



virtutech

Getting Started with Simics

Simics Version 4.2

Revision 3001
Date 2009-03-28

© 2006–2009 Virtutech AB
Drottningholmsvägen 22, SE-112 42 Stockholm, Sweden

Trademarks

Virtutech, the Virtutech logo, Simics, and Hindsight are trademarks or registered trademarks of Virtutech AB or Virtutech, Inc. in the United States and/or other countries.

The contents herein are Documentation which are a subset of Licensed Software pursuant to the terms of the Virtutech Simics Software License Agreement (the “Agreement”), and are being distributed under the Agreement, and use of this Documentation is subject to the terms the Agreement.

This Publication is provided “as is” without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.

This Publication could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein; these changes will be incorporated in new editions of the Publication. Virtutech may make improvements and/or changes in the product(s) and/or the program(s) described in this Publication at any time.

The proprietary information contained within this Publication must not be disclosed to others without the written consent of Virtutech.

Contents

1	Introduction	5
1.1	Who Should Read This Document?	5
1.2	Conventions	5
2	Virtualized Systems Development	6
2.1	Developing Software with Virtualized Systems Development	8
2.2	Incorporating Virtualized Systems Development	10
2.2.1	Impact on Software Development Tools	10
2.2.2	Impact on the Product Development Process	11
3	Product Definition & Structure	13
3.1	Simics Hindsight	13
3.2	Simics Accelerator	13
3.3	Simics Virtual Board/System	14
3.4	Simics Model Builder	14
3.5	Simics Ethernet Networking	14
3.6	Simics CPU Core Types	15
3.7	Simics Device Model Libraries	15
4	Tutorial	16
4.1	Launch Simulation	16
4.2	Running the Simulation	20
4.3	Checkpointing	21
4.4	Reverse Execution	22
4.5	Getting Files into a Simulated System	24
4.6	Debugging	25
4.7	Tracing	32
4.8	Scripting	35
5	Simics Hindsight User Interface	37
5.1	Overview of Simics Graphical User Interface	37
5.2	Simics Control Window	39
5.2.1	Toolbar	39
5.2.2	File Menu	41
5.2.3	Edit Menu	43
5.2.4	Run Menu	43

5.2.5	Debug Menu	44
5.2.6	Tools Menu	45
5.2.7	Window Menu	46
5.2.8	Help Menu	46
5.3	Console Window	47
5.3.1	Text Console	47
5.3.2	Graphics Console	49
5.4	Command Line Window	50
5.5	CPU Registers Window	52
5.6	Device Registers Window	52
5.7	Memory Contents Window	52
5.8	Disassembly Window	54
5.9	Hap Browser	57
5.10	Help Browser	58
5.11	Object Browser	58
5.12	Configuration Browser	60
5.13	Statistics Plot	62
5.14	Memory Mappings Browser	63
5.15	Preference Window	64
	Appearance Panel	64
	Startup Panel	65
	Advanced Panel	67
5.16	Source View Window	68
5.17	Stack Trace Window	68
6	Next Steps	70
	Index	71

Chapter 1

Introduction

This document provides users new to Virtutech Simics a quick introduction on how to effectively use Simics. It describes the concepts and benefits of Virtualized Systems Development™ when used to develop software, and the products in the Simics family. It also provides a tutorial and links to other Simics documents that will help you to become effective with Virtutech Simics.

1.1 Who Should Read This Document?

This document is intended for new Simics users who intend to use Simics standalone. If you intend to use Simics Eclipse read the Simics Eclipse User Guide instead.

1.2 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 should not be there.
```

The directory where Simics Base package or any add-on package 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.

Chapter 2

Virtualized Systems Development

Virtualized Systems Development™ (VSD) is product development without the use of, or need of, the target hardware platform that the software will eventually run on. For example, if the product's software is being developed for a target system that contains a PowerPC processor, an Ethernet controller, RapidIO, a UART and other peripherals, each developer of the software team would typically need this target hardware configuration in order to debug and test their software (see Figure 2.1). When target hardware is in short supply, or not available, this obviously becomes a bottleneck for the software development team.

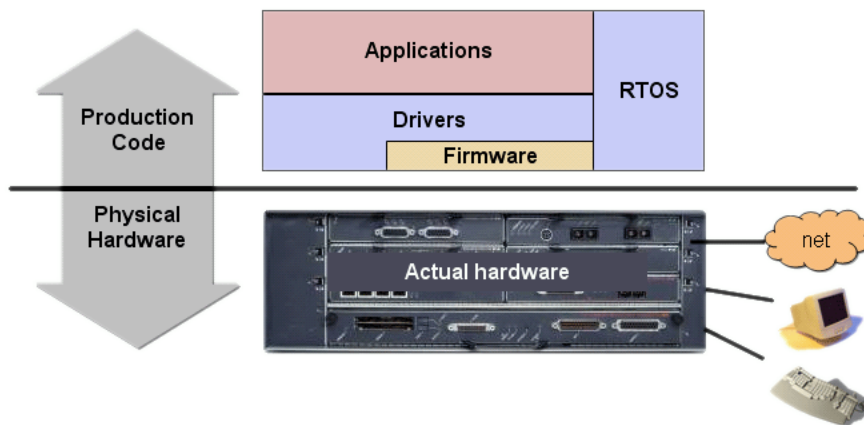


Figure 2.1: Traditional Software Development Environment

With Virtualized Systems Development, the target hardware is virtualized (i.e., simulated) and runs on each developer's development workstation. To the target software, the virtualized target hardware behaves exactly the same as the physical target hardware (see Figure 2.2). Virtualized System Development Platform Target application code, the real-time operating system, drivers, firmware, all can be debugged, tested, and executed using the virtualized target hardware instead of the physical target hardware. Furthermore, a vir-

virtualized systems development environment allows one to run the exact same binary that they would run on the physical target. This means there is no need for RTOS/OS API abstraction layers, stubbed out drivers or firmware, nor multiple build scripts that build the software one way for production environment and another way for a stubbed-out environment.

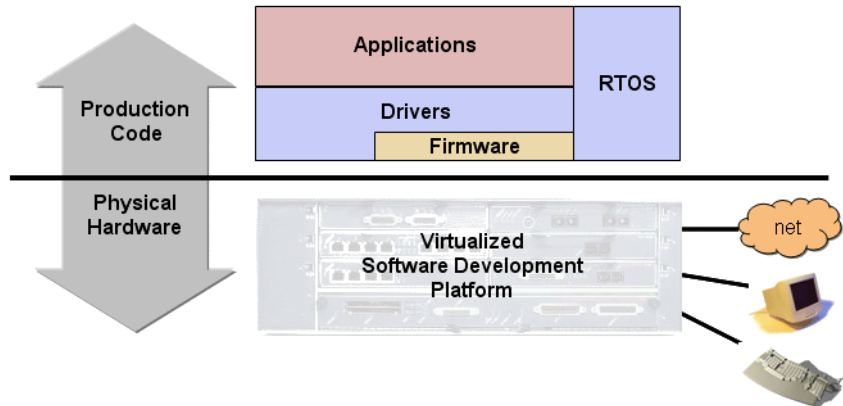


Figure 2.2: Virtualized Systems Development Environment

To accomplish this, a virtualized systems development environment provides the following things:

- an instruction set simulator for the microprocessor(s) in the target hardware;
- behavioral simulation of all devices in the target hardware that the target software interacts with;
- connections between, and among, simulated targets and the real-world (e.g., networks like Ethernet, MIL-STD-1553, ARINC 429, SpaceWire, FireWire, USB, ATM, and other mechanisms like disk images, memory images, etc.); and
- the ability to use the same tools (e.g., compilers, linkers, debuggers, IDEs, and RTOSs) and processes that the software developer would use with the physical hardware.

Furthermore, to be truly useful for software development and testing of the entire target software stack, a virtualized systems development platform must be fast enough to execute the target software. This is what differentiates a virtualized systems development platform from simulation tools provided by the electronic design automation (EDA) industry. These tools, while extremely accurate from a hardware implementation perspective, often are not fast enough or complete enough to run the entire target software stack.

2.1 Developing Software with Virtualized Systems Development

VSD allows software development tasks to occur more efficiently and without need or limitations of the physical target hardware. Initial software development and debug can occur even while the physical target hardware is in development. Furthermore, software testing does not have to be delayed because of a limited quantity of physical hardware targets. The following is a list of the benefits of using VSD for common software development tasks:

Decrease the time it takes for target system bring-up

Because low-level software can be developed, debugged, and tested long before the physical target hardware is available, it can be used to accelerate hardware bring-up once the physical target hardware is available. In addition, instead of debugging both the software and hardware simultaneously during hardware bring-up, the development team can focus on these activities separately and be more efficient with tracking down and fixing problems. As a result bring-up testing is now truly testing and not development.

Move risky development items earlier in the development process

Frequently the most risky project activities (such as system integration and testing) are postponed until late in the development life-cycle when they are the most expensive to fix. By moving these activities earlier in the process, they become less risky and less expensive to fix. Using VSD, one can develop the low-level software and integrate it with the application-level software, and even do system integration and testing.

Reduce the need for multiple hardware prototype iterations

With traditional product development, the hardware team is usually pressured to get working prototypes available as quickly as possible, even if that means that they have to create multiple iterations of the prototypes. In addition, multiple iterations of prototypes are usually necessary in order to evaluate the impact of a hardware change on the software. With Virtualized Systems Development, the rush to produce first prototype hardware is greatly reduced because the software team has virtual hardware to use. In addition, proposed changes to the hardware can be quickly and cheaply tested on the virtual platform, eliminating the need for multiple iterations of hardware prototypes.

Improve the quality of the software

Software quality can be improved in a number of ways using Virtualized Systems Development. First, the target platform is available to all software developers, including the software testers. The team does not have to compete for target hardware resources and has more time to test their software. Second, a virtualized systems development environment allows for system tests to be scripted and automated. In addition, software can be tested in ways that testing on physical hardware often prevents—being able to set devices to particular states, being able to inject faults into a device or the system, being able to record and replay all activity that occurs during the test run including the state of the processor(s), devices, and I/O.

Find software bugs in a fraction of the time

Some software bugs take days or weeks to track down, but only a fraction of the time

2.1. Developing Software with Virtualized Systems Development

to fix. They may be difficult to pinpoint because they are not deterministic (they do not exhibit the same behavior each time) or are sporadic. In addition, it may be difficult to determine if the bug is due to a real software problem or is caused by an underlying hardware problem. Using virtualized systems development platform features such as advanced breakpoints/watchpoints and reverse execution can significantly reduce the time it takes to find these bugs. Furthermore, once the bug is found, a snapshot of the execution leading up to the bug can be taken so that the bug can be easily reproduced, and thus fixed, by another person on the development team.

Improve individual developer's productivity

Virtualized software development can greatly improve the productivity of individual developers and testers. Features such as loading from a saved system state allows developers to eliminate time-consuming boot processes, or to save their debugging progress before leaving work and then being able to start from the same point later. In addition, since developers are not tied to physical target hardware, they can take the virtual target hardware with them, and work efficiently away from the office. Working in parallel with traditional tools such as debuggers, a virtualized software development environment can extend the capability of these tools to do things that they could not do before: reduce the need for having a JTAG interface or target debug agent running on the target, provide for advanced breakpointing capability (such as breaking on a range of addresses or a pattern written to the console), reverse execution capability, etc.

Improve robustness of software

Software needs to handle abnormal hardware and system states by detecting these states and recovering or shutting down safely. With physical hardware, these states can be difficult or impossible to force, thus testing that the software handles these states is practically impossible. With Virtualized Systems Development, abnormal system and hardware states can be created simply (fault injection), and can even be scripted for test automation.

Easily manage complex target environments

Some systems comprise tens or hundreds of other boards or systems all communicating via a network of some sort. Managing these physical target environments requires lab space, technicians, cables, and patience. Managing the virtual target environment equivalents of these complex systems is done through scripting and configuration files. This eliminates the need for lab space, for personnel dedicated to managing the test configurations, and the time needed to set up and change these complex systems.

Advanced debugging support for advanced technologies

With the increasing use of advanced technologies such as multi-core processors, multi-processor systems, multithreaded operating systems, and Systems on Chips (SoC), debugging software is becoming increasingly challenging. Virtualized Systems Development makes debugging these sorts of systems easier. By providing deterministic execution, bugs in these types of systems can be easily reproduced. Furthermore, debugging these problems is simpler because all devices in the system are visible—including internal state and registers. Tracing device activity, even when the device is

2.2. Incorporating Virtualized Systems Development

part of a SoC is feasible. Debugging is no longer limited to monitoring CPU registers and external bus activity.

Improved communication between hardware and software teams

In traditional development environments, problems frequently arise late in the development process when the software and hardware are integrated for the first time. This is often due to the fact that communication, in the form of hardware/software interface specifications, is incomplete or inaccurate. When this happens, the software team develops the wrong software and it does not work on the hardware. Changes to the software (or hardware) to correct these issues are very costly at this point. With VSD, the hardware and software teams communicate earlier in the product development process since the software team can begin development even before hardware is available. Even the process of developing a VSD model can detect errors in the hardware design or hardware specification.

Maintaining legacy systems

Some products need to be maintained years after they are available. Not only does this hardware become less reliable over time, but the costs to maintain space to house it is expensive. In addition, some products have many different configurations which can be difficult to maintain. VSD allows these systems to be maintained virtually. This provides a more reliable development environment and less effort to maintain.

Being able to execute faster than real time

In some cases, VSD enables a system to execute faster than real-time. This allows tests to execute faster than their counterparts on physical hardware.

2.2 Incorporating Virtualized Systems Development

Incorporating Virtualized Systems Development into your development process is as easy as replacing the physical target with the virtualized target, except that you connect to the virtual target through scripts instead of with cables.

2.2.1 Impact on Software Development Tools

Existing tools (compiler, linker, debugger, RTOS, build scripts, test tools) all should work with the virtualized target as they do with the physical target. There is no need to maintain two sets of source code or use different compile/link switches: the target software will run on the virtual target as it does on the physical target.

Even integrated development environments (IDE) and the debuggers that come with them will work with the virtual target. Most debuggers rely on a target debug agent that resides on the target. The same mechanism can be used with a virtual target as shown on figure 2.3.

There are some times when it is not desirable or possible for a debugger to use a target debug agent. Target debug agents are not usually helpful when debugging BSPs (board support packages), drivers, and RTOS code. In addition, sometime the target does not have

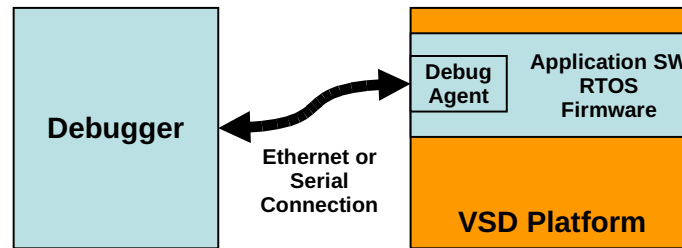


Figure 2.3: Debug Agent on a Virtualized Systems Development Platform

sufficient memory for a target debug agent or the user does not want the application software to be affected by the presence of a debug agent.

During these times, the software developer either relies on other forms of debugging (such as a hardware trace or probe), or uses a special debugger that attaches to a JTAG port on the physical target.

The VSD platform contains an API that allows debuggers to communicate directly with it eliminating the need for a Target Debug Agent and JTAG.

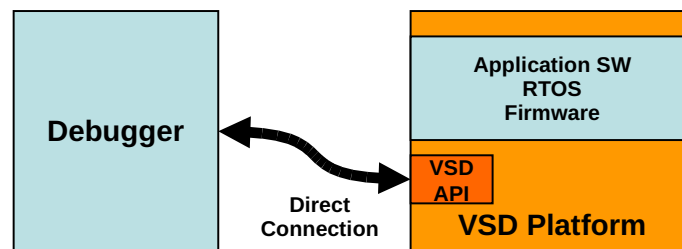


Figure 2.4: Direct Connection on a Virtualized Systems Development Platform

2.2.2 Impact on the Product Development Process

The efficiency of your product development process will improve when Virtualized Systems Development is utilized. There are immediate productivity improvements for the individuals of the software team by having access to unlimited virtual hardware and a superior debug platform.

In addition to this, significant improvements will be realized if the product development process is modified to take full advantage of Virtualized Systems Development. By parallelizing activities that are normally done sequentially, the project schedule can be reduced and risky items addressed earlier in the process where they are cheaper and easier to handle. For example:

2.2. Incorporating Virtualized Systems Development

- Defining and validating the hardware/software interfaces early in the process before the hardware goes to prototype.
- Developing, debugging, and testing low-level software on the virtual target while the hardware prototypes are being built.
- Having low-level software, RTOS, firmware, drivers working before prototype hardware is available so that when it becomes available, only a few days are needed to bring up the prototype hardware.
- Having the high-level application developers use the same set of tools and processes that the low-level software developers use to avoid incompatibilities and inconsistencies that are usually discovered during integration.
- Integrating the high-level software with the low-level software before the entire physical system is available or working.
- Creating and running system integration tests before the entire physical system is available or working.

Significant improvements occur when the entire product development team incorporates VSD into their process. Cost savings occur because multiple iterations of prototype hardware can be reduced or eliminated. Time to market and risk reductions occur because software and hardware can be developed simultaneously. In addition, the ability to try out virtual hardware changes on the software without need for physical hardware prototypes allows for much quicker feedback and supports true iterative development.

Chapter 3

Product Definition & Structure

The Simics product family is made up of multiple products and optional add-on products. This section defines what these products are and how they interact with each other. Please contact Virtutech directly for the licensing terms of each product.

3.1 Simics Hindsight

Simics Hindsight is the main/base Simics product used by software developers. It provides the user interface and software debugging interface to the Simics platform. Simics Hindsight can be used from within Eclipse, with the help of Simics Eclipse, or be used standalone. If you use it standalone you can connect it with other software debuggers such as GDB and the debuggers that come with popular integrated development environments.

Simics Hindsight provides source level debugging, breakpointing, watchpoints, scripting capabilities, and device and system logging. It also provides a unique feature—reverse execution. With this feature, you can actually run your software in reverse, which is extremely useful for finding elusive bugs. Simics Hindsight also includes the ability to save and load checkpoints. Checkpoints contain the complete state of the system, thus when you save a checkpoint you are saving the entire state of the system—including the processors, devices, and all the software. When you load a checkpoint, you begin executing from the exact place that you stopped executing when you saved that checkpoint.

Simics Hindsight attaches to a Simics Virtual Board or Virtual System, which is the actual simulation platform.

3.2 Simics Accelerator

Simics Accelerator provides a unique set of scalability and execution speed capabilities for Simics simulations. With Accelerator, Simics takes advantage of multiprocessor and multi-core hosts to accelerate the simulation of large target systems containing multiple machines. You can also distribute the simulation of a system across multiple machines. Simics Accelerator also contains page sharing technology to exploit the properties of the target system in order to further improve scalability by reducing the memory consumption and reusing decoded and compiled target code. For target configurations containing many boards and

many processor, Simics Accelerator can dramatically enhance the simulation speed and scalability.

The speedup for any particular target system setup will vary with the capacity of the host machine, as well as the properties of the target system hardware and software.

3.3 Simics Virtual Board/System

A Simics Virtual Board or Virtual System is the simulation platform and the model of the physical target hardware that is being simulated. A Simics Virtual Board or Virtual System can be as simple or complex as the physical target system is. It can contain anything from a simple CPU and RAM, to a complex single board computer containing a multi-core processor, or a personal computer complete with graphics console, disk drives, and mouse, and even a complex network of computers and systems. Simics Virtual Systems can connect to other Simics Virtual Systems or to the real world via networks like Ethernet.

A Simics Virtual System contains a “model” of the physical target. This “model” includes one or more high-performance CPU instruction set simulators as well as models of the devices that make up the system, and the connections available for the virtual target.

Within Simics Hindsight, a Simics Virtual Board/System is often referred to as “the target”.

Virtutech provides a number of off-the-shelf Virtual Boards/Systems. However, more commonly, customers build their own virtual boards using Simics Model Builder.

3.4 Simics Model Builder

Simics Model Builder is an optional add-on product that allows users to create, modify, configure, examine, and control the virtual boards and systems that run on the Simics platform. Simics Model Builder provides a compiler for the Device Modeling Language (DML). DML allows for device models to be created quickly and efficiently by modeling just the behavior of the device instead of how the device is physically implemented. Simics Model Builder also provides a convenient way for doing quick prototyping and running what-if scenarios to see how various hardware configurations affect the performance of the software. Simics Model Builder also makes it possible to integrate existing C, C++, and SystemC models into Simics.

3.5 Simics Ethernet Networking

Simics Ethernet Networking is an optional add-on product that provides Ethernet networking support for applications running on the Simics platform. Simics Ethernet Networking is needed when one or more virtual systems are connected to other virtual systems or to a real, physical Ethernet network.

3.6 Simics CPU Core Types

Simics CPU Core Types are instruction set simulators for particular microprocessors. CPU Core Types vary not only on CPU architecture (e.g., PowerPC, POWER, MIPS, Intel/AMD x86, SPARC, ARM, etc.) but also specific members of those architectures, such as the Freescale P4080.

If a user purchases a Virtual System from Virtutech, they usually do not specify a CPU Core Type as it is included with the Virtual System. However, if the user wants the flexibility to create their own virtual systems, then they typically purchase a license for one or more CPU Core Types.

A Simics Virtual System can include one or more of these CPU Core types.

3.7 Simics Device Model Libraries

Simics Device libraries are models of commercial-off-the-shelf (COTS) devices that Virtutech and its partners have created. These device models can be for devices such as interrupt controllers, Ethernet controllers, bridge chips, systems on a chip (SoC), etc. Typically, customers do not purchase these individually as these are included as part of a Virtual System. However, if a customer is designing their own virtual system, then they may purchase these device model libraries. Such libraries are generally not CPU core specific and can be used in any model.

Chapter 4

Tutorial

This tutorial is a step-by-step guide describing how to perform some common tasks with Simics Hindsight. The guide is based on a target computer built by AMCC and known as *Ebony*. *Ebony* is a simulated PowerPC 440GP board running a very small MontaVista™ Linux™ installation.

In order to follow this tutorial, you need to install the Firststeps add-on package, which contains the *Ebony* board. A detailed explanation is provided in the *Simics Installation Guide*.

4.1 Launch Simulation

This section will show you how to start Simics, create a workspace and launch a simulation session.

Let us start Simics:

- On Windows, you launch Simics via the Start menu, in the Virtutech Folder.
- On Unix, run the script `[simics]/bin/simics-gui` to start the graphical user interface (GUI), where `[simics]` refers to Simics's installation directory.

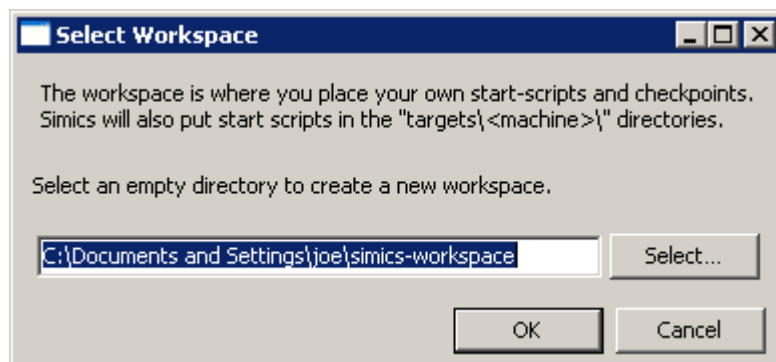


Figure 4.1: Create a Workspace

Since you start Simics for the first time, a dialog box will pop up (figure 4.1), asking you where to create a *workspace*. As a Simics installation can be shared among many users, it should preferably be kept read-only. *Workspaces* are personal areas where Simics will keep your own settings and where you can store all your working files. So pick a directory and let Simics make a workspace out of it.

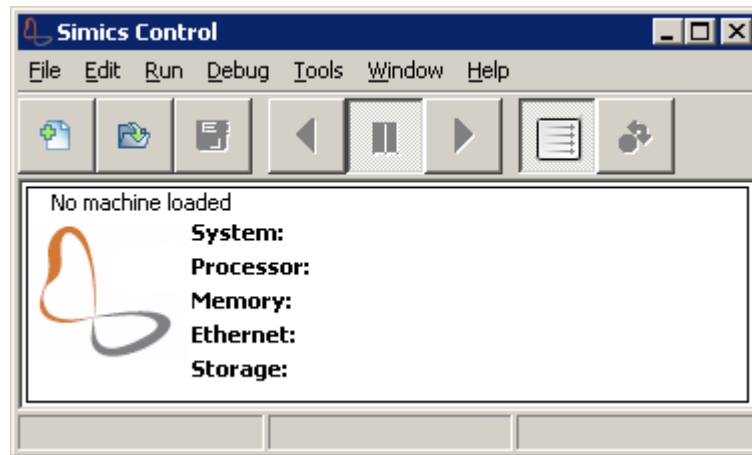



Figure 4.2: Simics Control Window

Once the workspace is created Simics will start. If this is the first time you run Simics, you will be asked to register to be able to participate on the Simics support forum. Then the *Simics Control* window opens (figure 4.2). The *Simics Control* window contains the icon toolbar and all the menus that control Simics, as well as a summary of the current simulation session. From its menus you can open additional windows. For this tutorial you need the *Simics Command Line*. Select **Tools** → **Command Line Window** to open it.

The *Command Line* window allows you to interact with Simics. Simics will print there warnings and error messages, and you as a user can enter commands to inspect and modify the simulation. Anything that can be done via the menus of the *Simics Control* window can also be done in CLI. Several of the features demonstrated in this guide makes use of CLI, as you will see later on.

Create a new session with the *Ebony firststeps* target machine by clicking  **New session from script**. Open the `ebony` directory and choose the file `ebony-linux-firststeps.simics`. After a few seconds, the *Simics Control* window should show you a summary of the simulated system you have just loaded, and a new window should have popped up, called *Serial Console on system_cmp0.soc.uart0* (see figure 4.4).

This new window is part of the simulation: it is connected to the serial port of the simulated *Ebony* board. All output from the simulated machine will be displayed here. It is also here that you can interact with the simulated software, as opposed to the Simics CLI window, which is used to control the simulation itself.



Figure 4.3: Simics Command Line Window (CLI)

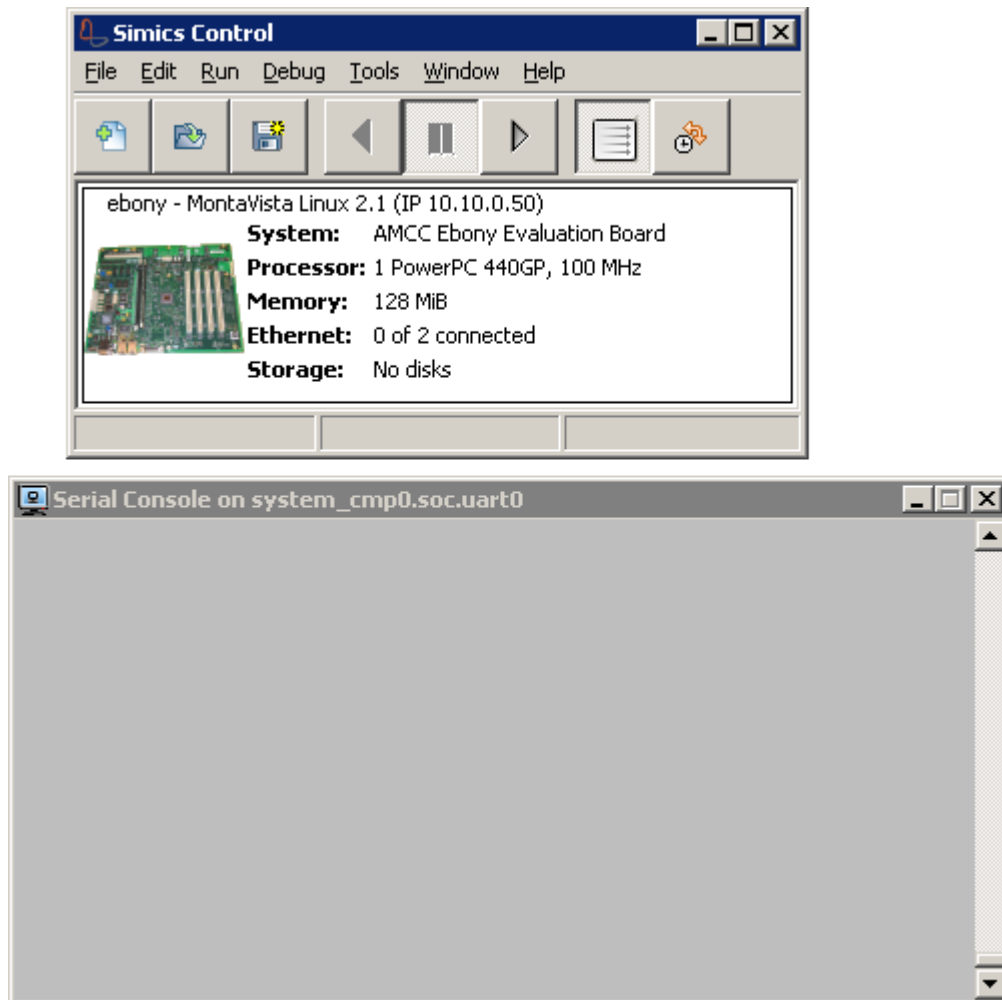


Figure 4.4: Ebony target system is loaded

4.2 Running the Simulation

The simulation does not start automatically. Simics is waiting for you to command it, either via the toolbar, or through commands entered at the Simics prompt (in the CLI window). You can start the simulation by clicking on **▶ Run Forward**, or by entering the command **continue** or **c** at the prompt.

A short side note: in all Simics documentation, interaction with Simics in the CLI window (figure 4.3) is presented in monospace font. User input will be presented in **bold font**.

