

Wind River® Simics®

GETTING STARTED

4.6

<i>Revision</i>	4081
<i>Date</i>	2012-11-16

Copyright © 2010–2012 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, Simics, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. The Wind River logo is a trademark of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:
www.windriver.com/company/terms/trademark.html

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location:
`installDir/LICENSES-THIRD-PARTY/`.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters

Wind River
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

Toll free (U.S.A.): 800-545-WIND
Telephone: 510-748-4100
Facsimile: 510-749-2010

For additional contact information, see the Wind River Web site:
www.windriver.com

For information on how to contact Customer Support, see:
www.windriver.com/support

Contents

1	Introduction	5
1.1	Who Should Read This Document?	5
1.2	Conventions	5
2	What is Simics	6
2.1	Simics Hindsight	6
2.2	Simics Analyzer	7
2.3	Simics Accelerator	7
2.4	Simics Virtual Board/System	7
2.5	Simics Model Builder	8
2.6	Simics Extension Builder	8
2.7	Simics Ethernet Networking	8
2.8	Simics CPU Core Types	8
2.9	Simics Device Model Libraries	9
3	Tutorial	10
3.1	Starting Simics	10
3.2	Running the Simulation	12
3.3	Checkpointing	15
3.4	Controlling Simulation Speed	16
3.5	Running the Simulation Backward	16
3.6	Getting Files into a Simulated System	18
3.7	Debugging	19
3.8	Tracing	27
3.9	Scripting	30
4	Simics Hindsight User Interface	32
4.1	Overview of Simics Graphical User Interface	32
4.2	Simics Control Window	34
4.2.1	Toolbar	34
4.2.2	File Menu	36
4.2.3	Edit Menu	38
4.2.4	Run Menu	38
4.2.5	Debug Menu	39
4.2.6	Tools Menu	40

4.2.7	Window Menu	41
4.2.8	Help Menu	42
4.3	Console Window	42
4.3.1	Text Console	43
4.3.2	Graphics Console	43
4.4	Command Line Window	45
4.5	CPU Registers Window	46
4.6	Device Registers Window	46
4.7	Memory Contents Window	46
4.8	Disassembly Window	48
4.9	Hap Browser	51
4.10	Help Browser	52
4.11	Object Browser	52
4.12	Configuration Browser	54
4.13	Statistics Plot	56
4.14	Memory Mappings Browser	57
4.15	Preference Window	58
	Appearance Panel	58
	Startup Panel	59
	Advanced Panel	62
4.16	Source View Window	62
4.17	Stack Trace Window	63
5	Next Steps	64
	Index	65

Chapter 1

Introduction

This document provides a quick introduction to new Simics users. It describes the concepts and benefits of Virtualized Systems Development for developing software, and the products in the Simics family. It also provides a tutorial and a short presentation of the rest of the Simics documentation.

1.1 Who Should Read This Document?


This document is targeted at new Simics users who will be using the native Simics interface. Simics users interested in the Simics Eclipse environment should probably refer to the *Eclipse User's 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

What is Simics

Wind River Simics is a fast, functionally-accurate, full system simulator. Simics creates a high-performance virtual environment in which any electronic system – from a single board to complex, heterogeneous, multi-board, multi-processor, multicore systems – can be defined, developed and deployed.

Simics enables companies to adopt new approaches to the product development life cycle resulting in dramatic reduction in project risks, time to market, and development costs while also improving product quality and engineering efficiency. Simics allows engineering, integration and test teams to use approaches and techniques that are simply not possible on physical hardware.

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 Wind River directly for the licensing terms of each product.

2.1 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.

Included with Simics Hindsight is one such virtual board, the Quick Start Platform (QSP). The QSP is a simple virtual platform that includes a limited set of synthetic, simulation-only

devices. A QSP can be used to get started with Simics and familiarize oneself with its features, and it can also be used to develop user-level application software with full access to all Simics features. It cannot be used to run real world firmware, device drivers, or BSPs, since its device models do not correspond to any real machine.

2.2 Simics Analyzer

Simics Analyzer is an add-on product to Simics Hindsight. It provides analysis and debugging tools for software applications running on the target system.

The foundation is an OS awareness system which provides a view of the software running on the target system. It provides a view of the target software, complete with kernels, processes, tasks and threads.

On this foundation Analyzer adds three major tools: a full system process list, a system execution timeline and a tool to perform code coverage.

The full system process list and system execution timeline are Eclipse views. The process list provides a snapshot view of the software running on the target system, while the system execution timeline records which software has run when and where on the system. The graphical timeline view in Eclipse presents this information for easy access and analysis.

The code coverage tool provides statement level code coverage for programs running on the target. This is presented as HTML for easy viewing or in a text based format for easy processing by scripts.

2.3 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.

2.4 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”.

Wind River provides a number of off-the-shelf Virtual Boards/Systems (including the QSP that comes with Simics Hindsight). However, more commonly, customers build their own virtual boards using Simics Model Builder.

2.5 Simics Model Builder

Simics Model Builder is an optional add-on product that allows users to create, modify and configure virtual boards and systems that run on the Simics platform. Simics Model Builder contains the DML tool, which contains a compiler for the DML language. 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.

2.6 Simics Extension Builder

Simics Extension Builder is an optional add-on product that complements Simics Model Builder and provides the means for users to develop simulator feature extensions, such as trace modules, real-world connections, statistics collectors and foreign processor models.

2.7 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.

2.8 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 Wind River, 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.

2.9 Simics Device Model Libraries

Simics Device libraries are models of commercial-off-the-shelf (COTS) devices that Wind River 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 3

Tutorial

This tutorial describes how to perform some common tasks with Simics. The example system we will use is called *MPC8641-Simple*. It is based on the Freescale's MPC8641 HPCN platform, although not all devices have been modeled. The *MPC8641-Simple* system is a simulated Power Architecture MPC8641D board, with two e600 cores, running a very small Linux installation.

In order to follow this tutorial, your Simics installation needs to include the Firststeps add-on package, which contains the *MPC8641-Simple* system. If the Firststeps add-on is not yet part of your Simics installation, the *Installation Guide* will provide instructions on how to install it. The Firststeps package is usually available to all customers, and you should be able to download it in the same way you downloaded Simics Hindsight.

If you get licensing errors when trying to follow this tutorial, please ask your Wind River representative for a new license, or post a request on the Simics support forums.

3.1 Starting Simics

Let us first show you how to start Simics, create a workspace and run a simulation.

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

As you start Simics for the first time, a dialog box will pop up (figure 3.1), asking you where to create a *workspace*. A Simics installation can be shared among many users, and 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 just pick a directory and let Simics make a workspace out of it.

As a first-time user, Simics will also propose you to register on the Simics support forums. You may register or skip this step, although you should keep in mind that the forums are the best place to ask for help and get answers from the Simics support team.

At this point, Simics is running. The *Simics Control* window has opened (figure 3.2). It 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

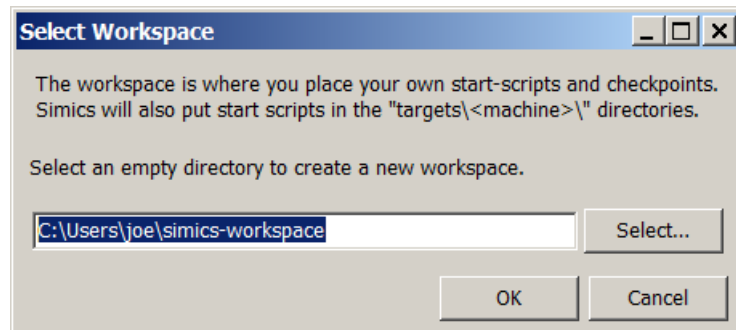


Figure 3.1: Create a Workspace

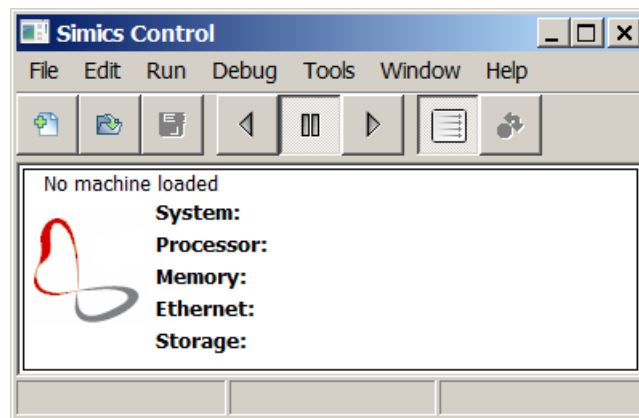


Figure 3.2: Simics Control Window

tutorial you need the *Simics Command Line* window. Select **Tools** → **Command Line Window** to open it.

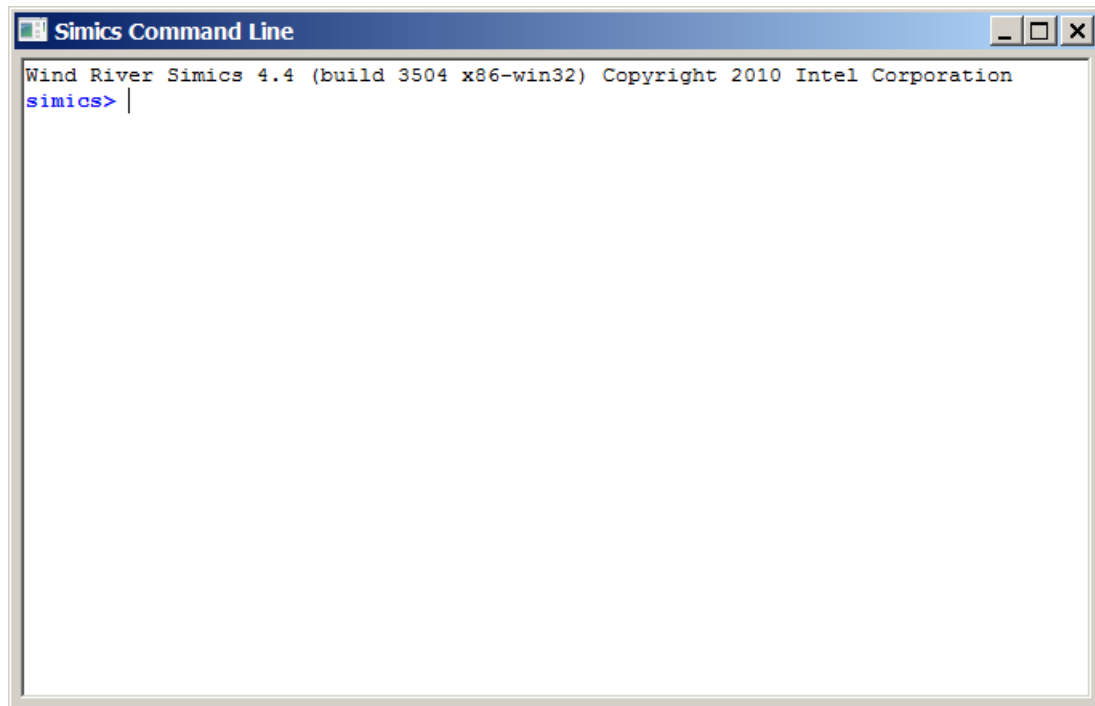



Figure 3.3: Simics Command Line Window

The *Command Line* window allows you to interact with Simics via a text prompt. 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 the command-line interface (CLI). Several of the features demonstrated in this guide make use of CLI, as you will see later on.

Create a new session with the *MPC8641 firststeps* target machine by clicking on  **New session from script**. Open the `mpc8641-simple` directory and choose the file `firststeps.simics`. After a few seconds, the *Simics Control* window will show you a summary of the simulated system you have just loaded, and a new window should have popped up, called *Serial Console on mpc8641d_simple.soc.uart[0]* (see figure 3.4).

This new window is part of the simulation: it is connected to the serial port of the simulated *MPC8641D-Simple* 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.

3.2 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. You can start the

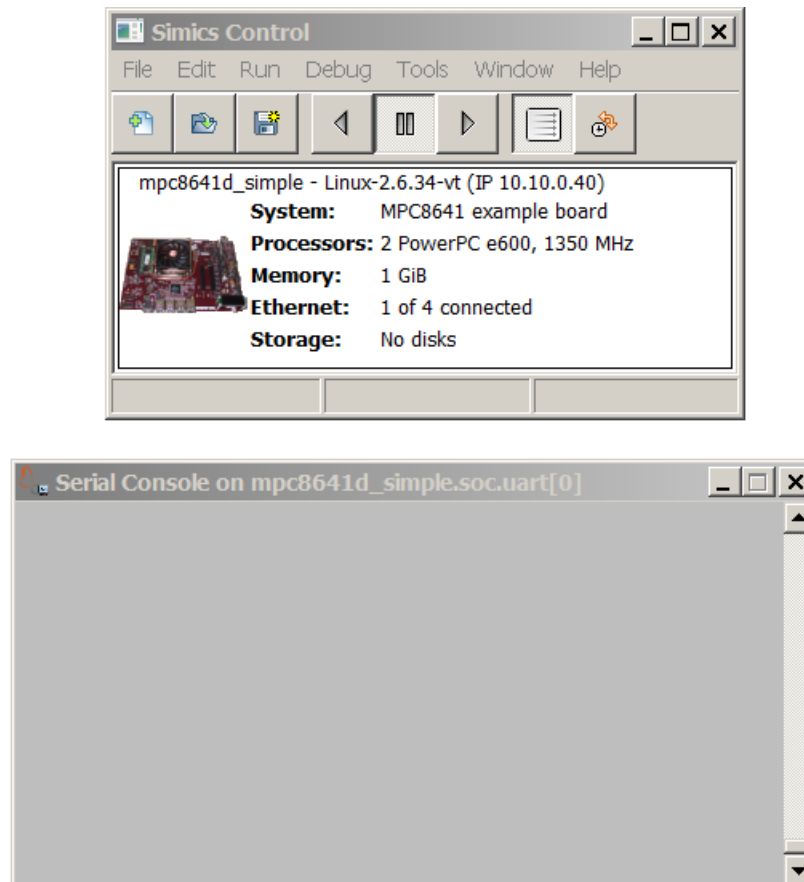


Figure 3.4: The MPC8641D target system is loaded

simulation by clicking on ► **Run Forward**, or by entering the command **continue** or **c** at the prompt.

