



virtutech

Using Simics Hindsight for Software Development

Simics Version 4.2

Revision 3001
Date 2009-03-28

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

Trademarks

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

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

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

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

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

Contents

1	Introduction	5
2	Simulation Concepts	6
2.1	The Limits of Simulation	6
2.2	Nonintrusive Inspection and Debugging	6
2.3	Simulated Time	7
3	Debugging Software with Simics	8
3.1	Breakpoints	8
3.1.1	Memory Breakpoints	9
3.1.2	Temporal Breakpoints	11
3.1.3	Control Register Breakpoints	11
3.1.4	I/O Breakpoints	12
3.1.5	Text Output Breakpoints	12
3.1.6	Graphics Breakpoints	12
3.1.7	Magic Instructions and Magic Breakpoints	13
3.2	Symbolic Debugging	14
3.2.1	Sample Session	15
3.2.2	Source Code Stepping	18
3.2.3	Symbolic Breakpoints	19
3.2.4	Reading Debug Information from Binaries	19
3.2.5	Loading Symbols from Alternate Sources	20
3.2.6	Multiple Debugging Contexts	20
3.2.7	Scripted Debugging	22
3.3	Reverse Execution	24
3.3.1	Using Reverse Execution	24
3.3.2	Performance	25
4	Using Simics for Hardware Bring-Up and Firmware Development	26
4.1	A Simple Example	26
4.2	Going Further	27
5	Using Simics for Application-Level Development	29
5.1	Launching an Application	29
5.2	Debugging an Application	30
5.3	Debugging Networked Applications	30

6	Using Simics with Wind River Tornado or Workbench	31
6.1	Starting the Simics WDB Agent	31
6.2	Connecting Tornado	32
6.3	Connecting Workbench	32
6.4	Limitations	33
7	Using Simics with GDB	34
7.1	Remote GDB and Shared Libraries	36
7.2	Using GDB with Reverse Execution	37
7.3	Compiling GDB	38
8	Using Simics with Other IDEs	40
	Index	41

Chapter 1

Introduction

This document explains how to use Simics Hindsight as a tool in software development. In this setting, Simics plays two roles:

execution platform

The software is run on a simulated target machine inside Simics.

debugger

Simics can either function as a stand-alone debugger, or as a debugger backend for an external debugger.

Using Simics as a debugger has some major benefits compared to debugging on real hardware:

- Debugging is completely non-intrusive.
- You can inspect and modify the state of the entire target system, at any level. This is especially convenient when debugging low-level code such as firmware and hardware drivers.
- The simulation is completely deterministic. Once you manage to trigger a bug, you can repeat it as often as you like.
- You have complete control over time. For example, you can freeze time while inspecting the target, save the simulated state in a checkpoint and restore it at a later time, and even run the whole simulation backward.

Note: It is recommended that you have read *Getting Started with Simics* before tackling this document.

Chapter 2

Simulation Concepts

2.1 The Limits of Simulation

Simics is a system-level instruction set simulator. This means that:

- Simics models the target system at the level of individual instructions, executing them one at a time.
- Simics models the binary interface to hardware in sufficient detail that any software that runs on the real hardware will also run on Simics.

In practice, what this means is that there is no code that is too “low-level”—Simics can run, and debug, any kind of software: firmware, hardware drivers, operating systems, user-level applications, whatever. There are some caveats, though:

- Simics’s model of time is rather simple; for example, it assumes by default that all instructions take the same amount of time to run. It is not difficult to write a program that uses this fact to detect the difference between Simics and real hardware. However, this is seldom a problem with real-world software.

You can read more about Simics and time in *Advanced Memory & Timing in Simics*.

- The hardware models must be detailed enough. Models of nontrivial pieces of hardware are not typically fully complete, so it is possible to write a program that detects the difference between Simics and real hardware by probing any unimplemented corners. However, any given piece of software can be accommodated by extending the hardware models to cover those corners.

You can read more about hardware modeling in *Modeling Your System in Simics*.

2.2 Nonintrusive Inspection and Debugging

Simics has powerful built-in inspection and debugging facilities. These include:

- Inspecting registers, memory, and hardware state.

- Modifying register and memory contents, and hardware state.
- Setting (and triggering) breakpoints and watchpoints.
- Powerful scripting support for all of this (see section 3.2.7 for some examples).

Because these are implemented in the simulator, no debugging software needs to be on the target at all. As a result, the debugging machinery is completely invisible to the target (and thus to any software running on it).

2.3 Simulated Time

One of the most powerful properties of full-system simulation is that time inside the simulation and time in the real world are two completely different things. This brings a number of substantial benefits:

- You can pause the simulation at any time, and the software running in the simulated simply *cannot* detect this. This allows you to inspect (and optionally modify) the state even at points where real hardware would be unable to stop.
- You can save the state of the simulation to disk (this is called a *checkpoint*), and start again from that point at any time, any number of times.
- The simulation is completely deterministic. Every time you start from the same state (such as a checkpoint), the *exact* same thing will happen. This can be tremendously useful when hunting down certain types of bugs.
- The simulation can be reversed, making it possible to answer questions such as “This value is garbage now. When did this happen, and who did it?”; simply set a write breakpoint and run backward until it triggers, and you have your culprit.

These advantages apply to the entire simulated system, whether it is a single target machine or an entire network.

Chapter 3

Debugging Software with Simics

This chapter explains two cornerstones of debugging in Simics: its powerful breakpoint support, and fully scriptable debug and symbol information handling. It also discusses the reverse execution capabilities of Simics.

Note: Much of the breakpoint support is available even if you use Simics as a backend for an external debugger (as in chapter 6 and 7), but most of the symbolic debugging features are not exposed through these interfaces. This is not a big deal in practice, since these debuggers have their own symbolic debugging support.

3.1 Breakpoints

Like any other debugger, Simics can set breakpoints on user code and data. But unlike most debuggers, Simics breakpoints are not limited by what the hardware can support; for example, there is no restriction on the number of read/write breakpoints (also known as watchpoints).

In Simics you can set breakpoints on:

- memory accesses: any range and combination of read/write/execute
- time (number of cycles or instructions executed)
- instruction types, such as control register accesses
- device accesses
- output in the console

Simics is fully deterministic, and debugging in Simics is fully non-intrusive. This makes it possible to narrow down the location of difficult bugs by re-running the *exact* same run as many times as you need.

But there is often no need to run by the bug multiple times, since breakpoints work even when the simulation is running backwards. For example, to find the code responsible for writing garbage to a pointer, run forward until your program crashes, then set a write

breakpoint on the (now clobbered) pointer, and run backward; the breakpoint will trigger at the point in time when the pointer was last written to.

3.1.1 Memory Breakpoints

Note: The examples in this section require the user to supply the desired address of the breakpoint directly. See section 3.2.3 for examples of how to set breakpoints by giving just the name of a function or variable.

A memory breakpoint stops the simulation whenever a memory location in a specified address interval is accessed. The address interval can be of arbitrary length and the type of the memory access can be specified as any combination of *read*, *write*, and *execute*.

The easiest way to set memory breakpoints is to use the **break** command:

```
simics> break p:0x10000
Breakpoint 1 set on address 0x10000 in 'system_cmp0.soc.plb' with ?
access mode 'x'
```

Prefix the address with **p:** or **v:** to get a physical or virtual address, respectively. As you can see in the following example, Simics defaults to interpreting a breakpoint address as virtual if you do not specify otherwise:

```
simics> break v:0x4711
Breakpoint 2 set on address 0x4711 in 'system_cmp0.cell0_context' with ?
access mode 'x'
simics> break p:0x4711
Breakpoint 3 set on address 0x4711 in 'system_cmp0.soc.plb' with access ?
mode 'x'
simics> break 0x4711
Breakpoint 4 set on address 0x4711 in 'system_cmp0.cell0_context' with ?
access mode 'x'
Note: overlaps with breakpoint 2
```

This way of setting breakpoints will attach them to the memory space (physical address) or context (virtual address) connected to the current processor. If this is not exactly what you want, read on.