```
simics> this is something that you should type
This is output from Simics
```

So in our example:

```
[...]
simics> continue
... the simulation is running ...
running> stop
[cpu0] v:0x07fd395c p:0x007fd395c  xor r3, r11, r9
... the simulation is now stopped ...
simics>
```

If you let the simulation run for a while, you will see Ebony printing out the boot process messages in the serial console. The simulation can be stopped at any time by entering the **stop** command or clicking on **■ Stop** in the toolbar. When stopping, the simulation will finish the current executing instruction to leave everything in a consistent state.

After some time, MontaVista Linux will be up and running and you will obtain a command line prompt in the serial console.

You can now try typing a few commands on the prompt:

```
root@firststeps: ~# pwd
/root
root@firststeps: ~# ls /
LICENSE      dev          host         lost+found  proc        tmp
bin          etc          lib          mnt         root        usr
boot         home        linuxrc     opt         sbin       var
root@firststeps: ~#
```

Note: If the serial console does not respond, make sure that the simulation is actually running, for example by clicking **▶ Run Forward**.

```

Serial Console on system_cmp0.soc.uart0
Mounting /tmp: done.
Starting syslogd: done.
Starting klogd: done.
Starting inetd: done.
Starting thttpd: eth0: Link is Up
eth0: Speed: 100, Full duplex.
eth1: Link is Up
eth1: Speed: 100, Full duplex.

MontaVista Linux 2.1, Preview Kit

(none) login: root

Welcome to MontaVista Linux 2.1, Professional Edition

BusyBox v0.60.2 (2002.08.28-16:54+0000) Built-in shell (ash)
Enter 'help' for a list of built-in commands.

# eth0: Speed: 100, Full duplex.
ifconfig eth0 10.10.0.50 netmask 255.255.255.0
# █

```

Figure 4.5: A booted MontaVista Linux

4.3 Checkpointing

In order to avoid booting Firststeps every time we need it, we use the facility known as *configuration checkpointing* or simply *checkpointing*. This enables Simics to save the entire state of the simulation to disk. The files are saved in a portable format and can be loaded at a later time.

To write a checkpoint:

1. Stop the simulation, with **Stop** or with a command:

```

running> stop
simics>

```

2. Click on **Save Checkpoint**. Press the **Up** until you reach the workspace, name the checkpoint `after_boot.ckpt` and press **Save**. The complete state is now saved.

Note: The checkpoint saved is actually a directory containing multiple files.

Let us try our newly saved checkpoint:

1. Select **File** → **New Empty Session** and confirm that this is indeed what you want to do.
2. Click on **Open Checkpoint** and open the `after_boot.ckpt` file. After a few seconds, the entire simulation is restored, including the serial console contents. Remember to resume the simulation (by entering **continue** or clicking on **Run Forward**) before typing more commands at the console.

You can read more about configuration and checkpointing in *Advanced Features of Simics*.


4.4 Reverse Execution

This section will introduce a special feature of Simics: the ability to run the simulation forward and backward in time, also called *reverse execution*.

The simulation speed of Simics varies a lot depending on what software that executes. In this example, the operating system is mostly idle, causing the simulation to become extremely fast. To avoid virtual time progressing too rapidly we first activate the real-time mode:


```
simics> enable-real-time-mode
simics>
```

This will cause the virtual time to never progress **faster** than the real time and also reduces how much host processor Simics use. See **help enable-real-time-mode** for further information on this. You can read more about performance in *Simulation Performance* chapter in *Advanced Features of Simics*.

To enable reverse execution, you just need to click on  **Enable/Disable Reverse Execution**. In reality, what this button does is to create a *time bookmark* called “start”, which in some ways is similar to a checkpoint. The reverse execution engine keeps such bookmarks and some additional information to jump from one bookmark to another and simulate backward in time.

In CLI, reverse execution is enabled by creating a time bookmark with the **set-bookmark** command:

```
simics> set-bookmark start
simics> c
```

By clicking on  **Enable/Disable Reverse Execution** again, you will disable reverse execution. In CLI, this would be done by running the **delete-bookmark -all** command.

To demonstrate the possibilities reverse execution gives, we will accidentally remove an important file on the simulated system. Enter the following commands in the simulated system’s serial console:

```
root@firststeps: ~# rm /bin/ls
root@firststeps: ~# ls /
ls: No such file or directory
```

The program **ls** has been removed. You can no longer print out the contents of a directory. Let us use reverse execution to recover **ls**.

To prove to ourselves that reversing actually works, let us take a look at how much virtual time has passed so far. For this purpose, we can use the **ptime** command:

```
simics> stop
simics> ptime
```

```
processor                steps          cycles  time [s]
system_cmp0.soc.cpu    16667617210  16667617210  166.676
simics>
```

In the example above, about 167 virtual seconds have passed.

Now we will jump back to the bookmark we just created, using the command **skip-to**. **skip-to** will not run the simulation backwards, but will jump directly to the bookmark.

```
simics> skip-to bookmark = start
[cpu0] v:0xc0003d24 p:0x000003d24 beq- cr7,0xc0003d0c
simics> ptime
processor                steps          cycles  time [s]
system_cmp0.soc.cpu    13586370338  13586370338  135.864
simics>
```

As shown above, we have now simulated about 30 seconds back in the past.

Another way is to actually run the simulation backwards. You can click on **Run Reverse** and you will quickly find yourself back to the moment where you enabled reverse execution (i.e., the first time bookmark). This will be slower than **skip-to**, but on its way back toward the past, it will do all sort of useful things like triggering breakpoints, and this will prove very useful when debugging.

In any case, the system is now in the state it was before the file was erased. Lets run forward again.

```
simics> c
```

When you type something in the terminal, you will notice that it does not respond any longer! But you will see the same commands as before being replayed. This behavior is intentional, and keeps the deterministic property of the simulation, which is invaluable when debugging. Keystrokes, network traffic and any other input is replayed until the last known time is reached. In our example, this is not what we want. To erase all knowledge about the future, run the **clear-recorder** command.

```
simics> stop
simics> skip-to bookmark = start
simics> clear-recorder
simics> c
```

Resume the simulation and enter the following command:

```
root@firststeps: ~# ls /
LICENSE      dev          host         lost+found  proc        tmp
bin          etc          lib          mnt         root        usr
boot        home        linuxrc     opt         sbin        var
```

```
root@firststeps: ~#
```

The `ls` command is back! You can read more about reverse execution in *Using Simics for Software Development*.

4.5 Getting Files into a Simulated System

Simics is a full system simulator, meaning that everything in the target system is simulated, including the disks the software is installed on. It is often necessary to transfer files from the real into the simulated world, i.e., from the host to the target. Simics provides a way to directly access the host from the simulated filesystem, called *SimicsFS*.

SimicsFS is a kernel filesystem module (available for Linux and Solaris) that talks to a simulated pseudo-device. Our *Ebony* machine is equipped with *SimicsFS* support; just `mount /host` to access it.

```
root@firststeps: ~# mount /host
[simicsfs] mounted
root@firststeps: ~# ls /host
AUTOEXEC.BAT                CONFIG.SYS
Documents and Settings      IO.SYS
ntldr                       MSDOS.SYS
pagefile.sys                NTDETECT.COM
Program Files               RECYCLER
System Volume Information   WINDOWS
[...]
```

The exact output of the last command depends on your host system. The output in the example above is from a Windows machine.

In the next section we will explore some of the available debugging features. For that purpose we need to transfer the program we are to debug to our *Ebony* target machine. The file is located in the Firststeps add-on package.