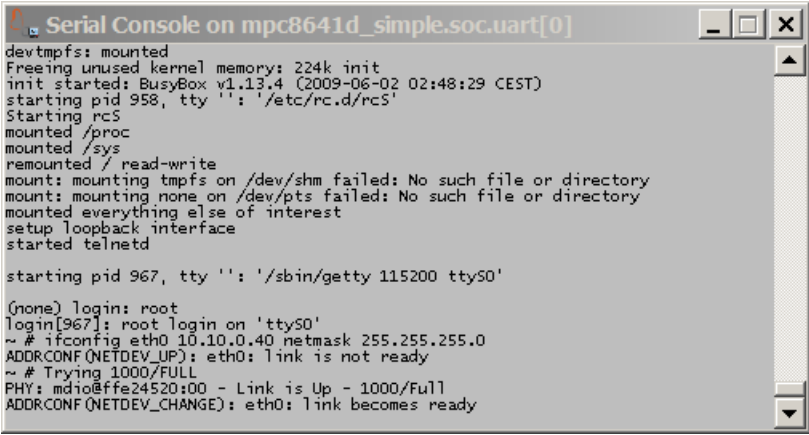
A short side note: in all Simics documentation, interaction with Simics in the CLI window (figure 3.3) is presented in monospace font. User input will be presented in **bold font**.

```
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, the boot messages will start in the serial console. The simulation can be stopped at any time by entering the **stop** command or clicking on ■ **Stop** in the toolbar.



```
Serial Console on mpc8641d_simple.soc.uart[0]
devtmpfs: mounted
Freeing unused kernel memory: 224k init
init started: BusyBox v1.13.4 (2009-06-02 02:48:29 CEST)
starting pid 958, tty \": '/etc/rc.d/rcS'
Starting rcS
mounted /proc
mounted /sys
remounted / read-write
mount: mounting tmpfs on /dev/shm failed: No such file or directory
mount: mounting none on /dev/pts failed: No such file or directory
mounted everything else of interest
setup loopback interface
started telnetd

starting pid 967, tty \": '/sbin/getty 115200 ttyS0'
(none) login: root
login[967]: root login on 'ttyS0'
~ # ifconfig eth0 10.10.0.40 netmask 255.255.255.0
ADDRCONF (NETDEV_UP): eth0: link is not ready
~ # Trying 1000/FULL
PHY: mdio@ffe24520:00 - Link is Up - 1000/Full
ADDRCONF (NETDEV_CHANGE): eth0: link becomes ready
```

Figure 3.5: A booted Linux system

After some time, 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 in the simulated system:

```
~ # cat /proc/cpuinfo
processor      : 0
cpu           : 7448, altivec supported
clock        : 1336.500000MHz
```

```

revision      : 0.0 (pvr 8004 0010)
bogomips      : 148.50

processor      : 1
cpu           : 7448, altivec supported
clock         : 1336.500000MHz
revision      : 0.0 (pvr 8004 0010)
bogomips      : 148.50

total bogomips : 297.00
timebase      : 74250000
platform      : MPC86xx HPCN
model         : Virtutech MPC8641-simple
Vendor        : Freescale Semiconductor
SVR           : 0x80900100
Memory        : 1024 MB
~ # ls /
bin           home      lib        proc        sys
dev           host      linuxrc    root        usr
etc           init      lost+found sbin        var
~ #

```

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

3.3 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, to be loaded at a later time. To save a checkpoint:

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

```



running> stop
simics>

```

2. Click on  **Save Checkpoint**. Press the **Up** button to get back to the workspace directory itself. Name the checkpoint `after_boot.ckpt` and press **Save**.

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 serial console.

You can read more about configuration and checkpointing in the *Hindsight User's Guide*.

3.4 Controlling Simulation Speed


The simulation speed of Simics depends a lot on what software it simulates. In this example, the operating system, once booted, is mostly idle, causing the simulation to become very fast. To avoid having virtual time progress too quickly, you can activate the real-time mode feature:

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

This will cause the virtual time never to progress *faster* than the real time. It will also reduce Simics' processor usage. See **help enable-real-time-mode** for further information on this topic. You can read more about performance in the *Simulation Performance* chapter in the *Hindsight User's Guide*.

3.5 Running the Simulation Backward

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

To enable reverse execution, just click on  **Enable/Disable Reverse Execution**. What this button does behind the scenes 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 offers, we will remove an important file on the simulated system. Enter the following commands in the serial console:


```

~ # rm /bin/ls
~ # ls /
-sh: ls: not found

```

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

To prove to ourselves that reversing actually works, we can 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]
mpc8641d_simple.soc.cpu[0] 10604631208 10604631208    7.855
simics>

```

In the example above, the simulation has run for about 10 virtual seconds. Now we will jump back to the bookmark we just created, using the command `skip-to`. `skip-to` will not run the simulation backward, but will jump directly to the bookmark.

```

simics> skip-to bookmark = start
[mpc8641d_simple.soc.cpu[0]] v:0xc000f238 p:0x00000f238 mtmsr r7
simics> ptime
processor                steps          cycles  time [s]
mpc8641d_simple.soc.cpu[0] 2485120854 2485120854    1.841
simics>

```

As shown above, we are now about 6 seconds back in the past.

It is also possible to *run* the simulation backward. You can click on **Run Reverse** and you will quickly find yourself back at 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 towards the past, Simics will do all sort of useful things like triggering breakpoints, and this will prove very practical when debugging.

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

```

simics> c

```

When you type something in the terminal, you will notice that it does not respond any longer! Instead, you will see the commands being replayed, as we typed them before. 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 on the serial console:

```
~ # ls /
bin          etc          host          linuxrc       proc          sys
dev          home         lib           lost+found    sbin          var
~ #
```

The **ls** command is back! You can read more about reverse execution in the *Hindsight User's Guide*.

3.6 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, or in other words, from the host to the target. Simics provides a way to directly access the host filesystem from the simulated machine, called *SimicsFS*.

SimicsFS is a kernel filesystem module (available for simulated Linux and Solaris) that talks to a simulated pseudo-device. Our MPC8641-Simple machine is equipped with SimicsFS support; just mount `/host` to access it.

```
~ # mount /host
[simicsfs] mounted
~ # ls /host
$Recycle.Bin
Documents and Settings
Drivers
MININT
MSOCCache
PerfLogs
Program Files
ProgramData
Recovery
System Volume Information
Users
Windows
[...]
~ #
```

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

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 *MPC8641-Simple* target machine. The file is located in the Firststeps add-on package.

```
~ # cd /host/Program\ Files/Simics/Simics\ 4.4/
Firststeps\ 4.4.0/targets/mpc8641-simple/images
/host/Program\ Files/Simics/Simics\ 4.4/Firststeps\ 4.4.0/
targets/mpc8641-simple/images # cp debug_example ~
/host/Program\ Files/Simics/Simics\ 4.4/Firststeps\ 4.4.0/
targets/mpc8641-simple/images # cd
~ # ls
debug_example
~ #
```

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/simics/simics-4.4/simics-firststeps-4.4.0/targets/mpc8641-simple/images/`

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

```
~ # umount /host
```

You can read more about SimicsFS and other ways to transfer files in the *Hindsight User's Guide*.

3.7 Debugging

Note: This part of the tutorial relies on features provided by the Simics Analyzer product.

This section demonstrates some of the source-level debugging facilities that Simics provides. The Firststeps add-on package includes an example code snippet called `debug_example.c`, located in the `targets, mpc8641-simple` directory. It contains the source code of the program we are going to debug. The intended purpose of `debug_example` is to print some information on the users of the system.

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

```
~ # ./debug_example
[...]
Got segmentation fault!
```

```
~ #
```

This output indicates that our program crashed. Let us use Simics features to debug it. We will need to use Simics' *OS awareness* to make sure we are debugging the right program, since Simics simulates the whole system, including task switches and whatever the OS is scheduling. We will also need symbolic debugging to understand what is going on at the source code level.

Let us first activate symbolic debugging. The binary executable `debug_example` we copied above is compiled with symbol information, so we will just create a **`syntable`** object to access it from Simics:

```
simics> new-syntable file = "C:\\Program Files\\Simics\\Simics-4.4\\p
Firststeps 4.4.0\\targets\\mpc8641-simple\\images\\debug_example"
Created symbol table 'debug_example'
ABI for debug_example is ppc-elf-32
debug_example set for context mpc8641d_simple.cell_context
simics>
```

We now have a **`debug_example`** object with all debugging information.

Next, we will utilize the Simics *OS Awareness* system, which will allow us to associate a *context object* with a specific instance of `debug_example` when it is executing.

Note that all usage of the OS Awareness system here requires an already configured tracker system. For more details refer to the section *Target Software Tracking* in the *Analyzer User's Guide*.

```
simics> mpc8641d_simple.software.track node = debug_example syntable = debug_example
[mpc8641d_simple.software.tracker info] enabling the tracker
Context debug_example0 will start tracking debug_example when it starts
simics>
```


We got back a context object called **`debug_example0`**. We can use it to set breakpoints on `debug_example`. The process tracker will make sure this context is active every time the operating system schedules `debug_example` on a processor. We can check that the process tracker is working by asking for the current list of processes (kernel not included):

```
simics> mpc8641d_simple.software.list
Process      Binary  PID  TID
kthreadd          2
migration/0       3
ksoftirqd/0       4
watchdog/0        5
migration/1       6
ksoftirqd/1       7
watchdog/1        8
```

```

events/0          9
events/1         10
khelper          11
async/mgr        14
sync_supers      98
bdi-default     100
kblockd/0       102
kblockd/1       103
kseriod         113
rpciod/0        132
rpciod/1        133
khungtaskd      159
kswapd0         160
aio/0           208
aio/1           209
nfsiod          217
crypto/0        223
crypto/1        224
kjournald       957
flush-1:0       967
init             1    1
sh              974   974
httpd           973   973
telnetd         971   971
simics>

```

Last, we need to make sure that reverse execution is enabled so we can use it for debugging. Click on on  **Enable/Disable Reverse Execution** if it is not active already.

Let us start the simulation again, but this time, using the context object:

```
simics> debug_example0.run-until-activated
```

In the Firststeps machine, we start `debug_example`:

```
~ # ./debug_example
```

Simics will stop as soon as the **debug_example** context becomes active:

```

[mpc8641d_simple.software info] Context debug_example0 is now tracking node 45
[mpc8641d_simple.soc.cpu[1]] v:0x48015e40 <whole instruction not mapped>
debug_example0 is now current context on mpc8641d_simple.soc.cpu[1]
simics>

```

We end up at the page fault caused by the first instruction execution of `debug_example`. However, we care little about debugging the Linux page fault handler, so let us skip directly to the program itself. We set a breakpoint on `main()` and run until it hits:

```
simics> break (pos main) -x
Breakpoint 232 set on address 0x10000650 in 'debug_example0' with access mode 'x'
simics> c
Breakpoint 232 on instruction fetch from 0x10000650 in debug_example0.
[mpc8641d_simple.soc.cpu[1]] v:0x10000650 p:0x03fca0650 stwu r1,-64(r1)
main (argc=0, argv=0x100ad700) at /tmp/debug_example.c:53
53      (file /tmp/debug_example.c not found)
```

Simics had now stopped at the beginning of `main()`, but complains that it can not find `/tmp/debug_example.c`. This is where the original file was when it was compiled, and thus what is built in the symbol information. Generally, if you compile your programs on the same host you run the simulation, this mismatch will never happen. In our case, we can fix it easily by telling Simics where to look instead.

Open the *Source View* window in the **Debug** menu. There, in the *Source File* item, click on **Find** and point at the correct location of `debug_example.c` (in the *FirstSteps* package, in `targets\mpc8641-simple\`). There is now a small green arrow pointing at the current execution location in the Source View, with the correct source code.

