SIM_register_typed_attribute()*** documentation in the *Simics Reference Manual* for more information on writing attribute methods.

12.4.4 Standard Component Commands

There are several commands in Simics that are used to create, inspect and manipulate components. The following table lists the most common ones. The *Simics Reference Manual* contains their complete documentation.

create-<component>
Creates a non-instantiated component.

new-<component>
Creates an instantiated component. This command is only applicable to standalone components.

connect-components
Same as <**component**>**.connect**, the <**component**>**.** version is preferred.

instantiate-components
Instantiate all, or a hierarchy of, non-instantiated components.

list-components
Prints a list of all existing components.

set-component-prefix
Set a name prefix used for all components and their configuration objects.

get-component-prefix
Return the component name prefix.

import-<**component-collection**>
Enable CLI commands for a collection of components.

<**component**>**.get-component-object**
Return the instance name of a specified configuration object in a component.

<**component**>**.connect**
Connect component to another component.

<**component**>**.disconnect**
Disconnect component from another component.

<**component**>**.info**
Print static configuration information about the component.

<**component**>**.status**
Print dynamic configuration information about the component.

12.4.5 User-defined Component Commands

It is possible to add additional commands to a component. This is done in the same way as for any other configuration class in Simics. See the **new_command()** documentation for more information.

12.5 Various Component Features

The following additional "good to know" information about components is provided to help the programmer gain a better understanding of components rather than as coding guidelines.

12.5.1 Checkpointing

Most of the state in a component is usually checkpointed via the attributes used to configure it. There are also separate attributes that are only used for checkpointing, discussed in sections 12.3.1 and 12.4.3.

Things to checkpoint include state calculated or received during the connection phase, since it may be needed to support later reconfiguration for hotplugging components.

All information about connectors and connections is checkpointed automatically by the component base class.

12.5.2 Automatic Queue Assignment

All configuration objects in Simics that handle time in some way need a *queue* attribute set. The queue is an object implementing an event queue, also called time queue, needed for the time to advance. All objects that have the same queue are said to be part of the same *time domain*. The only objects that can be currently used as queues are processors and objects of the class **clock**.

The component system automatically sets the queue attribute for all objects at instantiation time, based on the component hierarchy. To override the automatic queue assignment, for example on multi-processor boards where each processor should be its own queue, simply assign the queue attribute when adding the pre-configuration objects.

12.5.3 Automatic Recorder Assignment

Devices that handle input such as serial and network devices, keyboards and mice, usually implement a connection to a **recorder** object. All their input passes through the recorder so that it may record the input to the file and later replay the same input from the file.

The component system automatically creates a recorder and connects it to all input devices that have a recorder attribute. A component can override this automatic assignment by setting the recorder attribute itself for its objects.

12.5.4 Inheritance

Since components are written as classes in object-oriented Python, it is easy to create new components that are similar to existing ones by using inheritance. Instead of basing the component on the **component_object** base class, another component class can be used. The new component class can, for example, modify the connector list, override the methods for adding object and connection information in the parent class. Several south bridge components supported by Simics are all based on the same **south_bridge_component** class, that in turn is based on a class supplying legacy PC devices.

12.6 Standard Connector Types

The following is a list of common connector types found in many of the architecture models implemented by Simics. Machine-dependent connector types are not described in this section. The tables list all connector directions and the `connector_info` for connectors of

each direction, including check methods with different arguments than the connect method. The `connector_info` is used as argument to the connect function of the opposite component during connect, see 12.3.9.

one direction	<first argument>, <second argument>, ...
other direction(s)	<first argument>, ...

agp-bus
Used to connect AGP based graphics devices.

down	<AGP slot>, <agp-bus object>
up	[[<AGP function>, <agp-device object>]*]

ethernet-link
Used to connect Ethernet devices and Ethernet links together. *Up and down connectors of the ethernet-link type can not be connected together. They have to be connected to an "any" connector.*

any	<ethernet-link object>
down	-
up	-

fc-loop
Simplified Fibre-Channel connection. The controller should have a connect function that makes sure that disk IDs on the loop are unique.

down	<fc-controller object>
up	<fc-device object>, <loop ID>

graphics-console
Used to connect a graphical console to a graphics device.