```
root@firststeps: ~# cd /host/Program\ Files/Virtutech/Simics\ 4.2/
Firststeps\ 4.2.0/targets/ebony/images
root@firststeps: ~# ls
busybox-0.60.2.uimage      linux-2.4.31-fp.uimage  u-boot-1.1.2
debug_example              linux-2.4.31.uimage
root@firststeps: ~# cp debug_example ~
root@firststeps: ~# cd
root@firststeps: ~# ls
debug_example
root@firststeps: ~#
```

Note: The example above is done on a Windows host and corresponds to the standard installation directory for the Firststeps add-on package. On Unix, you would try to look in `/host/opt/virtutech/simics-4.2/simics-firststeps-4.2.0/targets/ebony/images/`

The file `debug_example` is now in the target system. At this point, you probably want to stop SimicsFS and save a checkpoint.

```
root@firststeps: ~# umount /host
```

You can read more about SimicsFS and other ways to transfer files in *Using Simics for Software Development* and *Advanced Features of Simics*.

4.6 Debugging

This section demonstrates some source-level debugging facilities that Simics provides. The Firststeps add-on package contains an example code snippet called `debug_example.c`, located in the `targets,ebony` directory. We recommend opening the file `debug_example.c` in an editor of your choice to follow the debugging example more easily. This file contains the code that we are going to debug. The program is supposed to print some information about the users on a system.

In the previous section we copied the executable into the simulated system by using SimicsFS. Now, run that program in the serial console:

```
root@firststeps: ~# ./debug_example
[...]
Got segmentation fault!
root@firststeps: ~#
```

This output indicates that our program crashed. Let us use Simics features to debug it. Simics needs to get some debugging information to map addresses to source code lines and variables. This information is stored in the executable. The `symtable` module in Simics handles symbolic debugging:

```
simics> new-symtable file = "C:\\Program Files\\Virtutech\\Simics 4.2\\Firststeps 4.2.0\\targets\\ebony\\images\\debug_example"
Created symbol table 'debug_example'
ABI for debug_example is ppc-elf-32
debug_example set for context system_cmp.cell0_context
simics>
```

To help debugging your programs, Simics can recognize *magic instructions*, that is, instructions that have no side-effects on a real machine, but can be programmed to do things when run inside Simics, such as stopping the simulation. The `debug_example` code con-

tains such an instruction at the beginning of the *main()* function. Type the following to make magic instructions stop the simulation:

```
simics> enable-magic-breakpoint
simics> c
```

Now, make sure that reverse execution is enabled and re-run `debug_example`:

```
root@firststeps: ~# ./debug_example
```

Simics will stop as soon as the magic instruction is executed. The CLI window should display the message *magic breakpoint*. Click on **Debug** → **Disassembly** to open up a window with the instructions the processor is currently running. The magic breakpoint instruction should be marked.

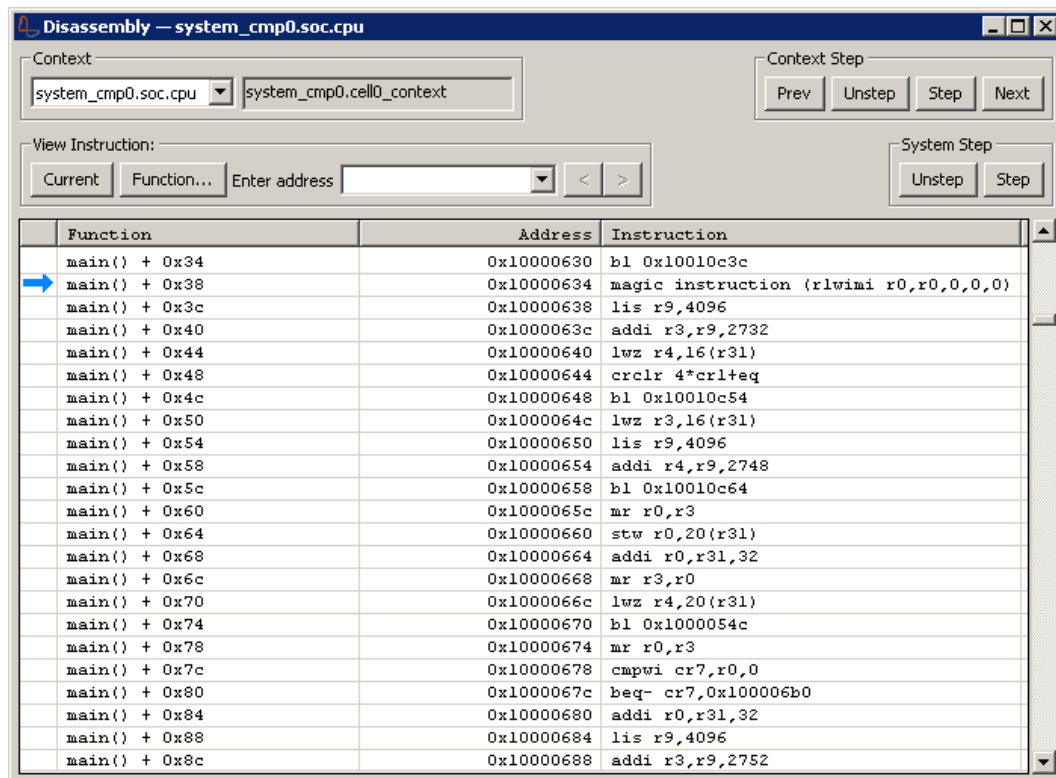


Figure 4.6: Disassembly Window

You can also view the current source code by clicking **Debug** → **Source View**. Since the `debug_example` binary was compiled in a directory that either does not exist at all on your host system, or that does not contain the source code `debug_example.c`, Simics will not be able to find the source code without a little help: select the source file by clicking **Find...** and browse to `debug_example.c`. You can also use a more dynamic approach

where part of the path in the debug information encoded in the binary is replaced with a path you specify:

```
simics> debug_example.source-path "/tmp>C:\\Program Files\\Virtutech\\Simics 4.2\\Firststeps 4.2.0\\targets\\ebony"
```

The command above basically says: whenever the path `/tmp` is found in the debug information, replace it with `C:\\Program Files\\Virtutech\\Simics 4.2\\Firststeps 4.2.0\\targets\\ebony`. You can get more information about the `<syntable>.source-path` command using `help`:

```
simics> help syntable.source-path
```

Note: All this source code gymnastic is due to the fact that `debug_example` was not compiled on your machine. When you compile your own programs, everything will work right out of the box.

Now that we have pointed Simics to the source code, the *Source Code* window should contain the source code of `debug_example`. The current line should be line 73, `MAGIC_BREAKPOINT`, indicated by a small arrow next to the line number (figure 4.7).

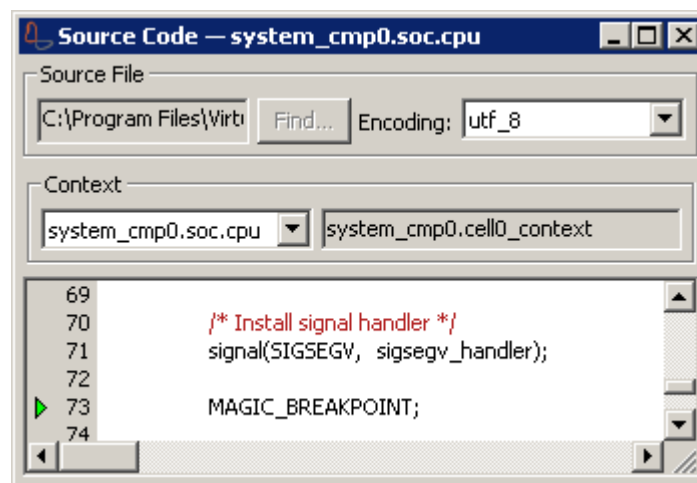


Figure 4.7: Source Code View

Let us find the cause of the segmentation fault. Place a breakpoint on the `sigsegv_handler()` function. The `sigsegv_handler()` function is called when the program receives a segmentation fault and will allow the program to exit gracefully.

```
simics> break (sym sigsegv_handler)
Breakpoint 3 set on address 0x10000520 with access mode 'x'
3
```

```
simics>
```

Resume the simulation. It will stop at the signal handler, and by running the **stack-trace** command, you can see the chain of function calls leading up to this point. This list gives you useful hints about where the crash occurred.

```
simics> c
Code breakpoint 1 reached.
simics> stack-trace
#0 0x10000520 in sigsegv_handler (sig=0)
    at /tmp/debug_example.c:35
#1 0x7ffff398 in ?? ()
#2 0xff06a44 in ?? ()
#3 0xff0ff60 in ?? ()
#4 0x100006ac in main (argc=1, argv=0x7ffffdf4)
    at /tmp/debug_example.c:82
#5 0xfed5fdc in ?? ()
#6 0x0 in ?? ()
simics>
```

The stack-trace is also accessible via **Debug** → **Stack trace** (figure 4.8).

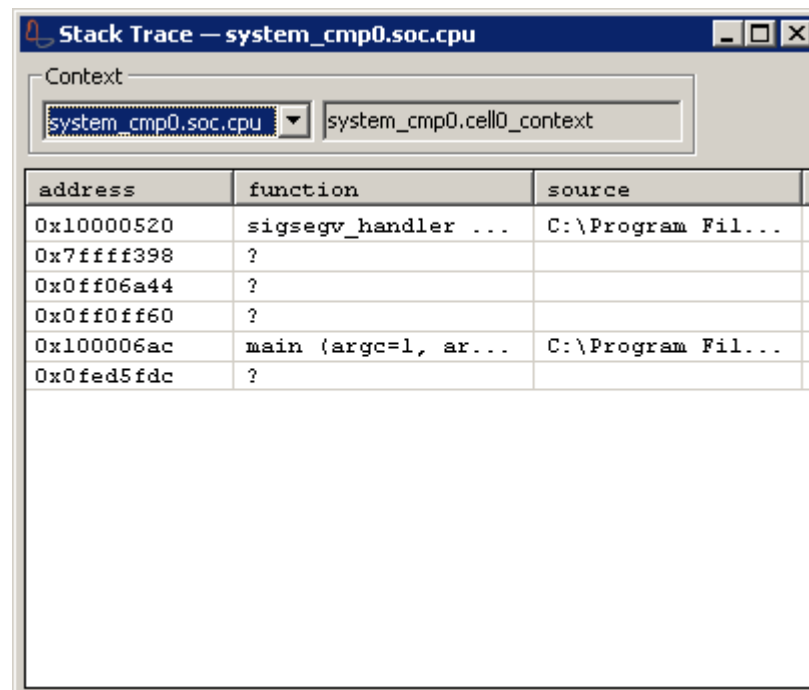


Figure 4.8: Stack trace when the segmentation fault was caught

Simics prints a ? when no symbol could be found for a given address. This can either be a bogus address or a function inside the standard library, for which no symbols have been loaded. The *Source Code* window has been updated and the current line now points to the *sigsegv_handler()* function (figure 4.9).

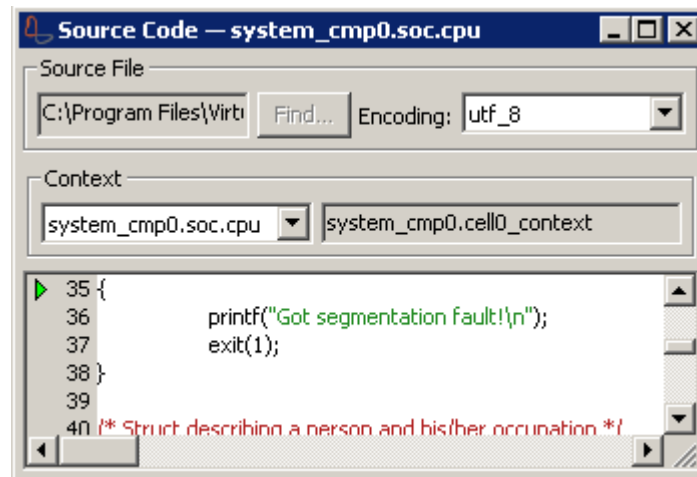


Figure 4.9: Source view when the segmentation fault was caught

A few frames down you will find the *main()* function, which caused the crash. Let us run the simulation backward into that function. The command **reverse-step-line** will run backwards until the previous known source line is reached.

```
simics> reverse-step-line
[system_cmp0.soc.cpu] v:0x100006a8 p:0x00737c6a8 b1 0x10010c54
main (argc=1, argv=0x7ffffe44)
    at C:\Program Files\Virtutech\[\...\]targets\ebony\debug_example.c:82
82         printf("Type: %s\n", user.type);
```

This line caused the crash (see also figure 4.10). Let us examine what *user.type* contains:

```
simics> psym user.type
(char *) 0xa94 (unreadable)
simics> psym user
{name = 0x7ffffdd0 "shutdown", type = (char *) 0xa94 (unreadable)}
simics>
```

As you can see, the *type* member points to an unreadable address, which caused the crash. Where does that value come from? What we want to find is where the last write to this pointer occurred. Let us set a write-access breakpoint on the memory occupied by the pointer and then run backward (using **reverse**) until the breakpoint is reached:

```
simics> break -w (sym "&user.type") (sym "sizeof user.type")
```

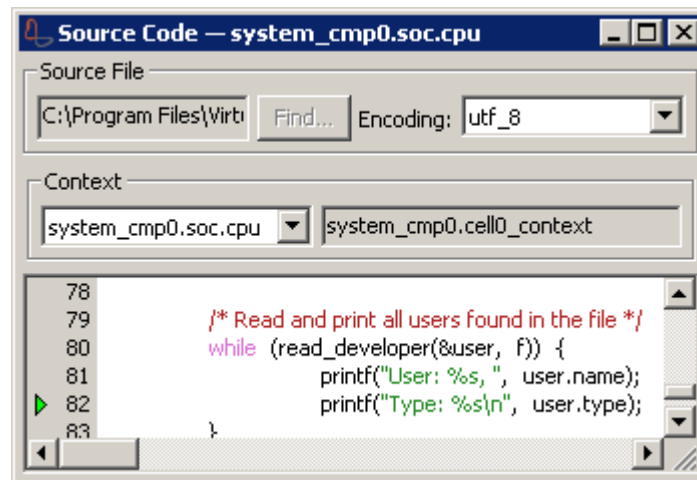


Figure 4.10: Line in source code causing the crash

```

Breakpoint 4 set on address 0x7ffffdd8, length 4 with access mode 'w'
4
simics> reverse
Breakpoint on write to address 0x7ffffdd8 in cell0_context.
Completing instruction @ 0xff365e0 on system_cmp0.soc.cpu.
simics>

```

Now, examine the stack trace:

```

simics> stack-trace
#0 0xff365e4 in ?? ()
#1 0x100005d8 in read_developer (p=0x7ffffdc0, f=0x10010ca0)
    at /tmp/debug_example.c:60
#2 0x10000674 in main (argc=1, argv=0x7ffffe44)
    at /tmp/debug_example.c:80
#3 0xfed5fdc in ?? ()
#4 0x0 in ?? ()
simics>

```

A call from `read_developer()` at line 60, has caused the pointer to be corrupted. Switch to that frame and display the code being run.

```

simics> frame 1
#1 0x100005d8 in read_developer (p=0x7ffffdd0, f=0x10010ca0)
    at /tmp/debug_example.c:60
simics> list read_developer 15
48 {
49     char line[100], *colon;

```

```

50
51     if (fgets(line, 100, f) == NULL)
52         return 0;          /* end of file */
53
54     /* Type is always developer */
55     p->type = "developer";
56
57     /* Everything until the first colon is the name */
58     colon = strchr(line, ':');
59     *colon = '\0';
60     strcpy(p->name, line);
61     return 1;
62 }
simics>

```

We can also use the *Stack Trace* window to select the frame; doing so will update the *Source Code* window as well (figure 4.11).

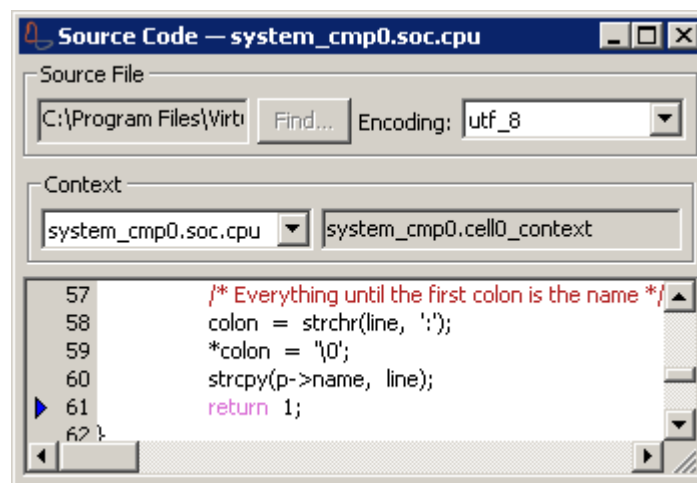


Figure 4.11: Call to *strcpy()*

On line 60, while the *name* field was filled in using *strcpy()*, our failing pointer was accidentally overwritten (remember that the breakpoint was placed on the *type* member). If you issue the command **psym line**, Simics will print out the string copied: "shutdown". A look into the declaration of `struct person` shows that the *name* field is only 8 bytes long, and hence has no space for the trailing null byte of *shutdown*. Check the contents of *p* after and before the actual write to verify it is overwritten:

```

simics> psym "*p"
{name = 0x7ffffdd0 "shutdown", type = (char *) 0xa94 (unreadable)}
simics> reverse-step-instruction
Completing instruction @ 0xff365e0 on system_cmp0.soc.cpu.

```

```

Breakpoint on write to address 0x7ffffdd8 in cell0_context.
simics> frame 1; psym "*p"
#1 0x100005d8 in read_developer (p=0x7ffffdd0, f=0x10010ca0)
    at /tmp/debug_example.c:60
{name = 0x7ffffdd0 "shutdown\020", type = (char *) 0x10000a94 "developer"}
simics>

```

To clean up after our debug session, we must remove the breakpoints that we have set. They are identified with a number and can be shown using **list-breakpoints**.

