SID Simulator User's Guide

SID Simulator User's Guide

Copyright © 2001 by Red Hat, Inc.

Print edition `$Date: 2001/10/16 16:25:21 $`

This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/[1]).

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

Table of Contents

1. Introduction **7**
- Benefits of Using the SID Simulator 7
- The Simulator Toolkit 7
- About this Manual 8
  - SID Architecture 8
  - SID Configuration File 8
  - SID Component Library 8
  - Using SID 9

2. The SID Architecture **11**
- Component Types 11
  - Hardware Model Components 11
  - Software Model Components 11
  - Bridge Components 12
  - Special Function Components 12
- Communication Between Components 12
- System Organization 13
  - Main Loop, Schedulers 13
  - Peripherals, Memory Mapping 13
  - CPU, Monitor Emulation, Debugger Interface 14

3. Working with the SID Configuration File **15**
- The Configuration File Contents 15
  - Technical Notes 15
- Creating a New Configuration File 15
- Command Syntax 16
  - The **load** Command 16
  - The **new** Command 17
  - The **connect-pin** and **disconnect-pin** Commands 18
  - The **connect-bus** and **disconnect-bus** Commands 19
  - The **set** Command 20
  - The **relate** and **unrelate** Commands 20
  - Lexer Issues 21

4. How to Understand Component Reference Information **23**
- Finding Component Reference Information 23
- Parts of the Component Documentation Template 23
  - Name 23
  - Synopsis 24
  - Functionality 24
  - Environment 26
  - SID Interface Reference 26
  - References 28

5. Using SID **29**
- Running SID 29
  - Direct invocation 29
  - Invocation through a wrapper 29
- SID Startup 31
- Creating an Application to Run on SID 31
- Debugging Using SID 32
  - Using GDB 32
  - Profiling 33

Chapter 1. Introduction

The SID simulator consists of an engine that loads and connects simulated components, based on a configuration file, and runs simulation sessions. The SID Simulation Toolkit provides a set of ready-made components that you can configure to create your own simulation environment.

Benefits of Using the SID Simulator

In recent years, there has been notable growth in the use and the application of embedded systems. However, the improvements to the design and testing tools have not kept pace with the rapid development of customized hardware parts. This simulation tool has been designed to help close this gap and meet the needs of embedded software developers. The simulation of the target environment enables embedded software developers to analyze and test their software, even in the absence of the physical hardware.

In many respects, the virtual target simulation enables a better development environment, since you can analyze a system design's functionality and performance in ways that are not possible on the physical hardware.

There are three key benefits that can be gained from using this tool:

- With a simulated target, the debugger can see different states of the software that cannot be viewed on the hardware. Consequently, testing on the simulator tool is often faster and more flexible that performing similar software testing on the physical hardware.
- Using the SID simulation environment, you can gain more insight into the functional behavior of the software on your system. This results in a shortened time-to-market.
- Testing on a simulated target allows you to use system building blocks to explore alternative solutions before the hardware components are created. This can reduce the overall cost of the system implementation.

However, in order to get the full value of the simulation tool. You do need to understanding the differences between the actual hardware and virtual target simulation environments.

The Simulator Toolkit

Designing an embedded system is the process of implementing a desired functionality by using a set of physical components. Using a similar process, the SID Simulation Toolkit enables you to create a virtual simulation environment using simulated components, even when physical components are unavailable.

Whether you are working with physical components or on a virtual simulation environment, your system can be viewed as a collection of simpler subsystems. In other words, the system is made up of components, with a list of methods or rules that tie all the pieces together in order to create the desired functionality.

The virtual simulation model that you use to test your embedded system software needs to be a natural one in order to help you debug your design. The virtual simulation model must be flexible so that you can choose different models in the different phases of the development process. The simulation of the model should also accommodate different levels of accuracy, such as functional and cycle-accurate. With a

functionally accurate simulator, you can compile and execute the design model directly on a host machine without paying much attention to the simulation time. With a cycle-accurate simulation, you can execute the design model in a timed fashion.

About this Manual

This manual is a guide to the SID Simulator. It is a reference guide for users who want to use and configure the simulation components provided by Red Hat.

The manual is organized according to how an user would first approach using the SID tool by providing information about the overall SID architecture; how to run, use and customize the SID tool for your specific requirements, and how to gather information from the running system.

SID Architecture

This release contains the tools and capabilities to customize the configuration of the virtual simulation model by providing a library of existing components; a documented interface to enable the designer to add other components. All SID components have been designed to enable pluggability into the SID framework. Each component exports a set of pines and bus connectors and therefore has no knowledge of the other components that are connected to it.

SID Configuration File

The SID configuration file is an editable file that configures a SID simulation run. The configuration file defines the simulation contents, topology, and initial state. We supply several configuration files that simulate selected target boards.

For information on how to create or modify a SID configuration file, refer to Chapter 3>.