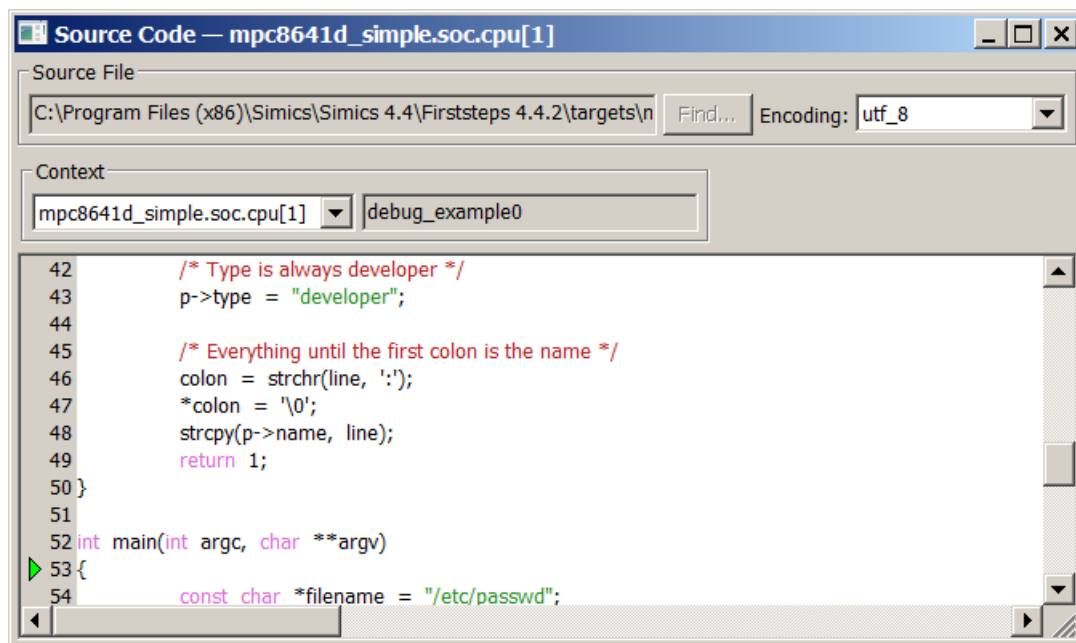
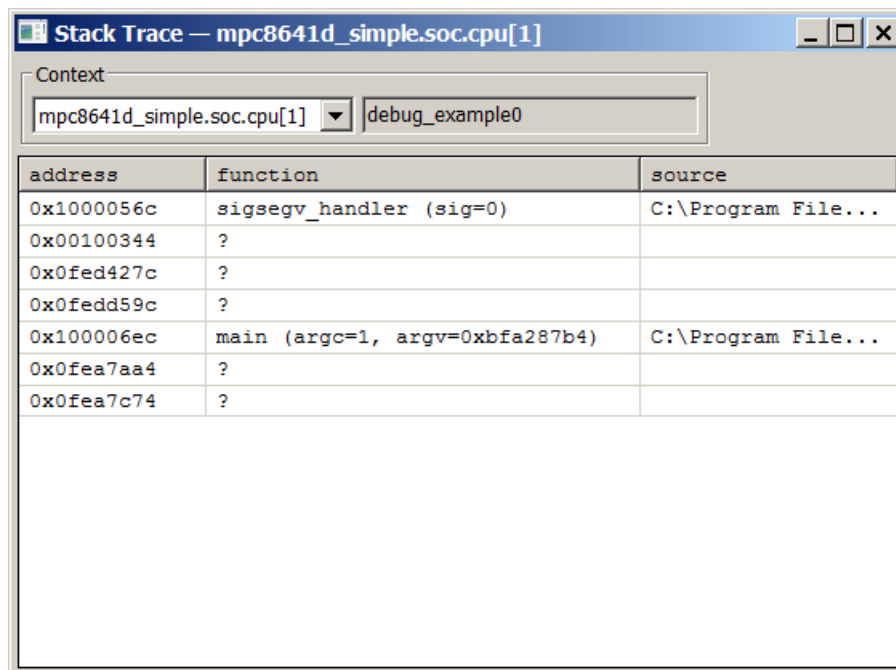


Figure 3.6: Source Code View

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 233 set on address 0x1000056c in 'debug_example0' with access mode 'x'
simics>
```



address	function	source
0x1000056c	sigsegv_handler (sig=0)	C:\Program File...
0x00100344	?	
0x0fed427c	?	
0x0fedd59c	?	
0x100006ec	main (argc=1, argv=0xbfa287b4)	C:\Program File...
0x0fea7aa4	?	
0x0fea7c74	?	

Figure 3.7: Stack trace when the segmentation fault was caught

Resume the simulation, which will stop at the signal handler. Now open the *Stack Trace* via **Debug** → **Stack trace** (figure 3.7). 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 3.8).

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 backward until the previous known source line is reached.

```
simics> reverse-step-line
End of input recording.
[mpc8641d_simple.soc.cpu[1]] v:0x100006e8 p:0x03fca06e8 bl 0x100008d0
main (argc=1, argv=0xbfdf2f84) at ./debug_example.c:70
70                                     printf("Type: %s\n", user.type);
```

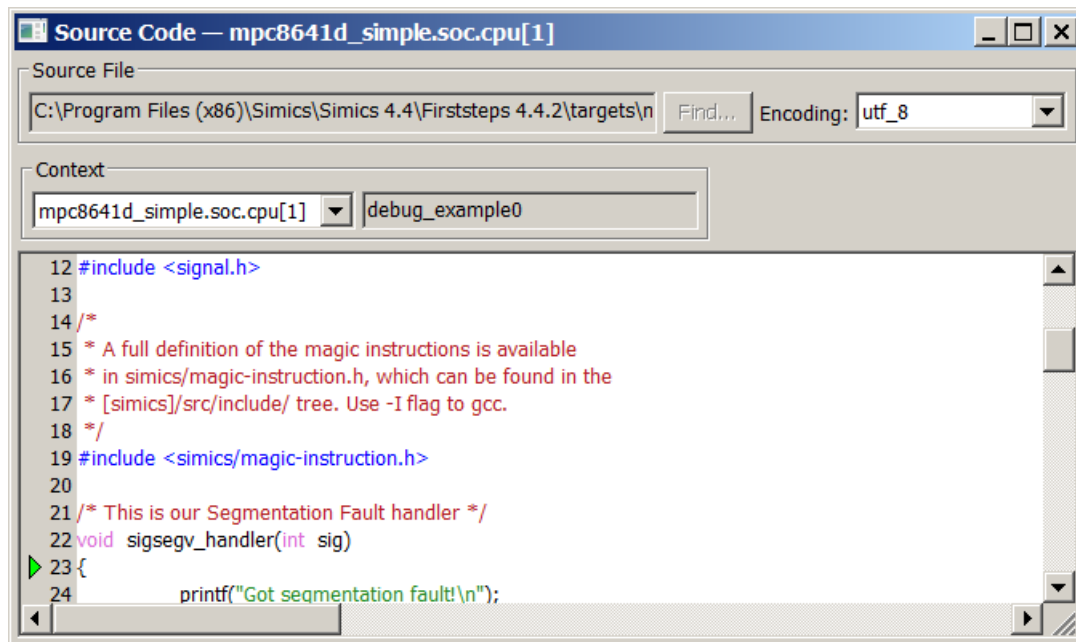


Figure 3.8: Source view when the segmentation fault was caught

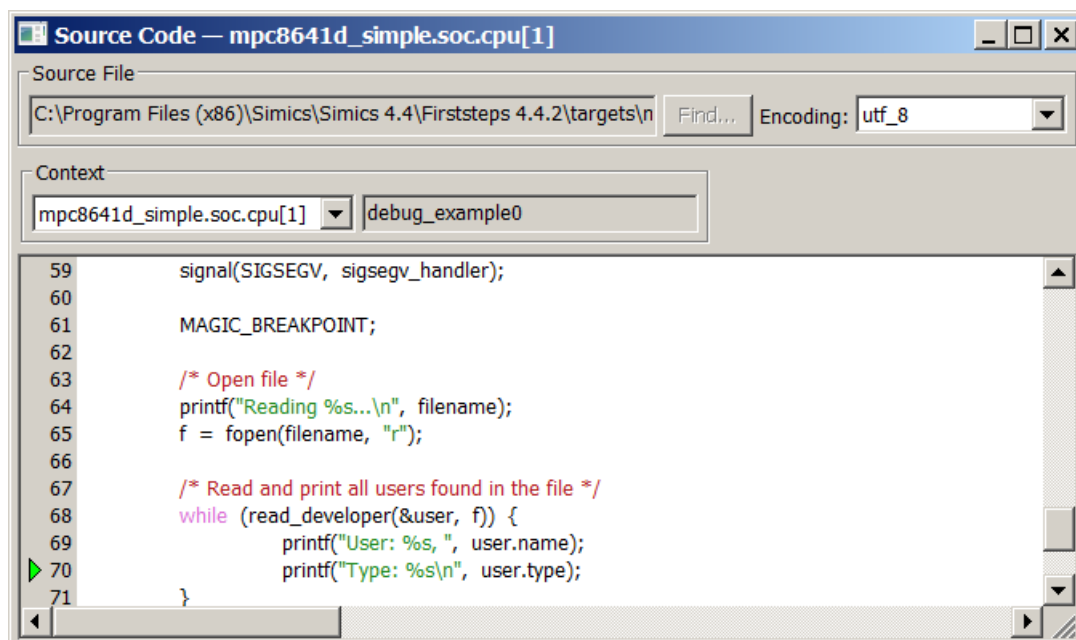


Figure 3.9: Line in source code causing the crash

This line caused the crash (see also figure 3.9). Let us examine what *user.type* contains:

```
simics> psym user.type
(char *) 0x9d4 (unreadable)
simics> psym user
{name = 0xbfdf2ce0 "shutdown", type = (char *) 0x9d4 (unreadable)}
simics>
```

As you can see, the *type* member points to an unreadable address with a very suspicious look. 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")
Breakpoint 466 set on address 0xbfdf2ce8 in 'debug_example0', length 4 with access mode 'w'
simics> reverse
Breakpoint 466 on write to 0xbfdf2ce8 in debug_example0.
[mpc8641d_simple.soc.cpu[1]] v:0x0ff0bda4 p:0x03fe10da4 beqlr
simics>
```

Now, examine the stack trace:

```
simics> stack-trace
#0 0xff0bda4 in ?? ()
#1 0x10000628 in read_developer (p=0xbfdf2ce0, f=0x10011008)
    at ./debug_example.c:48
#2 0x100006fc in main (argc=1, argv=0xbfdf2f84) at ./debug_example.c:68
#3 0xfea7aa4 in ?? ()
#4 0xfea7c74 in ?? ()
#5 0x0 in ?? ()
simics>
```

A call from *read_developer()* at line 48, has caused the pointer to be corrupted. Switch to that frame and display the code being run.

```
simics> frame 1
#1 0x10000628 in read_developer (p=0xbfb0ae60, f=0x10011008)
    at [...] \targets\mpc8641-simple\debug_example.c:48
simics> list read_developer 15
36 {
37     char line[100], *colon;
38
39     if (fgets(line, 100, f) == NULL)
40         return 0;        /* end of file */
41
```

```

42         /* Type is always developer */
43         p->type = "developer";
44
45         /* Everything until the first colon is the name */
46         colon = strchr(line, ':');
47         *colon = '\\0';
48         strcpy(p->name, line);
49         return 1;
50     }
simics>

```

We can also use the *Stack Trace* window to select the frame; doing so will update the *Source Code* window as well (figure 3.10).

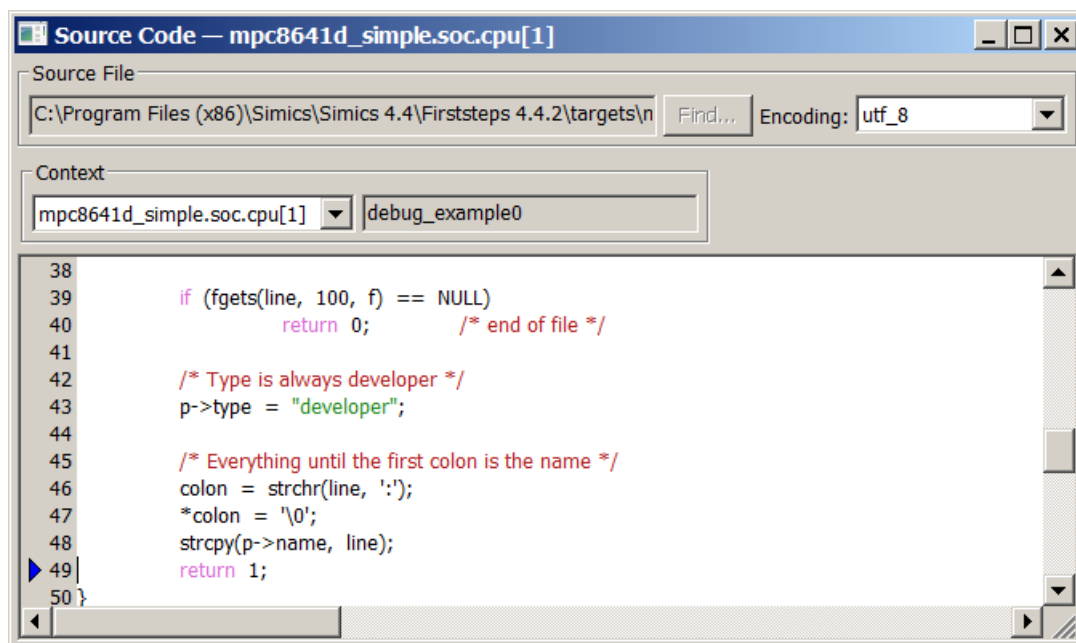


Figure 3.10: Call to *strcpy()*

On line 48, 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 = 0xbfdf2ce0 "shutdown", type = (char *) 0x9d4 (unreadable)}
simics> reverse-step-instruction

```

```
[mpc8641d_simple.soc.cpu[1]] v:0x0ff0bda4 p:0x03fe10da4 beqlr
Breakpoint 466 on write to 0xbfdf2ce8 in debug_example0.
[mpc8641d_simple.soc.cpu[1]] v:0x0ff0bda0 p:0x03fe10da0 stb r0,4(r5)
simics> frame 1; psym "*p"
#1 0x10000628 in read_developer (p=0xbfb0ae60, f=0x10011008)
    at [...] \targets\mpc8641-simple\debug_example.c:48
{name = 0xbfb0ae60 "shutdown\020", type = (char *) 0x100009d4 "developer"}
simics>
```

You can read more about debugging in the *Hindsight User's Guide*.

3.8 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 `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 *mpc8641-simple*. We execute 5000 instructions without tracing first to reach a sequence of instructions that includes memory accesses:

```
simics> c 5000
[mpc8641d_simple.soc.cpu[0]] v:0xeff04a54 p:0x0eff04a54 xor r11,r11,r9
simics> trace0.start
Tracing enabled. Writing text output to standard output.
simics> c 10
inst: [      1] CPU  0 <v:0xeff04a54> <p:0x0eff04a54> 7d6b4a78 xor r11,r11,r9
inst: [      2] CPU  0 <v:0xeff04a58> <p:0x0eff04a58> 7d600278 xor r0,r11,r0
inst: [      3] CPU  0 <v:0xeff04a5c> <p:0x0eff04a5c> 540015ba rlwinm r0,r0,2,22,29
inst: [      4] CPU  0 <v:0xeff04a60> <p:0x0eff04a60> 7d27002e lwzx r9,r7,r0
data: [      1] CPU  0 <v:0xeff4b248> <p:0x0eff4b248> Read  4 bytes 0x58684c11
inst: [      5] CPU  0 <v:0xeff04a64> <p:0x0eff04a64> 556bc23e srwi r11,r11,8
inst: [      6] CPU  0 <v:0xeff04a68> <p:0x0eff04a68> 88080004 lbz r0,4(r8)
data: [      2] CPU  0 <v:0xeff60100> <p:0x0eff60100> Read  1 bytes 0xff
inst: [      7] CPU  0 <v:0xeff04a6c> <p:0x0eff04a6c> 7d295a78 xor r9,r9,r11
inst: [      8] CPU  0 <v:0xeff04a70> <p:0x0eff04a70> 7d200278 xor r0,r9,r0
inst: [      9] CPU  0 <v:0xeff04a74> <p:0x0eff04a74> 540015ba rlwinm r0,r0,2,22,29
inst: [     10] CPU  0 <v:0xeff04a78> <p:0x0eff04a78> 7d67002e lwzx r11,r7,r0
data: [      3] CPU  0 <v:0xeff4b210> <p:0x0eff4b210> Read  4 bytes 0xabd13d59
```

```
[mpc8641d_simple.soc.cpu[0]] v:0xeff04a7c p:0xeff04a7c srwi r9,r9,8
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.