down	<graphics-device object>
up	<graphics-console object>

keyboard
Used to connect a keyboard device to a console for receiving real keyboard input.

down	<keyboard object>
up	<console object>

ide-slot
Provides connection between an IDE controller and IDE disks and CD-ROMs.

down	-
up	<ide-device object>

isa-bus
Used to connect legacy ISA devices to an ISA bus. The connect function in the component with the "down" connector should detect port number collisions.

down	<port-space object>, <memory-space object>, <interrupt object>, <dma object>
up	-
up check	[<port-number>*]

mem-bus

Connection for SDRAM components providing SPD information. The connect function should make sure that only SDRAMs of the correct type and width are inserted in a memory slot.

down	<i2c-bus object>, <i2c-bus address>
up	<memory-megs>, <memory-ranks>
up check	<type>, <memory_megs>, <memory-ranks>, <bit-width>, <ecc-width>

mouse

Used to connect a mouse device to a console for receiving real mouse input.

down	<mouse object>
up	<console object>

pci-bus

Used to connect PCI devices to a standard PCI bus. A component defining the "down" connector must make sure that the pci-bus object has both the *memory_space* and *io_space* attributes set, and that the pre-configuration objects referenced by them each have a *map* attribute.

down	<PCI slot>, <pci-bus object>
up	[[<PCI function>, <pci-device object>]*]

pcmcia-slot

Used to connect PCMCIA (PC-CARD) devices into a PCMCIA controller. *The arguments exported by the up connector is expected to change in a future Simics version.*

down	down <pcmcia-device object>, <slot-id>
up	<attr-space object>, <common-space object>, <io-space object>

sb-interrupt

South Bridge interrupt routing connection.

down	<interrupt object>, <io-apic object>
up	<interrupt object>

scsi-bus

Used to connect SCSI disks and CD-ROM drives to a standard SCSI bus. The component with the "any" connector should make sure that devices are not connected to the bus using the same ID. *Up and down connectors of the scsi-bus type can not be connected together. They have to be connected to an "any" connector.*

any	<scsi-bus object>
down	<scsi-device object>, <SCSI ID>

up	<scsi-device object>, <SCSI ID>

`serial`

Used to connect serial devices together, and to different kinds of serial consoles. *Either the link argument, or the console/device has to be supported, not both*.

down	<link>, <serial-device object>, <console title>
up	<serial-link>, <text-console object>

Chapter 13

Programming the Memory Hierarchy Interface

The *Memory Transactions* chapter in *Simics User Guide* describes how to connect listener objects to the memory hierarchy interface, and how to control the transactions sent to these objects. This section explains how to develop a new listener object by implementing the `timing-model` or the `snoop-memory` interfaces.

13.1 Implementing the Interface

The `timing-model` and `snoop-memory` contains only one function called ***operate()***:

```
static cycles_t
my_timing_model_operate(conf_object_t         *mem_hier,
                        conf_object_t         *mem_space,
                        map_list_t            *map_list,
                        generic_transaction_t *mem_op);
```

The four arguments are:

`conf_object_t *`*mem_hier*
This points to the timing model or snooper itself; it can be safely cast to the actual type of the listening object.

`conf_object_t *`*mem_space*
The *mem_space* argument is the memory space object that the timing model or snooper is connected to.

`map_list_t *`*map_list*
The *map_list* argument describes the entry in the *map* attribute list that this memory operation matched (as returned from a call to the ***space_lookup()*** function in the `lookup` interface).

generic_transaction_t *mem_op

This is a pointer to information about the current memory operation. The `generic_transaction_t` data structure is explained in detail in the Simics API Data Types section of the API chapter in the *Simics Reference Manual*.

13.2 Changing the Behavior of a Memory Transaction

In a Timing Model

An object listening on the `timing-model` interface is presented with memory transactions before they have been executed, and may therefore change both their semantics and their timing. Here is a list of changes that a timing model is authorized to perform:

Setting mem_op->ignore

If set to 1, the memory transaction will not be executed.