SID Component Library

SID comes with a number of components, each of which can be modified, configured, or connected to any other independently. They are self-contained, and individually documented. Existing components include:

- CPUs, such as `hw-cpu-arm7` or `hw-cpu-m32r/d`
- disks, such as `hw-disk-ide`
- rom-monitors or low-level operating systems, such as `sw-gloss-arm_angell`
- glue, such as `hw-glue-bus-mux` or `hw-glue-probe-bus`
- interrupts, such as `hw-interrupt-arm_ref`
- LCDs, such as `hw-visual-lcd` or `hw-lcd-char-display`
- binary loaders, such as `sw-load-elf`
- mappers, such as `hw-mapper-basic`
- memory, such as `hw-memory-ram_rom-basic`
- MMUs, such as `hw-remap_pause-arm_ref`
- UARTs, such as `hw-uart-ns16550`

- parallel ports, such as `hw-parport-ps_2`
- RTCs, such as `hw-rtc-ds1x42`
- timers, such as `hw-timer-arm_ref`
- audio codecs, such as `hw-audio-sid`
- console IO facilities, such as `sid-io-stdio` or `sid-io-socket`

For information on how to understand the SID component documentation, refer to Chapter 4>.

Adding new components is straightforward and does not require any modifications to SID. For more information on the SID Component Library, refer to the SID Component Developers Kit Reference Guide.

Using SID

SID is packaged as a standalone command-line program that reads and executes a configuration file (see Chapter 3>). A typical session with SID begins with compiling or assembling code for the simulated system to run, using standard a cross-development toolchain, and proceeds through loading the target binary into the simulation environment, stepping the simulator through a supervised execution, shutting the simulation down and analyzing the results. While running a simulation, SID can interface with POSIX standard I/O, a TK-based visual simulation monitor, the `gdb` debugger, and the `gprof` profiler.

For information on invoking, customizing, and interfacing with SID, refer to Chapter 5>.

Chapter 2. The SID Architecture

A simulated environment provides embedded software developers with a number of advantages over a typical hardware model. Working in a simulated environment provides a rapid way to examine the complex interactions of the embedded software on the simulated hardware system.

SID is a simulation framework for supporting embedded systems software development. It features a modular architecture of loosely-coupled software components that interact with each other to simulate the behavior of physical hardware parts. This modularity allows users or programs to monitor or interact with a running simulation.

SID insists on a decomposition of a simulation into independent components that share a fixed low-level API, which defines all possible inter-component communication mechanisms. The API is small and general, on which application- or device-specific behaviors are layered.

SID typically reads configuration files to create and manage the web of low-level connections between all needed components. A configuration file may spell out the model of an entire hardware development board, or may add instrumentation to an existing model.

Component Types

The SID package includes many types of components. The component types fall into the following categories:

- hardware model
- software model
- bridge
- special function

Hardware Model Components

A hardware model type component simulates some sort of target hardware device, so it tends to connect to other components through hardware-oriented connections.

Examples include:

- a CPU, `hw-cpu-arm7t`
- a hard drive interface, `hw-disk-ide`
- an address space decoder/mapper, `hw-mapper-basic`
- memory, `hw-memory-ram/rom-basic`
- a teletype, `hw-visual-tty`

These tend to be the busiest components in a running simulation.

Software Model Components

A software model type component crosses the hardware-software abstraction boundary by emulating the operation of target software.

Examples include:

- a ROM monitor, `sw-gloss-arm/angel`
- a debugging stub, `sw-debug-gdb`
- profiling instrumentation, `sw-profile-gprof`
- an executable loader, `sw-load-elf`

Bridge Components

A bridge type component crosses simulation implementation language boundaries by mapping between a different API and the SID low-level API. It allows a foreign simulation system to impersonate a SID component, and sometimes vice versa.

Examples include:

- a Tcl/Tk bridge, `bridge-tcl`

Special Function Components

A special function type component carries out tasks related to the infrastructure of the simulation, or crossing the target/host abstraction boundary.

Examples include:

- an event scheduler, `sid-sched-sim`
- a configuration manager, `sid-control-cfgroot`
- a host network interface, `sid-io-socket`

Communication Between Components

The SID API provides only a few ways for components to communicate with each other. Hardware type components tend to use only the *pin* and *bus* mechanisms, and other types also use the *attribute* and *relation* mechanisms. All of these communication mechanisms may be set up in a SID configuration file.

The SID *pin* mechanism allows a component to send or receive broadcast events to or from connected components. Each event carries a small integer value as a parameter; how this value is interpreted is up to the involved components. This is an abstraction of individual electrical signal lines that connect hardware parts, modelling a level transition by a function call. A component may maintain several different pin connection networks. Typically, clocking and initialization or shutdown signals are distributed through dedicated pin connections.

The SID *bus* mechanism allows a component to read or write addressable memory or registers of another component. This is an abstraction of the address, data, and control buses that connect hardware parts together, modelling protocols and signals by a function call from a component (a bus master) to another (a bus slave). A bus mastering component may have several named accessors. Each may point to a bus slave component's named bus.

The SID *attribute* mechanism allows a component to read or write named internal states of another component. These states are considered to be opaque strings and may represent anything from hexadecimal values of registers through configuration variables to a binary dump of the entire state of the component. Attributes may be categorized, so that standard formatting and interpretation may be applied to groups of attributes.

The SID *relation* mechanism allows a component to make any SID API call to another component. A component that supervises others has a specific named relationship to them. A supervisor's relation stores pointers to the supervised components.

System Organization

One may consider the fixed low-level SID API as a socket, or a standardized connector between simulated components. Think of SID components as integrated circuits that plug into these sockets and configuration files as the circuit wiring that connects the sockets to each other. To simulate an entire board, many different components will be needed, and their connections need to be set up just right for SID to work.