Note: The current processor can be set with the **pselect** command:

```
simics> pselect system_cmp0.soc.cpu
```

Without an argument, **pselect** prints the current processor:

```
simics> pselect
Currently selected processor is system_cmp0.soc.cpu
```

Physical memory breakpoints are handled by memory space objects. A memory space represents a physical address space; they sit between the processor and the actual hardware devices (e.g. RAM) that can be accessed with read and write instructions. Breakpoints are created with the memory space's **break** command:

```
simics> system_cmp0.soc.cpu->physical_memory
"system_cmp0.soc.plb"
simics> system_cmp0.soc.plb.break address = 0x10000 length = 16 -w
Breakpoint 1 set on address 0x10000 in 'system_cmp0.soc.plb', length 16
with access mode 'w'
```

Virtual memory breakpoints are handled by context objects. A context represents a virtual address space; you can learn more about them in chapter 3.2. Essentially, they provide a level of indirection between processors and virtual memory breakpoints; a processor has a current context, which in turn has virtual breakpoints:

```
simics> system_cmp0.soc.cpu->current_context
"system_cmp0.cell0_context"
simics> system_cmp0.cell0_context.break 0x1fff00
Breakpoint 1 set on address 0x1fff00 in 'system_cmp0.cell0_context' with
access mode 'x'
```

Note that by default, all simulated processors in a cell share one context (**cell_n_context**). If you want a virtual breakpoint to apply only to a subset of the processors, create a new context just for them:

```
simics> new-context foo
simics> ebony0_system_cmp0.soc.cpu.set-context foo
simics> foo.break 0xfffffffffbfc008b8
```

Execution breakpoints can be modified with filter rules to only trigger when instructions match certain syntactical criteria. This feature is mainly useful with breakpoints covering large areas of memory. The commands available are **set-prefix** (to match the start of an instruction), **set-substr** (to match a particular substring), and **set-pattern** (to match the bit pattern of the instruction). The commands work by modifying an existing breakpoint, so

you first have to set an execution breakpoint and then modify it to match only particular expressions.

For example, to stop when an instruction with the name `add` is executed in a memory range from `0x10000` to `0x12000`, use the following commands:

```
simics> break 0x10000 0x2000 -x
Breakpoint 1 set on address 0x10000 in 'ebony0_system_cmp0.cell10_context',
length 8192 with access mode 'x'
simics> set-prefix 1 "add"
```

Simics will now break on the first `add` instruction encountered (or the last, if the simulation runs backward). For more information, see the *Simics Reference Manual* or use the **help break** command.

3.1.2 Temporal Breakpoints

Unlike an ordinary debugger, Simics can handle temporal breakpoints, i.e., breakpoints in time. Since the concept of time is based on steps and cycles, a temporal breakpoint refers to a specific step or a cycle count as measured by a given processor:

```
simics> cpu0.cycle-break 100
simics> cpu0.step-break 100
```

In the example above, the breakpoints are specified relative to the current time. It is also possible to give temporal breakpoints in absolute time (where 0 refers to the time when the original configuration was set up in Simics). You may set breakpoints in the past as well as in the future.

```
simics> system_cmp0.soc.cpu.cycle-break-absolute 100
simics> system_cmp0.soc.cpu.step-break-absolute 100
```

All the commands **cycle-break**, **step-break**, **cycle-break-absolute**, and **step-break-absolute**, can be given without prefixing them with the CPU. This will set a breakpoint for the current processor.

3.1.3 Control Register Breakpoints

A control register breakpoint is triggered when a selected control register is accessed. The control register is specified either by name or number, and the access type can be any combination of *read* and *write*. For example:

```
simics> break-cr register = msr
```

Note that the exact arguments to this command depend on the target architecture. A list of available control registers can be obtained by tab-completing the *register* argument. See the documentation for **break-cr** in the *Simics Reference Manual* for more information.

3.1.4 I/O Breakpoints

An I/O breakpoint is always connected to a specific device object. The breakpoint is triggered when that device is accessed. The breakpoint is set using the **break-io** command, which take the device name as a parameter. For example, to break on accesses to a device called **hme0**, we would use the following syntax:

```
simics> break-io device = system_cmp0.soc.emac0_dev
```

A list of devices can be obtained by tab-completing the *device* argument.

3.1.5 Text Output Breakpoints

Many simulated machines have a *text console*—a terminal window hooked up to a serial port on the target machine, so that you can type commands to the target and get replies.

A text console can halt the simulation on the occurrence of a given character sequence in the output; this is called a *text output breakpoint*.

- To set a breakpoint, use the command **console.break string**. Simics will stop when *string* appears in the output.
- Use **console.unbreak string** to remove a particular breakpoint.
- All breakpoints can be listed using the **console.list-break-strings** command.

Note: To find out if a specific simulated machine uses a text console, look for an object of class **text-console** in the list provided by **list-objects** once the configuration is loaded.

3.1.6 Graphics Breakpoints

If your target machine has a graphical display (as opposed to just a text console), you can set graphical breakpoints on it. A graphical breakpoint is a (small or large) bitmap image and a pair of coordinates; when the pixels at those coordinates on the simulated display exactly match the breakpoint image, the simulation will halt.

The following commands can be used to save and set breakpoints for a graphics console:

gfx-console.save-break-xy filename left top right bottom [comment]

Let the user specify a rectangular area inside the graphics console using the top left and bottom right corners coordinates. The selected area will be saved as a binary graphical breakpoint file. You can add an optional comment that will be put at the beginning of the breakpoint file.

gfx-console.break filename

Activate a previously saved breakpoint and return a breakpoint id. When a graphical breakpoint is reached, it is immediately deleted and Simics halts execution and returns to the command prompt.

gfx-console.delete id

Delete the breakpoint associated with *id*.

3.1.7 Magic Instructions and Magic Breakpoints

For each simulated processor architecture, a special nop (no-operation) instruction has been chosen to be a **magic instruction** for the simulator. When the simulator executes such an instruction, it triggers a `Core_Magic_Instruction` hap and calls all the callback functions registered on this hap. (*Advanced Features of Simics* explains what haps are, and how to write hap handler functions that get called when a given hap triggers.)

Since magic instructions are just no-operation instructions on hardware, you can run code containing magic instructions on hardware as well as in the simulator, but you will not get any of the extra behavior Simics implements for the magic instruction.

If the architecture makes it possible, an immediate value is encoded in the magic instruction. When the hap is triggered, this value is passed as an argument to the hap handlers. This provides the user with a rudimentary way of passing information from the simulated system to the hap handler.

Magic instructions have to be compiled into the binary files that are executed on the target. The file `magic-instruction.h` in `[simics]/src/include/simics/` defines a `MAGIC(n)` macro that can be used to place magic instructions in your program, where *n* is the immediate value to use.

Note: The declaration of the macros are heavily dependent on the compiler used, so you may get an error message telling you that your compiler is not supported. In that case, you will have to write the inline assembly corresponding to the magic instruction you want to use. The GCC compiler should always be supported.

Note: The magic instruction macro is directly usable only from C and C++; if your program is written in another language, you will have to call a C function that uses the macro, or an assembly function that includes the magic instruction. (If the language supports inline assembly, that can of course be used as well.) For example, in Java it would be necessary to use the JNI interface. Check your compiler and language documentation for details.

A complete list of magic instructions and the range of the parameter *n* is provided in figure 3.1.