Setting mem_op->reissue

If set to 1 and the memory hierarchy is stalling, the memory transaction will be sent again to the timing model after the stalling was performed. If set to 0, the transaction will be executed without further calls to the timing model.

Setting mem_op->block_STC

If set to 1, the transaction won't be cached in the STCs, ensuring that the next access to the same memory area will be sent to the timing model. Default is 0, unless the transaction has side-effects that would prevent it from being cached.

Stalling the Memory Transaction

By returning a non-zero number of cycles, the ***operate()*** function will stall the memory transaction that was passed as argument for that amount of time.

If a zero stall time is returned, some additional operations are allowed:

Setting mem_op->exception

If set to an exception, the transaction will be interrupted and an exception will be taken. Default is no exception (`Sim_PE_No_Exception`).

Setting mem_op->user_ptr

This `void *` pointer is not touched by Simics during the memory operation execution. It can be used to pass information from a timing model to a snoop device.

Memory Store Value

Since the memory operation has not been executed yet, it is possible to change the value of a store operation in the timing model. However, it is *important* to restore the original value in the snoop device once the operation has been performed. The ***SIM_get_mem_op_value()*** and ***SIM_set_mem_op_value()*** functions (and their variants) can be used to get and set the value of a given memory transaction.

In a Snoop Device

An object listening on the `snoop-memory` interface is presented with memory transactions after they have completed. It cannot influence the execution of the operation and it may not return a non-zero value for stalling, but it is allowed to modify the value of the memory operation. Since the data returned by read operations are available at this stage, the snoop device is also an ideal place to trace memory transactions.

The following actions are allowed:

Change a Memory Store Value
If the value of a memory store has been changed in the timing model, it should be reset in the snoop device.

Change a Memory Load Value
Since the operation has been performed, the snoop device is the right place to change the value of a load. This is done with the usual ***SIM_get/set_mem_op_value()*** functions.

13.3 Chaining Timing Models

Sometimes it is desirable to chain timing models, e.g., if you are implementing a multi-level cache model and want to model each level of the cache as an individual class. To do this, the ***operate()*** function must call the corresponding functions of the lower levels (a *lower* or *next* level cache means a cache further away from the CPU, closer to the actual memory).

The `g-cache` source code included with Simics is an example of how to do this. Whenever there is a miss in the cache, the **g-cache** object creates a new memory operation and calls the ***operate()*** method of the `timing-model` interface from the next level cache specified by the *timing_model* attribute.

Chapter 14

The dbuffer library

When sending blocks of data around in the simulation, it is often very important that this can be done in an efficient manner, which usually means that memory copies need to be avoided as much as possible. The typical case for this is in the simulation of networks or other communication links where the hardware model constructs network packets, often by reading from simulated memory, and sends the them over a network link to one or more receivers. These receivers can then each do their modifications to the packets and hand them over to software running on the simulated target system. By using the `dbuffer_t` type, Simics network simulation can do this with a minimum of overhead.

14.1 Dbuffer fundamentals

A *dbuffer* is an opaque data structure that is accessed using functions that updates it or read information from it. Its purpose is to store blocks of data, and the dbuffer library doesn't try to interpret any of the data it contains.

The API is designed so that the implementation can be highly efficient, especially by making the following operations cheap:

- Adding data to the beginning, end, or middle of the buffer
- Removing data from the beginning, end, or middle of the buffer
- Making a copy of the buffer, or parts of it
- Creating new buffers, and releasing them after use
- Reading the data out of the buffer

All of the above mentioned operations can usually be done in constant time, with no copying of data already in the buffers. For the add operation, it is usually necessary to copy the actual data into the buffer, but it is sometimes possible to reusing existing data pointers.

14.2 Usage

This section gives an overview of the most common operations in the dbuffer API.

To create a new dbuffer, call the ***new_dbuffer*** function. This will return a pointer to a `dbuffer_t` that is used in later API calls to manipulate the dbuffer.

To add data to the buffer, the most commonly used function is ***dbuffer_append***. It will extend the size of the dbuffer by the given amount of bytes and return a pointer to the newly added bytes. This pointer can be used to move data into the buffer by using it as the destination address of ***memcpy*** call, or any other suitable way. The pointer returned by ***dbuffer_append***, or the similar functions ***dbuffer_prepend*** and ***dbuffer_insert*** can only be used to write data to the newly added part of the dbuffer, since the memory allocated for the dbuffer may not be contiguous.