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 mpc8641d_simple.soc.uart[0]
simics> c 72_500
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4501 SZ 1 0x0 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4503 SZ 1 0x83 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4500 SZ 1 0x0 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4501 SZ 1 0x0 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4503 SZ 1 0x3 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4504 SZ 1 0x3 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4502 SZ 1 0x7 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4503 SZ 1 0x83 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4500 SZ 1 0xa1 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4501 SZ 1 0x0 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4503 SZ 1 0x3 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Read from mpc8641d_simple.soc.cpu[0]: PA 0x4505 SZ 1 0x60 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4500 SZ 1 0xd (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Read from mpc8641d_simple.soc.cpu[0]: PA 0x4505 SZ 1 0x60 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4500 SZ 1 0xa (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Read from mpc8641d_simple.soc.cpu[0]: PA 0x4505 SZ 1 0x60 (BE)
[mpc8641d_simple.soc.uart[0] trace-io] Write from mpc8641d_simple.soc.cpu[0]: PA 0x4500 SZ 1 0xd (BE)
[mpc8641d_simple.soc.cpu[0]] v:0xeff25910 p:0xeff25910 lwz r9,-32768(r30)
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 mpc8641d_simple.soc.uart[0]
simics> trace-cr -all
```

```

simics> c 6_000_000
[mpc8641d_simple.soc.cpu[0] trace-cr] msr <- 0x1030
[mpc8641d_simple.soc.cpu[0] trace-cr] dec <- 0x1220a
[mpc8641d_simple.soc.cpu[0] trace-cr] msr <- 0x9030
[mpc8641d_simple.soc.cpu[0] trace-cr] sprg0 <- 0x0
[mpc8641d_simple.soc.cpu[0] trace-cr] sprg1 <- 0x1
[mpc8641d_simple.soc.cpu[0] trace-cr] sprg2 <- 0x0
[mpc8641d_simple.soc.cpu[0] trace-cr] srr0 <- 0xff95100
[mpc8641d_simple.soc.cpu[0] trace-cr] srr1 <- 0x1032
[mpc8641d_simple.soc.cpu[0] trace-cr] dec <- 0x1220a
[mpc8641d_simple.soc.cpu[0] trace-cr] msr <- 0x1032
[mpc8641d_simple.soc.cpu[0] trace-cr] srr0 <- 0xff95360
[mpc8641d_simple.soc.cpu[0] trace-cr] srr1 <- 0x9030
[mpc8641d_simple.soc.cpu[0]] v:0x0ff9533c p:0x00ff9533c cmpw r3,r5
simics>

```

We can single-step with the `-r` flag, to see what registers each instruction changes. You can also open the *CPU Registers* window: register changes will be highlighted in yellow as instructions are executed.

```

simics> untrace-cr -all
simics> step-instruction -r 10
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95340 p:0x00ff95340 bne+ 0xff95330
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95344 p:0x00ff95344 blr
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95360 p:0x00ff95360 subfc r4,r4,r7
r4 <- 615481
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95364 p:0x00ff95364 subfe. r3,r3,r6
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95368 p:0x00ff95368 bge+ 0xff9535c
[mpc8641d_simple.soc.cpu[0]] v:0x0ff9535c p:0x00ff9535c bl 0xff95330
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95330 p:0x00ff95330 mftbu r3
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95334 p:0x00ff95334 mftbl r4
r4 <- 337452
[mpc8641d_simple.soc.cpu[0]] v:0x0ff95338 p:0x00ff95338 mftbu r5
[mpc8641d_simple.soc.cpu[0]] v:0x0ff9533c p:0x00ff9533c cmpw r3,r5
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 time-stamp:

```

simics> trace-exception System_call
simics> c
[mpc8641d_simple.soc.cpu[0] trace-exception] {mpc8641d_simple.soc.cpu[0] 0xc0013f18 278855516}
Exception 14: System_call
[...]
[mpc8641d_simple.soc.mcm info] {mpc8641d_simple.soc.cpu[0] 0xc049aa6c 277937920} Enabling core 1
running> stop
[mpc8641d_simple.soc.cpu[0]] v:0xc038d88c p:0x00038d88c cmpwi cr7,r0,0
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-**.

3.9 Scripting

The Simics command line has some built-in scripting capabilities. When that is not enough, Python can be used instead. Restart Simics with `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.mpc8641d_simple.soc.cpu[0])
simics> @msr = conf.mpc8641d_simple.soc.cpu[0].msr
simics> @while conf.mpc8641d_simple.soc.cpu[0].msr == msr: SIM_continue(1)
[...]
simics> @end_cycle = SIM_cycle_count(conf.mpc8641d_simple.soc.cpu[0])
simics> @print "Executed", end_cycle - start_cycle, "instructions"
Executed 444 instructions
simics>
```

After you enter `@while conf.mpc8641d_simple.soc.cpu[0].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 the *Hindsight User's Guide*. The full description of the Simics API is available in the *Model Builder Reference Manual* and *Extension Builder Reference Manual*.

Chapter 4

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 *Installation Guide*.

4.1 Overview of Simics Graphical User Interface

Figure 4.1 shows the windows opened by Simics when opening a simulation of a system.

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 4.2).

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 4.4).

Serial Console

This is an output window of the simulated system, a text serial console connected to the serial port of the system. This is commonly used as output for systems that are not using graphical output.

Graphics Console

This window is available if the system has graphical output. It is connected to the graphics device of the system. This will be the display used if you run Windows™ or other graphical operation systems in your simulation.

4.1. Overview of Simics Graphical User Interface

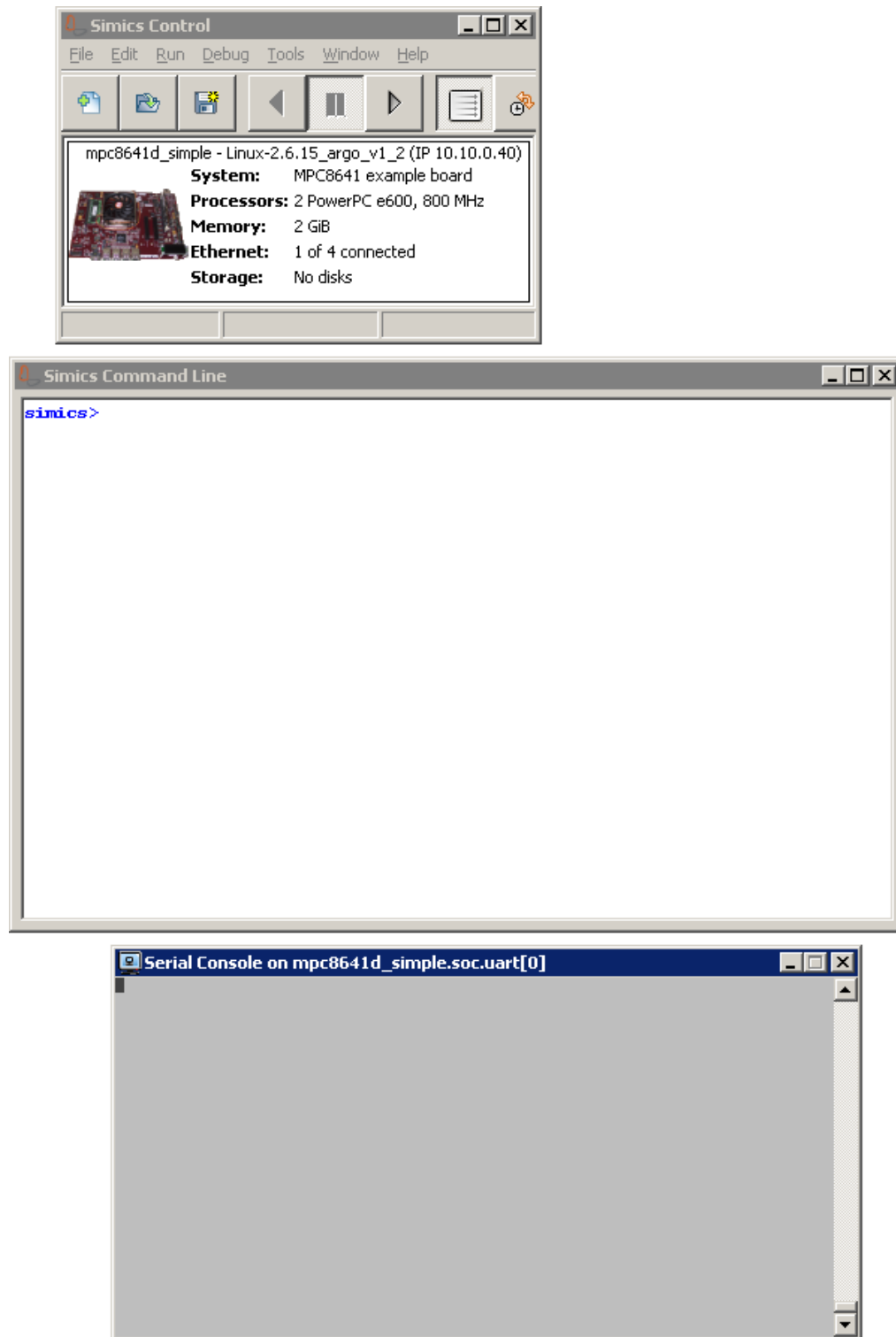


Figure 4.1: A Simics Simulation Session

Note: 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 4.3).

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

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.

4.2 Simics Control Window

The *Simics Control* window (the first window in figure 4.1) contains the toolbar and menus for Simics. It also presents information about the active simulation.



Figure 4.2: Running a Simulation

When the simulation is running, the title of the control window will change to *Simics Control: Running* (see figure 4.2). 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.

4.2.1 Toolbar



Figure 4.3: 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**.

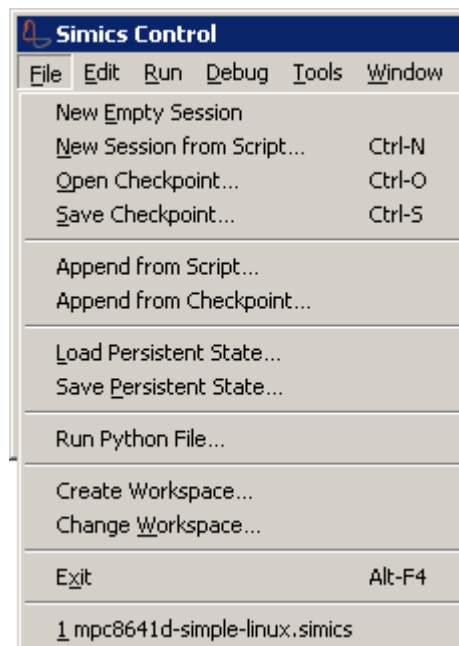
4.2.2 File Menu

Figure 4.4: 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 the *Hindsight User's Guide* 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.

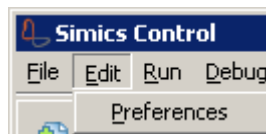
4.2.3 Edit Menu

Figure 4.5: Edit Menu

Edit → Preferences

Opens the *Preferences* windows described in section [4.15](#).

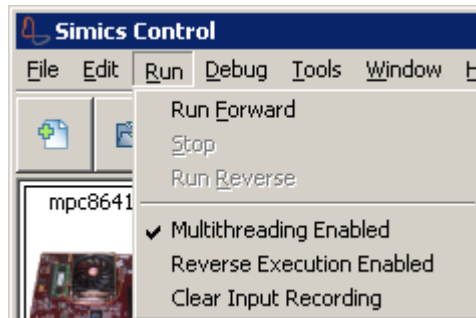
4.2.4 Run Menu

Figure 4.6: 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.

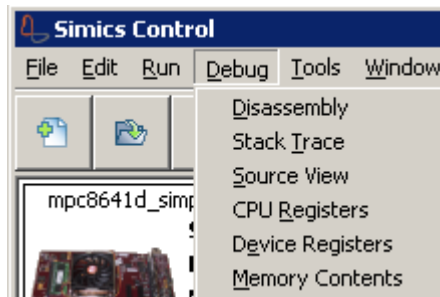
4.2.5 Debug Menu

Figure 4.7: Debug Menu

Debug → Disassembly

Shows the *Disassembly* window, which presents the CPU execution in assembly instructions. This window is described in section 4.8.

Debug → Stack Trace

Shows the *Stack Trace* window, which shows the stack trace of the currently running process. This window is described in section 4.17.

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 4.16.

Debug → CPU Registers

Shows the *CPU Registers* window, which presents the contents of various CPU registers. This window is described in section 4.5.

Debug → Device Registers

Shows the *Device Registers* window, which presents the contents of various device registers. This window is described in section 4.6.

Debug → Memory Contents

Shows the *Memory Contents* window, which presents the contents of physical and logical memory spaces. This window is described in section 4.7.