Here is a simple example of how to use magic instructions:

```
#include "magic-instruction.h"

int main(int argc, char **argv)
{
    initialize();
    MAGIC(1);                               // tell the simulator to start
                                           // the cache simulation
}
```

Target	Magic instruction	Conditions on n
ARM	orreq rn, rn, rn	$0 \leq n < 15$
H8300	brn n	$-128 \leq n \leq 127$
MIPS	li %zero, n	$0 \leq n < 0x10000$
PowerPC	rlwimi x, x, 0, y, z	$0 \leq n < 32768, n = x \ll 10 \mid y \ll 5 \mid z$
SH	mov rn, rn	$0 \leq n < 16$
SPARC	sethi n, %g0	$1 \leq n < 0x400000$
x86	cpuid %bx, %bx	$0 \leq n < 0x100000$
Cell SPU	or n, n, n	$0 \leq n < 128$

Figure 3.1: Magic instructions for different Simics Targets

```

do_something_important();
MAGIC(2); // tell the simulator to stop
           // the cache simulation

clean_up();
}

```

This code needs to be coupled with a callback registered on the magic instruction hap to handle what happens when the simulator encounters a magic instruction with the arguments 1 or 2 (in this example, to start and stop the cache simulation).

Simics implements a special handling of magic instructions called **magic breakpoints**. A magic breakpoint occurs if magic breakpoints are enabled and if the parameter n of a magic instruction matches a special condition. When a magic breakpoint is triggered, the simulation stops and returns to prompt.

Magic breakpoints can be enabled and disabled with the commands **enable-magic-breakpoint** and **disable-magic-breakpoint**. The condition on n for a magic instruction to be recognized as a magic breakpoint is the following:

```
n == 0 || (n & 0x3f0000) == 0x40000
```

Note that the value 0 is included for architectures where no immediate can be specified. The file `magic-instruction.h` defines a macro called `MAGIC_BREAKPOINT` that places a magic instruction with a correct parameter value in your program.

3.2 Symbolic Debugging

A vital part of a debugger's task is to understand the system being debugged at a higher level than just machine instructions and memory contents. The user thinks in terms of processes, functions, and named variables, so the tool should understand these concepts.

Three classes implement these abstractions in Simics:

context

A **context** object represents a virtual address space. Each processor in the simulated system has a *current context*, which represents the virtual address space currently visible to code running on the processor. Virtual-address breakpoints are properties of contexts; different context objects have separate sets of virtual breakpoints, and by changing a processor's current context, you change its set of virtual breakpoints.

The correctness of the simulation does not depend on contexts in any way; the concept of multiple virtual address spaces is useful for *understanding* the simulated software, but not necessary for just running it. What contexts to create and how to use them is entirely your business; Simics does not care.

By default, each processor has the object **celln-context** as its current context. You may create new contexts and switch between them at any time. This allows you, for example, to maintain separate debugging symbols and breakpoints for different processes in your target machine. When a context is used in this manner (active when and only when a certain simulated process is active), the context is said to *follow* the process.

symtable

A **symtable** object stores debugging and symbol information for an address space. Symtables are properties of **context** objects; each context can have zero or one symtable associated with it.

This symbol information, produced by the compiler, allows Simics to translate back and forth between source code locations and code addresses, variable names and memory locations, and so on.

process trackers, such as linux-process-tracker

Unlike ordinary debuggers, Simics is a full-system debugger: the debugger is in control of the entire simulated system, and not just one of its processes. However, debugging often involves one or a few specific processes, and it would be convenient to hide the rest of the system. This is accomplished by having a separate **context** for each process you care about, and setting it as the current context for a processor precisely when that process is running.

Simics Hindsight's OS awareness provides process trackers that does this job for you. They know enough about the target machine's operating system to be able to see the difference between processes, and swap Simics context objects on OS context switches.

Simics comes with process trackers for some targets, but far from all. *Advanced Features of Simics* describes process trackers in more detail, including how to build your own.

Let us start with an example of how all these pieces can be used together.

3.2.1 Sample Session

Here we inspect a user-space program—the `zsh` shell—running on a 32-bit PowerPC target. Two things are required for this session: a `zsh` binary built with `debug info` (see section 3.2.4), and its source code.

Start by creating a process tracker:

```

simics> new-linux-process-tracker kernel = ppc32-linux-2.4.31
Using parameters suitable for ppc32 Linux 2.4.31.
New process tracker tracker0 created.
simics> tracker0.add-processor system_cmp0.soc.cpu
simics> new-context-switcher tracker = tracker0
New context switcher switcher0 created.

```

Note that we had to tell the process tracker which kernel we use—or rather, what we have to tell it is the value of a number of numerical parameters:

```

simics> tracker0.status
Status of tracker0 [class linux-process-tracker]
=====

                Active : No (0 users)
                Processors : system_cmp0.soc.cpu
                Processor type : ppc32

Process tracking parameters:
    kernel_start : 3221225472
        ts_comm  : 574
        ts_next  : 72
    ts_next_relative : 0
        ts_pid   : 124
        ts_prev  : 76
        ts_state : 0
    ts_thread_struct : 616

```

ppc32-linux-2.4.31 is simply a convenient name for the set of parameters that work with the 32-bit PowerPC Linux 2.4.17 kernel; such predefined parameter sets exist for some of the kernels in the machines shipped by Virtutech. If you want to do process tracking on a kernel for which there is no such predefined parameter set, you will want to look up the process tracker’s **autodetect-parameters** command in the *Simics Reference Manual*.

We also created a context switcher. It handles the rather boring task of listening to the process tracker and actually switching contexts at the right moment, so that one context will follow the process that runs `zsh`. (Context switchers are covered in more details in *Advanced Features of Simics*.)

Now create the symbol table and load the symbols. Note that for this to work, the `zsh` binary must have been built with debug info (this typically means giving the `-g` flag to your compiler).

```

simics> new-symtable zsh_sym
Created symbol table 'zsh_sym'
zsh_sym set for context cell10_context

```

```
simics> zsh_sym.load-symbols ~/zsh-4.2.3/Src/zsh
```

Tell the context switcher to use a special context for the zsh process. Make sure that the new context uses the symbol table:

```
simics> switcher0.track-bin zsh zsh_context
Context 'zsh_context' will be tracking the first process
that executes the binary 'zsh'.
simics> zsh_context->syntable = zsh_sym
```

We would like to start debugging the program at the beginning of its main function. The symbol table can tell us where that is:

```
simics> psym main
{int (int, char **)} 0x100001f8
simics> whereis (sym main)
in main() at /home/jane/zsh-4.2.3/Src/main.c:92
```

sym is like **psym**, except that it only returns the value, and not its type—which is exactly what other commands are expecting as input.

We set a breakpoint at `main`, and let the simulation run:

```
simics> zsh_context.break -x (sym main)
Breakpoint 1 set on address 0x100001f8 in 'zsh_context' with access mode 'x'
simics> c
```

Note that as long as you do not execute a binary named “zsh”, you can run whatever program you want without triggering this breakpoint. That is because it is set on the **zsh_context** context, which will not be activated until zsh is run:

```
$ ls /
bin  dev  home  lib          mnt  proc  sbin  usr
boot etc  host  lost+found  opt  root  tmp   var
$ sh -c 'echo foo'
foo
$ zsh
```

Now the simulation stops:

```
Code breakpoint 1 reached.
[cpu0] v:0x100001f8 p:0x07a2f1f8 stwu r1,-32(r1)
main (argc=0, argv=0x0) at /home/jane/zsh-4.2.3/Src/main.c:92
92      {
```

We can single-step through the code:

```

simics> zsh_context.step
[cpu0] v:0x10000214 p:0x07a2f214  lwz r3,8(r31)
93          return (zsh_main(argc, argv));
simics>
[cpu0] v:0x10043914 p:0x079f6914  stwu r1,-64(r1)
init_jobs (argv=0x1, envp=0x7ffffe14)
        at /home/jane/zsh-4.2.3/Src/jobs.c:1465
1465     {

```

(Just pressing Return at the prompt repeats the last stepping command.)