While the configuration process is described in detail in a later section(see Chapter 3>), there are common patterns of configuration in typical SID simulation runs. These sketches outline some of these patterns.

Main Loop, Schedulers

In a running SID simulation, activity occurs in a single thread. It passes from a central top-level loop down through components and back, by calling SID API functions. More specifically, the top level loop signals each iteration on a designated pin. Each component that connects a pin to it will be invoked to do some work. This work may include internal calculations, changing attributes, or could involve making pin- or bus-related function calls to other components. Then those components do some work, and so on. When these work units are complete, control returns to the top level loop, which then goes around for another iteration.

There also exist control pins that cause and indicate the initialization and shutdown of a simulation run. This allows components to monitor or effect changes to the top-level looping. For example, simulated CPU executes the target program's `exit()` system call, the component that interprets system calls will signal this event on a pin. If that pin is connected to the main loop's shutdown control pin, the simulation will shortly end.

Many components may wish to regulate the rate at which control is passed to them. They may have nothing to do at times (like an idle DMA controller). Some may wish to act at independent frequencies (like a CPU and a timer). Several special function components exist to keep simulation time and dispatch events to other components only at intervals of interest: these are the scheduler (`sid-sched`) components.

There is usually one host-time scheduler and one target-time scheduler in a simulation run. Both are connected to the top level activity pin. Components that interface between the host and the simulated target often use the host-time scheduler (`sid-sched-host`), so they get activity signals at rates related to passing wall-clock time. Components that model the hardware of the target system are generally connected to the target-time scheduler (`sid-sched-sim`), so target activity may be suspended/resumed en masse by a debugger.

Peripherals, Memory Mapping

A typical SID configuration one or more CPU models, some banks of memory, and several peripherals. When the CPU executes a load or store instruction, the correct SID component must be addressed to carry out the memory access. This decoding operation is performed by an intermediary: a mapper component that is configured with the target board's memory map.

The CPU component is usually connected to the slave side of the mapper, and the master side of the mapper is connected to multiple slave peripheral components. Other bus master components may also connect to the mapper similarly. When a bus master makes a read or write call, the interposed mapper component decodes the address, and passes the read or write call to the specific component being addressed.

In effect, the mapper transforms the CPU's view of the address space to peripheral-centric views. This way, neither the CPU nor the peripheral is aware of the target board's memory map: this knowledge is encapsulated in a mapper.

There may be several mapper components in a simulation run. This is appropriate if there are separate instruction and data address spaces for a Harvard architecture CPU, or if address spaces are nested, or partly shared between multiple masters.

CPU, Monitor Emulation, Debugger Interface

Chapter 3. Working with the SID Configuration File

To create a simulated system, you need to select the components you want to use, describe their properties, and specify how they should connect together. All of this information is specified in the simulator configuration file.

The simulator configuration file consists of a series of commands. The instructions are interpreted by a special SID component with type `sid-control-cfgroot`. This component is always present in the SID main-line program **sid** or **sid.exe** .

The Configuration File Contents

The configuration file consists of three major sections:

- A listing of component libraries, which are to be dynamically loaded libraries
- A set of commands to instantiate any required SID components
- A set of commands that connect and configure the components.

The configuration process consists of reading each command from a configuration file, from top to bottom. If any lines are not recognized as syntactically valid commands, or if any commands fail (see below), the configuration process continues, but the simulation will abort before entering the main loop. Error messages are produced in these cases.

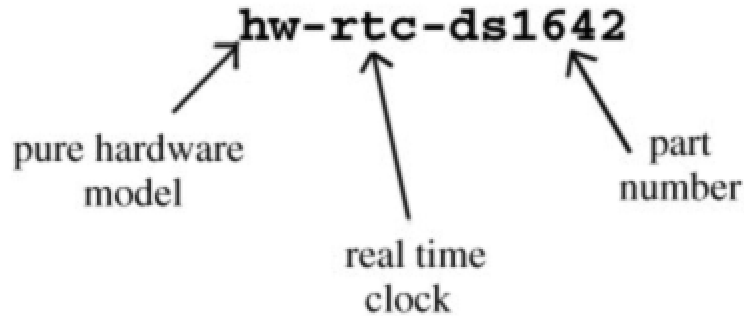
Technical Notes

In a way, the simulator configuration language corresponds to the common low-level API of SID components. This API is described in the Component Developers Kit Reference manual. Each configuration command maps to a small number of low-level SID API calls. Any SID component could make the same calls at any time during a run. This means that configuration is not really a special stage of SID processing, but potentially an ongoing activity to continually change the running system. This also means that any particular SID configuration file could be supplanted by SID components that perform the work by making the equivalent SID API calls.

Creating a New Configuration File

The configuration file can have eight types of commands, interspersed with blanks and comments. Commands take a number of whitespace-separated arguments. Each parameter is a string, optionally enclosed in double-quotes. Comments start with "#" and extend until the end of line.

- The **load** command registers component libraries. These are dynamically-loaded libraries that contain a collection of related SID component types. Each component type has a unique structured name that describes its type of functionality. Names are typically structured to describe the component's role. For example:



- The **new** command instantiates components of a given type and assigns unique reference names for the new instances. Other commands use these names to reference the component instances.
- The **set**, **connect-pin**, , **disconnect-pin**, **connect-bus**, **disconnect-bus**, and **relate** commands configure and manage connections between component instances.