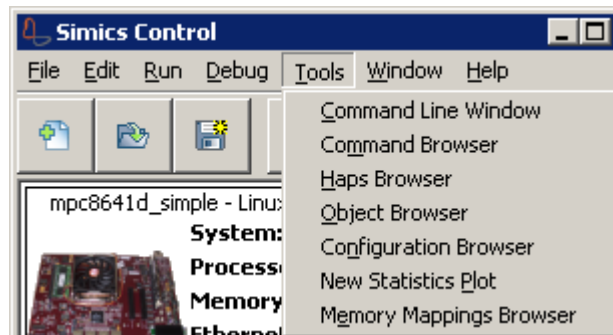
4.2.6 Tools Menu

Figure 4.8: 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 4.4.

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 4.9.

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 4.11.

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 4.12.

Tools → New Statistics Plot

Creates a new *Statistics Plot* window, which can plot statistics collected by Simics. See section 4.13.

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 4.14.

4.2.7 Window Menu

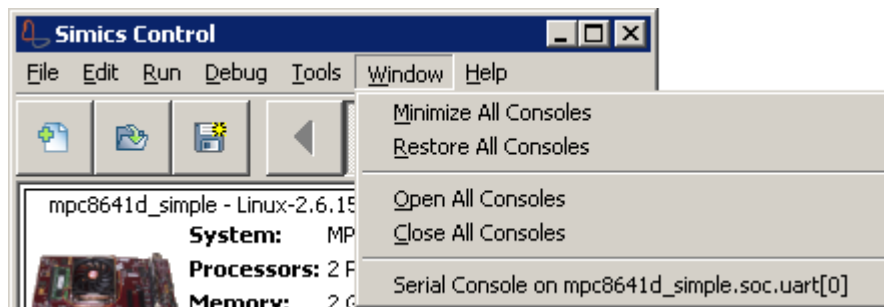


Figure 4.9: Window Menu

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.

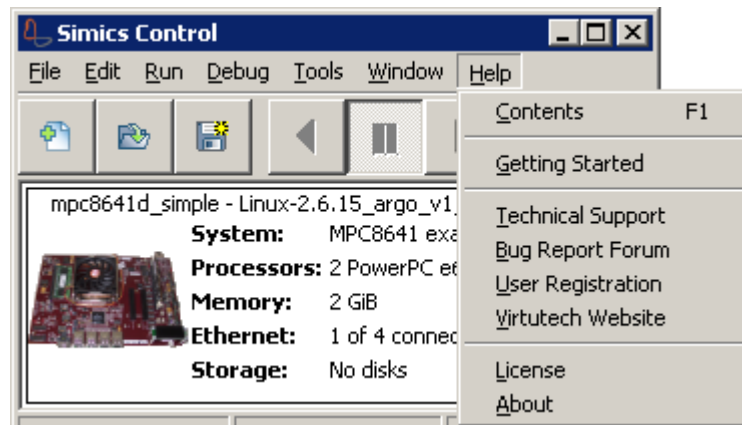


Figure 4.10: Help Menu

4.2.8 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 4.10.

Help → Getting Started

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

Help → Technical Support

Opens the technical support page of the Wind River Simics website.

Help → Bug Report Forum

Opens your personal support forum page on the Simics website.

Help → Wind River Website

Opens the Wind River company website.

Help → License

Displays the *Simics Software License Agreement*.

Help → About

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

4.3 Console Window

The *Console* window shows the output of the simulated machine. It can be graphical, as shown in figure 4.12, or text-based, as shown in figure 4.11. Note that the graphics console can also emulate a text console, which is the case when booting a simulated PC.

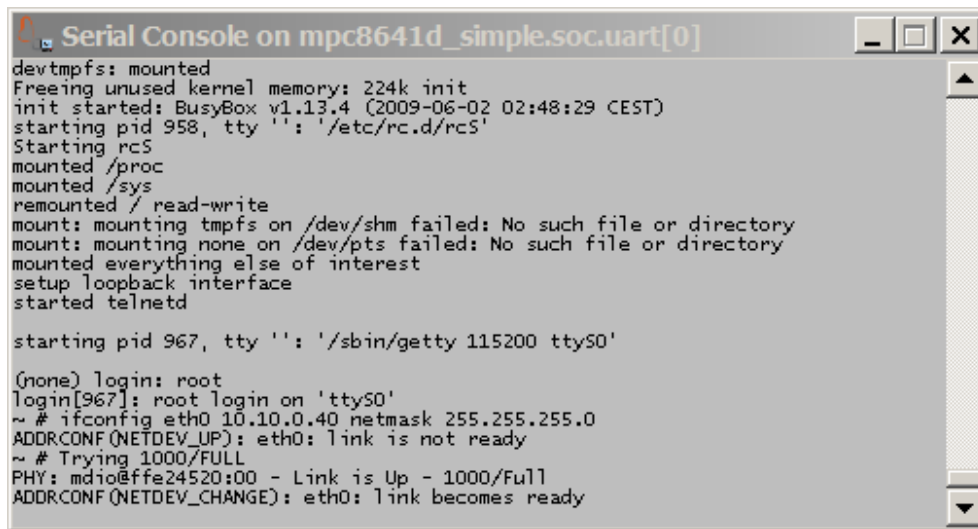


Figure 4.11: A Text Console Window

4.3.1 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.

4.3.2 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.



Figure 4.12: A Graphics 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.

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.

4.4 Command Line Window

The *Command Line* window (figure 4.13) 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, those shortcuts are documented in the *Hindsight User's Guide*.

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 the *Hindsight User's Guide*.

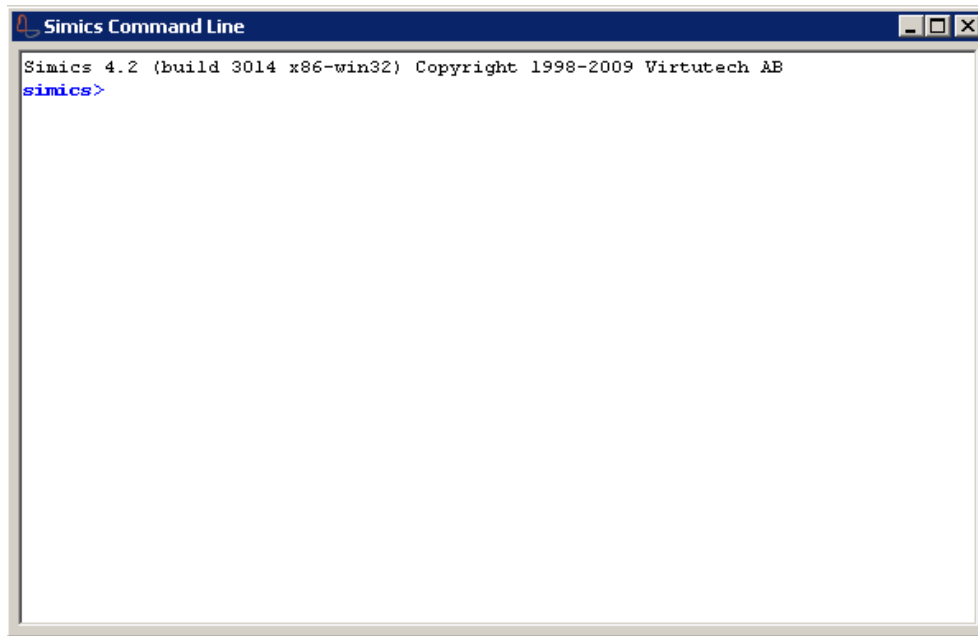


Figure 4.13: Command Line Window

4.5 CPU Registers Window

The *CPU Registers* window (figure 4.14) 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 4.15). The groups are architecture specific.

4.6 Device Registers Window

The *Device Registers* window (figure 4.16) 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.

4.7 Memory Contents Window

The *Memory Contents* window (figure 4.17) shows the contents of memory spaces and the virtual memories of processors. It is modeled after hex editors, but currently it is only a

4.7. Memory Contents Window

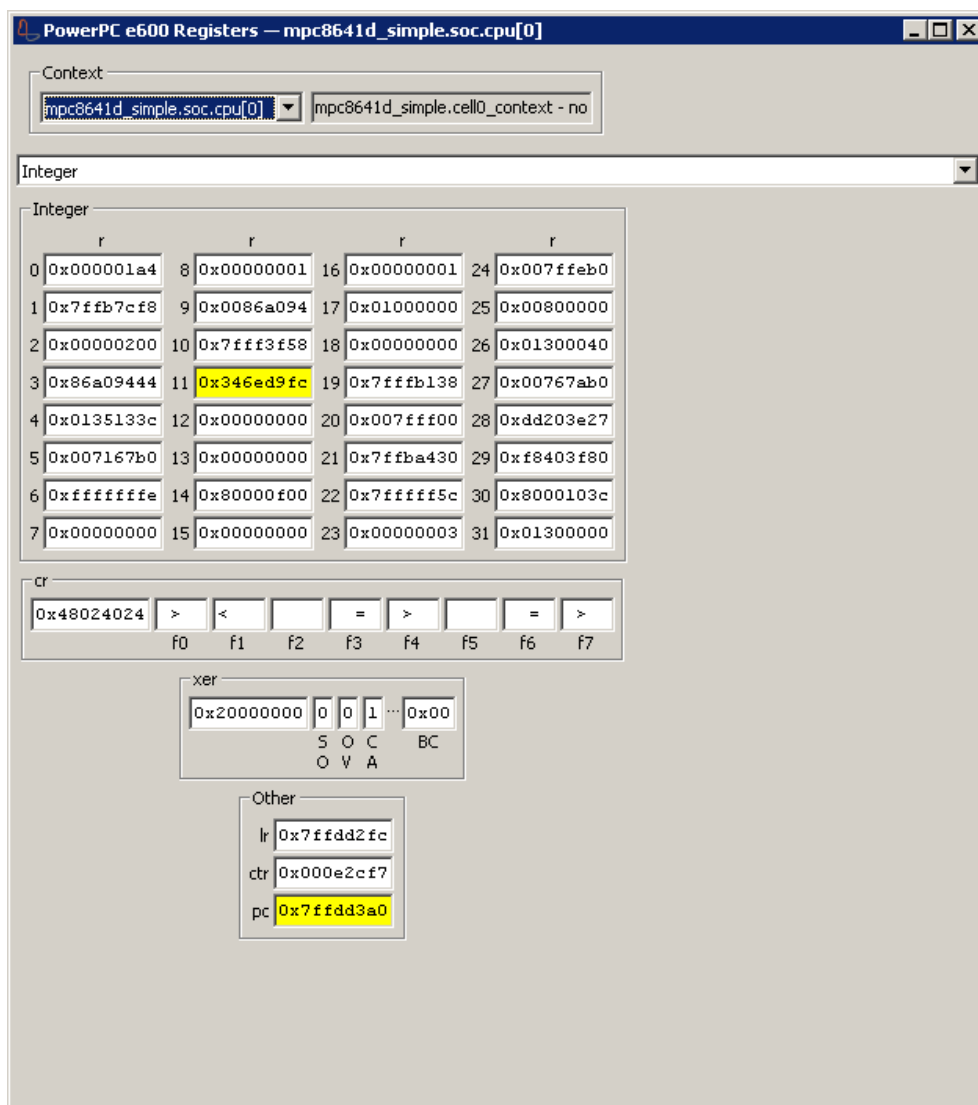


Figure 4.14: CPU Register Window

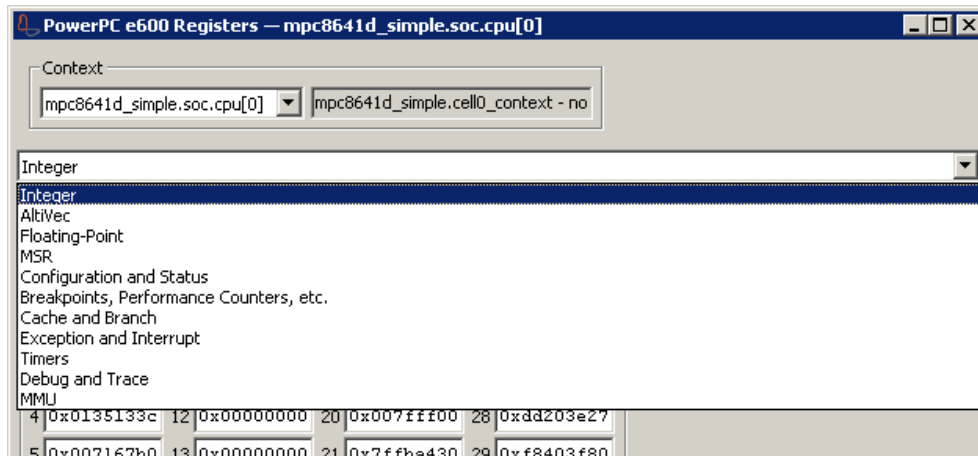


Figure 4.15: CPU Register Types

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 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 a dot. Error values are also shown as dots.

4.8 Disassembly Window

The *Disassembly* window is shown in figure 4.18. 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.)