We can also examine the contents of variables (note that some C expressions must be quoted to prevent the command line parser from eating them):

```

simics> psym envp
(char **) 0x7ffffe14
simics> psym "envp[0]"
(char *) 0x7ffffefa "zsh"

```

Looking at the stack, we can see that we have made two function calls that have not returned since we started single-stepping:

```

simics> stack-trace
#0 0x10043914 in init_jobs (argv=0x1, envp=0x7ffffe14)
    at /home/jane/zsh-4.2.3/Src/jobs.c:1465
#1 0x1003d394 in zsh_main (argc=1, argv=0x7ffffe14)
    at /home/jane/zsh-4.2.3/Src/init.c:1219
#2 0x10000220 in main (argc=1, argv=0x7ffffe14)
    at /home/jane/zsh-4.2.3/Src/main.c:93
#3 0x10129830 in __libc_start_main () in zsh
#4 0x0 in ?? ()

```

3.2.2 Source Code Stepping

Context objects have other source code stepping functions besides **step**. **next**, for example, steps to the next source line without descending into function calls like **step** does. (This is what happened when **step** skipped `setlocale`, but **next** will do this with every function call whether or not we have line number information for them.) And **finish** runs the simulation until the current function returns:

```

simics> zsh_context.finish

zsh_main (argc=1, argv=0x7ffffe14) at /home/jane/zsh-4.2.3/Src/init.c:1219
1219     ttytab['\0'] |= IMETA;

```

All these stepping commands have reverse counterparts: **rstep**, **rnext**, and **uncall**.

Note: The stepping commands expect to step through a single single-threaded process. If the context does not properly follow a single process, or if that process is multithreaded, they may terminate too soon, or too late, or not at all.

The reason for this is that all but the simplest stepping commands rely on the stack pointer to be well-behaved—in particular, that it keeps pointing to the same stack. The presence of multiple threads—or multiple processes not hidden by a process tracker—breaks this assumption.

3.2.3 Symbolic Breakpoints

We saw earlier how **sym** could be used to set a breakpoint on a function. **pos** can be used to set a breakpoint on a source line:

```
simics> pos jobs.c:1465
268712212
simics> hex (pos jobs.c:1465)
"0x10043914"
simics> zsh_context.break -x (pos jobs.c:1465)
Breakpoint 2 set on address 0x10043914 in 'zsh_context' with access
mode 'x'
```

It is also possible to set a breakpoint on data (a watchpoint). The following example sets a data breakpoint on the variable “*argc*”, causing the simulation to stop whenever this variable is read from or written to. The second parameter is the extent of the breakpoint, in bytes.

```
simics> zsh_context.break -r -w (sym "&argc") (sym "sizeof argc")
Breakpoint 3 set on address 0x7ffffd88 in 'zsh_context', length 4 with
access mode 'rw'
```

See section 3.1 for more information about how to use breakpoints.

3.2.4 Reading Debug Information from Binaries

Symbolic information is normally read from file using the `<symtable>.load-symbols` command as in the example above. Currently only ELF binaries can be used, and the debug info must be in either DWARF or STABS format. Also, the files must be present on the host machine—Simics cannot read directly from the file system of the simulated machine.

Here are some things to think about when preparing a binary for debugging:

- Some versions of the Sun WorkShop (Forte) C compiler do not put the debug information in the final executable, but expect a debugger to read it from the object files

directly. This is not supported by Simics, so be sure to use the `-xs` option when compiling.

- If getting sensible stack traces is important, adhere to the target machine's calling and stack frame conventions. In other words, avoid optimizations such as GCC's `-fomit-frame-pointer`.
- Currently, only C is supported, not C++. It is possible to debug programs built from a mixture of C and C++ source, but then only symbols from the C part (and those declared `extern "C"`) will be reliably recognized, for name mangling reasons.
- It is possible to debug dynamically loaded code by specifying the base address of each module when using **load-symbols**, but it is easier to just link the code statically when possible. See section 7.1 for how to find the base address on some systems.

3.2.5 Loading Symbols from Alternate Sources

Sometimes it is desirable to read symbols from a source other than a binary file—perhaps all you have is a text file listing the symbols. The `<symtable>.plain-symbols` command reads symbols from a file in the output format of the BSD `nm` command. Example:

```
000000000046b7e0 T iunique
000000000062ba40 B ivector_table
00000000005a6338 D jiffies
```

The hexadecimal number is the symbol value, usually the address. The letter is a type code; for this purpose, D, B, and R are treated as data and anything else as code.

The symbols do not have any C type or line number information associated with them, but you will at least be able to print stack traces and find the location of statically allocated variables and functions.

3.2.6 Multiple Debugging Contexts

The process tracker and context switcher can without problem handle separate contexts (and symbol tables) for two or more processes at once:

```
simics> new-symtable encode_sym
Created symbol table 'encode_sym'
simics> encode_sym.load-symbols ~/sharutils-4.3.80/src/uuencode
[symtable] Symbols loaded at 0x10000000
simics> switcher0.track-bin uuencode context = encode_context
Context 'encode_context' will be tracking the first process
that executes the binary 'uuencode'.
simics> encode_context->symtable = encode_sym

simics> new-symtable decode_sym
```

```

Created symbol table 'decode_sym'
simics> decode_sym.load-symbols ~/sharutils-4.3.80/src/uudecode
[symtable] Symbols loaded at 0x10000000
simics> switcher0.track-bin uudecode context = decode_context
Context 'decode_context' will be tracking the first process
that executes the binary 'uudecode'.
simics> decode_context->symtable = decode_sym

```

Here, we have created separate contexts for the programs `uuencode` and `uudecode`, loaded their symbols into two `symtables`, and asked the context switcher to associate these contexts with the first processes that execute “`uuencode`” and “`uudecode`”, respectively.

We would like to step through `uudecode` first:

```
simics> decode_context.step
```

The simulation just runs freely now, waiting for us to reach a source line while `decode_context` is active—which will happen as soon as we start `uudecode`:

```

$ ls -laR / | uuencode - | uudecode | wc

[cpu0] v:0x100011a0 p:0x079c1a0 mr r9,r1
main (argc=0, argv=0x0)
    at /home/jane/sharutils-4.3.80/src/uudecode.c:432
432     {

```

We started four programs at once, here. First, `ls` prints a listing of every file on the target machine. This listing is fed to `uuencode`, which encodes it, then feeds the coded result to `uudecode`, which decodes it. The decoded file listing (which is identical to the original listing produced by `ls`) is then fed to `wc`, which counts the number of words in it and prints the result on the terminal.

This description makes it sound like the four programs are run in sequence, one after the other. This is not the case. They all run simultaneously—or rather, since this is a single-processor system, they run interleaved. It works more or less like this:

1. `ls` runs, accumulating output in a buffer. When that buffer is full, it makes a system call that passes the buffered output to the next process, `uuencode`.
2. `uuencode` runs until it has consumed all available input, or until it has to flush its output buffer.
3. `uudecode` runs until it has consumed all available input, or until it has to flush its output buffer.
4. `wc` runs until it has consumed all available input.
5. Repeat until `wc` has finished.

Things may not happen in exactly that order; the only constraint is that a process that is waiting for input cannot be run until some input is available, and a process that has filled its output buffer must wait for the operating system to empty it. As long as the amount of data to be passed is large enough to fill the programs' output buffers many times over (and a directory listing of the entire file system should certainly be large enough), execution will alternate between the different programs. So, when we reach the first source line in `uudecode`, `uuencode` should still be running.