```

simics> delete 1
simics> delete 4
simics> disable-magic-breakpoint
simics>

```

You can read more about debugging in *Using Simics for Software Development*.

4.7 Tracing

Tracing is a way to observe what is going on during the simulation. This section describes how to trace memory accesses, I/O accesses, control register writes, and exceptions in Simics. The tracing facility provided by the **trace** module will display all memory accesses, both instruction fetches and data accesses.

Launch the `ebony-linux-firststeps.simics` configuration, but do not boot it yet. Start by creating a tracer:

```

simics> new-tracer
Trace object 'trace0' created. Enable tracing with 'trace0.start'.
simics>

```

We are going to trace a few instructions executed when booting *Ebony*. We execute 300 instructions without tracing first to reach a sequence of instructions that includes memory accesses:

```

simics> c 300
[system_cmp0.soc.cpu] v:0xfffff160 p:0x1fffff160 tlbwe r1,r4,1
simics> trace0.start
Tracing enabled. Writing text output to standard output.

simics> c 6
inst: [      1] CPU  0 <v:0xfffff160> <p:0x1fffff160> 7c240fa4 tlbwe r1,r4,1
inst: [      2] CPU  0 <v:0xfffff164> <p:0x1fffff164> 7c4417a4 tlbwe r2,r4,2
inst: [      3] CPU  0 <v:0xfffff168> <p:0x1fffff168> 38840001 addi r4,r4,1
inst: [      4] CPU  0 <v:0xfffff16c> <p:0x1fffff16c> 4200ffdc bdnz+ 0xfffff148

```

```

inst: [      5] CPU  0 <v:0xfffff148> <p:0x1fffff148> 84050004 lwzu r0,4(r5)
data: [      1] CPU  0 <v:0xfffff1b8> <p:0x1fffff1b8> Read  1 bytes  0xc0
data: [      2] CPU  0 <v:0xfffff1b9> <p:0x1fffff1b9> Read  1 bytes  0x0
data: [      3] CPU  0 <v:0xfffff1ba> <p:0x1fffff1ba> Read  1 bytes  0x12
data: [      4] CPU  0 <v:0xfffff1bb> <p:0x1fffff1bb> Read  1 bytes  0x10
inst: [      6] CPU  0 <v:0xfffff14c> <p:0x1fffff14c> 2c000000 cmpwi r0,0
[system_cmp0.soc.cpu] v:0xfffff150 p:0x1fffff150 beq- 0xfffff170
simics>

```

- Lines beginning with `inst:` are executed instructions. Each line contains the address (both virtual and physical) and the instruction itself, in both hexadecimal form and mnemonic.
- Lines beginning with `data:` indicate that some instructions are performing memory operations. Each line contains the operation address (again, both virtual and physical), the type of operation (read or write), the size and the value.

Note: In the trace you can see one four-byte memory read that is split into four single-byte reads. This reflects how Simics models the accesses to the flash memory accessed at this point.

It is also possible to only trace accesses to a certain device. This is done with the `trace-io` command. In this example we are looking at the interaction with the UART device.

```

simics> trace0.stop
Tracing disabled
simics> trace-io system_cmp0.soc.uart0_dev
simics> c 30_000
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000203 SZ 1 0x80 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000200 SZ 1 0x48 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000201 SZ 1 0x0 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000203 SZ 1 0x3 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000202 SZ 1 0x0 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000204 SZ 1 0x0 (BE)
[system_cmp0.soc.uart0_dev trace-io] Read from system_cmp0.soc.cpu: PA 0x140000205 SZ 1 0x60 (BE)
[system_cmp0.soc.uart0_dev trace-io] Read from system_cmp0.soc.cpu: PA 0x140000200 SZ 1 0x0 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000207 SZ 1 0x0 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000201 SZ 1 0x0 (BE)
[system_cmp0.soc.uart0_dev trace-io] Read from system_cmp0.soc.cpu: PA 0x140000205 SZ 1 0x60 (BE)
[system_cmp0.soc.uart0_dev trace-io] Write from system_cmp0.soc.cpu: PA 0x140000200 SZ 1 0xd (BE)
[system_cmp0.soc.uart0_dev trace-io] Read from system_cmp0.soc.cpu: PA 0x140000205 SZ 1 0x0 (BE)
[system_cmp0.soc.cpu] v:0xfff8a634 p:0x1fff8a634 bl 0xfff8a608
simics>

```

Note: You can use underscores anywhere in numbers to make them more readable. The underscores have no meaning and are ignored when the number is read.

trace-cr turns on tracing of changes in the processor's control registers.

```
simics> untrace-io system_cmp0.soc.uart0_dev
simics> trace-cr -all
simics> continue 6_500_000
[system_cmp0.soc.cpu trace-cr] tcr <- 0x0
[system_cmp0.soc.cpu trace-cr] dec <- 0x0
[system_cmp0.soc.cpu trace-cr] decar <- 0x0
[system_cmp0.soc.cpu trace-cr] tsr <- 0x8000000
[system_cmp0.soc.cpu trace-cr] decar <- 0x51615
[system_cmp0.soc.cpu trace-cr] dec <- 0x51615
[system_cmp0.soc.cpu trace-cr] tcr <- 0x4400000
[system_cmp0.soc.cpu trace-cr] ivpr <- 0x0
[system_cmp0.soc.cpu trace-cr] msr <- 0x29000
[system_cmp0.soc.cpu] v:0x07fd8638 p:0x007fd8638 subfc r4,r4,r7
simics>
```

We can single-step with the **-r** flag, to see what registers each instruction changes:

```
simics> untrace-cr -all
simics> step-instruction -r 10
      r4 <- 1364
[system_cmp0.soc.cpu] v:0x07fd863c p:0x007fd863c subfe. r3,r3,r6
[system_cmp0.soc.cpu] v:0x07fd8640 p:0x007fd8640 bge+ 0x7fd8634
[system_cmp0.soc.cpu] v:0x07fd8634 p:0x007fd8634 bl 0x7fd8608
[system_cmp0.soc.cpu] v:0x07fd8608 p:0x007fd8608 mftbu r3
[system_cmp0.soc.cpu] v:0x07fd860c p:0x007fd860c mftbl r4
      r4 <- 6529681
[system_cmp0.soc.cpu] v:0x07fd8610 p:0x007fd8610 mftbu r5
[system_cmp0.soc.cpu] v:0x07fd8614 p:0x007fd8614 cmpw r3,r5
[system_cmp0.soc.cpu] v:0x07fd8618 p:0x007fd8618 bne+ 0x7fd8608
[system_cmp0.soc.cpu] v:0x07fd861c p:0x007fd861c blr
[system_cmp0.soc.cpu] v:0x07fd8638 p:0x007fd8638 subfc r4,r4,r7
simics>
```

Output from the trace commands can be controlled with the **log-setup** command. For example each log message can be prepended with a time-stamp, indicating the processor, program counter and the step count when the event occurred.

```
simics> log-setup -time-stamp
simics>
```

Simics can also monitor exceptions. Here we will trace all system calls with a timestamp:

```
simics> trace-exception System_call
simics> c
[system_cmp0.soc.cpu trace-exception] {system_cmp0.soc.cpu 0xc0004c10 203963444} Exception 8: System_call
[system_cmp0.soc.cpu trace-exception] {system_cmp0.soc.cpu 0xc0004c10 205608843} Exception 8: System_call
[system_cmp0.soc.cpu trace-exception] {system_cmp0.soc.cpu 0xc0004c10 205617361} Exception 8: System_call
[...]
running> stop
simics> untrace-exception -all
simics>
```

Note: There are variants of **trace-io**, **trace-cr** and **trace-exception** that will stop the simulation when the respective event occurs. These commands begin with **break-**.

4.8 Scripting

The Simics command line has some built-in scripting capabilities. When that is not enough, Python can be used instead. Restart Simics with `ebony-linux-firststeps.simics` to follow the examples in this section. We will use a **trace** object as our example:

```
simics> new-tracer
Trace object 'trace0' created. Enable tracing with 'trace0.start'.
simics>
```

There are two commands available for this object: `(trace).start` and `(trace).stop`. For example, to start tracing:

```
simics> trace0.start
Tracing enabled. Writing text output to standard output.
simics>
```

It is also possible to access an object's *attributes* using CLI. The state of an object is contained in its attributes:

```
simics> trace0->classname
"base-trace-mem-hier"
simics> trace0->enabled
1
simics>
```

Variables in CLI are prefixed with \$, and can hold a string, a number, or an object reference. In the following example the variable `my_tracer` references our `trace0` object (i.e., it is *not* a copy of the trace object).

```
simics> $my_tracer = trace0
simics> $my_tracer->enabled
1
simics>
```

It is also possible to access the tracer from Python. All lines beginning with a @ are evaluated as a Python statement.

```
simics> @trace_obj = SIM_get_object("trace0")
simics> @trace_obj
<the base-trace-mem-hier 'trace0'>
simics> @trace_obj.enabled
1
simics>
```

The Simics API is directly accessible from Python. The script below counts the number of instructions that are executed until the register `msr` is modified. It imitates the functionality of `break-cr msr`.

```
simics> @start_cycle = SIM_cycle_count(conf.system_cmp0.soc.cpu)
simics> @msr = conf.system_cmp0.soc.cpu.msr
simics> @while conf.system_cmp0.soc.cpu.msr == msr: SIM_continue(1)
[...]
simics> @end_cycle = SIM_cycle_count(conf.system_cmp0.soc.cpu)
simics> @print "Executed", end_cycle - start_cycle, "instructions"
Executed 366 instructions
simics>
```

After you enter `@while conf.system_cmp0.soc.cpu.msr [...] command`, the simulation starts, and continues until the `msr` register is modified. When that happens, the simulation stops and the rest of the commands can be entered.

You can read more about scripting in chapter *Advanced Features of Simics*. The full description of the Simics API is available in the *Simics Reference Manual*.

Chapter 5

Simics Hindsight User Interface

This section provides a complete description of the Simics Hindsight Graphical User Interface (GUI). On Windows, starting Simics through the Start menu will automatically bring up the GUI. On Unix, this is achieved by running the `simics-gui` script, whether from a workspace or directly from the Simics installation directory.

The first time the GUI is running, it will propose to create a workspace for storing your working files. This is a good suggestion, but if you want to know more, you can refer to the *Workspace Management* section in the *Simics Installation Guide*.

5.1 Overview of Simics Graphical User Interface

Figure 5.1 shows a running Simics simulating a PowerPC card running MontaVista Linux. There are three visible windows:

Simics Control

This is the *control* window: it provides all the menu entries and icons to control Simics. It also describes the currently loaded simulation (see also section 5.2).

Simics Command Line

This is the window that accepts written commands from you. As Simics is quite complex, many tasks are easier to perform with a command line interface. This chapter provides useful command line equivalents for many menu entries, when the menu entries are described. Other manuals will introduce more commands as they become useful (see also section 5.4).

Serial Console on `uart0`

This is the output of the simulated PowerPC card, a text serial console connected to the serial port `uart0`. Depending on the current simulation, you may have one or more of these output windows connected to various ports of the simulated system (serial port, VGA output, etc. See also section 5.3).

Many other windows are available via the menus, depending on your needs. They will be described later in this chapter.

5.1. Overview of Simics Graphical User Interface

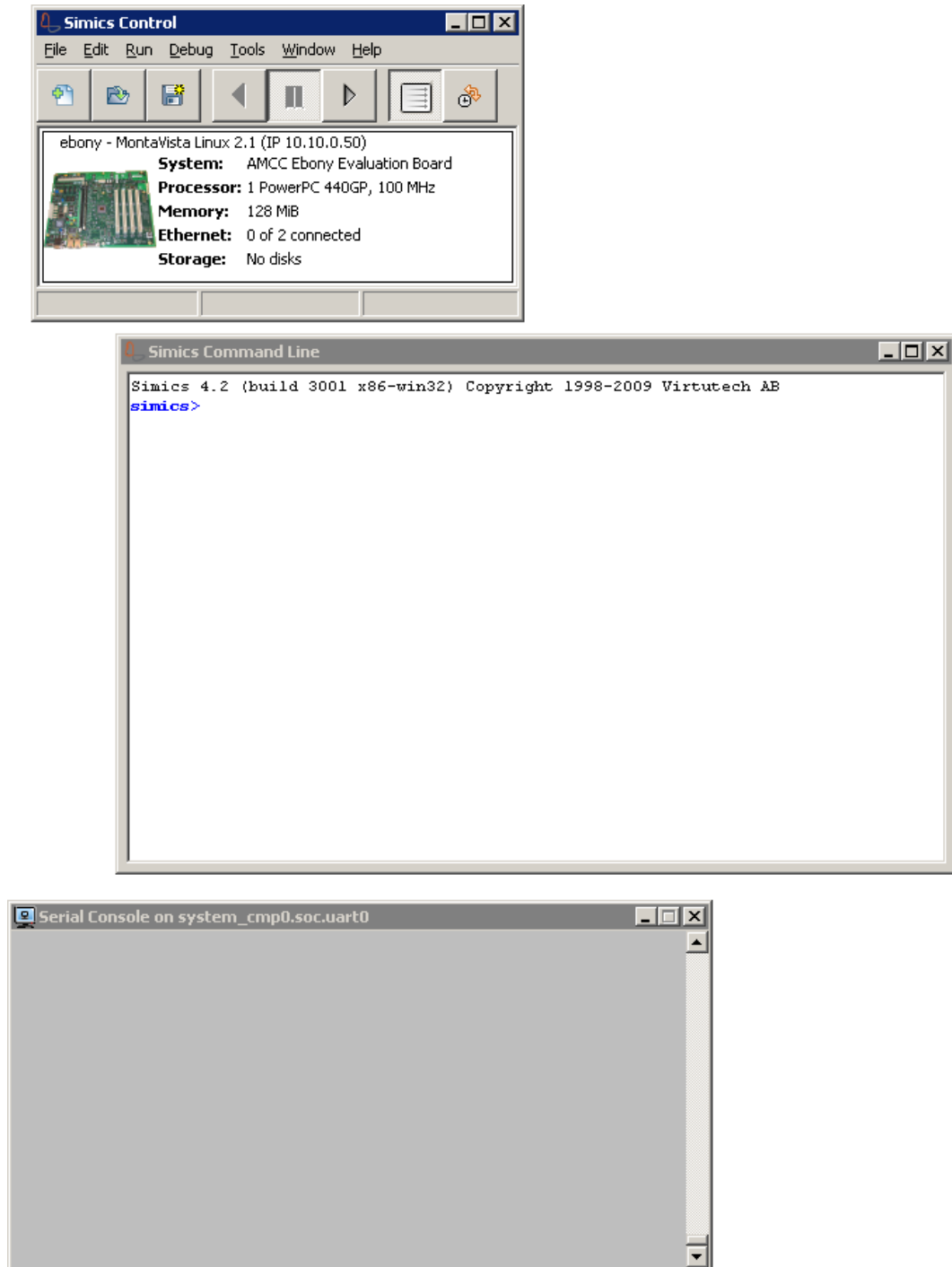


Figure 5.1: A Simics Simulation Session

The Simics GUI is based on the concept of a *simulation session*. When Simics is started, no session is active. Sessions are begun and ended using menu commands or toolbar buttons.

Note: In practice, each session corresponds to running Simics from the command line with a given target configuration. However, it is possible to run several sessions in sequence without quitting the Simics GUI.

5.2 Simics Control Window

The *Simics Control* window (the first window in figure 5.1) contains the toolbar and menus for Simics. It also presents information about the active simulation.



Figure 5.2: Running a Simulation

When the simulation is running, the title of the control window will change to *Simics Control: Running* (see figure 5.2). The status bar at the bottom of the window shows the virtual time elapsed since the start of the simulation. It will also provide information on the current execution status when running with reverse execution enabled.

5.2.1 Toolbar



Figure 5.3: Simics Control Toolbar

The toolbar in the *Simics Control* window offers quick access to the most commonly used commands. The following icons are available:

New Session From Script...

Starts a new Simics session. You will be asked to open a Simics script that will configure a simulated target. Example scripts for each system you have installed will be available in your workspace, in the `targets` directory. Note that when a new session starts, the previous session is automatically closed.