Command Syntax

The syntax for each command type follows.

The load Command

Syntax

The syntax is: *LIBRARYFILE LIBRARYSYMBOL*

where:

- The *LIBRARYFILE* parameter is a file name.
- The *LIBRARYSYMBOL* parameter is a symbol name.

For a list of appropriate *LIBRARYFILE* and *LIBRARYSYMBOL* names, refer to the component reference information on each component.

Example 3-1. load

```
load
     libcache.la cache_component_library
load libmemory.la memory_component_library
```

Function

Registers a component library. Loads the shared library from the file named by the first string. Enumerates the available component types by searching the library for

the `sid::component_library` symbol named by the second string. Add all its component type names to the catalog of known types for subsequent new commands. Nearly all SID components are packaged in shared libraries.

Technical Notes

The *`LIBRARYFILE`* parameter is interpreted as a filename. It may be the name of a Windows DLL or the name UNIX shared object (`.so`) file, although libtool archives ending with the extension `.la` are preferred. The file must exist and be loadable on the host platform using normal library loading routines like `dlopen()`. If the filename is not absolute, several paths are searched. Among these are a colon-separated UNIX-style directory list given in the SID_LIBRARY_PATH environment variable, the PATH environment variable in standard platform notation, and the installation root directory for SID given by the SID_EXEC_PREFIX environment variable.

The *`LIBRARYSYMBOL`* parameter is the name of an exported data symbol in the shared library. It must be resolvable using a standard function like `dlsym()` and refer to an initialized instance of the `sid::component_library` structure that lists the low-level entry points to the component factory functions.

The **load** command checks the magic version number in the structure against the number compiled into the sid main-line program. If the shared library is deemed compatible, the list of component types that it contains are obtained, using the `sid::component_library::` call, and registered internally. From this point, any further request to instantiate a component type whose name is on that list will be directed to this shared library. The special string *`-static-`* is also understood as a *`LIBRARYFILE`* parameter. It is used in cases where sid::component_library structures are specially linked directly into the SID main-line program. This is an advanced technique and is not discussed further. Referring to an invalid *`LIBRARYFILE`* or *`LIBRARYSYMBOL`* will cause an error.

The new Command

Syntax

The syntax is:

```
new
     TYPENAME       REFERENCE
```

where:

- The *`TYPENAME`* parameter is the component type.
- The *`REFERENCE`* parameter is the reference name of the new component instance.

Example 3-2. new

```
new
      hw-glue-sequence-8 init-sequence
new sw-debug-gdb gdb-interface
```

Function

Instantiates a new component. The first string is the component type–one of those made available by a prior load command. The second string is the freely chosen reference name for the new component instance. That name must be unique in a SID run. It is possible to instantiate the same type of component multiple times, but each instance must have a unique reference name. A `sid-control-cfgroot` type component is automatically instantiated and given the reference name main, so that name is reserved. Do not use an unknown *`TYPENAME`*, or a duplicate *`REFERENCE`* ; this will cause an error.

Technical Notes

If this command is successful, it will cause the `sid::component_library::create_component()` hook to be called on the component library that has last registered TYPENAME.

The connect-pin and disconnect-pin Commands

Syntax

The syntax is:

```
connect-pin
     COMPONENT1  PIN1
     DIRECTION      COMPONENT2
     PIN2

     disconnect-pin COMPONENT1
     PIN1  DIRECTION
     COMPONENT2
     PIN2
```

where:

- The *`COMPONENT1`* and *`COMPONENT2`* parameters are reference names of existing component instances.
- The *`PIN1`* and *`PIN2`* parameters are pin names belonging to *`COMPONENT1`* and *`COMPONENT2`*, respectively.
- The *`DIRECTION`* parameter is either " `->`", "`<-`", or "`<->`", indicating the directionality of the pin connection. The first two establish or destroy a single pin connection in opposite directions, and the last establishes or destroys two independent connections in the opposite directions. The arrow shape suggests the direction of signalling: output to input.

Example 3-3. connect-pin and disconnect-pin

```
connect-pin main perform-activity -> host-sched advance
connect-pin cpu trap <-> gdb-interface trap
disconnect-pin host-sched advance <- main perform-activity
```

Function

Adds or removes a pin connection between a pair of component instances. Each SID component generally defines its own set of pins. These may be input and/or output. See the reference documentation for the list for any particular component type. It is possible to connect many input pins to a given output pin, and it is also possible to connect many output pins to a given input pin. It is an error to make the same pin-to-pin connection twice, or to connect an input-only and an output-only pin in the wrong direction. Similarly, it is an error to try to remove a nonexistent connection.

Technical Notes

The **connect-pin** command causes a `sid::component::find_pin()` call on the component and pin on the input side of the arrow, and passes the result to a `sid::component::connect_pin()` call on the output side of the arrow. (The bidirectional <-> case simply repeats this with the two sides exchanged.) The **disconnect-pin** command does the same, except it uses `sid::component::disconnect_pin()` instead. With a pin connection, the component on the output side of the arrow can make `sid::pin::driven` calls to the given input pin.