Let's first step a few lines in `uudecode`:

```
simics> decode_context.step
[cpu0] v:0x100012e4 p:0x079cd2e4  lis r9,4102
433      int opt;
simics>
[cpu0] v:0x100012f4 p:0x079cd2f4  li r0,0
438      outname = NULL;
```

This is just like when we were debugging a single program. Now, let's test our assumption by stepping to the next line in `uuencode`. The `step` command will let the simulation run until we reach a new line *in that context*, so we will either stop when `uuencode` gets to run again, or continue running forever if `uuencode` has already finished.

```
simics> encode_context.step
[cpu0] v:0x10000b28 p:0x07a2fb28  mr r0,r3
try_putchar (c=39)
      at /home/jane/sharutils-4.3.80/src/uuencode.c:130
130      if (putchar (c) == EOF)
```

As expected, `uuencode` was still running. In fact, it was busy outputting text for `uudecode` when it was interrupted so that `uudecode` could be started.

If we step to the next line in `uudecode` again, we see that it has now run to the point where it was blocking, waiting for `uuencode` to produce more output:

```
simics> decode_context.step
[cpu0] v:0x10000234 p:0x079cf234  mr r0,r3
read_stduu (inname=0x100464ec "stdin", outname=0x7fffbcf8 "-")
      at /home/jane/sharutils-4.3.80/src/uuencode.c:123
123      int n;
```

3.2.7 Scripted Debugging

It is often useful to access data symbolically from Python scripts. Scripts access the debugging facilities using the `symtable` interface and attributes of the `symtable` class. These are documented in the *Simics Reference Manual*.

For instance, here is a short script to print out the contents of one of the linked lists that `zsh` uses. It uses the `eval_sym` function, which takes a C expression and returns a

(*type, value*) pair. The expression parsed by *eval_sym* may contain casts, struct member selection and indexing.

```
eval_sym = SIM_get_class_interface("syntable", "syntable").eval_sym

def eval_expr(cpu, expr):
    return eval_sym(cpu, expr, [], 'v')

def ptr_str(typed_val):
    (type, val) = typed_val
    return "%(s)0x%x" % (type, val)

def print_linklist(list):
    cpu = current_processor()
    ll = eval_expr(cpu, list)
    first = eval_expr(cpu, ptr_str(ll) + "->first")
    l = []
    def print_tail(node):
        type, val = node
        if val == 0:
            return # end of list
        type, val = eval_expr(cpu, ptr_str(node) + "->dat")
        type, val = eval_expr(cpu, ptr_str(("char *", val)))
        l.append(val)
        next = eval_expr(cpu, ptr_str(node) + "->next")
        print_tail(next)
    print_tail(first)
    print l
```

zsh uses these lists for many different things, among them to store the directory stack. After having given the command `pushd` a few times on the zsh prompt, we can inspect the directory stack by stopping in the `bin_cd` function and printing the linked list "dirstack":

```
simics> zsh_context.break -x (sym bin_cd)
Breakpoint 1 set on address 0x1000282c in 'zsh_context' with access mode 'x'
simics> c
Code breakpoint 1 reached.
[cpu0] v:0x1000282c p:0x07a2b82c stwu r1,-672(r1)
bin_cd (nam=0x30000190 "/", argv=0x0, ops=0x101b0000, func=2147481984)
    at /home/jane/zsh-4.2.3/Src/builtin.c:772
772     {
simics> @print_linklist("dirstack")
['/var/tmp', '/tmp', '/usr', '/home', '/sbin', '/bin', '/root']
None
```

3.3 Reverse Execution

Simics has the capability to run a simulation in reverse, which in many cases can greatly simplify the debugging of complicated software problems.

From a user perspective, there is little difference between running the simulation forwards and backwards; all the standard debugging tools like breakpoints and watchpoints can be used when the simulation is run in reverse. There are, however, some reverse execution specific issues, which are discussed in the following sections.

3.3.1 Using Reverse Execution

Before any reverse operations can be performed, there has to be at least one time bookmark; reverse operations are possible in the region following the first (oldest) bookmark. Depending upon how Simics is invoked (and various user preferences), a time bookmark denoted “start” is sometimes added automatically at the beginning of the simulation. Bookmarks can also be created by enabling reverse execution in an external debugger.

Bookmarks are managed through the commands

- **set-bookmark** *label*,
- **list-bookmarks** and
- **delete-bookmark** [*label* | -all].

Note: From a performance and resource utilization perspective, it might be advantageous to put the first bookmark as close to the region of interest as possible.

Reverse execution support is disabled when all time bookmarks are deleted.

Once reverse execution has been enabled (by the creation of a time bookmark), it is possible to both run the simulation in reverse and to skip backwards in time. The Simics commands to do this are

- **reverse**,
- **reverse-to** *position* and
- **skip-to** *position*.

The **reverse** and **reverse-to** commands run the simulation backwards until a breakpoint occurs or till the oldest time bookmark is reached. The **skip-to** command jumps to a particular point in time, ignoring intermediate breakpoints. It should be noted that skipping can be significantly faster than reversing.

The **skip-to** and **reverse-to** commands take either an absolute step count or a time bookmark as argument.

Most forward executing commands used for debugging has a corresponding reverse variant obtained by adding a reverse prefix. The reverse variant of **step-instruction** is for instance **reverse-step-instruction**.

Any external input (like a human typing on a virtual serial console) is replayed when the simulation is being run forward after a reversal; this guarantees that the simulation will follow the same path as it did originally. For the same reason, all external input is ignored until the point is reached where the first reverse operation was initiated. It is possible to override this behavior with the **clear-recorder** command. This command discards all recorded input and allows an alternate future to take place.

Some changes to the simulation from entities outside the simulation are not recorded by recorders, for example manual changes from the command line interfaces or the graphical user interface. Such changes can make reverse execution operate somewhat unpredictably. It is recommended that any time bookmarks before a non-replayable change of the simulation state is deleted explicitly.

3.3.2 Performance

The usage of time bookmarks has a certain impact on overall performance since it implicitly enables reverse execution support. Normal performance is always obtained if all time bookmarks are removed.

The reverse execution engine optimizes performance for certain reverse operations that are expected to be common. One example of this is that skipping to a bookmark (or to the region just after a bookmark) can be significantly faster than skipping to some other location.

It is possible to tune certain reverse execution parameters in order to optimize for a particular usage pattern (although the default settings should work well in most situations). Tradeoffs exists between:

- reverse performance
- forward performance
- memory utilization
- scope of reversibility

The **rexec-limit** command is the primary tool for adjusting the balance, e.g.

```
simics> rexec-limit steps = 20000000
simics> rexec-limit size_mb = 200
```

The steps limit indicate that the scope of interest is at most the specified number of steps. By imposing a steps limit, resources can be spent more effectively with the drawback that reversal past the limit may not possible.

The size limit imposes a limit on the amount of memory reverse execution may use. If the limit is exceeded, reverse performance will be traded for less memory consumption.

Chapter 4

Using Simics for Hardware Bring-Up and Firmware Development

Simics makes hardware bring-up, firmware development, and other low-level programming tasks easier in a number of ways:

Hardware replacement

A simulator replaces hardware. This has two key benefits during hardware bring-up: you can start working on the software before the hardware is available, and you can have as many copies of the simulated hardware as you like. Both of these translate directly to reduced total development time for the combined hardware+software product.

Inspection and modification

You can inspect the state of the entire simulation—memory, processor registers, device registers, anything—all entirely non-intrusively. And time is simply paused while you do so. You can run backward in time, modify memory or register contents, and then run forward again and see the effects of this change.

Full debug support

The full power of Simics debugging (see chapter 3), with breakpoints, symbolic debugging, reverse execution, scripting, and so on, is available everywhere, even at the very lowest levels.

