



FlexModel User's Manual

To search the entire manual set, press this toolbar button.
For help, refer to [intro.pdf](#).



Copyright © 2001 Synopsys, Inc.
All rights reserved.
Printed in USA.

Information in this document is subject to change without notice.

SmartModel, ModelAccess, ModelTools, SourceModel Library, LM-1200, and Synopsys Eagle[®] are registered trademarks; MemPro, MemSpec, MemScope, FlexModel, LM-family, LM-1400, Logic Model, ModelSource, and SourceModel are trademarks of Synopsys, Inc.

All company and product names are trademarks or registered trademarks of their respective owners.

Contents

Preface	9
About This Manual	9
Related Documents	9
Manual Overview	10
Typographical and Symbol Conventions	10
Getting Help	11
The Synopsys Website	12
Synopsys Common Licensing (SCL) Document Set	12
Comments?	12
 Chapter 1	
FlexModel Overview	13
What Are FlexModels?	13
FlexModel Structure and Interface	14
Installing FlexModels	14
FlexModel Installation Tree	15
FlexModel Licensing	15
FlexModel Limitations	16
 Chapter 2	
Using FlexModels	17
Introduction	17
Running flexm_setup	17
Setting Up the Model	18
The flex_get_inst_handle Command	19
Using Multiple FlexModel Instances	19
Controlling the FlexModel Command Flow	20
Resetting the Simulation	20
Transferring Control to a C Testbench	21
Using Multiple Command Streams in a C Testbench	21
Using Uncoupled Mode in a C Testbench	22
Burst Transfers	24
Non-pipelined Bus Operations	24
Pipelined Bus Operations	25
Synchronizing the Command Flow	27
FlexModel Timing	28

Selecting Function-only or Timing Model	28
Controlling Timing Checks and Delays	29
FlexModel Interrupts	30
Interrupt Service Routines	30
Detecting and Servicing Interrupts	31
Developing HDL Interrupt Routines	32
Developing C Interrupt Routines	34
Developing VERA Interrupt Routines	36
Chapter 3	
FlexModel Command Modes	39
Introduction	39
Using HDL Command Mode	39
VHDL Control	41
Verilog Control	41
HDL Control Between Model and Testbench	41
Using C Command Mode	43
Creating an External C File	44
Compiling an External C File	45
Switching Control to an External C Program	47
Using VERA Command Mode	47
FlexModel VERA Classes	48
VERA Files in the LMC_HOME Tree	49
The <i>Model/Fx</i> Class Constructor	49
Examples with Top-level Testbenches	50
Accessing the Current Error Status	51
FlexModel Logging from the VERA Class	52
Chapter 4	
FlexModel Command Reference	53
Introduction	53
Model-Specific and Global Commands	53
About the Commands	54
Bus and Zero-Cycle Commands	54
The <i>inst_handle</i> Parameter	54
The req and rslt Command Suffixes	55
Command Result Identifiers	55
The wait_mode Parameter	56
The status Parameter	56
Command Syntax Differences in VERA Command Mode	58
Global FlexModel Commands	59

Global FlexModel Command Descriptions	60
flex_clear_queue	60
flex_define_intr_function	62
flex_define_intr_signal	63
flex_get_cmd_status	65
flex_get_coupling_mode	67
flex_get_inst_handle	68
flex_print_msg	70
flex_run_program	72
flex_set_coupling_mode	74
flex_start_program	75
flex_switch_intr_control	76
flex_synchronize	77
flex_wait	79
 Chapter 5	
FlexModel C Testbench Interface	81
Introduction	81
What Are FLEX_VEC Vectors?	81
Creating FLEX_VEC Vectors	82
FLEX_VEC Lexical Rules	83
FLEX_VEC Error Handling	84
FLEX_VEC Command Descriptions	85
C Testbench Example	95
 Appendix A	
Reporting Problems	101
Introduction	101
Model Versions and History	101
Running FlexModel Diagnostics	102
Creating FlexModel Log Files	102
Command Logging	103
Stimulus Logging	104
Message Logging	105
Sending the Log Files to Customer Support	105
 Index	107

Figures

Figure 1:	FlexModel Structure and Interface	14
Figure 2:	FlexModel Structure in LMC_HOME Tree	15
Figure 3:	Pipelined Bus Operations	25
Figure 4:	Interrupt Detection and Servicing	31
Figure 5:	Read_req/read_rslt Pair for Testbench	42
Figure 6:	Multiple Commands within a Single Clock Cycle	42
Figure 7:	Accessing a C Testbench from HDL	43
Figure 8:	VERA Model Class Hierarchy	48

Tables

Table 1:	VERA Files in the LMC_HOME Directory	49
Table 2:	FlexModel Command Types	53
Table 3:	Status Parameter Error Codes	56
Table 4:	Global FlexModel Command Summary	59
Table 5:	VHDL 9-State to 4-State Conversion	83
Table 6:	Stimulus Logging Format	104

Preface

About This Manual

This manual explains how use FlexModels in your test environment. FlexModels are a type of SmartModel and they share many characteristics in common with them, but there are significant differences. For example, FlexModels have advanced features like the ability to issue model commands from an HDL, C, or VERA testbench. Those capabilities and other enhancements to traditional SmartModel usage are explained in this manual.