The connect-bus and disconnect-bus Commands

Syntax

The syntax is:

```
connect-bus
     MASTER  ACCESSOR
     SLAVE
     BUS

     disconnect-bus MASTER ACCESSOR SLAVE BUS
```

where:

- The *MASTER* and *SLAVE* parameters are reference names of existing component instances.
- The *ACCESSOR* parameter is the name of an accessor of the *MASTER* component.
- The *BUS* parameter is the name of a bus of the *SLAVE* component.

Example 3-4. connect-bus and disconnect-bus

```
connect-bus angel target-memory bus access-port

disconnect-bus cpu insn-memory
      bus access-port
```

Function

Adds or removes an accessor-to-bus connection between two components. Each SID component defines its own set of buses and accessors. See the component reference sections for the list for any particular component type.

Technical Notes

The **connect-bus** command causes a `sid::component::find_bus()` call on the *SLAVE* component for the *BUS* name, and will pass the result to a `sid::component::connect_accessor()` call on the *MASTER* component for the *ACCESSOR* name. The **disconnect-bus** command does the same, except it uses `sid::component::disconnect_accessor()` instead. With a bus connection, the *MASTER* component can make `sid::bus` read or write calls via the given accessor to the given bus of the *SLAVE* component.

The set Command

Syntax

The syntax is:

```
set
    COMPONENT  ATTRNAME
    ATTRVALUE
```

where:

- The *COMPONENT* parameter is a reference name of an existing SID component instance.
- The *ATTRNAME* parameter is the name of an attribute defined by that component.
- The *ATTRVALUE* parameter is the value string.

Example 3-5. set

```
set cpu r0 0x1ea90e42
set angel command-line "sid hello world"
```

Function

Sets an attribute value. Each SID component generally defines its own set of attributes and rules for interpreting their values. See the component reference sections for the list for any particular component type.

Technical Notes

The component instance will receive a `sid::component::set_attribute_value()` call with verbatim copies of these two parameters. If the `sid::component::set_attribute_value()` call fails (returns something other than sid::component::ok), the configuration process will fail.

The relate and unrelate Commands

Syntax

The syntax is:

```
relate
     COMPONENT1       RELATIONSHIP
     COMPONENT2

     unrelate COMPONENT1
     RELATIONSHIP      COMPONENT2
```

where

- The *COMPONENT1* and *COMPONENT2* parameters are reference names of existing component instances.
- The *RELATIONSHIP* parameter is the name of a component relationship supported by *COMPONENT1*.

Example 3-6. relate and unrelate

```
relate
      gdb-interface target cpu
unrelate cpu coprocessor cop
```

Function

Adds or removes a supervisory relationship between the named components. Each SID component may define its own set of component relationships. See the component reference sections for the list of relationships it supports. It is an error to create duplicate relationships, for example by repeating the same command.

Technical Notes

This command causes a SID component API call to be placed on *COMPONENT1*, supplying *RELATIONSHIP* and *COMPONENT2*. For more information on the SID component API, refer to the Component Developer's Kit Reference manual. With a component relationship, *COMPONENT1* is able to make any low-level SID API call on *COMPONENT2*.

Lexer Issues

The configuration file may contain blank lines. Comments begin with the `#` character and terminate at end-of-line. Blank lines and comments are ignored during parsing. All commands take some arguments. All arguments are strings. A string is a white-space-separated sequence of printable (`isprint()`) characters. Strings may be enclosed in double quotes. If started with double-quotes, all characters between opening and closing quotes are included in the string. Backslash characters ("\") escape double-quotes and backslash characters, in the usual C convention. The string `\n` is

interpreted as a C `\n`. Without double-quotes, strings are taken to be whitespace-separated words. A `#` character found where a string is expected is interpreted as a comment to end-of-line, and the search for the next string found is returned instead.

Chapter 4. How to Understand Component Reference Information

Every SID component has an associate text file that documents the component according to a standard template. The template is meant to include enough information for you to use the component in a simulation run. It includes a basic functional description and detailed low-level configuration information.

The component documentation does not attempt to describe an application programmer's view of a hardware-oriented component, so it is not a programming manual. You should refer to the hardware manufacturer's data books for this kind of information.

Finding Component Reference Information

To search for and display installed component documentation files that match a given substring, use the **siddoc** command.

Usage:

```
siddoc [-l] name
   ...
```

where

name

is a substring of the desired component type name

-l

specifies that you only want matching component types to be listed, instead of the actual documentation. Without *-l*, each matching documentation file will be displayed on the screen by running the **more** program.The default **more** may be overridden by setting the PAGER environment variable.

Examples:

siddoc -l hw-cpu

will show `hw-cpu-arm7t hw-cpu-m32r/d` and any other CPU-type components that have documentation files.

siddoc memory-flash uart

will display the documentation files of both `hw-memory-flash-*` components, then the `hw-uart-*` components.

Parts of the Component Documentation Template

The simulation component documentation template is broken into the following sections.

Name

This item provides the SID component-type name. In cases where a family of components may be described together, this item may list several names.

SID component-type names are structured so that the first part of the component-type name indicates the category of thing that is simulated:

- `hw` usually indicates that the simulated component represents hardware.
- `sw` indicates that the simulated component represents software.
- `sid` indicates that the simulated component performs special simulation management tasks.