4.1 A Simple Example

It is easy to write a handful of instructions directly to memory, fill the registers with any necessary values, and manually single-step through this little program:

```
simics> load-file test.bin 0x10000
simics> set-pc 0x10000
simics> %r17 = 4711
simics> si
[cpu0] v:0x00010004 p:0x00010004 mtmsr r3
```

```
simics> si
[cpu0] v:0x00010008 p:0x00010008 xor r0,r0,r0
```

As always in Simics, this kind of thing can be scripted if you expect to run it more than once:

test.simics:

```
run-command-file orange-common.simics
load-file test.bin 0x10000
set-pc 0x10000
%r17 = 4711
continue 12
expect %r17 4713
expect %pc 0x1001c
quit
```

```
$ ./simics test.simics
** Values differ in expect command: 0 4713
$
```

Here, we first call another simics script to set up the machine for us, then run our little test case. The **expects** will cause Simics to exit with an error code (as shown) if the conditions are not met; otherwise, the **quit** will cause Simics to quit successfully.

4.2 Going Further

The simple script in the last section can be extended in several directions:

- **load-file** simply writes the contents of a file directly to memory. There are at least two other options:
 - Using **set** to write values directly to memory. This is useful if the test program is truly just a few instructions long.
 - Using **load-binary** to load an executable in one of the formats Simics recognizes, such as ELF. Unlike **load-file**, this command automatically loads the executable at the right address, and returns the entry point address.
- You can have more complicated stop conditions than simply “run twelve instructions”; for example, you can use execution or data breakpoints (section 3.1.1), control register breakpoints (section 3.1.3), device I/O breakpoints (section 3.1.4), or magic instruction breakpoints (section 3.1.7).

- Various conditions cause Simics to trigger *hops*; for example breakpoints, privilege level changes, magic instructions, and traps. You can easily write a small hop callback function that gets called whenever this happens; such a callback could terminate the simulation (indicating success or failure), or simply log or change some value. Hops and hop callbacks are covered in *Advanced Features of Simics*.

Chapter 5

Using Simics for Application-Level Development

5.1 Launching an Application

Launching an application program in Simics is done just like on real hardware: get the program onto the machine's disk, and instruct the operating system to run the program—by command line, double-click, or whatever.

The only challenge here is to import files into the simulation; there are two options:

- You can create a disk image containing your files, and insert this image in the simulated target machine. Typically, you would create a CD-ROM image and insert into the target's CD-ROM drive. This is described in *Advanced Features of Simics*.
- You can use SimicsFS to directly access the host's file system from the target. (This is supported only for Linux and Solaris targets.) This is described in *Advanced Features of Simics*.
- You can use a *real-network* connection to transfer the files from host to target using any network file transfer protocol, such as HTTP, FTP, NFS, and so on. This is described in *Ethernet Networks in Simics*. Unless you are going to use a real-network connection anyway, this is more work to set up than the first two options.

Note: SimicsFS and real-network are a way for information from the real world to leak into the simulation. This makes the simulation indeterministic; if you want a deterministic simulation, you should disable them after finishing the file transfer.

Once you have imported your files, consider checkpointing the state at that point, so that you can easily restart your debugging session if you are not one of those who always find the bug on the first try. Checkpointing is detailed in *Advanced Features of Simics*.

5.2 Debugging an Application

When debugging a userspace application, you will typically make use of a slightly different feature set than when debugging low-level software (see chapter 4). Breakpoints will still be useful, of course, especially magic breakpoints and regular memory breakpoints on virtual addresses (see chapter 3.1). And reverse execution just gets more useful as your program increases in size.

For userspace application debugging on Linux, you will want to familiarize yourself with process trackers, which provide a way to hide the parts of the system that are not your application. See section 3.2. This chapter also covers many other userspace debugging topics.

You also have the option of controlling Simics with an external debugger, such as GDB (see chapter 7).

5.3 Debugging Networked Applications

There are two qualitatively different ways to set up a networked system in Simics:

Real network

You can let a simulated target machine communicate with real machines over a real-network connection. This lets you keep the simulated system small, but has one serious drawback: the real world cannot be checkpointed or scripted from within Simics, and is not deterministic. See section 2.3 for a list of what you will be missing out on.

Virtual network

If you set up two or more target machines inside the simulation and connect them in a network, you have a purely virtual network. The combined system is just as deterministic and checkpointable as a single simulated target, so you lose none of the advantages of simulation.

Both of these options are described at length in *Ethernet Networks in Simics*.

Chapter 6

Using Simics with Wind River Tornado or Workbench

Simics can function as a debugger backend to Wind River®'s Tornado® and Workbench IDEs.

The normal way to control a target machine from these IDEs is to have a *WDB agent* running on the target. It talks to the IDE using the Wind River Debug (WDB) protocol, and carries out low-level debugging tasks such as starting, stopping, setting breakpoints, and reading memory.

Simics comes with a built-in WDB agent, **wdb-remote**. Just like a WDB agent that runs on the target machine, it listens for commands on a UDP port, and can start, stop, step, and inspect the target. However, unlike an agent running on the target, it is completely non-intrusive since the whole protocol is handled by the simulator. No debugging code runs on the target, and it is unaware of being halted since it is the entire simulation that stops. You can single-step, use breakpoints and inspect memory contents, and the target will never know.

6.1 Starting the Simics WDB Agent

Instances of **wdb-remote** are created with the **new-wdb-remote** command. At the simics console, do for example

```
simics> new-wdb-remote processor = cpu0 cpu-type = PPC604 \  
          mem-size = 0x1000000 host-pool-base = 0x0 \  
          host-pool-size = 0x10000
```

Use the tab completion to get suggestions for possible *processor* and *cpu-type* arguments. *mem-size* is the number of bytes the agent will claim that the target has; it is not vital to get this right. *host-pool-base* and *host-pool-size* define a client-owned area of the target's memory; if you are not going to download code onto the target or something like that, this area will not be used, and you can give an arbitrary region, for example the one given here.

For a more thorough explanation, including optional parameters, see the documentation for the **new-wdb-remote** command in the reference manual.

6.2 Connecting Tornado

First create a **wdb-remote** agent, as described in section 6.1. Then let Tornado connect to it:

1. From the Tornado menu, choose **Tools** → **Target Server** → **Configure**.
2. Press *New* to create a new target server configuration.
3. Choose **Target Server Properties** → **Target Server File System**; make sure that *Enable File System* is not enabled.
4. Choose **Target Server Properties** → **Back End**; select *wdbrpc*.
5. Choose **Target Server Properties** → **Core File and Symbols**; select *File*, and fill in the name of your VxWorks image.
6. For *Target Name/IP Address*, fill in the name of the host where Simics is running, or *localhost* if it is the same computer that you run Tornado on.
7. Press *Launch*.

Thankfully, most of these steps only have to be carried out the first time you connect.

Tornado is now connected to Simics. For example, you can inspect the target with the browser (**Tools** → **Browser**). (Note that if you have not let the target machine boot before you try this, or if the target machine does not run VxWorks, parts of the browser will not work.)

6.3 Connecting Workbench

First create a **wdb-remote** agent, as described in section 6.1. Then let Workbench connect to it:

1. From the Workbench menu, choose **Target** → **New Connection**. Choose *VxWorks 6.x Target Server connection*.
2. In the dialog that appears, fill in the hostname of the computer you are running Simics on, or *localhost* if it is the same computer that you run Workbench on. Press **Next**, then **Next** again.
3. Make sure that *Query target object lists and target object states on connect* is not enabled. Press **Finish**.
4. An icon for your new target connection appears in the *Target Manager* window. (It autoconnects when you first create it; if you want to reconnect later, just use this icon.) Right-click on the sub-icon that appears once the connection is established. Choose **Attach to Kernel (System Mode)**.