This is equivalent to the **File** → **New Session From Script...** menu entry. You can also start a new session by running the command **run-command-file** on the Command Line Interface (CLI), but you must close the existing session first.

Open Checkpoint...

Opens a previously saved checkpoint in Simics. You will be asked to provide a checkpoint file to open. Note that this will start a new session and automatically close the previous session.

This is equivalent to the **File** → **Open Checkpoint...** menu entry. You can also read an existing checkpoint using the **read-configuration** command at the prompt, but you must close the existing session first.

Save Checkpoint...

Saves the current simulation state in a checkpoint file. You will be asked to provide a name for the checkpoint to save. You can later open this checkpoint to restore the state of the simulation using the **Open Checkpoint...** button.

This is equivalent to the **File** → **Save Checkpoint...** menu entry. You can also save a checkpoint by using the **write-configuration** command.

Run Reverse

Runs the simulation backward in simulated time. This button is only active when reverse execution is enabled for the current simulation (see below).

This is equivalent to the **Run** → **Run Reverse** menu entry. You can also run backward using the **rev** command at the prompt.

Stop

Stops the simulation at the current virtual time.

This is equivalent to the **Run** → **Stop** menu entry. You can also stop the simulation by entering the **stop** command at the prompt.

Run Forward

Runs the simulation forward.

This is equivalent to the **Run** → **Run Forward** menu entry. You can also run the simulation forward by using the **continue** or **c** command at the prompt.

Enable/disable multithreaded simulation

This button is a switch to enable or disable multithreaded simulation. When multithreading is enabled, Simics will try to run the simulation using multiple threads.

This is equivalent to the **Run** → **Multithreading Enabled** switch.

Enable/Disable Reverse Execution

This button is a switch to enable or disable support for reverse execution. When

reverse execution is enabled, Simics will make sure to keep information to be able to reverse the simulation. This will enable the **Run Reverse** button in the toolbar. When disabling reverse execution, Simics will clean up all the saved information and it will not be possible to go back in time anymore.

This is equivalent to the **Run → Reverse Execution Enabled** switch. You can also start the reverse execution mechanism using the **set-bookmark** command at the prompt. To stop it, use the command **delete-bookmark -all**.

5.2.2 File Menu

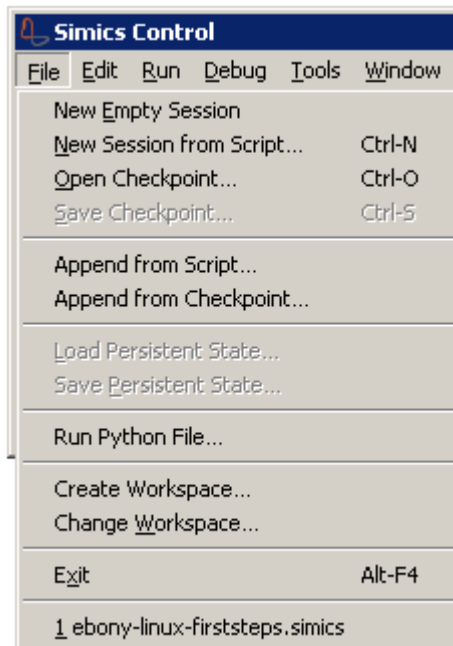


Figure 5.4: File Menu

File → New Empty Session

Creates a new empty simulation session, closing the current session and terminating the simulation.

File → New Session from Script...

Starts a new Simics session. You will be asked to provide a Simics script that will configure a simulated target. Examples scripts for each system you have installed will be available in your workspace, in the `targets` directory. Note that when a new session starts, the previous session is automatically closed. This is equivalent to the **New Session from Script** toolbar icon.

File → Open Checkpoint...

Opens a previously saved checkpoint in Simics. You will be asked to provide a

checkpoint file to open. Note that this will start a new session and automatically close the previous session. This is equivalent to the **Open Checkpoint** toolbar icon.

File → **Save Checkpoint...**

Saves the current simulation state in a checkpoint file. You will be asked to provide a name for the checkpoint to save. You can later open this checkpoint to restore the state of the simulation using the **Open Checkpoint...** menu entry. This is equivalent to the **Save Checkpoint** toolbar icon.

File → **Append from Script...**

Appends a configuration from a script to the current session. The new configuration will be loaded in Simics along with the previous one. This makes it possible to repeatedly load the same target definition to obtain several instances in one simulation. Note that this is possible only when the previous session has not been run yet.

File → **Append from Checkpoint...**

Appends a checkpoint to the current session. This works in the same way as **Append from Script**. However, you cannot load a checkpoint from the same target as the current session, since it would create name collision in the objects used to define the simulation. Note that this is possible only when the previous session has not been run yet.

File → **Load Persistent State...**

Loads the persistent state of a simulation previously saved with **Save Persistent State**. The persistent state usually includes everything that survives a reboot (disk contents, flash memories, ...).

File → **Save Persistent State...**

Saves the persistent state of the simulation. It usually includes everything that survives a reboot (disk contents, flash memories, ...). In order to have a consistent persistent state, it is necessary to stop the operating system running on the target, or at least to flush all caches that could prevent data from being written to the storage devices.

File → **Run Python File...**

Runs a Python script in the current session. Refer to *Advanced Features of Simics* for more information on scripting with Python.

File → **Create Workspace...**

Creates a new workspace and select it as the current workspace.

File → **Change Workspace...**

Changes the current workspace.

File → **Exit**

Terminates the current session and quit Simics (Notice on Unix the keyboard shortcut is Ctrl+Q).

File → *Previous Files*

List of scripts or checkpoints previously opened in Simics.

5.2.3 Edit Menu

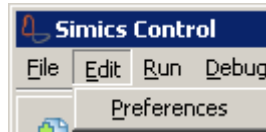


Figure 5.5: Edit Menu

Edit → Preferences

Opens the *Preferences* windows described in section [5.15](#).

5.2.4 Run Menu

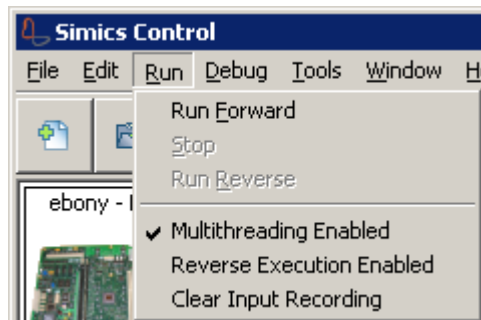


Figure 5.6: Run Menu

Run → Run Forward

Runs the simulation. This is equivalent to the **Run Forward** toolbar icon.

Run → Stop

Stops a running simulation. This is equivalent to the **Stop** toolbar icon.

Run → Run Reverse

Runs the simulation backward in time. Note that you must enable reverse execution before this feature is available. This is equivalent to the **Run Reverse** toolbar icon.

Run → Multithreading Enabled

Enables or disabled the multithreading feature. Multithreading can improve performance of the simulation if the configuration supports it.

This is equivalent to the **Enable/Disable Multithreaded Simulation** toolbar icon.

Run → Reverse Execution Enabled

Enables or disables the reverse execution feature. Reverse execution can affect the

performance slightly since Simics has to keep track of previous states to be able to run backward in time if asked to. The overhead is usually very small.

This is equivalent to the **Enable/Disable Reverse Execution** toolbar icon.

Run → Clear Input Recording

Discard recorded input events (e.g. human console input) which allows an alternate future to take place.

5.2.5 Debug Menu

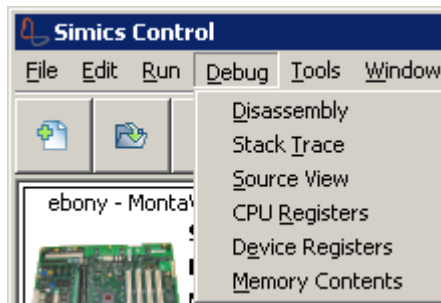


Figure 5.7: Debug Menu

Debug → Disassembly

Shows the *Disassembly* window, which presents the CPU execution in assembly instructions. This window is described in section 5.8.

Debug → Stack Trace

Shows the *Stack Trace* window, which shows the stack trace of the currently running process. This window is described in section 5.17.

Debug → Source View

Shows the *Source View* window, which follows the source code of the current process, if enough information is available. This window is described in section 5.16.

Debug → CPU Registers

Shows the *CPU Registers* window, which presents the contents of various CPU registers. This window is described in section 5.5.

Debug → Device Registers

Shows the *Device Registers* window, which presents the contents of various device registers. This window is described in section 5.6.

Debug → Memory Contents

Shows the *Memory Contents* window, which presents the contents of physical and logical memory spaces. This window is described in section 5.7.

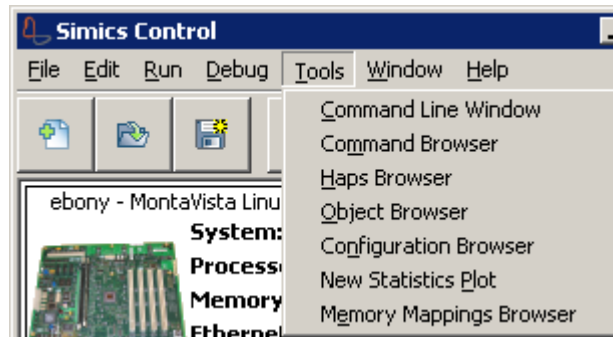


Figure 5.8: Tools Menu

5.2.6 Tools Menu

Tools → Command Line Window

Shows the *Command Line* window, with which you can type commands for Simics to execute. This window is described in section [5.4](#).

Tools → Command Browser

Shows the *Command Browser*, which presents all CLI commands in Simics with their documentation. There is support for text search in both command names and the description texts.

Tools → Hap Browser

Shows the *Hap Browser*, which allows you to browse through the available haps and check what functions are currently registered. This window is described in section [5.9](#).

Tools → Object Browser

Shows the *Object Browser*, which presents the objects that build the simulation and allows you to inspect them. This window is described in section [5.11](#).

Tools → Configuration Browser

Shows the *Configuration Browser*, which presents the components of the model and allows you to create new components and connect them to each other. This window is described in section [5.12](#).

Tools → New Statistics Plot

Creates a new *Statistics Plot* window, which can plot statistics collected by Simics. See section [5.13](#).

Tools → Memory Mappings Browser

Shows the *Memory Mappings Browser*, which presents the memory mappings browser which allows you to inspect the memory spaces of the system and which devices are mapped into them. This window is described in section [5.14](#).

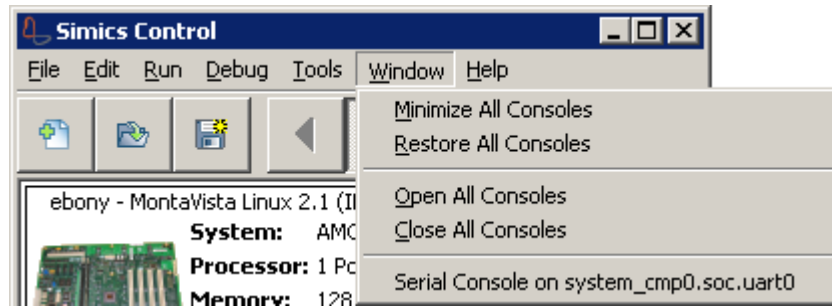


Figure 5.9: Window Menu

5.2.7 Window Menu

Window → Minimize All Consoles

Minimizes all target output windows like serial consoles and windows representing graphical output.

Window → Restore All Consoles

Restores all target output windows like serial consoles and windows representing graphical output.

Window → Open All Consoles

Opens all target output windows that were previously closed.

Window → Close All Consoles

Closes all target output windows.

Window → Console List

List of the currently opened target output windows.

5.2.8 Help Menu

Help → Contents

Opens the *Help Browser* which contains all the Simics on-line help (including all Simics manuals). This window is described in section 5.10.

Help → Getting Started

Opens the *Help Browser* and point it at this manual.

Help → Technical Support

Opens the technical support page of the Virtutech website.

Help → Bug Report Forum

Opens your personal support forum page on the Virtutech website.

Help → Virtutech Website

Opens the Virtutech company website.

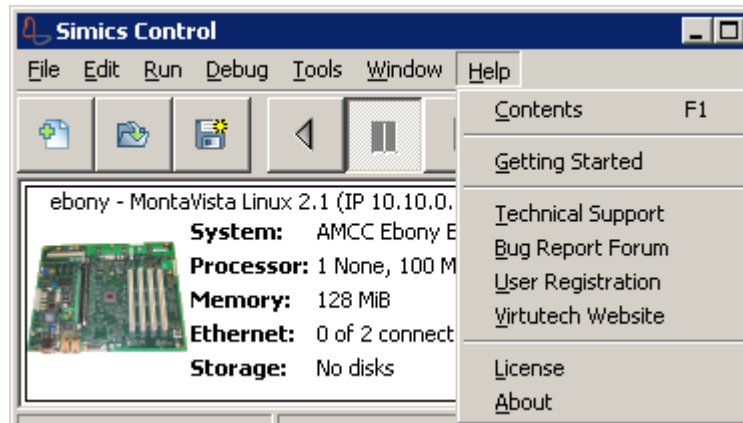


Figure 5.10: Help Menu

Help → **License**

Displays the *Simics Software License Agreement*.

Help → **About**

Describes the current version of Simics and all the configured add-on packages.

5.3 Console Window

The *Console* window shows the output of the simulated machine. It can be graphical, as shown in figure 5.11, or text-based, as shown in figure 5.1. Note that the graphics console can also emulate a text console, which is the case when booting a simulated PC. Usually the graphics or the text console object is called **con0**, when you want to use commands.

5.3.1 Text Console

You can enter text into the text console by selecting the window and typing away. To copy the output from the text console, just drag your mouse inside the window to select text. On Windows, it behaves like a regular command line window, on Unix, like a classic terminal.

To send text to the text console from the Simics command line, you can use the command *console.input string*. This is very handy to automate installation scripts, for example.



Figure 5.11: A Graphics Console Window

Note: If the console is attached via a simulated serial line, as is the case for the simulated *Sarek* and *Donut* machines, the window can be resized using the following sequence of commands:

```
simics> con0.close
Closing console window.
simics> con0->width = 128
simics> con0->height = 60
simics> con0.open
Opening console window.
```

To make the simulated machine understand that the console has changed its size, you would use a command like `stty rows 60 cols 128` (once again, this example is for simulated Solaris).

5.3.2 Graphics Console

A graphics console is used to display what would be seen on the monitor of a real physical computer, but in a window. It's also used to generate keystrokes and mouse moves for the simulated system. Any character typed into the graphics console window on the host system can be forwarded to an appropriate simulated keyboard device.

By clicking the right mouse button with the Shift key pressed when the pointer is in the graphics console, the mouse pointer is captured and the mouse will behave as if controlling the target system. When input is grabbed, most window manager keyboard shortcuts, as well as special key combinations like Ctrl-Alt-Delete, will be intercepted and will not work as usual.

- On Unix, special X11 key combinations usually continue to work though so watch out, Ctrl-Alt-Backspace might still kill your X server.
- On Windows, some key combinations that have special global effects, (such as Ctrl-Escape to bring up the Start menu and Alt-Tab to switch programs) will retain their usual effect and are not grabbed by the console window.

To release the mouse, just press Shift and right click again. Note that regular keystrokes will be sent to the simulated machine whenever the console window is in focus, even when the mouse is not captured.

If you want to change the keyboard/mouse combination to capture the mouse, use the command `console.grab_setup`. Use `help console.grab_setup` to find the available options for this command. Usually the commands will be written as `con0.grab_setup`.

To avoid sending keystrokes and mouse movements to the simulated computer by accident, the command `console.disable-input` can be used. This will prevent the graphics console from passing on any input events to the target. To enable input again use `console.enable-input`.

The graphics console window is not guaranteed to exactly match what would be seen on a real display. For example, if the target system is running in 24-bit color depth and the host

display provides only 16-bit, the colors values will be scaled down and some information will be lost.

In most cases the dimensions (in pixels) of the graphics console exactly matches those of the simulated display, i.e., a simulated 640x480 display will pop up as a window with dimensions of 640x480. This means that the aspect ratio of the simulated display sometimes will be incorrect. For example, when simulating a 640x480 display on a 1280x1024 host display, the simulated display will be a little bit shrunk in the vertical direction.

To capture the contents of the graphics console to a file, use the *console.save-bmp* command.

Note that there is a *graphics device* in the simulated machine which is a separate entity from the graphics console; the graphics console displays what the graphics device has drawn. The display of the graphics console is not updated for every access to the video card but at regular target time intervals, so the graphics seen in the console might not always match what is actually in the video memory of the simulated graphics card. To refresh the console display, use the command *graphics-device.redraw* (which usually translates into *vga0.redraw*). The refresh rate can also be set using the *graphics-device.refresh-rate [rate]* command, where rate is measured in times per target second. Keep in mind that since the time measured is target time, the real world refresh rate depends a great deal on what code is being executed on the simulated machine.

5.4 Command Line Window

The *Command Line* window provides you a prompt on which to type commands. Simics will also use that window for messages and feedback to your commands. The shortcuts used for editing commands can be customized in the *Preferences* window. By default, they follow the standard of the machine you are running on (Windows shortcuts on Windows, and GTK shortcuts on Unix). You can change to Readline (i.e., Emacs-like) shortcuts if you wish to, in which case the following combinations are supported:

control-a:

go to the beginning of the input prompt.

control-e:

go to the end of the input prompt.

control-k:

cut out the text on the right of the cursor.

control-y:

paste the contents of the clipboard at the cursor.

control-f:

move the cursor forward one character.

control-b:

move the cursor back one character.

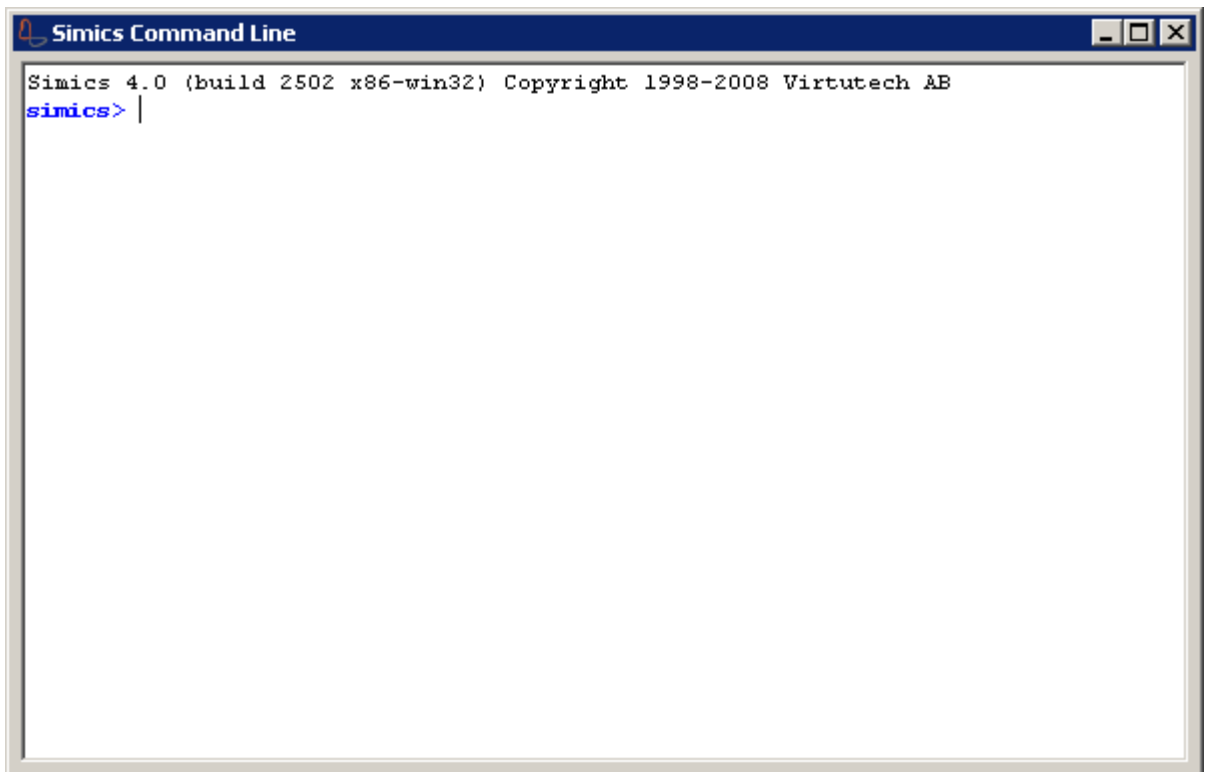


Figure 5.12: Command Line Window

control-n:

recall the next history entry.

control-p:

recall the previous history entry.

control-t:

switch the order of the two characters on each side of the cursor.

control-w:

delete a word to the left of the cursor.

control-d:

delete a character to the right of the cursor.

Using the prompt, you can type any command available in Simics, and you can even write scripts via the custom command line scripting or Python. For more information, read the *Command Line Interface* and *Scripting* chapters in *Advanced Features of Simics*.

5.5 CPU Registers Window

The *CPU Registers* window presents the contents of the selected CPU registers.

CPU registers that have changed value since the last time the window was updated are shown with a yellow background. This feature is mostly useful when single-stepping through assembly instructions.

CPUs usually have a lot of registers, so they have been divided in several groups. Select a specific group to change the registers shown in the window (see figure 5.14). The groups are architecture specific, and the list shown here corresponds to a PowerPC 440GP processor used in the *Ebony* first-steps example machine.

5.6 Device Registers Window

The *Device Registers* window presents the contents of the selected Device's registers. Device registers that have changed value since the last time the window was updated are shown with a yellow background.

The top of the window contains a drop down box which allows you to select the device to inspect and below that is a table with the devices of the device. The table is divided in to sections, with one section for each bank of the device.

5.7 Memory Contents Window

The *Memory Contents* window shows the contents of memory spaces and the virtual memories of processors. It is modeled after hex editors, but currently it is only a viewer. At the top of the window you can select which memory space to inspect and which address in the memory space to go to. Below that is the memory viewer. It shows the contents of memory line by line. Each line shows 16 bytes of memory. The viewer has 3 columns. The leftmost

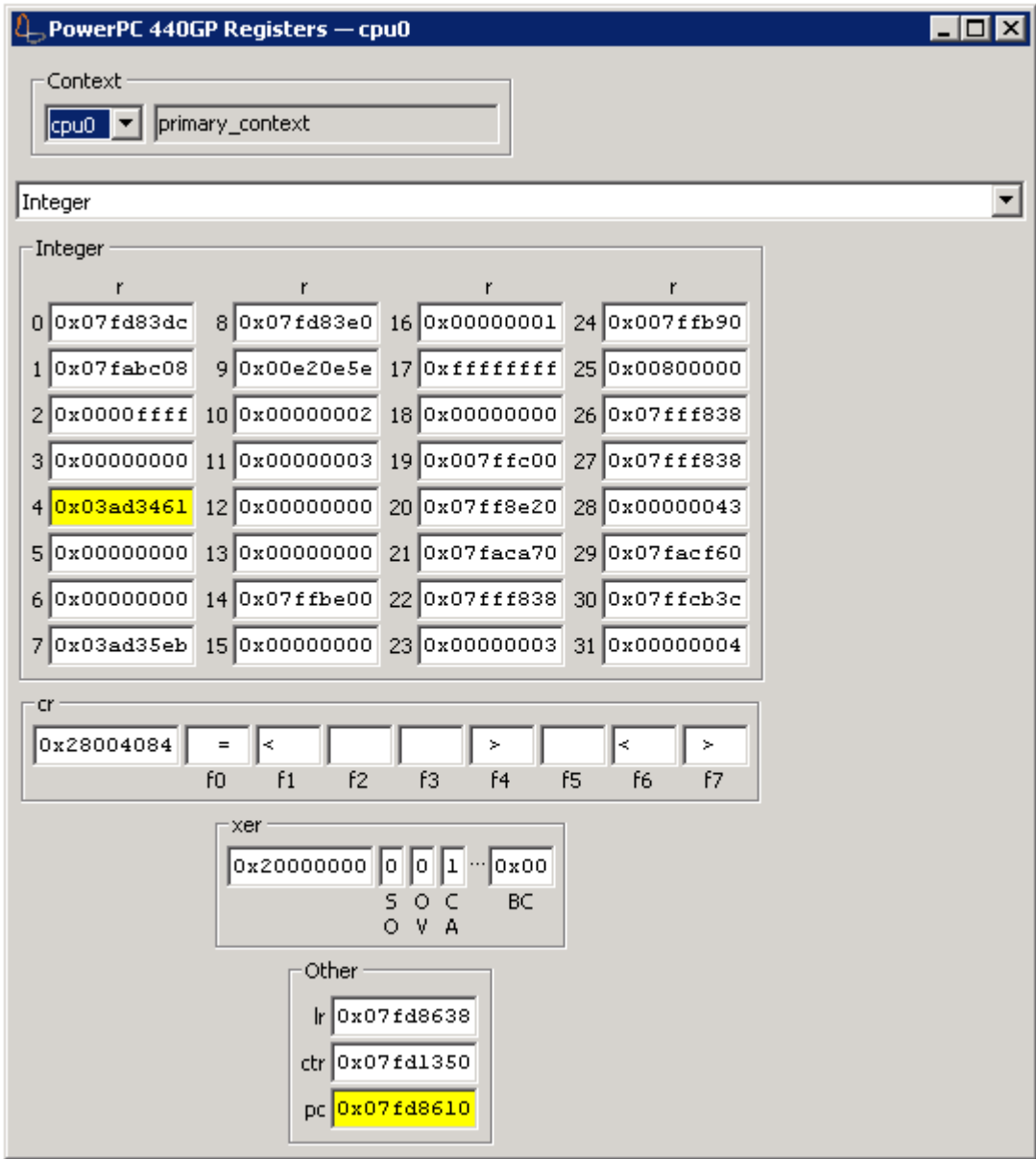


Figure 5.13: CPU Register Window

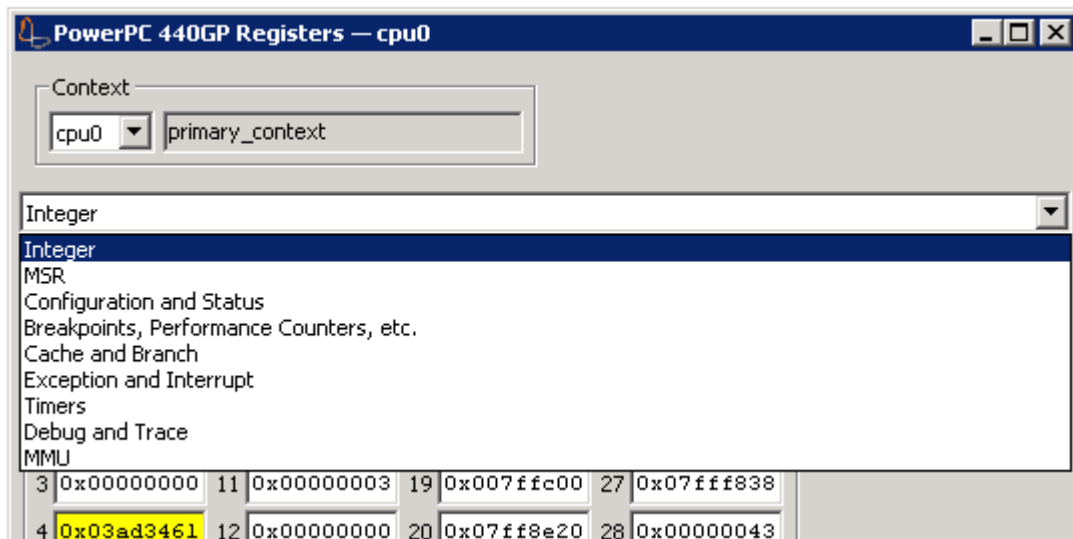


Figure 5.14: CPU Register Types

column shows the address for each line as a hexadecimal value. The middle column shows the contents of the line interpreted as hexadecimal values. The right columns shows the contents of the line interpreted as ASCII values. You can select a range of memory in the viewer by dragging, like in a text editor. To the right of the viewer is an inspector which shows the value of the current selection interpreted as an integer. A pair of radio buttons allow you to chose if you want to interpret the selection as a big endian value or a little endian value. If you selection is larger than 1024 bytes the inspector will not try to interpret the selection as an integer.

Some of the addresses in a memory space can be invalid. This is shown by special tokens for the bytes in question. For addresses outside memory the viewer displays ** instead of a hexadecimal value, for virtual addresses which the processor failed to translate to physical addresses the viewer shows -- instead and for addresses in devices which do not support inquiry accesses the viewer shows ??. If the selection contains one of these special error values the viewer can not interpret the selection as an integer.

The ASCII column will display value outside the range 1-126 as .. Error values are also shown as ..

5.8 Disassembly Window

The *Disassembly* window is shown in figure 5.17. When Simics is stopped, it shows the assembly code surrounding the current instruction.

The System **Step** and **Unstep** buttons will single-step the processor one instruction forward or backward, respectively. (Note that unstepping requires reverse execution to be enabled.)

The Context **Step** and **Unstep** buttons will single-step the processor's *current context*; this is only useful if you have set up this context to follow a particular process, as described

Offset	S..	Value	Register	Description
0x0	4	1046272	CCSRBAR	
0x8	4	0	ALTCBAR (uni...	
0x10	4	0	ALTCAR (unimpl)	
0x20	4	0	BPTR	
0xc08	4	0	LAWBAR[0]	
0xc10	4	0	LAWAR[0]	
0xc28	4	0	LAWBAR[1]	
0xc30	4	0	LAWAR[1]	
0xc48	4	0	LAWBAR[2]	
0xc50	4	0	LAWAR[2]	
0xc68	4	0	LAWBAR[3]	
0xc70	4	0	LAWAR[3]	
0xc88	4	0	LAWBAR[4]	
0xc90	4	0	LAWAR[4]	
0xca8	4	0	LAWBAR[5]	
0xcb0	4	0	LAWAR[5]	
0xcc8	4	0	LAWBAR[6]	
0xcd0	4	0	LAWAR[6]	
0xce8	4	0	LAWBAR[7]	
0xcf0	4	0	LAWAR[7]	
0xd08	4	0	LAWBAR[8]	
0xd10	4	0	LAWAR[8]	
0xd28	4	0	LAWBAR[9]	
0xd30	4	0	LAWAR[9]	

Figure 5.15: Device Register Window

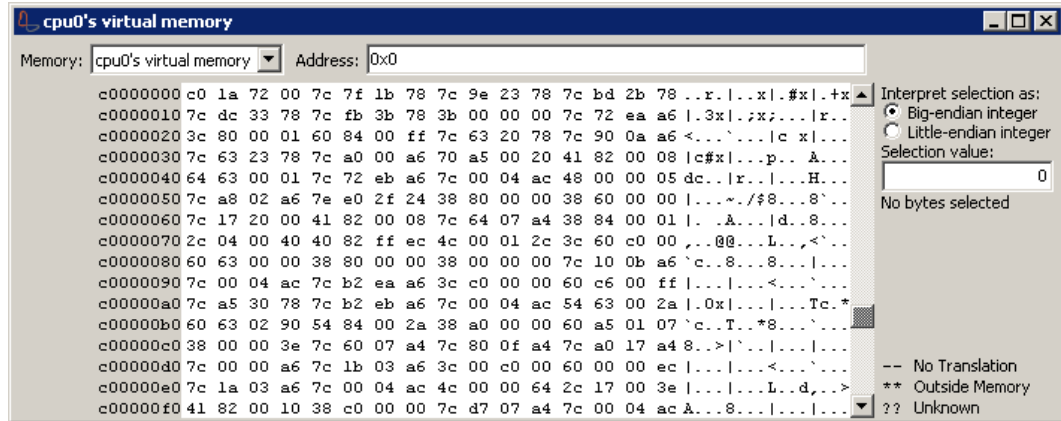


Figure 5.16: Memory Contents Window

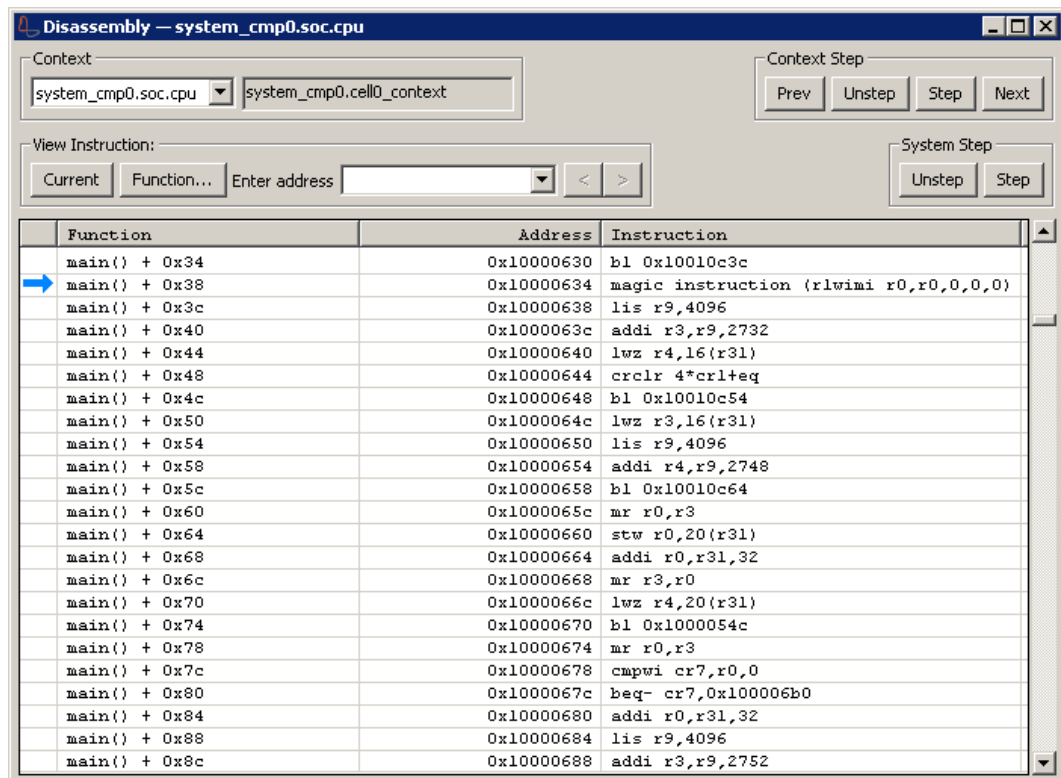


Figure 5.17: The Disassembly Window

in *Using Simics for Software Development* and *Advanced Features of Simics*. Then, these buttons will single-step in the instruction stream of that process, and ignore all other processes.

The Context **Next** and **Prev** buttons work just like **Step** and **Unstep**, except that they will skip subroutine calls: **Next** will step directly from a subroutine call instruction to the point where the call returns; **Prev** does the same but in reverse.

By typing an address in the address field and then clicking **Go to Address**, the disassembly window can disassemble code anywhere in memory. Click on **Program Counter** to get back to the current instruction.

5.9 Hap Browser

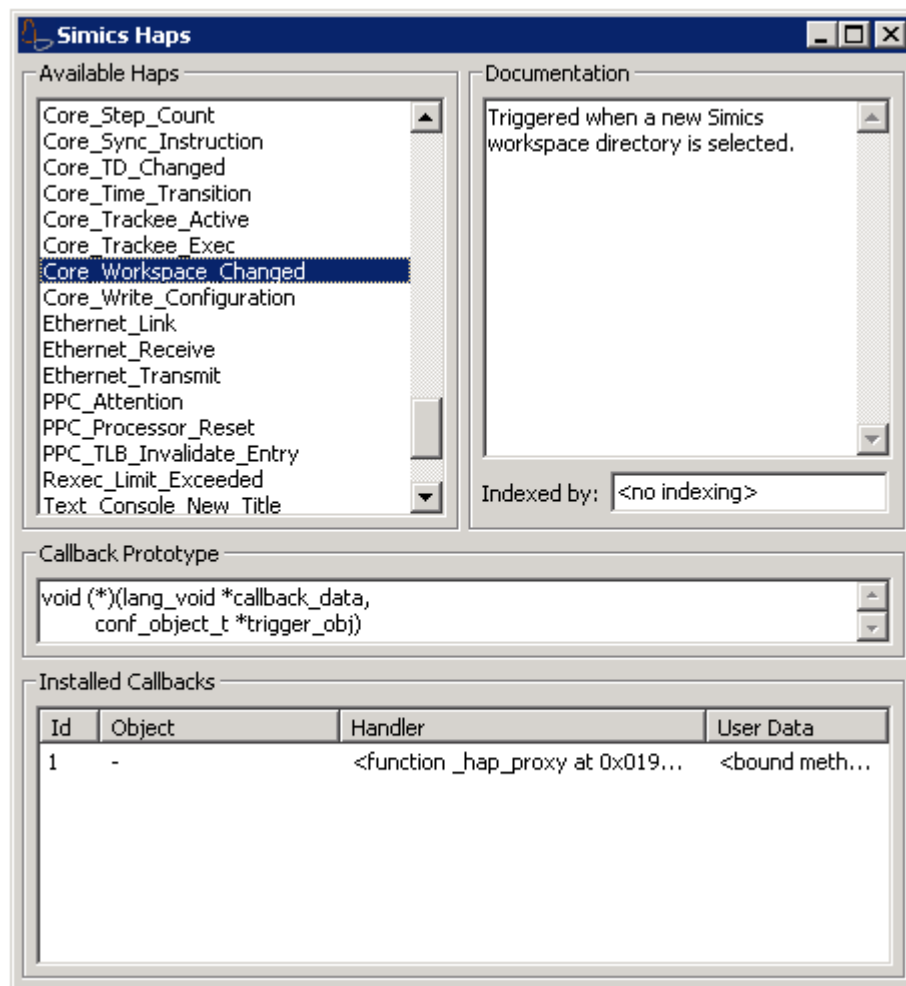


Figure 5.18: Hap Browser