Synopsis

The Synopsis summarizes the component's purpose and lists its external interfaces in a condensed form. The summary includes a complete list of the names of the component's pins, buses, accessors, attributes, and relationships. These names may be referenced in SID configuration files, or by other supervisory components. The last items in this section list the component library file and symbol names in which this component-type is found. Supply these two names to a load configuration file command.

Functionality

This section gives detailed information about how this component operates. It has several subsections.

Modelling

The modelling subsection discusses the relationship of this SID component to the real thing (maybe hardware) being modelled. The scope of simulation focuses on whatever important differences exist between a simulated program's view of the SID model versus reality.

Behaviors

The behaviors subsection describes in detail how the SID component operates. Each separate behavior (stage of life, or independent processing activity) is described separately. Each behavior section describes what triggers the behavior (usually a pin, bus, or attribute call), what the component does internally in response, and what external results may be visible. These sections reference the low-level pin, bus, etc. names.

SID Conventions

The conventions subsection describes which SID behaviors the component supports. These are called conventions, since they are high-level behaviors expressed exclusively in terms of lower-level constructs (pins, attributes, and so on). The following is a list of SID conventions.

Supervisory versus Functional

A supervisory component may manipulate other components using the full SID API. A functional component uses only the pin and bus APIs to communicate with its

neighbors. Most hardware modelling components are functional; most simulator control components are supervisory.

Target View Support

A SID component that supports the target view specially exports some attributes. The component classifies these selected attributes with one of the category names pin, register, or setting. The component internally associates these attributes with operations on actual SID pins or other state.

The target view will allow attributes of these three categories to be interacted with (display = read, update = write). The write routines associated with these attributes should be generous to permit the user a wide range of valid input formats.

A SID component that does not support the target view may still be a first class member of a SID configuration. It will be harder for a user to interact with the component, that's all.

State Save/Restore Support

A SID component that supports standard state save/restore function does so by exporting an attribute named state-snapshot. When read, the attribute value represents a subset of the internal state of the component; this is the state save operation. When written to with the string from a previous read, the internal state of the component will be reset according to the supplied string; this is the restore operation.

Normally, components endavour to save and restore the subset state that may be visible to an executing simulated application. Some components may save/restore additional state like configuration preferences, or transitory attributes.

A SID component that does not support state save/restore may not be safe to interoperate with other components after a restore operation.

Triggerpoint Support

A SID component that supports triggerpoints allows some internal state to be watched or monitored for certain conditions. Watch criteria include any change, matching a given value, or matching a given value with a given boolean mask. Whenever a component detects that an active watching criterion succeeds, it drive a special output pin.

The watchable internal state values are advertised as attributes with the additional category watchable. Watching such a state value involves, at the low level, connecting to a virtual output pins that has a special name. The name encodes the watchable item and a specification of the watch criterion.

A SID component that does not support triggerpoints may still be a first class member of a SID configuration. Monitoring the component's internal state may simply be slow and cumbersome.

Reentrancy

Some SID components defend themselves against reentrancy. Since SID components tend to perform activity by calling other SID components, it is conceivable that the called component may in turn call back to the original component. It is a possible problem because some components may have internal data structures that are not

in a consistent state at the moment of a reentrant call. It is also possible that infinite recursion could result.

A SID component that defends itself against reentrancy does this by run-time checks around certain incoming calls, like specific pin or bus signals, or attribute accesses. When it detects that more than some number of calls are in progress, it signals an error message and may even shut SID down.

A SID component that does not defend itself against reentrancy may nevertheless be vulnerable to the problem. It may be unable to detect the situation before it becomes a failure.

In order to avoid triggering these detection measures, it is important that a SID configuration avoid the possibility of violation of the reentrancy restrictions of a component. For example, avoid creating pin or bus connections that cause gratuitous loops, and abusing supervisory components.

Environment

This section talks about how instances of this component interact with their environment.

Related Components

SID components are meant to connect to each other. This section discusses what other components work well together with this one. It may include configuration file fragments to demonstrate minimal interconnection instructions.

Host System

Since SID components are software modules that run on the host, they naturally interact with the host system. In some cases, there may be significant resource consumption issues that a user needs to be aware of. This section talks about what kind of impact this component may have on the host, including host I/O, security concerns, memory consumption, performance, and error handling conventions.

SID Interface Reference

This section discusses the interfaces of this SID component in a tabular reference format. There are several formats, one per low level interface. In all cases, the table rows cross-reference to the Behaviors section, where fuller details about the use of that item may be found (see Behaviors for more details).

Pins

Column 1: Name

Gives the string that the connect-pin configuration command may use to reference this pin.

Column 2: Directionality

Identifies this pin as a possible input, output, or both.

Column 3: Legal Values

Lists the range of values that may be driven into this pin (if an input), or what values may be driven outward (if output). The values may be limited to a subrange of the host_int_4 type.

Column 4: Behavior

References the behaviors associated with this pin

Buses

Column 1: Name

Gives the string with which the connect-bus configuration command may reference this bus.

Column 2: Address Range

Identifies the supported range of addresses for accesses on this bus.

Column 3: Access Types

Lists what subset of size, endianness, and read/write accesses are supported.

Column 4: Behavior

References the behaviors associated with this bus.

Accessors

Column 1: Name

Gives the string with which the **connect-bus** configuration command may reference this accessor.

Column 2: Access Types

Lists what subset of size, endianness, and read/write accesses may be made by this component using this accessor.