There are two ways to read data from the dbuffer, and to write data to it. The first is the simple way of using ***dbuffer_read***, ***dbuffer_update*** and ***dbuffer_replace*** which will return a pointer to a memory block that contains the data that is to be read from or written to. However, sometimes the data in the dbuffer needs to be reshuffled or copied to produce a contiguous block of memory. To minimize the overhead, it is better to use the ***dbuffer_read_some***, ***dbuffer_update_some*** and ***dbuffer_replace_some*** functions instead.

Another important aspect of the dbuffer API is that it is cheap to make a copy of a dbuffer. The ***dbuffer_clone*** function will return a new dbuffer that contains the same data as a given dbuffer, but the two dbuffers are completely independent, so any changes you make to one copy will not affect the other. The cloning is done using a copy-on-write mechanism internally, which means that it doesn't need to copy any data from the original dbuffer, but if one of the dbuffers is updated in a way that would affect the data in the other, a copy of at least parts of the data will be done behind the scenes. Some updates, such as appending to the dbuffer, is still without overhead.

Finally, when the dbuffer is no longer needed, the ***dbuffer_free*** function should be called.

See the API section for the full documentation of the dbuffer API.

14.3 Conventions

As with any memory management API, it is important to know how to handle the creation and releasing of dbuffers, in other words how to manage ownership of the dbuffers. The conventions used in Simics, which are assumed by many interfaces that employ dbuffers, are described in this section.

The conventions are based on two simple rules:

- Ownership of a dbuffer is never passed in a function call
- A dbuffer received in a function call may not be modified

The first rule means that whoever creates the dbuffer is responsible for releasing it. This often means that the dbuffer is created, filled with data, passed to other functions, and released in the same function. For some long-lived dbuffers, a reference may be stored to the dbuffer, and code in the same module will create it and release it.

Furthermore, this means that if a function receives a dbuffer reference as one of its parameter, and wishes to store it for later use, it must make a copy of it and store the copy. Fortunately, this is a cheap operation.

The second rule means that a function that takes a dbuffer reference as one of its parameter must also make a copy of it if it wants to make any changes to it.

Of course, there are exceptions to this rule. Sometimes the owner of a dbuffer will call a function with the explicit intent that the dbuffer should be updated. But these exceptions should always be documented in the interface description.

Let's take a look at a small example of how these rules work in practice. Consider a small example where a network device reads a packet from memory, adds a header to the packet and sends it to a network model that delivers it to a number of receivers. The receivers will strip away the header and store it in a queue. The dequeur is later run from an event handler and writes the incoming packet to local memory.

```
// This is part of the sender model
void sender(void)
{
    dbuffer_t *buf = new_dbuffer();
    long len;
    char *dataref = get_packet_from_mem(&len);
    memcpy(dbuffer_append(buf, len), dataref, len);
    write_header(dbuffer_prepend(buf, HEADERSIZE));
    send_to_network(buf);
    dbuffer_free(buf);
}

// This is part of the network model
void send_to_network(dbuffer_t *buf)
{
    // This function doesn't modify or store the dbuffer.  It
    // sends the same dbuffer to all receivers that match the
    // first six bytes of the header.
    for (i = 0; i < nreceivers; i++) {
        if (memcmp(dbuffer_read(buf, 0, 6), receivers[i]->addr) == 0)
            receive(receivers[i], buf);
    }
}

// This is part of the receiver model
void receive(receiver_t *receiver, *inbuf)
{
    if (receiver->enabled) {
        dbuffer_t *buf = dbuffer_clone(inbuf);
        dbuffer_remove(buf, 0, HEADERSIZE);
        enqueue(receiver->queue, buf);
```

```
    }
}

void dequeue_packet(receiver_t *receiver)
{
    dbuffer_t *buf = dequeue(receiver->queue);
    write_packet_to_mem(dbuffer_read_all(buf), dbuffer_len(buf));
    dbuffer_free(buf);
}
```

Notice how the sender code sends the dbuffer to the network link, and the network links sends the same dbuffer to all receivers. The network code will not try to send separate copies to each receiver, but instead let the receiver clone it if it needs to, since only the receiver code knows whether a copy needs to be made.