The hap browser is designed to help script writers to use haps and hap callbacks. Refer to *Advanced Features of Simics* for more information on scripting and haps.

The hap browser shows the list of the haps currently registered in the session in the top-left pane. For each hap, it prints the description, the index and the callback prototype that should be used. It can also list the functions currently triggered by this hap.

5.10 Help Browser

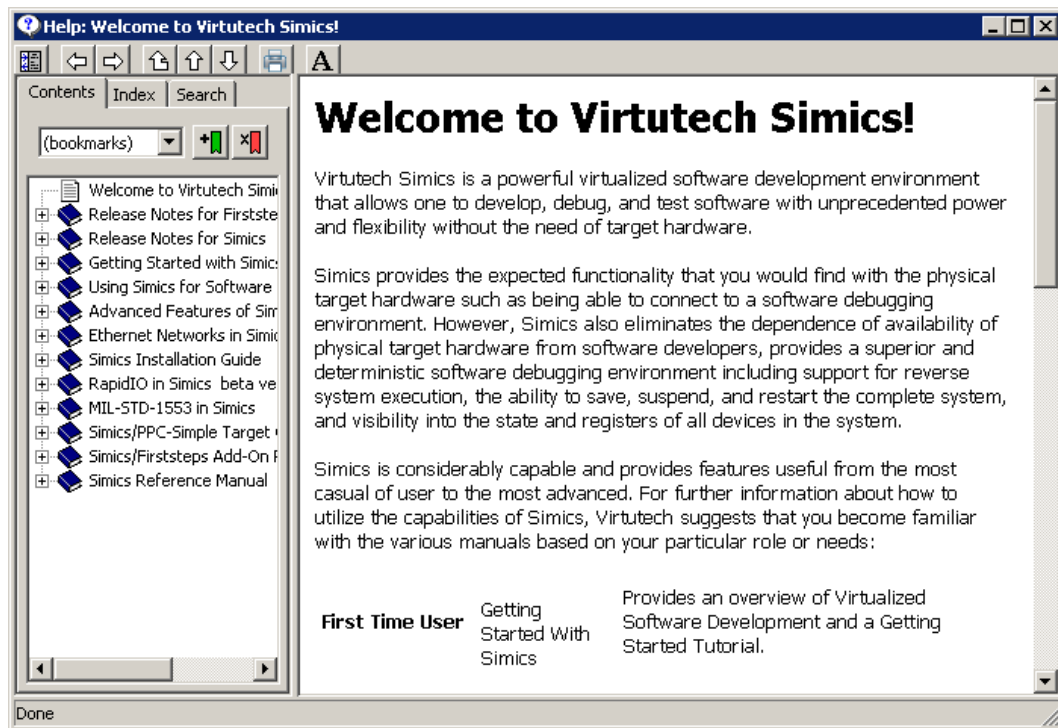


Figure 5.19: Help Browser

The Help Browser (figure 5.19) is a standard help system presenting all Simics manuals and guides. It allows you to navigate in the documentation and set bookmarks on interesting topics.

The *Index* panel let you search and browse the index of the documents. Do not forget to click on **Show All** to see all index entries.

The *Search* panel let you search for a specific string in the documentation.

5.11 Object Browser

The *Object Browser* window (figure 5.20) lets you inspect all the objects that compose the target configuration. The top-left list contains all the components, while the bottom-left list contains all the objects that belong to the simulation. Clicking on any of these will present information in the right side panel. To understand much of this information, you will need

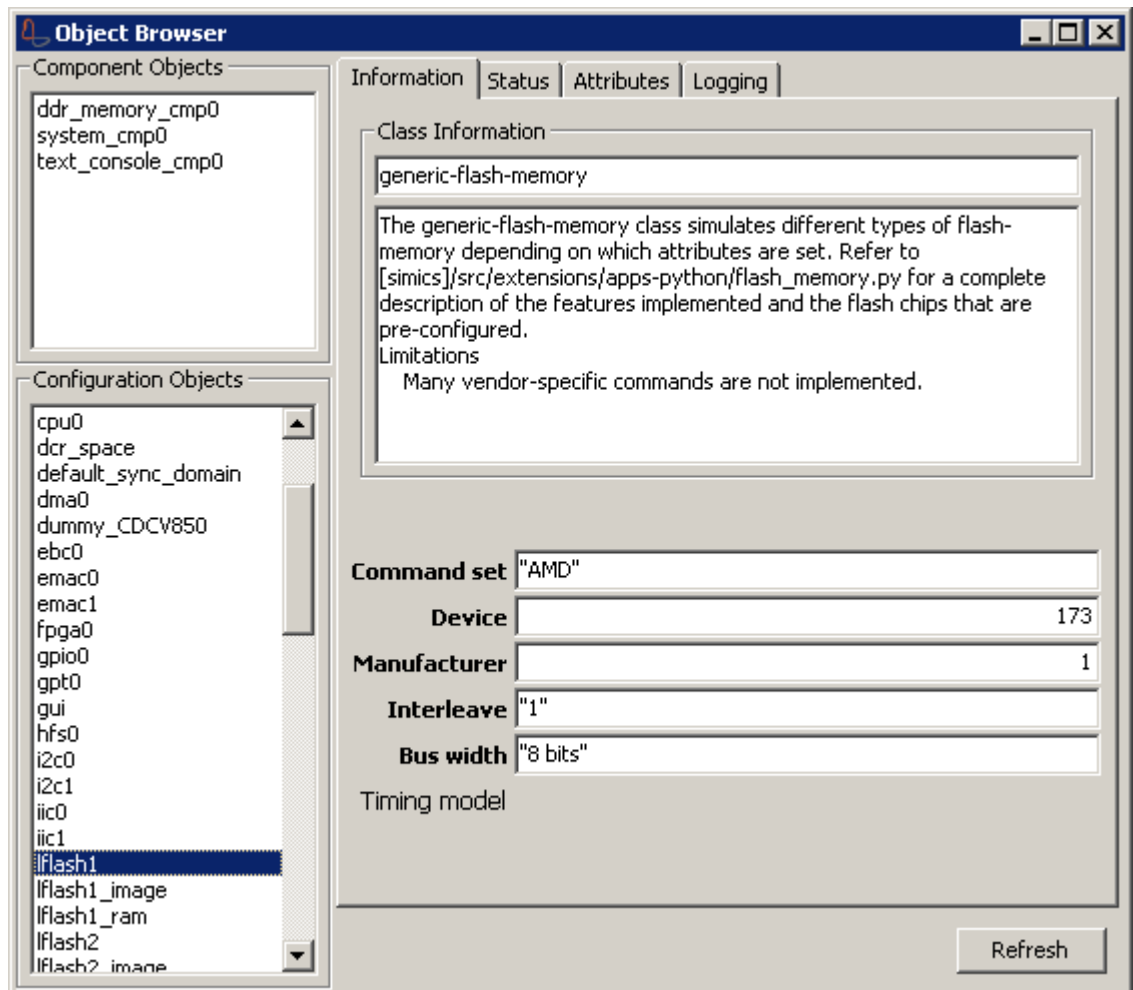


Figure 5.20: Object Browser

to get acquainted with the way Simics build configurations, which is described in *Advanced Features of Simics*.

In figure 5.20, the selected object is a flash-memory called **iflash1**.

Information

The class of the object will be indicated first, followed by the class description. The rest of the information is object-specific. In our example, it describes the type of flash memory configured (8-bits bus with no interleave, AMD command-set).

Status

The *Status* panel contains current information about the object state. In the case of our flash memory, it simply shows that the flash memory is in read-array mode at this point, which means that it behaves like normal memory for read operations.

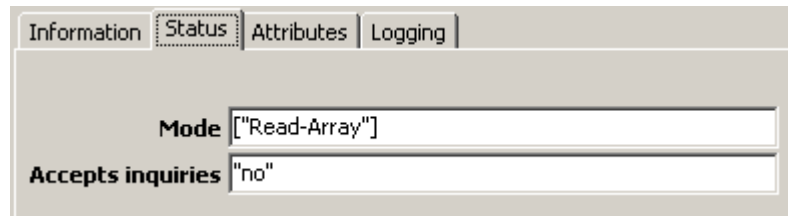


Figure 5.21: Object Browser: status

Attributes

The *Attributes* panel contains a list of all attributes that constitute the state of the object. Attributes represent both the static state (like configuration parameters) and the dynamic state (like the current operation, etc.). Clicking on an attribute will show its current value, its kind (Required, Optional, etc.), its format and its documentation in the bottom fields.

5.12 Configuration Browser

The *Configuration Browser* window (figure 5.23) lets you inspect and edit the components of the target configuration. You can also create new components. This is a three stage process: first you create the components, then you connect the components to each other, and finally you instantiate the components.

The window is divided into two major parts. The left part shows the components in the system and how they are connected. Here you create and instantiate new components. You can also create connections between components by drag and drop.

The graph of components connected by connections is very tree like and the left part of the window shows it as such. If a component is present in several branches of the tree, which is common for components like Ethernet links, a reference is placed in each branch and the component is shown at the top level of the tree. The references are identified by having an arrow icon (↵) instead of their usual component icons.

The right part of the window shows the currently selected component and its connectors and makes it possible to create and break connections between connectors. Just choose a connector and click the *Connect* or *Disconnect* buttons.

The *Configuration Browser* is also linked to the *Object Browser* (see section 5.11). By double-clicking a component you open it in the Object Browser.

5.13 Statistics Plot

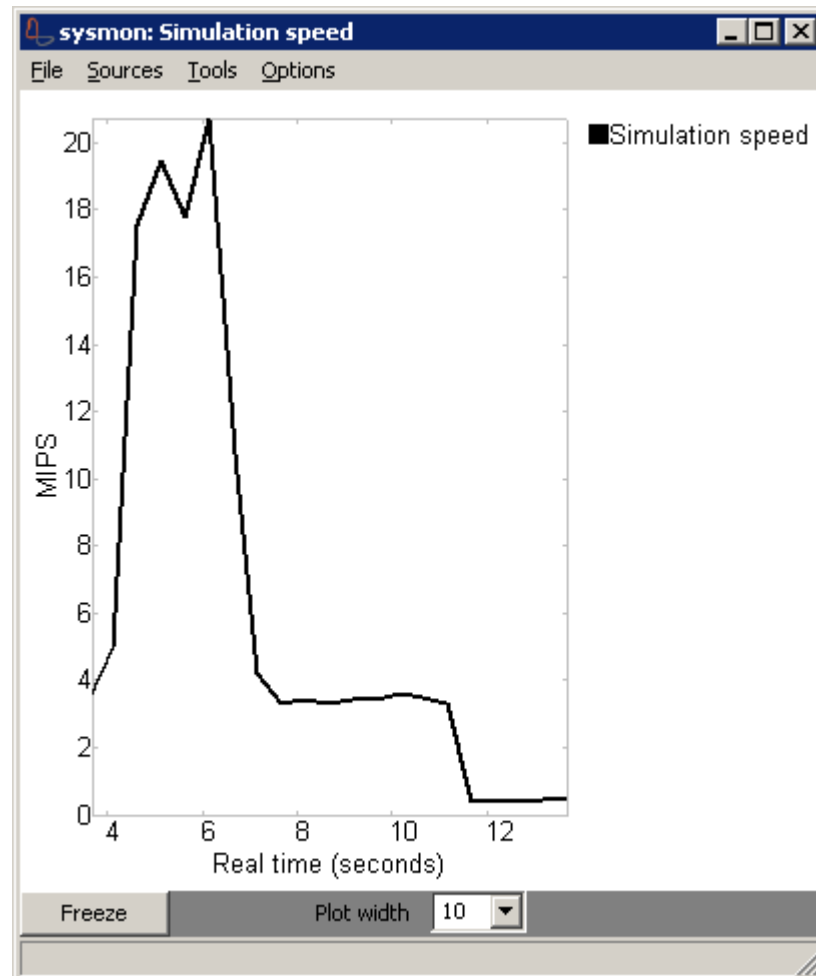


Figure 5.24: Statistics Plot

A *Statistics Plot* window (figure 5.24) lets you plot statistics collected by Simics. Specifically, some Simics objects publish data on how a scalar value (such as the simulation speed) changes over time, and the plot window displays the changing value graphically.

In the **Sources** menu, you are presented with a selection of the available statistics sources—Simics objects that publish statistics. The **sysmon** data source is always available, and provides statistics about Simics itself, such as host CPU usage or simulation speed (shown in

the screen shot). Other data sources include **g-cache**, which publishes cache statistics such as read misses, write misses, etc.—but note that most configurations do not come equipped with caches by default. You can add cache **g-cache** based caches yourself, but such models should only be added when detailed cache and timing information is needed.

You may open any number of plot windows. Each plot window can plot multiple data sources at once, as long as they have the same y-axis units. If you select a source that is incompatible with one you have already selected, the new source will replace the old.

5.14 Memory Mappings Browser

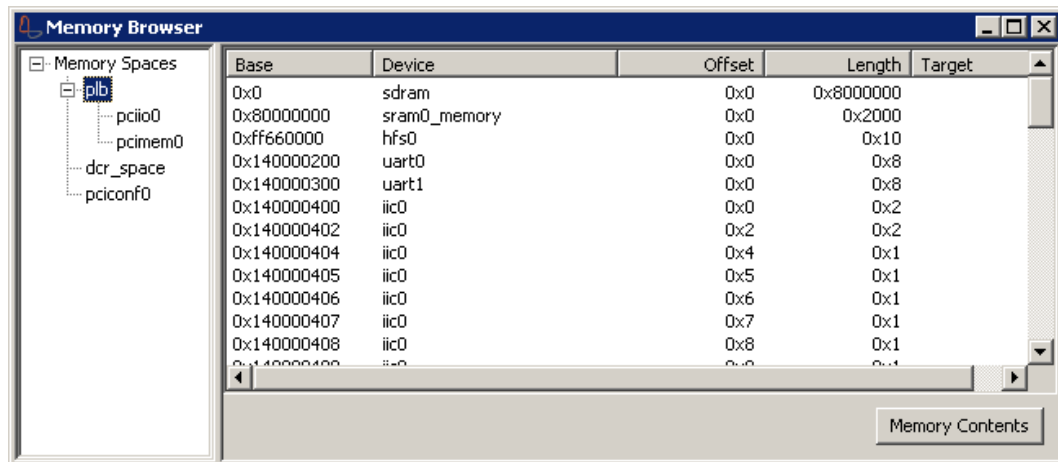


Figure 5.25: Memory Mappings Browser

The *Memory Mappings Browser* window (figure 5.25) shows you the memory spaces in your configuration with the devices mapped into them. The window is divided into two major parts. The left part is a tree which shows all the memory spaces in the system and how they are connected to each other and the right part a table which shows the devices mapped into the memory space. Select a memory space in the tree to the left to inspect it in the table to the right.

In the tree control with the memory spaces the roots of the trees are the memory spaces which are physical memories of processors in the system, memory spaces not mapped into any other memory spaces and finally, just to make sure all memory spaces are shown, all memory spaces not reachable from any of the first two sets. There is no division between the three categories of roots, but they are sorted. First all the physical memories are shown, then unmapped memory spaces and finally the rest. Each memory space is present as a child of every memory space it is mapped into. If you have cycles between the memory spaces the cycle will be broken the second time the same memory space is present in a branch from the root. This node in the tree will not have any children, but you can still see all the devices mapped into it in the right part of the window.

The table with devices mapped into the memory space lists all the banks mapped into the memory space. A bank in this case can be either a device, a port of a device or a function

of a device. Each line contains the start address of the bank, the name of the bank, the size of the window, the offset in the bank where this window is placed and finally, if the mapping is a translation mapping, the target of the translation. Below the table there is a button which opens a memory contents viewer for the memory space. See section 5.7 for documentation about the viewer it opens.

5.15 Preference Window

The *Preferences* window is organized in three panels: *Appearance* (figure 5.26), *Startup* (figure 5.27), and *Advanced* (figure 5.28). You can cancel all the changes done in the three panels by using the **Revert to Saved** button at the bottom left corner.

Appearance Panel

The *Appearance* (figure 5.26) panel let you define how input and output will be handled:

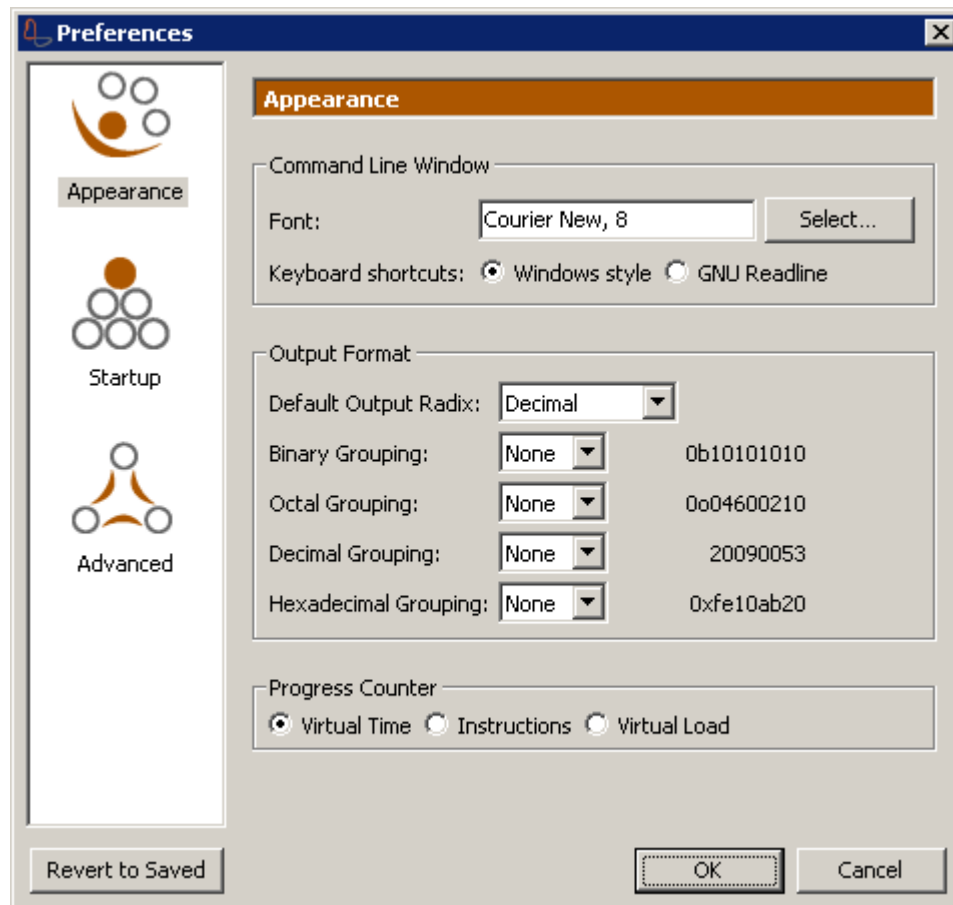


Figure 5.26: Preferences: Appearance

Command Line Window

This option box let you customize the font used in *Simics Command Line* window. You can also control the shortcuts used when typing at the command line:

- On Windows, you can opt for standard Windows shortcuts, or Unix Readline shortcuts (i.e., Emacs-like).
- On Unix, you can choose between standard GTK shortcuts or the Readline shortcuts.

Output Format

The output format option box lets you decide how Simics should show numbers like addresses or CPU registers in the *Command Line* window. You can choose in which base numbers will be displayed by default, and how numbers in a specific base should be printed out, inserting some “_” (underscores) to increase their readability. Note that you can always type “_” inside numbers if you wish to, they will be ignored by the command line regardless of these settings.

Progress Counter

This option controls what is shown in the status bar of the *Simics Control* window. Virtual time is common to the whole simulation, and is counted in seconds since the simulation was set up. Instruction counter and virtual load on the other hand, applies to the first CPU created in the simulation. The instruction counter shows the number of instructions executed by the CPU since the simulation was set up, while the virtual load shows how busy the CPU is at this point of time.

Startup Panel

The *Startup* panel (figure 5.27) lets you control how Simics should behave when started:

Startup Options

Execution Mode controls the type of CPU models that will be created when starting a new session. *Stall* CPUs are slower models that have interesting capabilities in terms of improved timing and instruction or data profiling. You can find more information on these in *Advanced Features of Simics* and—in the *Model Builder* package—*Advanced Memory and Timing in Simics*.

Enable reverse execution on startup controls whether reverse execution is enabled by default or not when starting a new session.

Enable multithreading on startup controls whether multithreading is enabled by default or not when starting a new session.

GUI

This box controls two aspects of how Simics handles its GUI. The first controls whether you can open windows when running Simics in command line mode and the second which windows Simics should open at start up.

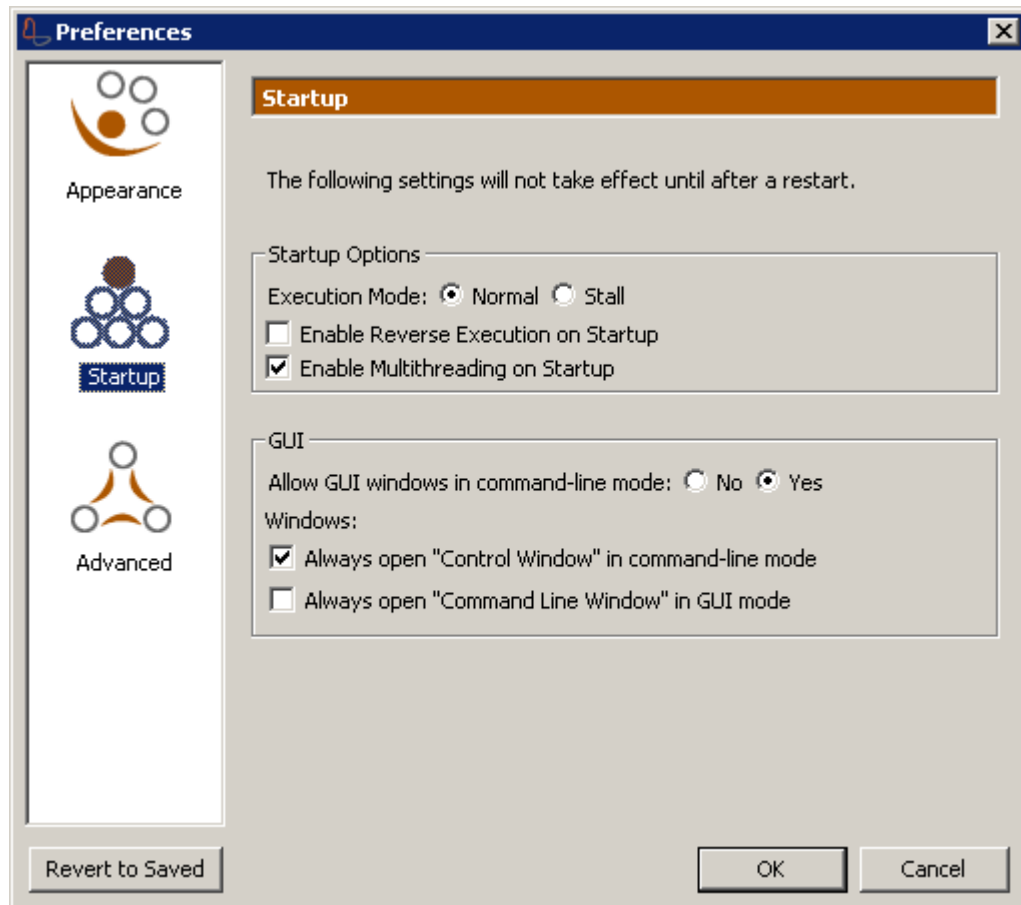


Figure 5.27: Preferences: Startup

Advanced Panel

The *Advanced* panel (figure 5.28) provides control over more advanced features:

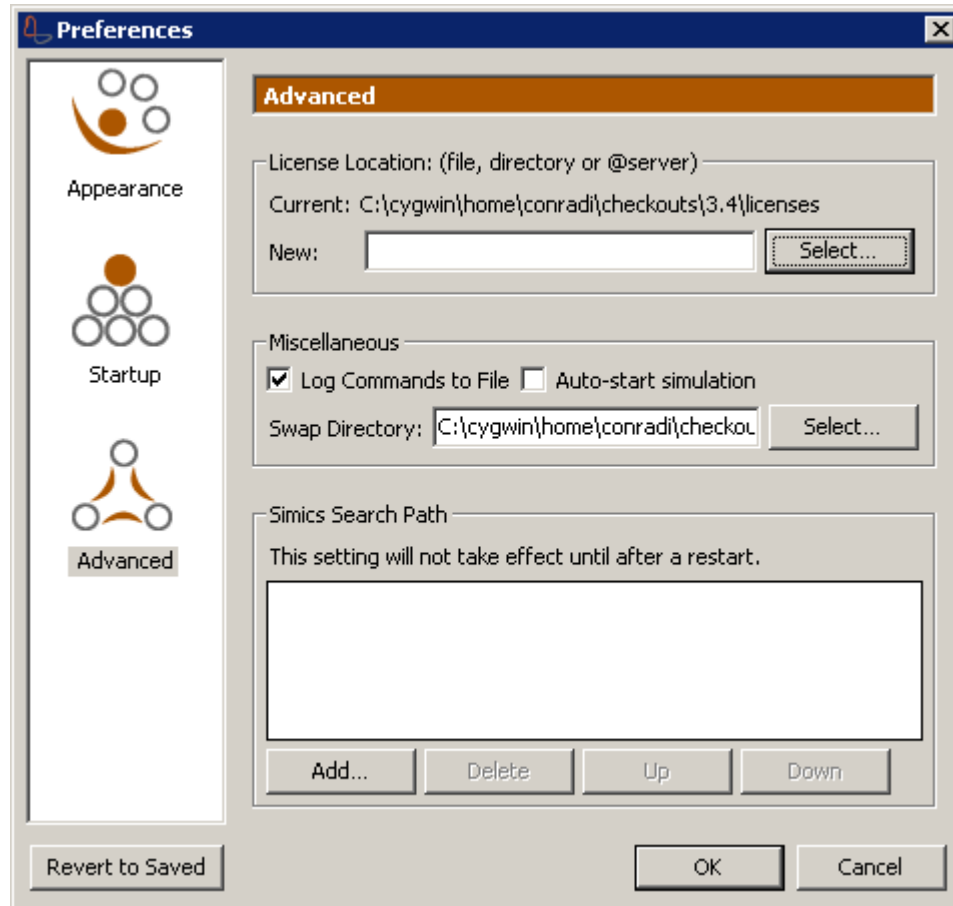


Figure 5.28: Preferences: Advanced

License File

By default, Simics looks for a license file in its own installation directory, under the `licenses` directory. You can also specify a license file by providing it here.

Miscellaneous