Workbench is now connected to Simics; you should see a disassembly of the memory region surrounding the current address of the program counter.

6.4 Limitations

Unlike an agent running inside the target OS, **wdb-remote** has no OS awareness. This means that it cannot operate on specific VxWorks or Linux tasks; it only supports the system context. For example, it is only possible to set breakpoints, single step, and suspend the whole target machine, not individual tasks.

Chapter 7

Using Simics with GDB

This chapter describes how to use **gdb-remote**, a Simics module that lets you connect a GDB session running on your host machine to the simulated machine using GDB's remote debugging protocol, and use GDB to debug software running on the target machine.

If you load the **gdb-remote** module in Simics, you can use the remote debugging feature of GDB, the GNU debugger, to connect one or more GDB processes to Simics over TCP/IP. In order to do this, you need a GDB compiled to support the simulation's target architecture on whichever host you're running. The **gdb-remote** module only supports version 5.0 or later of GDB, but other versions may work as well. Unfortunately GDB's remote protocol does not include any version checking, so the behavior is undefined if you use other versions. For information on how to obtain and compile GDB, see section 7.3.

Note: In order to use GDB with Simics running on a Windows host you need to either compile GDB for Cygwin, or run GDB remotely from a UNIX/Linux machine. The examples in this chapter assume the former.

To connect a GDB session to Simics, start your Simics session and run the **new-gdb-remote** command, optionally followed by a TCP/IP port number, which defaults to 9123 otherwise. This will automatically load the **gdb-remote** module.

When a configuration is loaded, Simics will start listening to incoming TCP/IP connections on the specified port. Run the simulated machine up to the point where you want to connect GDB. To inspect a user process or dynamically loaded parts of the kernel, the easiest solution might be to insert magic instructions at carefully chosen points. For static kernel debugging, a simple breakpoint on a suitable address will solve the problem.

Note: When debugging the start-up phase of an operating system, it might happen that gdb gets confused by the machine state and disconnects when you try to connect. In this case, execute a few instructions and try again.

Once Simics is in the desired state, start your GDB session, load any debugging information into it, and then connect it to Simics using the **target remote host:port** command, where *host* is the host Simics is running on, and *port* is the TCP/IP port number as described above. Here is a short sample session using *bagle*, a Sun UltraSPARC machine running Linux:

```
(gdb) set architecture sparc:v9a
(gdb) symbol-file vmlinux
Reading symbols from vmlinux...done.
(gdb) target remote localhost:9123
Remote debugging using localhost:9123
time_init () at /usr/src/linux/include/asm/time.h:52
(gdb)
```

Note: For some architectures, you need to give a command to GDB before connecting (the **set architecture** command in the session above). These are tabulated in the reference manual's section on **gdb-remote**, and will also be printed on the Simics console when you run **new-gdb-remote**.

From this point, you can use GDB to control the target machine by entering normal GDB commands such as **continue**, **step**, **stepi**, **info regs**, **breakpoint**, etc.

Note that while a remote GDB session is connected to Simics, the Simics prompt behaves a little differently when it comes to stopping and resuming the simulation. While the GDB session is at prompt, it is impossible to continue the simulation from within Simics (e.g., by using the **continue** command). However, once you continue the execution from GDB, you can stop it from GDB (by pressing control-C), which causes the simulation to stop and makes both GDB and Simics return to their prompts, or you can stop the simulation from the Simics prompt (also by pressing control-C). This only makes Simics return to prompt, while GDB will still think the target program is running. In this state, you should continue the simulation from the Simics prompt before attempting to use GDB.

You can also force GDB back to prompt using the **gdb0.signal 2** command in Simics, which tells the GDB session that the simulated machine got a **SIGINT** signal. **gdb0** here refers to the configuration object created on the fly when the GDB session connected to Simics. You can connect several GDB sessions to one Simics; each connection will be associated to one **gdbm** object.

Since GDB is not the most stable software, especially when using remote debugging, it unfortunately hangs now and then. To force Simics to disconnect a dead connection, you can use the **gdb0.disconnect** command.

Note that the **gdb-remote** module does not have any high-level information about the OS being run inside Simics. This means that in order to examine memory or disassemble code, the data or code you want to look at has to be in the active TLB.

Note: When using **gdb-remote** with targets supporting multiple address sizes (such as x86-64 and SPARC), you must have a GDB compiled for the larger address size. For SPARC, run GDB's configure script with the **--target=sparc64-sun-solaris2.8** option.

7.1 Remote GDB and Shared Libraries

It takes some work to figure out how to load symbol tables at the correct offsets for relocatable object modules in GDB. This is done automatically for normal (non-remote) targets, but for the remote target, you have to do it yourself. You need to find out the actual address at which the shared module is mapped in the current context on the simulated machine, and then calculate the offset to use for GDB's **add-symbol-file** command.

To find the addresses of the shared libraries mapped into a process' memory space under Solaris, use the **/usr/proc/bin/pmap pid** command. The start address of the text segment can be obtained from the `Addr` field in the `.text` line of the output from **dump -h file**.

Under Linux, the list of memory mappings can be found in the file `/proc/pid/maps` (plain text format). The `VMA` column of the `.text` line of the output from **objdump -h file** contains the start address of the text segment.

Using these two values, *map address* and *text address*, you should use *map address + text address* as the offset to **add-symbol-file** (it has to be done this way to compensate for how GDB handles symbol loading).

The following example uses a SPARC running Linux (`sim-sh#` denotes the shell in the simulated computer):

```
sim-sh# ps
  PID TTY          TIME CMD
  :
  461 ttyS0    00:00:00 bash
sim-sh# cat /proc/461/maps
0000000000010000-0000000000060000 r-xp 0000000000000000 08:11 90115  /bin/bash
000000000006e000-0000000000076000 rwxp 000000000004e000 08:11 90115  /bin/bash
  :
0000000070040000-0000000070138000 r-xp 0000000000000000 08:11 106505 /lib/libc-2.1.3.so
0000000070138000-0000000070140000 ---p 00000000000f8000 08:11 106505 /lib/libc-2.1.3.so
0000000070140000-000000007014e000 rwxp 00000000000f0000 08:11 106505 /lib/libc-2.1.3.so
  :
sim-sh# objdump -h /lib/libc-2.1.3.so

/lib/libc-2.1.3.so:      file format elf32-sparc

Sections:
Idx Name              Size              VMA                LMA                File off  Algn
  :
  14 .text              000ce338          000000000001e400  000000000001e400  0001e400  2**9
```

From this output, we derive that the bash process with PID 461 has `/lib/libc-2.1.3.so` located at starting address `0x70040000`. The `.text` symbols starts at address `0x1e400`, so if we connect GDB to Simics we have to add the symbols with an offset of `0x70040000 + 0x1e400 = 0x7005e400`. Before running the following commands, we stopped Simics using control-C while it was executing code in the bash process:

```
(gdb) dir ~/glibc-2.1.2/malloc
Source directories searched: /home/joe/glibc-2.1.2/malloc:$scdir:$scwd
(gdb) add-symbol-file libc.so.6 0x7005e400
add symbol table from file "libc.so.6" at
      .text_addr = 0x7005e400
(y or n) y
Reading symbols from libc.so.6...done.
(gdb) target remote localhost:9123
Remote debugging using localhost:9123
__libc_malloc (bytes=0x14) at malloc.c:2691
2691         if (victim == q)
(gdb) next
2693         q = next_bin(q);
(gdb)
```

7.2 Using GDB with Reverse Execution

`gdb-remote` supports an extension to the GDB remote protocol that allows the debugger to control the reverse execution functions in Simics. An unmodified GDB does not know about this extension. Virtutech provide two patches to GDB, one for GDB-6.3 and one for GDB-6.8. The patches can be downloaded from <https://www.simics.net/pub/>. The patches can be used with standard GDB to compile GDB with reverse execution capabilities, see section 7.3.