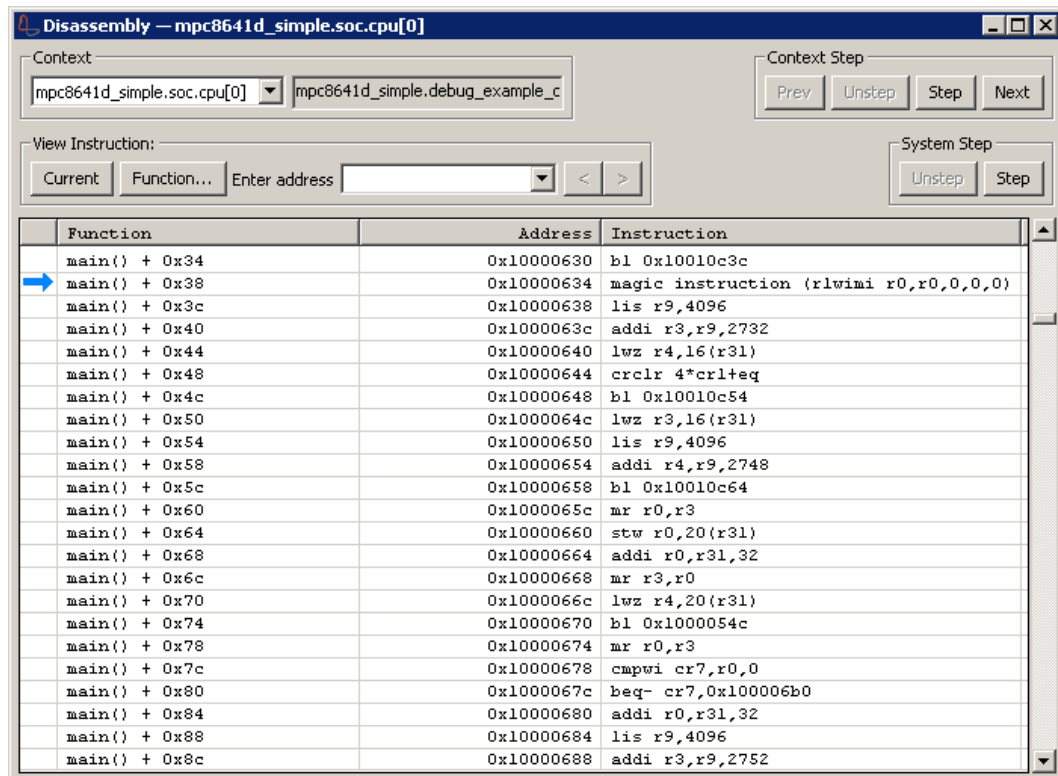


Figure 4.18: The Disassembly Window

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 in the *Hindsight User's Guide*. 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.

4.9 Hap Browser

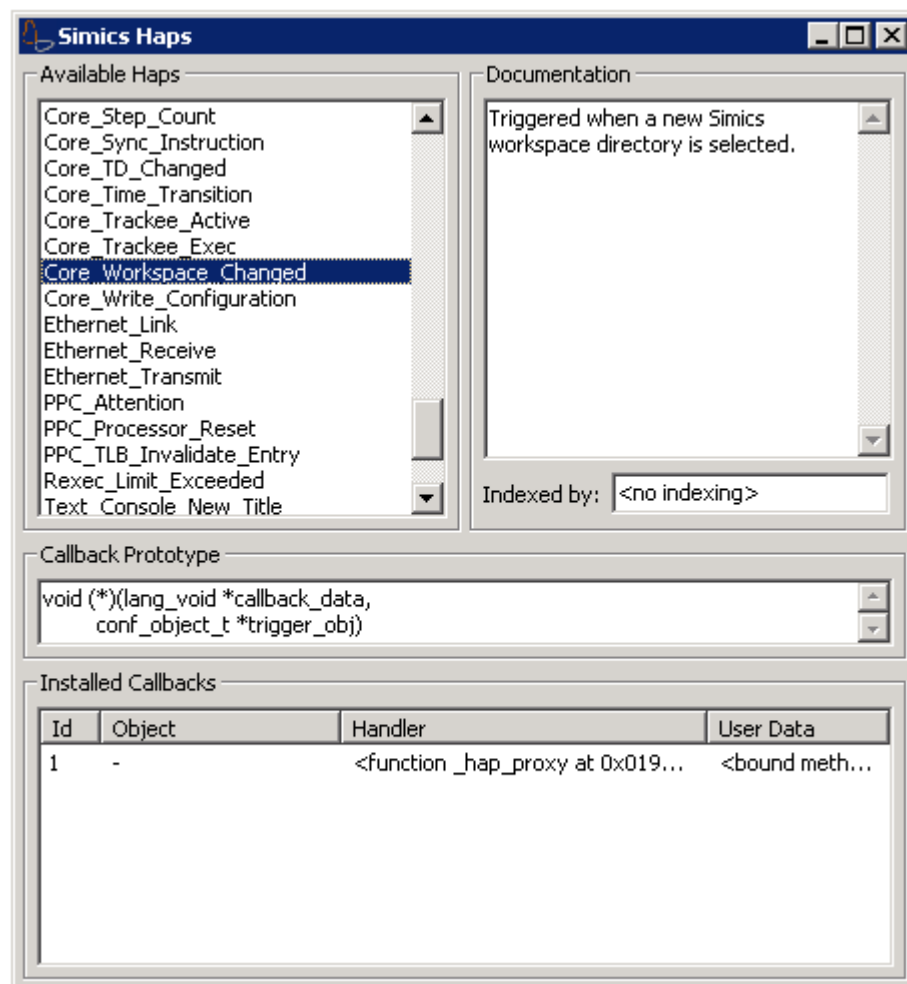


Figure 4.19: Hap Browser

The hap browser (figure 4.19) is designed to help script writers to use haps and hap callbacks. Refer to the *Hindsight User's Guide* 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.

4.10 Help Browser

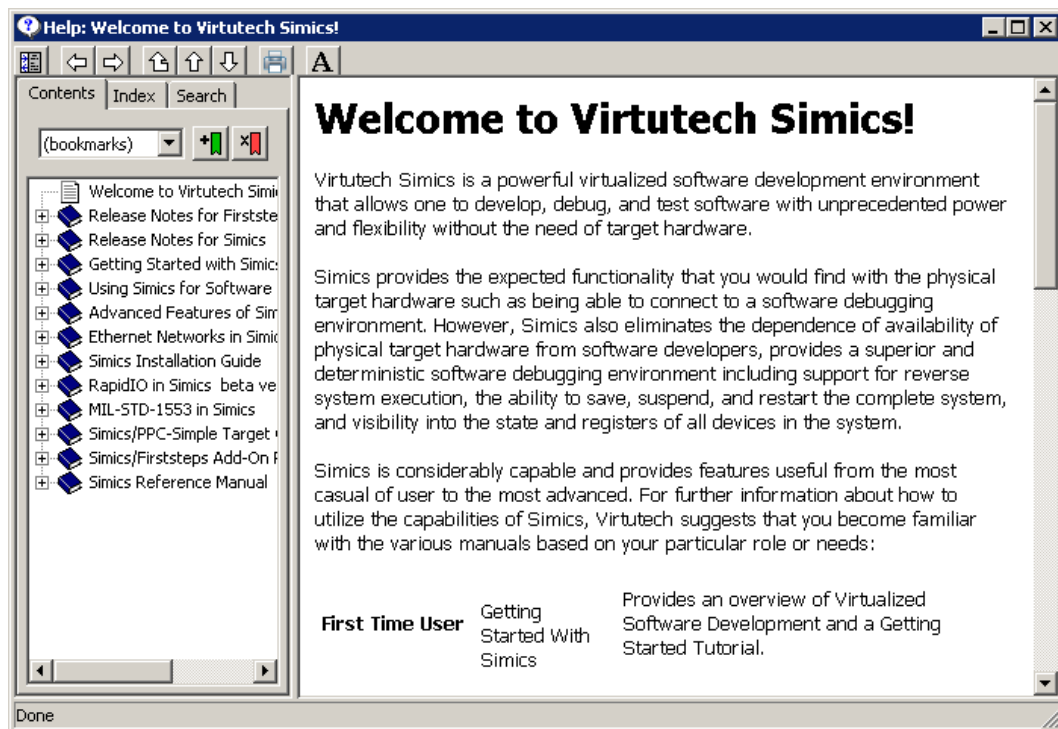


Figure 4.20: Help Browser

The Help Browser (figure 4.20) 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.

4.11 Object Browser

The *Object Browser* window (figure 4.21) 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

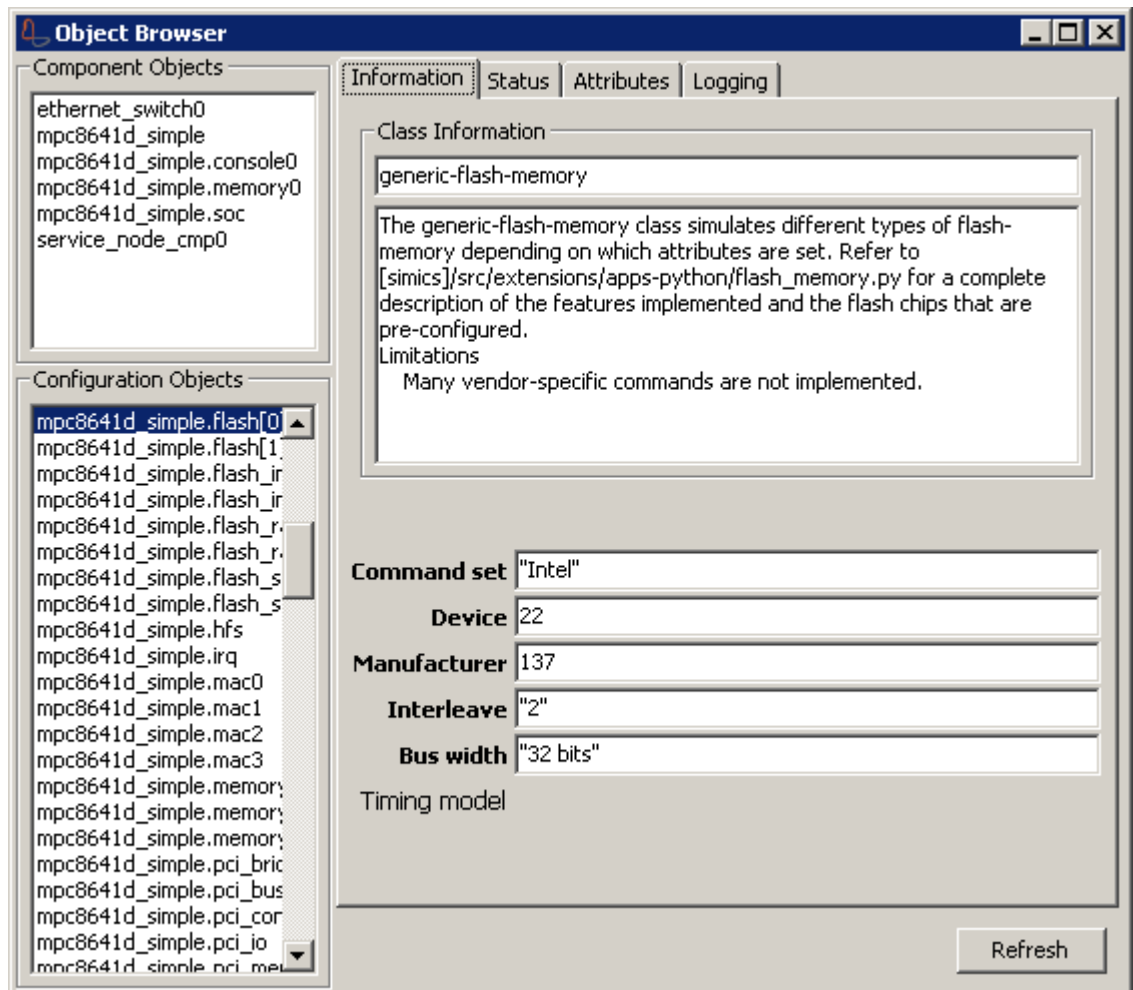


Figure 4.21: Object Browser

information in the right side panel. To understand much of this information, you will need to get acquainted with the way Simics build configurations, which is described in the *Hindsight User's Guide* and the *Model Builder User's Guide*.

Information

The class of the object will be indicated first, followed by the class description. The rest of the information is object-specific.

Status

The *Status* panel (figure 4.22) contains current information about the object state.

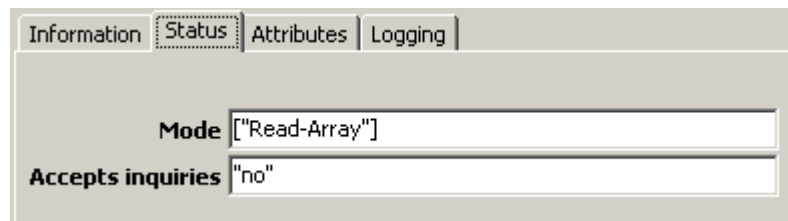


Figure 4.22: Object Browser: status

Attributes

The *Attributes* panel (figure 4.23) 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.

4.12 Configuration Browser

The *Configuration Browser* window (figure 4.24) 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 parts 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 4.11). By double-clicking a component you open it in the Object Browser.