Log commands to file controls whether Simics will create a log file for all commands entered in the command line window. This log file is available in `~/ .simics/4.2/ log` on Unix and in `Application Data\Virtutech\Simics\4.2\log` in your personal folder on Windows.

Auto-start simulation will tell Simics to start a simulation directly after loading it, instead of waiting for you to press the **Run Forward** button.

The **Swap directory** will be used by Simics when running simulations requiring a lot of memory. By default, Simics will not use swapping, but if you ask it to do so, it will

store temporary files in the directory indicated here. See *Advanced Features of Simics* for more information on memory usage and swapping.

Simics Search Path

The Simics Search Path tells Simics where to find files if they are not in the current directory, or at a specified place. This is useful to avoid absolute paths and to write more generic scripts. The files concerned are mainly disk images and binary files. You will find more information on this feature in *Advanced Features of Simics*.

5.16 Source View Window

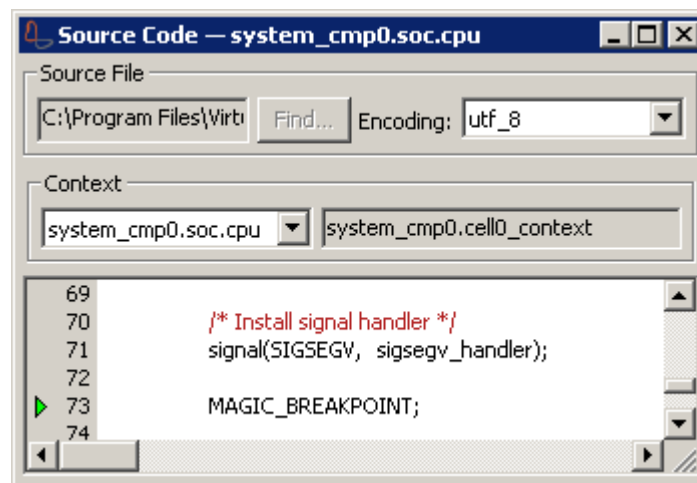


Figure 5.29: The source view window

The *Source View* window is shown in figure 5.29. When Simics is stopped, it shows the source code surrounding the current instruction. An arrow in the left margin indicates the source line containing the current instruction.

In order for this feature to work, the current context of the current processor (shown in **Context**) must have access to debug symbols for the running program. *Using Simics for Software Development* explains how to set this up.

5.17 Stack Trace Window

The *Stack Trace* window is shown in figure 5.30. It lists the active *stack frames*; that is, the functions that have been called but not yet returned. The outermost function is at the bottom, and the innermost (the one currently containing the program counter) is at the top.

If the current context of the current processor (shown in **Context**) has access to debug symbols for a stack frame, the function name and parameters will be shown for that frame, as seen in the top two lines in figure 5.30. If no debug symbols are available, only the frame's

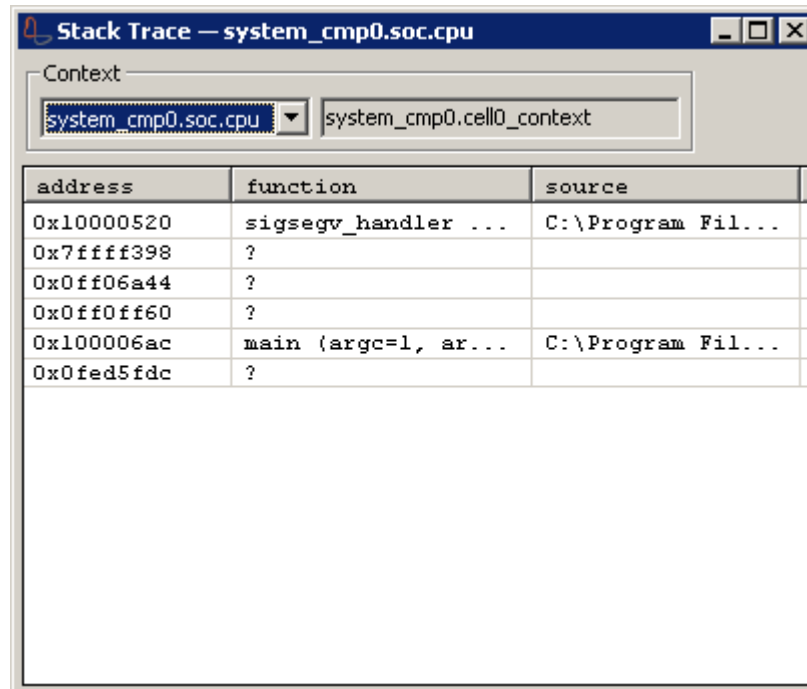


Figure 5.30: The stack trace window

address will be shown, as seen in the bottom two lines. *Using Simics for Software Development* explains how to load debug symbols.

Chapter 6

Next Steps

This *Getting Started with Simics* guide has provided an introduction and overview of the Simics platform and Virtualized Systems Development. To become more familiar with the features and capabilities of Simics, you should review the additional documentation that is provided.

Software developers who want to learn about the software debugging capabilities provided with the Simics platform should read the *Using Simics for Software Development* manual, and particularly the chapter regarding debugging with Simics.

Because Simics can be used as a back-end to many existing software debuggers, you should review the appropriate chapters in the *Using Simics for Software Development* to understand how to use Simics with your particular software debugger.

If you want to learn more about Simics simulation models or want to create or modify your own model, then you should review the *Modeling Your System in Simics* manual, and particularly the tutorial provided in this manual. Note that *Modeling Your System in Simics* is only available to those who purchased the *Simics Model Builder* product.

Index

Symbols

@, 36

–
in numbers, 65

[simics], 5

[workspace], 5

A

About, 47

Accelerator, 13

API, 36

Append from Checkpoint, 42

Append from Script, 42

assembly code, 54

attributes, 35, 60

B

bookmark, 22

break, 29

break-cr, 35

break-exception, 35

break-io, 35

Bug Report Forum, 46

C

callback, 58

callbacks, 57

capture

screen, 50

Change Workspace, 42

checkpoint

append, 42

open, 40, 41

save, 40, 42

checkpointing, 21

class, 60

Clear Input Recording, 44

clear-recorder, 23

CLI, 37, 45, 50

CLI variables, 35

Close All Consoles, 46

Command Browser, 45

Command line, 50

Command Line Window, 45, 65

component, 58, 60

configuration, 58, 60

Configuration Browser, 45, 60

Console, 37

Console Window, 46, 47

Contents, 46, 58

context, 54, 68

CPU Core Types, 15

CPU Registers, 44, 52

Create Workspace, 42

D

Debug menu, 44

CPU Registers, 44

Device Registers, 44

Disassembly, 44

Memory Contents, 44

Source View, 44

Stack Trace, 44

debug symbol, 68

debug symbols, 68

debugging, 25

delete, 32

delete-bookmark, 22

Device Model Libraries, 15

Device Registers, 44, 52

Disable Multithreading, 43

Disable Reverse Execution, 43

disable-magic-breakpoint, 32

Disable/Enable Multithreaded, 40

Disable/Enable Reverse Execution, 40

Disassembly, 44

disk images, 68

E

Edit menu, 43

 Preferences, 43

Enable Multithreading, 43

Enable Reverse Execution, 43

enable-magic-breakpoint, 26

Enable/Disable Multithreaded, 40

Enable/Disable Reverse Execution, 22, 40

Ethernet Networking, 14

execution mode, 65

Exit, 42

F

File menu, 41

 Append from Checkpoint, 42

 Append from Script, 42

 Change Workspace, 42

 Create Workspace, 42

 Exit, 42

 Load Persistent State, 42

 New Empty Session, 41

 New Session from Script, 41

 Open Checkpoint, 41

 previous files, 42

 Quit, 42

 Run Python File, 42

 Save Checkpoint, 42

 Save Persistent State, 42

first-steps, 16

frame, 30

G

Getting Started, 46

Go to Address, 57

Graphics Console, 49

GUI, 65

H

Hap Browser, 45, 57

haps, 57

Help, 58

 index, 58

 search, 58

help, 27

Help Browser, 58

Help menu, 46

 About, 47

 Bug Report Forum, 46

 Contents, 46

 Getting Started, 46

 License, 47

 Technical Support, 46

 Virtutech Website, 46

Hindsight, 13

I

index, 58

L

License, 47

license, 67

License File, 67

list-breakpoints, 32

Load Persistent State, 42

log, 67

M

magic breakpoint, 26

magic instruction, 25

Memory Contents, 44, 52

Memory Mappings Browser, 45, 63

memory usage, 68

Minimize All Consoles, 46

Miscellaneous, 67

Model Builder, 14

mouse, 49

Multithreading, 13

multithreading, 65

 enable/disable, 43

N

New Empty Session, 41

New Session From Script, 39

New Session from Script, 41

New Statistics Plot, 45

new-tracer, 32

Next, 57

O

object, 58

- Object Browser, [45](#), [58](#)
 - Attributes, [60](#)
 - Information, [60](#)
 - Status, [60](#)
- Open All Consoles, [46](#)
- Open Checkpoint, [21](#), [40](#), [41](#)
- Output Format, [65](#)
- P**
- Page Sharing, [13](#)
- persistent state
 - load, [42](#)
 - save, [42](#)
- Preferences, [43](#), [64](#)
 - Advanced, [67](#)
 - Appearance, [64](#)
 - Startup, [65](#)
- Prev, [57](#)
- product
 - CPU Core Types, [15](#)
 - definitions, [13](#)
 - Device Model Libraries, [15](#)
 - Ethernet Networking, [14](#)
 - Hindsight, [13](#)
 - Model Builder, [14](#)
 - Simics Accelerator, [13](#)
 - Virtual Board/System, [14](#)
- Program Counter, [57](#)
- Progress Counter, [65](#)
- prompt, [50](#)
- psym, [29](#)
- ptime, [22](#)
- python, [36](#)
 - Run Python File, [42](#)
- Q**
- Quit, [42](#)
- R**
- Readline, [50](#)
- refresh rate, [50](#)
- register
 - types, [52](#)
- registers, [52](#)
- Restore All Consoles, [46](#)
- reverse, [29](#)
 - reverse execution, [22](#), [54](#), [65](#)
 - enable/disable, [40](#), [43](#)
 - Run Reverse, [40](#)
 - reverse-step-line, [29](#)
 - Run Forward, [20](#), [40](#), [43](#)
 - Run menu, [43](#)
 - Multithreading Enabled, [43](#)
 - Reverse Execution Enabled, [43](#)
 - Run forward, [43](#)
 - Run Reverse, [43](#)
 - Stop, [43](#)
 - Run Python File, [42](#)
 - Run Reverse, [40](#), [43](#)
- S**
- Save Checkpoint, [21](#), [40](#), [42](#)
- Save Persistent State, [42](#)
- screenshot, [50](#)
- scripting, [35](#)
- search, [58](#)
- Serial Console, [17](#), [37](#)
- session, [37](#)
 - append, [42](#)
 - new, [41](#)
 - new from script, [39](#), [41](#)
- set-bookmark, [22](#)
- shortcuts, [50](#), [65](#)
- Simics Accelerator, [13](#)
- Simics Command Line, [37](#)
- Simics Control, [37](#)
- Simics Control window, [39](#)
- Simics Search Path, [68](#)
- simics user interface, [37](#)
- SimicsFS, [24](#)
- skip-to, [23](#)
- source code, [68](#)
- Source View, [26](#), [44](#)
- source-path, [27](#)
- stack frame, [68](#)
- Stack Trace, [31](#), [44](#)
- stack-trace, [28](#)
- stall, [65](#)
- Startup Options, [65](#)
- Statistics Plot, [45](#), [62](#)
- status, [60](#)
- status bar, [65](#)

Step, [54](#)
 step-instruction, [34](#)
 Stop, [20](#), [40](#), [43](#)
 swapping, [67](#)
 symbolic debugging, [25](#)

T

target, [14](#)
 Technical Support, [46](#)
 Text Console, [37](#), [47](#)
 toolbar, [39](#)

- Enable/Disable Multithreaded, [40](#)
- Enable/Disable Reverse Execution, [40](#)
- New Session From Script, [39](#)
- Open Checkpoint, [40](#)
- Run Forward, [40](#)
- Run Reverse, [40](#)
- Save Checkpoint, [40](#)
- Stop, [40](#)

 Tools menu, [45](#)

- Command Browser, [45](#)
- Command Line Window, [45](#)
- Configuration Browser, [45](#)
- Hap Browser, [45](#)
- Memory Mappings Browser, [45](#)
- New Statistics Plot, [45](#)
- Object Browser, [45](#)
- Statistics Plot, [45](#)

 trace-cr, [34](#)
 trace-exception, [35](#)
 trace-io, [33](#)
 tracing, [32](#)
 tutorial, [16](#)

U

underscores

- in numbers, [65](#)

 Unstep, [54](#)

V

Virtual Board, [14](#)
 Virtual System, [14](#)
 Virtualized Systems Development, [6](#), [10](#)
 Virtutech Website, [46](#)
 VSD, [6](#)

W

Window menu, [46](#)

- Close All Consoles, [46](#)
- Console list, [46](#)
- Minimize All Consoles, [46](#)
- Open All Consoles, [46](#)
- Restore All Consoles, [46](#)

 windows

- command line, [50](#)
- configuration browser, [60](#)
- control window, [39](#)
- cpu registers, [52](#)
- device registers, [52](#)
- disassembly, [54](#)
- graphics console, [49](#)
- hap browser, [57](#)
- help browser, [58](#)
- memory contents, [52](#)
- memory mappings browser, [63](#)
- object browser, [58](#)
- preference window, [64](#)
- source view, [68](#)
- stack trace, [68](#)
- statistics plot, [62](#)
- text console, [47](#)
 - resizing, [49](#)

 workspace, [16](#), [37](#)

- change, [42](#)
- create, [42](#)



Virtutech, Inc.

2001 Gateway Place
Suite 201E
San Jose, CA 95110
USA

Phone +1 408-392-9150
Fax +1 408-608-0430

<http://www.virtutech.com>