Simics also comes with a special package including prebuilt GDB-6.8 binaries with reverse execution support for the most common target architectures and hosts.

The patches adds a number of reverse execution commands to GDB. The commands in the GDB-6.3 patch all have the **`reverse-`** prefix, and are more or less just the reverse version of the standard commands. The GDB-6.8 patch adds a number of commands with the same functionality, but with new names. The list below lists the *GDB-6.3/GDB-6.8* commands and a short description.

`reverse-continue/reverse`

Run in reverse until a breakpoint or watchpoint is hit, or to the point where the reverse execution machinery runs out of history.

`reverse-next/previous`

Run in reverse and stop at the previous source code line. Will skip subfunction calls.

`reverse-nexti/previousi`

Run in reverse and stop at the previous instruction. Will skip subfunction calls.

`reverse-step/unstep`

Run in reverse and stop at the previous source code line. Will enter subfunction calls.

reverse-stepi/unstepi

Run in reverse and stop at the previous instruction. Will enter subfunction calls.

reverse-finish/uncall

Run in reverse till the point where the current function is called.

Normal break- and watchpoints set with **break** and **watch** will also be triggered when running in reverse using **reverse-continue**

A small example of how to use **reverse-next**:

```

22         for (i = 0; i < 10; i++)
(gdb) p i
$2 = 0
(gdb) n
24         c = foo (i) + c;
(gdb) p i
$3 = 1
(gdb) reverse-next
22         for (i = 0; i < 10; i++)
(gdb) p i
$4 = 0

```

The amount of history that reverse execution keeps is limited; it is only possible to reverse back to the point where reverse execution was enabled. If GDB recognizes that reverse execution has ran out of history, it will report an error. Note that this is not a fatal error, and the debugging session can continue, but without the possibility to reverse further than to the point where GDB reported the error.

```

(gdb) reverse-continue
Continuing.

```

```

No more history to reverse further.
_start () at start.c:17
17     {

```

7.3 Compiling GDB

If you do not want to (or cannot) use one of the GDB executables in *host/sys/bin/*, you will most likely have to compile GDB from source, even if your system already has GDB installed. The reason for this is that a given GDB executable is specialized both for the architecture of the computer you run it on (host), and the architecture of the computer that runs the programs you want to debug (target). Any GDB already installed on your computer will have target identical to host, but this is often not what you want when your target is a simulated machine.

The first step is to get the GDB source code. You can either get the unmodified GDB from <ftp://ftp.gnu.org/>, or a reverse execution-aware GDB from <https://www.simics.net/pub/>. In either case, the source will be packaged in a `.tar.gz` file.

The second step is to make sure you have all the tools necessary to compile GDB, such as GNU Make and a C compiler. On a Linux or Solaris system, you probably have them already. On Windows, you will have to install Cygwin; get it at <http://www.cygwin.com/>.

That done, unpack and configure GDB like this:

```
~> tar zxfv gdb-6.3.tar.gz
~> cd gdb-6.3
~/gdb-6.3> ./configure --target=powerpc64-elf-linux
```

(On Windows, be sure to enter these commands in the `bash` shell installed as part of Cygwin.)

The `--target` flag to `configure` specifies which target architecture your new GDB binary will be specialized for (in this example, a 64-bit PowerPC). These flags are tabulated in the reference manual's section on `gdb-remote`, and will also be printed on the Simics console when you run `new-gdb-remote`.

```
~/gdb-6.3> make
```

The build process takes a while; when done, it will have left a `gdb` executable in the "gdb" subdirectory. You can execute it directly from that location:

```
~/gdb-6.3> ./gdb/gdb
```

Chapter 8

Using Simics with Other IDEs

Simics does not explicitly support IDEs other than those listed in this manual. If you would like to use Simics with such an IDE, there are a few options you could try:

- Use Simics's own user interface to control the debugging process (see chapter 3).
- If your IDE speaks the WDB (Wind River Debug) protocol, you can try using a **wdb-remote** Simics object to connect (see chapter 6).
- If your IDE speaks the GDB serial protocol, try connecting it directly to Simics with the help of a **gdb-remote** object (see chapter 7).
- If your IDE uses GDB as a debugger backend, try to make this GDB instance connect to Simics with **gdb-remote**.

Index

B

- bookmarks, [24](#)
- breakpoint, [8](#)
 - control register access, [11](#)
 - graphics, [12](#)
 - I/O, [12](#)
 - magic breakpoint, [13](#)
 - memory, [9](#)
 - set-pattern, [10](#)
 - set-prefix, [10](#)
 - set-substr, [10](#)
 - symbolic, [19](#)
 - temporal, [11](#)
 - text output, [12](#)

C

- C compiler
 - GCC, [20](#)
- C compiler
 - Forte, [19](#)
 - GCC, [13](#)
 - Sun Workshop, [19](#)
- C++, [20](#)
- CD-ROM, [29](#)
- context, [15](#)
 - current, [15](#)
- context switcher, [16](#)
- current context, [15](#)

D

- debug information, [19](#)
- debugging, [14](#)
 - GDB, [34](#), [38](#)
 - memory spaces, [20](#)
 - multiple processes, [20](#)
 - remote, [34](#)
 - reverse execution, [37](#)
 - scripted, [22](#)

- shared libraries, [36](#)
 - symbolic, [14](#), [34](#)

- disable-magic-breakpoint, [14](#)
- DWARF, [19](#)

E

- ELF, [19](#), [27](#)
- enable-magic-breakpoint, [14](#)
- eval_sym, [22](#)
- expect, [27](#)

F

- Forte, [19](#)
- FTP, [29](#)

G

- GCC, [13](#), [20](#)
- GDB, [34](#), [40](#)
 - compiling, [38](#)
- GDB serial protocol, [40](#)
- gdb-remote, [34](#), [40](#)
- graphics breakpoints, [12](#)

H

- haps, [28](#)
- hex, [19](#)
- HTTP, [29](#)

I

- indeterminism, [29](#)

L

- linked list, [22](#)
- linux-process-tracker, [15](#)
- load-binary, [27](#)
- load-file, [26](#)

M

- magic breakpoint, [13](#)

magic instruction, 13
 passing arguments, 13
 memory mappings, 36
 multithreaded process, 19

N

network
 real network, 30
 virtual network, 30
 new-wdb-remote, 31
 NFS, 29
 nm, 20
 output format, 20

P

plain-symbols, 20
 pos, 19
 process
 follow, 15
 process tracker, 15
 psym, 17, 18
 Python, 22

R

Real network, 30
 real-network, 29
 remote GDB, 34
 reverse execution, 24
 bookmarks, 24
 performance, 25
 run-command-file, 27

S

set-pattern, 10
 set-pc, 26
 set-prefix, 10
 set-substr, 10
 sharutils, 20
 SimicsFS, 29
 simulation
 limits, 6
 time, 7
 single-step, 17
 STABS, 19
 stack-trace, 18
 Sun WorkShop, 19

sym, 17
 symbols
 loading, 19, 20
 symtable, 15

T

text console, 12
 text output breakpoint, 12
 time, 7
 Tornado, 31, 32

U

uudecode, 21
 uuencode, 20

V

Virtual network, 30
 VxWorks, 31

W

watchpoint, 19
 WDB agent, 31
 WDB protocol, 31, 40
 wdb-remote, 31, 40
 whereis, 17
 Wind River Debug protocol, 40
 Workbench, 31, 32

Z

zsh, 15



Virtutech, Inc.

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

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

<http://www.virtutech.com>