Chapter 15

The time server and the time client

When interfacing Simics, timing may in some cases pose a problem. For example, test programs may specify a timeout, but since Simics can run both faster and slower than real time, the test may either timeout when it should not, or not timeout when it actually should.

The time server solves this by exporting the virtual time ("Simics time") to the outer world. Clients can communicate with the time server through TCP/IP, using the time client library that provides a C-interface to the time server.

15.1 Time client library

The client library provides non-threaded C primitives for all features of the time server protocol. Using the time client library, interfacing to the time server is as easy as a C function call.

The following primitives exists:

```
typedef enum {
        Simtime_No_Error,
        Simtime_Socket_Error,         // errno contains last error
        Simtime_Timeout,              // global timeout (as specified in
                                      // simtime_connect)
        Simtime_Receive_Buffer_Full,  // the received message did not fit
                                      // in the buffer. It's probably a bug
                                      // in the time client library if it
                                      // happens
        Simtime_Parse_Error           // received message could not be
                                      // parsed
} simtime_error_t;


simtime_error_t simtime_query_time(simtime_context_t *ctx,
                                   double *time);

simtime_error_t simtime_sleep(simtime_context_t *ctx,
```

```
                                    double seconds,
                                    double *time);

simtime_error_t simtime_periodic_ping(simtime_context_t *ctx,
                                      double interval,
                                      double how_long,
                                      simtime_callback_t cb,
                                      void *user_data);
```

simtime_query_time will return the current virtual time. It is not expected to block very long (the time server will respond immediately when it receives the query).

Although tempting, ***simtime_query_time*** should not be used in a tight polling loop, as it will degrade Simics performance quite noticeable.

Depending on what the timing code looks like, one of the other primitives should be used instead.

The ***simtime_sleep*** function will block for a specified number of virtual seconds. Note that this function will never return if Simics is not simulating, unless a global timeout is specified (see the ***simtime_connect*** function). This is obviously because the virtual time is not progressing when the simulation is not running.

If the timeout is not known beforehand, it may not be possible to use ***simtime_sleep***. In this case, ***simtime_periodic_ping*** might be the solution. It will cause Simics to send periodic pings to the client. The interval between each ping is in real seconds, i.e. host time, not Simics time. It is also possible to specify a duration, also in real seconds, after which Simics will stop sending ping messages. If *duration* is negative or zero, Simics will continue to send ping messages until told to stop (non-zero return value from ping message callback function).

For every ping message, the callback function ***cb*** will be called:

```
typedef int (*simtime_callback_t)(void *data,
                                  simtime_context_t *ctx,
                                  int seq_no, double time);
```

seq_no contains the sequence number of the received ping message and *time* contains the current virtual time.

If the callback function returns a value other than zero, the periodic pings will be aborted, even though *duration* real seconds has not yet passed.

Note that ***simtime_periodic_ping*** will not return until *duration* real seconds has passed, or until the callback function returns a non-zero value. Also note that no simtime API calls should be made from the callback.

Before any of the above primitives can be used, a connection to the time server has to be established:

```
simtime_context_t *simtime_connect(const char *host,
                                   int port,
                                   int global_timeout);
```

```
void simtime_disconnect(simtime_context_t *ctx);
```

The time client will connect to a time server on *host* at port *port*. If *global_timeout* is larger than zero, all calls to simtime will timeout after *global_timeout* real seconds. This is useful to detect, for example, a crashed Simics session. But note that a call to e.g. ***simtime_sleep*** may take very long real time, depending on how fast Simics is simulating (and, of course, how long the sleep time is).

To disconnect from the time server, call the function ***simtime_disconnect***.

If the client is threaded, care must be taken to not make more than one API call at a time. The time client library is not designed to handle simultanious calls.

Index

T

U

W

Virtutech, Inc.

1740 Technology Dr., suite 460
San Jose, CA 95110
USA

Phone +1 408-392-9150
Fax +1 408-608-0430

http://www.virtutech.com