Column 3: Behavior

References the behaviors associated with this accessor.

Attributes

Column 1: Name

Gives the string with which the **set** configuration command may reference this attribute.

Column 2: Categories

Lists the category names under which this attribute is classified.

Column 3: Legal Values / Default Values

Describes legal values of this attribute, for both reading and writing, and may include the default value. There are several common attribute value formats:

- *numeric* strings are C-style integral numeric literals like `0`, `0xEEEF`, `0377`, `0b1101101`, and `-75`.
- *boolean* strings are values such as true, false, n, yes, 0, and 1.
- *general* strings containing a sequence of unrestricted binary bytes.

Column 4: Behavior

References the behaviors associated with this attribute.

Relationships

Column 1: Name

Gives the string with which the relate configuration command may reference this relationship.

Column 2: Behavior

References the behaviors associated with this relationship.

References

This section may refer the reader to further material describing the component, or its real-world image.

Chapter 5. Using SID

Running SID

SID is essentially a batch process. It reads a configuration file (see Chapter 3>) and executes its contents. Thus the basic invocation interface is simple; the hard part is writing a complete, useful configuration file. There are two ways, commonly, of using SID: direct invocation on a task-specific configuration file, and through a wrapper that sets up a more generic simulation environment.

Direct invocation

The most direct (and flexible) way of running SID is with the command-line:

sid [-h] [-n] [-f *FILE*] [-e *LINE*] [*CONFIG_FILE*...]

Table 5-1. SID Options

Option	Meaning
-h	print this help
-n	load/check configuration but do not run simulation
-f *FILE*	also read configuration *FILE*
-e *LINE*	also do configuration *LINE*

All `-f`/`-e` options are performed first, in sequence. Any *CONFIG_FILE* names supplied without `-f` are done last, in sequence.

Example 5-1. SID invocations

`sid arm7t-config`

runs the simulation environment described in `arm7t-config`

`sid -e 'set angel command-line "simulated banana"' arm7t-config`

executes the supplied configuration line (setting the command line to "simulated banana") and then runs the simulation environment described in `arm7t-config`

`sid -f special-settings arm7t-config`

executes any directives in the file `special-settings`, then runs the simulation environment described in `arm7t-config`

Invoking SID this way is simple, but requires a detailed configuration file `CONFIG_FILE` in order to perform any interesting simulation tasks. Since many components and connections will vary only slightly between simulation runs, it is often beneficial to introduce a "wrapper" program which automatically constructs configuration files from common idioms.

Invocation through a wrapper

A common wrapper around the `sid` binary is the perl script `configrun-sid`, which accepts more customization options and writes a temporary configuration file containing a relatively common arrangement of CPU, memory, gloss, scheduler, mapper, I/O and debugger components, representative of an embedded system "target board". It is invoked with the command-line

configrun-sid [–help] {–cpu=*CPU*} [–verbose] [–save-temps]
[–trace-extract] [–trace-semantics] [–trace-core]
[–memory-region=*BASE*,*SIZE* [,read-only | ,alias=*BASE2* | ,file=*FILENAME* | ,mmap]]
[–gdb=*PORT*] [–board=*BOARD*] [–engine= [scache | pbb]] [-EB | -EL]
[–tksm] [–persistent] [–no-run] [–insn-count=*N*] [–gprof] [–wrap=*COMPONENT*]

Table 5-2. configrun-sid Options

Option	Meaning	Default
–help	Print this help message.	
–cpu=CPU	Select target processor	none
–verbose	Turn on various run-time verbosity settings.	no
–save-temps	Keep generated sid configuration file.	no
–trace-extract	Turn on CPU insn decode tracing.	no
–trace-semantics	Turn on CPU insn execute tracing.	no
–trace-core	Turn on bus access tracing.	no
–memory-region=BASE,SIZE	Add RAM region from BASE to BASE+SIZE-1. Additional options may be appended, separated by commas: `read-only` flags this region as read-only, `alias=`*BASE2* adds an alias to this region at *BASE2*, `file=`*FILENAME* loads and saves this memory region from a file, and `mmap` memory-maps such a file rather than loading it.	no
–gdb=*PORT*	Add a gdb/debugger interface on TCP port.	none
–board=*BOARD*	Model given board or system.	gloss
–engine=scache|pbb	Set given cgen CPU engine.	pbb
-EB | -EL	Set powerup CPU mode to big/little endian.	auto

Option	Meaning	Default
–tksm	Add an experimental Tk system monitor.	no
–persistent	Rerun top-level loop indefinitely.	no
–no-run	Make config file (–save-temps) and exit.	no
–insn-count=*N*	Block of uninterrupted ticks for insns.	10000
–gprof	GPROF-profile, collect every N ticks.	no
–wrap=*COMPONENT*	Turn on SID API tracing for named component.	none