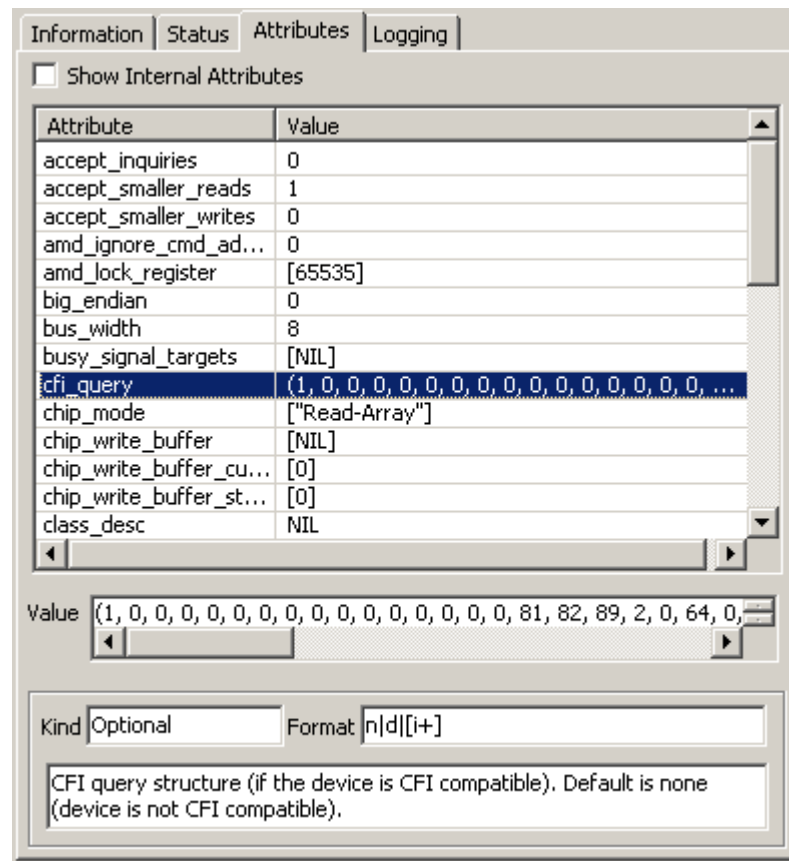


Figure 4.23: Object Browser: attributes

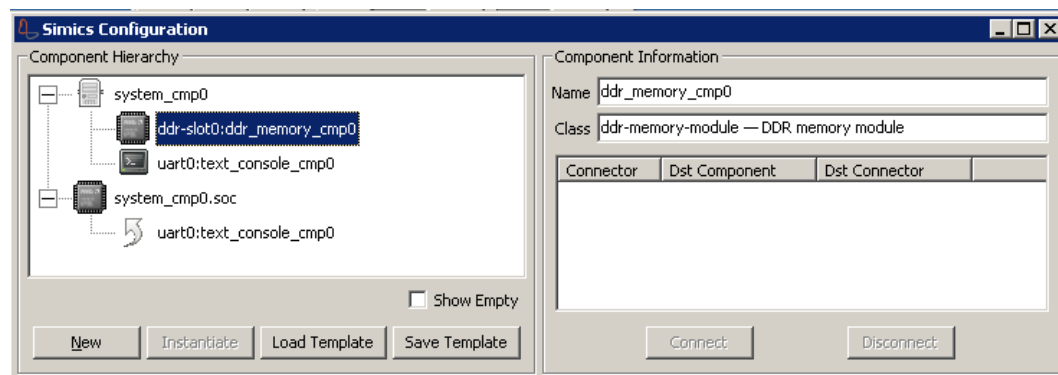


Figure 4.24: Configuration Browser

4.13 Statistics Plot

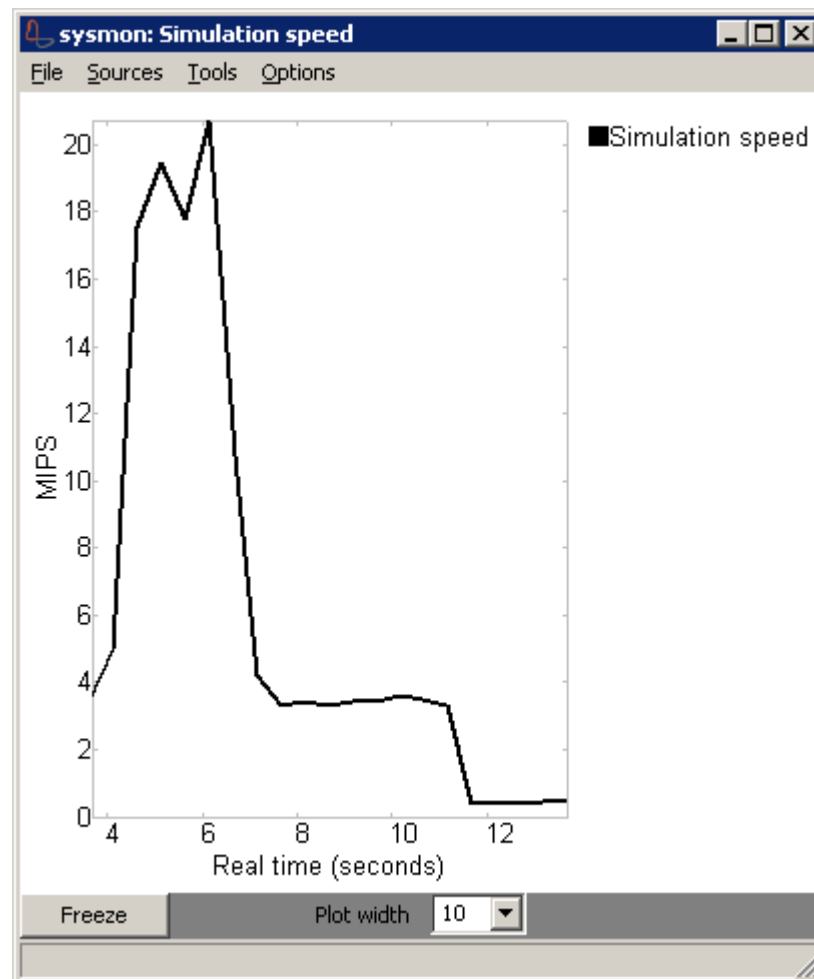


Figure 4.25: Statistics Plot

A *Statistics Plot* window (figure 4.25) 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.

4.14 Memory Mappings Browser

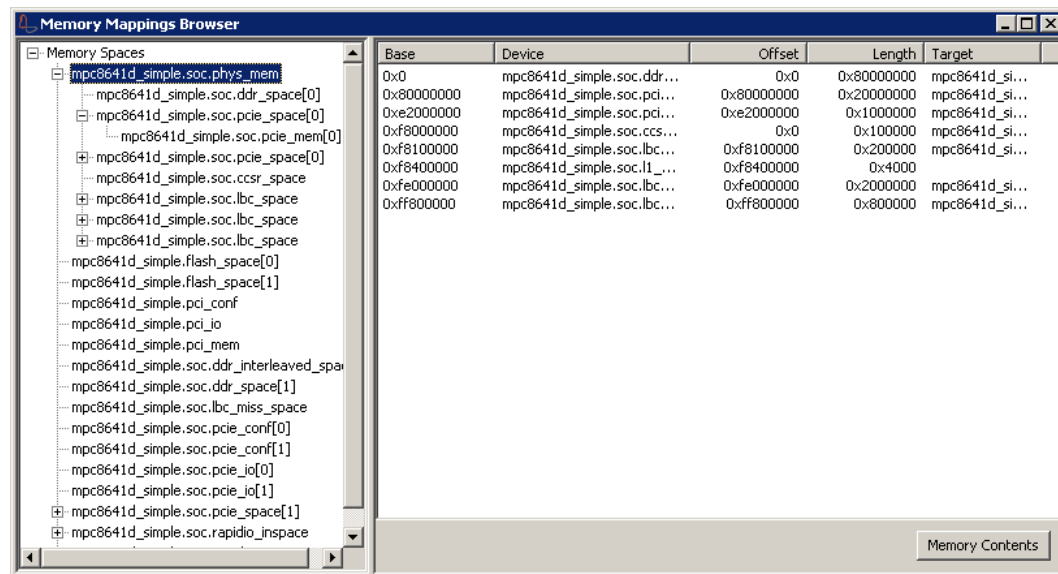


Figure 4.26: Memory Mappings Browser

The *Memory Mappings Browser* window (figure 4.26) 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 left part of the view, the roots of the trees are 1) the memory spaces which are physical memories of processors in the system, 2) memory spaces not mapped into any other memory spaces, 3) 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 4.7 for documentation about the viewer it opens.

4.15 Preference Window

The *Preferences* window is organized in three panels: *Appearance* (figure 4.27), *Startup* (figure 4.28), and *Advanced* (figure 4.29). 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 4.27) panel let you define how input and output will be handled:

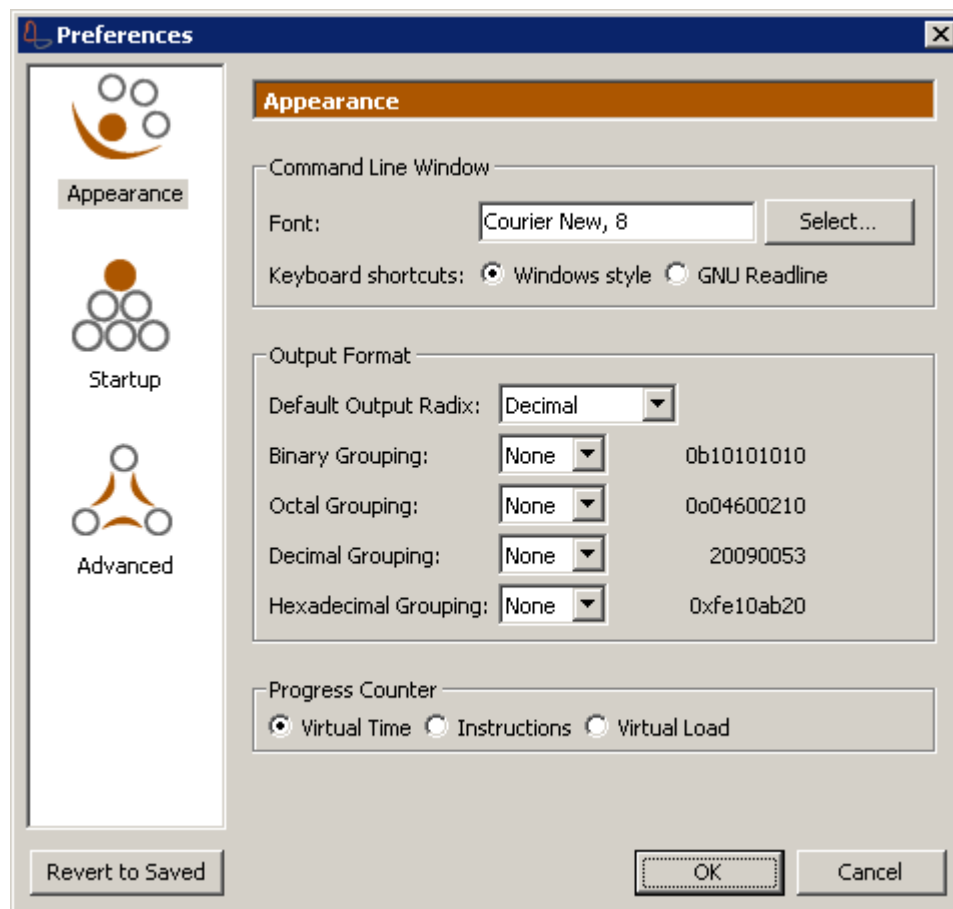


Figure 4.27: Preferences: Appearance

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 style shortcuts or GNU 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 4.28) 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 the *Analyzer User's Guide* and—in the *Extension Builder User's Guide*.

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.

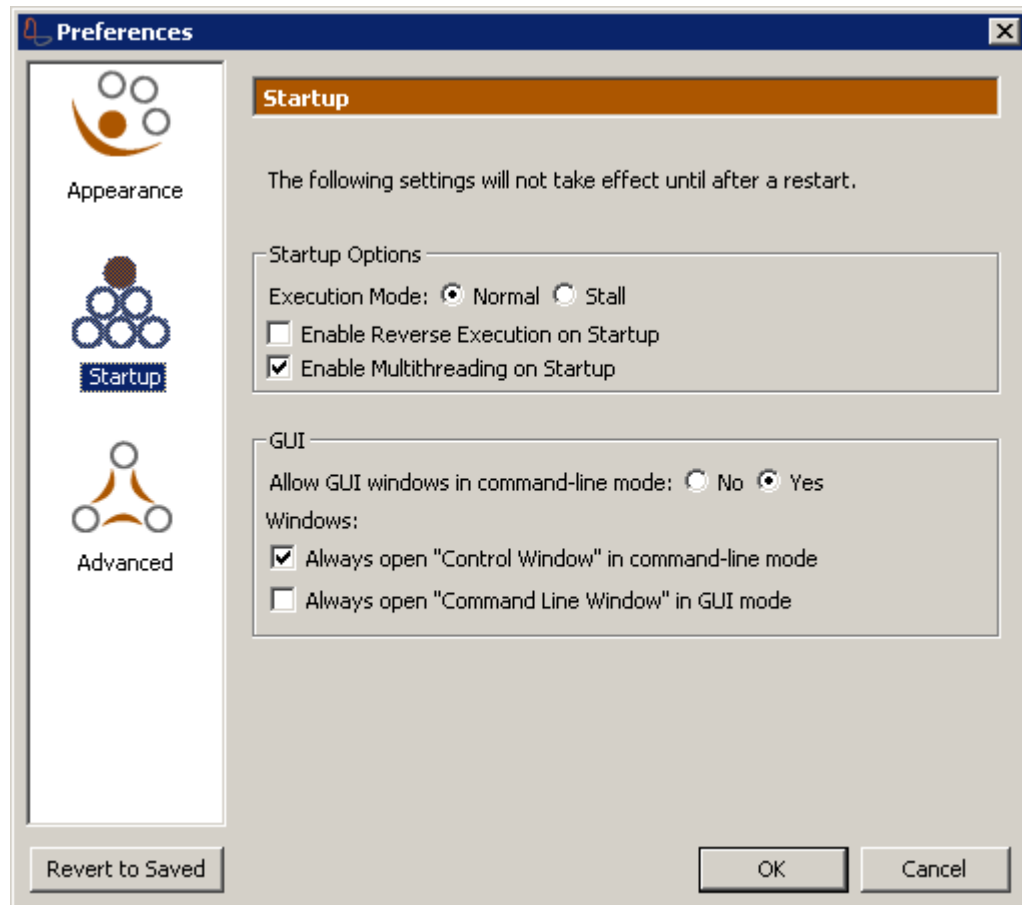


Figure 4.28: Preferences: Startup

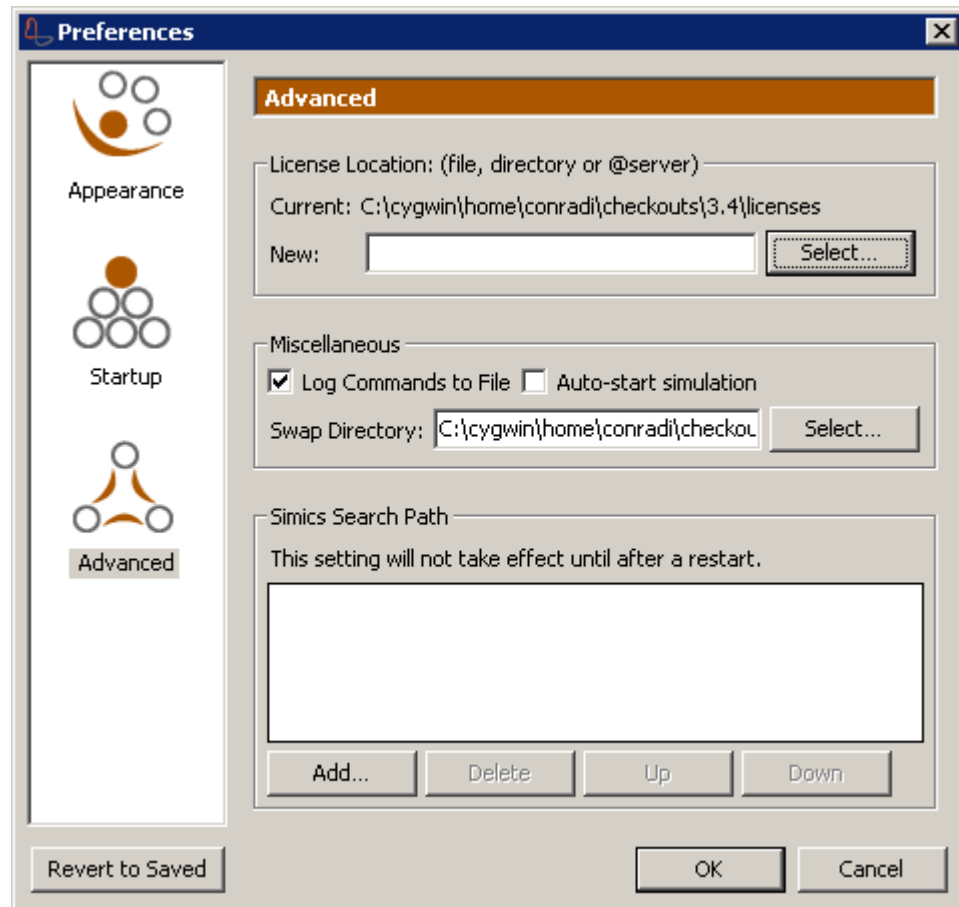


Figure 4.29: Preferences: Advanced

Advanced Panel

The *Advanced* panel (figure 4.29) provides control over more advanced features:

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/4.4/log` on Unix and in `Application Data\Simics\4.4\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 the *Hindsight User's Guide* 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 the *Hindsight User's Guide*.

4.16 Source View Window

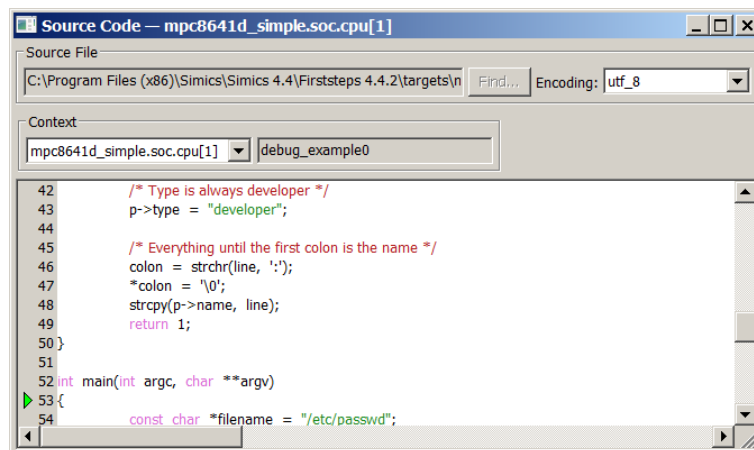


Figure 4.30: The source view window

The *Source View* window is shown in figure 4.30. 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. The *Hindsight User's Guide* explains how to set this up.

4.17 Stack Trace Window

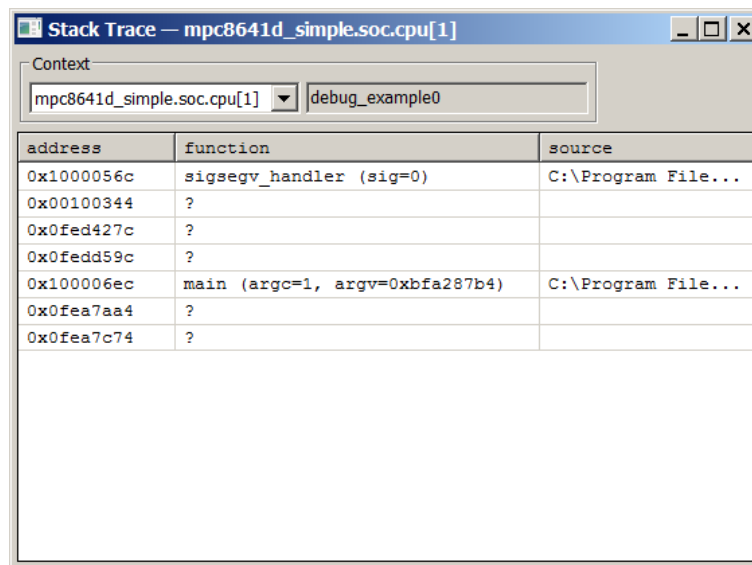


Figure 4.31: The stack trace window

The *Stack Trace* window is shown in figure 4.31. 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. If no debug symbols are available, only the frame's address will be shown. The *Hindsight User's Guide* explains how to load debug symbols.

Chapter 5

Next Steps

This *Getting Started* document 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 *Hindsight User's Guide* 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 *Hindsight User's Guide* 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 *Model Builder User's Guide* manual, and particularly the tutorial provided in this manual. Note that *Model Builder User's Guide* is only available to those who purchased the *Simics Model Builder* product.

Index

Symbols

@, [31](#)

—
 in numbers, [59](#)
[simics], [5](#)
[workspace], [5](#)

A

About, [42](#)
Accelerator, [7](#)
Analyzer, [7](#)
API, [31](#)
Append from Checkpoint, [37](#)
Append from Script, [37](#)
assembly code, [48](#)
attributes, [30](#), [54](#)

B

bookmark, [16](#)
break, [25](#)
break-cr, [30](#)
break-exception, [30](#)
break-io, [30](#)
Bug Report Forum, [42](#)

C

callback, [52](#)
callbacks, [52](#)
capture
 screen, [45](#)
Change Workspace, [38](#)
checkpoint
 append, [37](#)
 open, [35](#), [37](#)
 save, [35](#), [37](#)
checkpointing, [15](#)
class, [54](#)
Clear Input Recording, [39](#)

clear-recorder, [17](#)
CLI, [32](#), [40](#), [45](#)
CLI variables, [30](#)
Close All Consoles, [41](#)
Command Browser, [40](#)
Command line, [45](#)
Command Line Window, [40](#), [59](#)
component, [52](#), [54](#)
configuration, [52](#), [54](#)
Configuration Browser, [40](#), [54](#)
Console, [32](#)
Console Window, [41](#), [42](#)
Contents, [42](#), [52](#)
context, [51](#), [63](#)
CPU Core Types, [8](#)
CPU Registers, [40](#), [46](#)
Create Workspace, [37](#)

D

Debug menu, [39](#)
 CPU Registers, [40](#)
 Device Registers, [40](#)
 Disassembly, [39](#)
 Memory Contents, [40](#)
 Source View, [39](#)
 Stack Trace, [39](#)
debug symbol, [63](#)
debug symbols, [63](#)
debugging, [19](#)
delete-bookmark, [16](#)
Device Model Libraries, [9](#)
Device Registers, [40](#), [46](#)
Disable Multithreading, [39](#)
Disable Reverse Execution, [39](#)
Disable/Enable Multithreaded, [36](#)
Disable/Enable Reverse Execution, [36](#)
Disassembly, [39](#)
disk images, [62](#)

E

Edit menu, [38](#)
 Preferences, [38](#)
 Enable Multithreading, [39](#)
 Enable Reverse Execution, [39](#)
 Enable/Disable Multithreaded, [36](#)
 Enable/Disable Reverse Execution, [16](#), [21](#),
 [36](#)
 Ethernet Networking, [8](#)
 execution mode, [59](#)
 Exit, [38](#)
 Extension Builder, [8](#)

F

File menu, [36](#)
 Append from Checkpoint, [37](#)
 Append from Script, [37](#)
 Change Workspace, [38](#)
 Create Workspace, [37](#)
 Exit, [38](#)
 Load Persistent State, [37](#)
 New Empty Session, [36](#)
 New Session from Script, [37](#)
 Open Checkpoint, [37](#)
 previous files, [38](#)
 Quit, [38](#)
 Run Python File, [37](#)
 Save Checkpoint, [37](#)
 Save Persistent State, [37](#)
 first-steps, [10](#)
 frame, [25](#)

G

Getting Started, [42](#)
 Go to Address, [51](#)
 Graphics Console, [43](#)
 GUI, [59](#)

H

Hap Browser, [40](#), [51](#)
 haps, [51](#)
 Help, [52](#)
 index, [52](#)
 search, [52](#)
 Help Browser, [52](#)
 Help menu, [42](#)

About, [42](#)
 Bug Report Forum, [42](#)
 Contents, [42](#)
 Getting Started, [42](#)
 License, [42](#)
 Technical Support, [42](#)
 Wind River Website, [42](#)

Hindsight, [6](#)

I

index, [52](#)

L

License, [42](#)
 license, [62](#)
 License File, [62](#)
 Load Persistent State, [37](#)
 log, [62](#)

M

Memory Contents, [40](#), [46](#)
 Memory Mappings Browser, [41](#), [57](#)
 memory usage, [62](#)
 Minimize All Consoles, [41](#)
 Miscellaneous, [62](#)
 Model Builder, [8](#)
 mouse, [43](#)
 Multithreading, [7](#)
 multithreading, [59](#)
 enable/disable, [39](#)

N

New Empty Session, [36](#)
 New Session From Script, [35](#)
 New Session from Script, [37](#)
 New Statistics Plot, [41](#)
 new-tracer, [27](#)
 Next, [51](#)

O

object, [52](#)
 Object Browser, [40](#), [52](#)
 Attributes, [54](#)
 Information, [54](#)
 Status, [54](#)
 Open All Consoles, [41](#)

Open Checkpoint, [16, 35, 37](#)
 Output Format, [59](#)

P

Page Sharing, [7](#)
 persistent state
 load, [37](#)
 save, [37](#)
 Preferences, [38, 58](#)
 Advanced, [62](#)
 Appearance, [58](#)
 Startup, [59](#)
 Prev, [51](#)
 product
 Analyzer, [7](#)
 CPU Core Types, [8](#)
 definitions, [6](#)
 Device Model Libraries, [9](#)
 Ethernet Networking, [8](#)
 Extension Builder, [8](#)
 Hindsight, [6](#)
 Model Builder, [8](#)
 Simics Accelerator, [7](#)
 Virtual Board/System, [7](#)
 Program Counter, [51](#)
 Progress Counter, [59](#)
 prompt, [45](#)
 psym, [25](#)
 ptime, [17](#)
 python, [31](#)
 Run Python File, [37](#)

Q

Quit, [38](#)

R

Readline, [45](#)
 refresh rate, [45](#)
 register
 types, [46](#)
 registers, [46](#)
 Restore All Consoles, [41](#)
 reverse, [25](#)
 reverse execution, [16, 48, 59](#)
 enable/disable, [36, 39](#)
 Run Reverse, [35](#)

reverse-step-line, [23](#)
 Run Forward, [14, 35, 38](#)
 Run menu, [38](#)
 Multithreading Enabled, [39](#)
 Reverse Execution Enabled, [39](#)
 Run forward, [38](#)
 Run Reverse, [39](#)
 Stop, [38](#)
 Run Python File, [37](#)
 Run Reverse, [35, 39](#)

S

Save Checkpoint, [15, 35, 37](#)
 Save Persistent State, [37](#)
 screenshot, [45](#)
 scripting, [30](#)
 search, [52](#)
 Serial Console, [12, 32](#)
 session, [34](#)
 append, [37](#)
 new, [36](#)
 new from script, [35, 37](#)
 set-bookmark, [16](#)
 shortcuts, [45, 59](#)
 Simics Accelerator, [7](#)
 Simics Command Line, [32](#)
 Simics Control, [32](#)
 Simics Control window, [34](#)
 Simics Search Path, [62](#)
 simics user interface, [32](#)
 SimicsFS, [18](#)
 skip-to, [17](#)
 source code, [63](#)
 Source View, [39](#)
 stack frame, [63](#)
 Stack Trace, [26, 39](#)
 stall, [59](#)
 Startup Options, [59](#)
 Statistics Plot, [41, 56](#)
 status, [54](#)
 status bar, [59](#)
 Step, [48](#)
 step-instruction, [29](#)
 Stop, [14, 35, 38](#)
 swapping, [62](#)

T

target, 8
 Technical Support, 42
 Text Console, 32, 43
 toolbar, 34

- Enable/Disable Multithreaded, 36
- Enable/Disable Reverse Execution, 36
- New Session From Script, 35
- Open Checkpoint, 35
- Run Forward, 35
- Run Reverse, 35
- Save Checkpoint, 35
- Stop, 35

 Tools menu, 40

- Command Browser, 40
- Command Line Window, 40
- Configuration Browser, 40
- Hap Browser, 40
- Memory Mappings Browser, 41
- New Statistics Plot, 41
- Object Browser, 40
- Statistics Plot, 41

 trace-cr, 28
 trace-exception, 30
 trace-io, 28
 tracing, 27
 tutorial, 10

U

underscores

- in numbers, 59

 Unstep, 48

V

Virtual Board, 7
 Virtual System, 7

W

Wind River Website, 42
 Window menu, 41

- Close All Consoles, 41
- Console list, 41
- Minimize All Consoles, 41
- Open All Consoles, 41
- Restore All Consoles, 41

 windows

command line, 45
 configuration browser, 54
 control window, 34
 cpu registers, 46
 device registers, 46
 disassembly, 48
 graphics console, 43
 hap browser, 51
 help browser, 52
 memory contents, 46
 memory mappings browser, 57
 object browser, 52
 preference window, 58
 source view, 62
 stack trace, 63
 statistics plot, 56
 text console, 43
 workspace, 10, 32

- change, 38
- create, 37