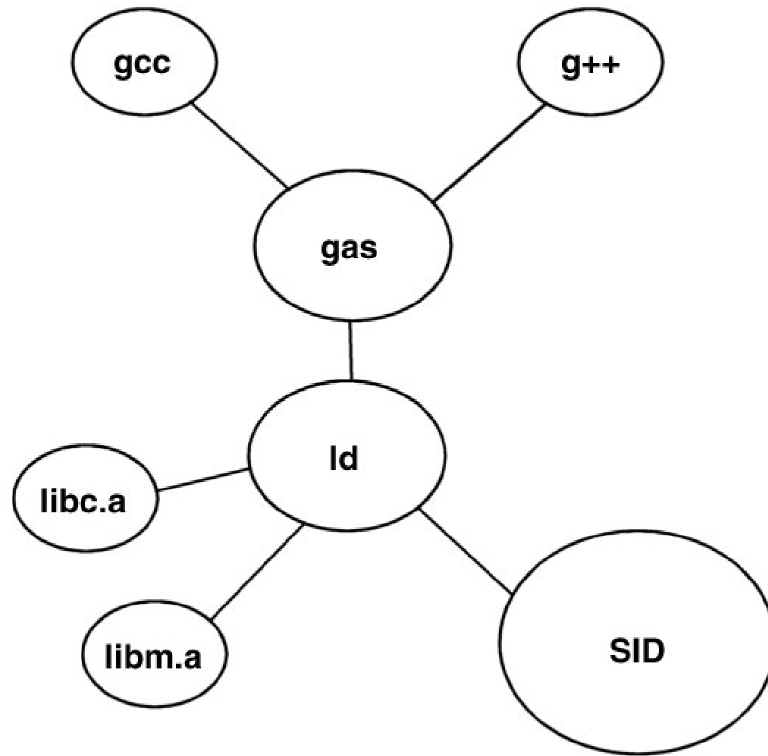
Typically, if SID is built along with a GNU toolchain, the build process also produces a wrapper shell script around `configrun-sid`, such as `arm-elf-sid`, which in turn executes `configrun-sid --cpu arm`. This is simply for uniform naming with the other target-specific portions of the toolchain, such as `arm-elf-gcc` or `arm-elf-ld`.

SID Startup

SID is started up when the `run!` pin is down, which results in the simulation top level loop beginning. The first action is that all the components reported in the SID configuration file are configured together in the specified manner in the configuration file. After the relationships are established, the main loop regains control to complete any further steps and after the main loop is completed, the control is returned to the function that sent the **run!** command.

Creating an Application to Run on SID

The target code is developed in a similar manner as you would for developing code to run on the target hardware. The source code is either written in C or C++ and is then compiled, assembled and linked for arm-elf format. The user then has the ability to download to the target hardware or on SID. The simulator supports the ARM architectures for this release in both big-endian and little-endian and in elf file format. The default is big-endian. Once the code is loaded on SID, the execution and debugging proceed in the usual manner.



Debugging Using SID

Aside from providing interactive monitoring of a simulation through a console or TK visual interface, SID provides mechanisms to debug and profile the simulation environment with conventional GNU development tools such as `gdb` and `gprof`. The following sections assume familiarity with these GNU tools, and focus on describing their interfaces with SID. The debugging and profiling components provided with SID are normal components, and certainly other such debugging and profiling components could be made. They do not have nor need special access to anything outside the normal SID component API; they merely serve as bridges between SID and other existing tools.

Using GDB

The debugging interface is designed resemble working with a remote "target board", using the GDB remote protocol. The SID simulation is run with a configuration that instantiates a `sw-debug-gdb` GDB stub component connected to a `sid-io-socket-server` socket component, which serves a connection between the GDB stub component and a remote debugger. The GDB stub component is configured to monitor and control other components within the simulation environment, such as CPU and system scheduler, in response to commands it receives from the socket component. A simulation user wishing to debug the simulation runs `gdb` as usual, on any machine capable of connecting to the host running the simulation, but directs it to the TCP

socket configured into the socket component in the simulation. The user can then proceed to debug the simulation exactly as if it were a "real" target environment.

Configuring the GDB stub and socket components is somewhat lengthy and verbose, but is performed automatically by the `configrun-sid` command when invoked with the `--gdb=`*`PORT`* option. For complete details of the connection process, see `siddoc sw-debug-gdb`.

Profiling

The profiling facility consists of the standard GNU profile-analysis tool `gprof`, along with a SID component `sw-profile-gprof`. The component is loaded into a SID session and configured to sample any other component's numeric attribute, and record a histogram. The histogram is then output to a file, and `gprof` interprets the file as a "flat profile". The profiling component has no call-graph collection facility.

When the profiling component's `sample` pin is driven, it uses its `value-attribute` attribute as the *name* of an attribute to sample on the component in its `target-component` relation. The sample is taken immediately, interpreted as a numeric string, and stored in a histogram bucket. The histogram bucket's width, in bytes, is specified by the profiling component's `bucket-size` attribute.

When the profiling component's `store` pin is driven, the histogram is dumped to the file named in the profiling component's `output-file` attribute, which has a default value of `gmon.out`. The histogram is not cleared by this action, however. To clear the histogram, the profiling component's `reset` must be driven. Since dumping and resetting are separate operations, one can collect multiple cumulative profiles of a value across a single simulation run.

A simple configuration example is given below, showing a connection between the profiling component and a cpu (assumed to already be instantiated). For complete reference on the profiling component, see `siddoc sw-profile-gprof`.

Example 5-2. Profiling a CPU component's program counter

```
load libprof.la prof_component_library
new sw-profile-gprof gprof

relate gprof target-component cpu

set gprof value-attribute pc
set gprof bucket-size 4

connect-pin main perform-activity -> gprof sample
connect-pin main stopping -> gprof store
```