This manual works in tandem with the individual FlexModel datasheets. General information that pertains to all FlexModels is presented here, whereas information that is specific to individual FlexModels is documented in the model datasheets.

Related Documents

For general information about SmartModel Library documentation, or to navigate to a different online document, refer to the [Guide to SmartModel Documentation](#). For the latest information on supported platforms and simulators, refer to [SmartModel Library Supported Simulators and Platforms](#).

For detailed information about specific models in the SmartModel Library, use the Browser tool (\$LMC_HOME/bin/sl_browser) to access the online model datasheets.

Manual Overview

This manual contains the following chapters:

Preface	Describes the contents of this manual and provides references to other sources of information about FlexModels. Also describes conventions and terminology used in this manual.
Chapter 1: FlexModel Overview	General information about FlexModel architecture, features, and benefits.
Chapter 2: Using FlexModels	How to set up one or more FlexModels as a testbench and use model commands to coordinate the command flows. Also how to use FlexModel timing and interrupts.
Chapter 3: FlexModel Command Modes	How to use the HDL, VERA, and C command modes to control FlexModels.
Chapter 4: FlexModel Command Reference	Common features of FlexModel commands and a command reference for global FlexModel commands.
Chapter 5: FlexModel C Testbench Interface	How to use the FlexModel C functions and operators to define and manipulate FLEX_VEC vectors for use with FlexModel commands.
Chapter A: Reporting Problems	How to enable FlexModel logging and report problems to Customer Support.

Typographical and Symbol Conventions

- Default UNIX prompt
Represented by a percent sign (%).
- **User input** (text entered by the user)
Shown in **bold monospaced** type, as in the following command line example:

```
% cd $LMC_HOME/bin
```
- **System-generated text** (prompts, messages, files, reports)
Shown in **monospaced** type, as in the following system message:

```
VALIDATION PASSED: No Mismatches during simulation
```

- **Variables** for which you supply a specific value

Shown in italic type, as in the following command line example:

```
% setenv LMC_HOME prod_dir
```

In this example, you substitute a specific name for *prod_dir* when you enter the command.

- **Command syntax**

Choice among alternatives is shown with a vertical bar (|) as in the following:

```
termination_style, 0 | 1
```

In this example, you must choose one of the two possibilities: 0 or 1.

Optional parameters are enclosed in square brackets ([]) as in the following:

```
pin1 [pin2 ... pinN]
```

In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* ... *pinN*]).

Getting Help

If you have a question while using Synopsys products, use the following resources:

1. Start with the available product documentation installed on your network or located at the root level of your Synopsys CD-ROM. Every documentation set contains overview information in the [intro.pdf](#) file.

Additional Synopsys documentation is available at this URL:

<http://www.synopsys.com/products/lm/doc>

Datasheets for models are available using the Model Directory:

<http://www.synopsys.com/products/lm/modelDir.html>

2. Visit the online Support Center at this URL:

<http://www.synopsys.com/support/lm/support.html>

This site gives you access to the following resources:

- SOLV-IT!, the Synopsys automated problem resolution system
- product-specific FAQs (frequently asked questions)
- the ability to open a support help call
- the ability to submit a delivery request for some product lines

3. If you still have questions, you can call the Support Center:

North American customers:

Call the Synopsys EagleI and Logic Modeling Products Support Center hotline at 1-800-445-1888 (or 1-503-748-6920) from 6:30 AM to 5 PM Pacific Time, Monday through Friday.

International customers:

Call your local sales office.

The Synopsys Website

General information about Synopsys and its products is available on the Web:

<http://www.synopsys.com>

Synopsys Common Licensing (SCL) Document Set

Synopsys common licensing (SCL) software is delivered on a CD that is separate from the tools that use this software to authorize their use. The SCL documentation set includes the following publications, which are located in (root)/docs/scl on the SCL CD and also available on the Synopsys FTP server (<ftp://ftp.synopsys.com>):

- *[Licensing QuickStart](#)*—(142K PDF file)
This booklet provides instructions for obtaining an electronic copy of your license key file and for installing and configuring SCL on UNIX and Windows NT.
- *[Licensing Installation and Administration Guide](#)*—(2.08M PDF file)
This guide provides information about installation and configuration, key concepts, examples of license key files, migration to SCL, maintenance, and troubleshooting.

You can find general SCL information on the Web at:

<http://www.synopsys.com/keys>

Comments?

To report errors or make suggestions, please send e-mail to:

doc@synopsys.com

To report an error that occurs on a specific page, select the entire page (including headers and footers), and copy to the buffer. Then paste the buffer to the body of your e-mail message. This will provide us with information to identify the source of the problem.

1

FlexModel Overview

What Are FlexModels?

FlexModels are binary simulation models that represent the bus functionality of microprocessors, cores, digital signal processors, and bus interfaces. FlexModels are essentially advanced SmartModels, and therefore use the SWIFT interface. FlexModels have the following features:

- Built with a cycle-accurate core and a controllable timing shell so that you can run the model in function-only mode for higher performance or with timing mode on when you need to check delays. You can switch between timing modes dynamically during simulation using simple commands in your testbench.
- Feature multiple different control mechanisms. You can coordinate model behavior with simulation events, synchronize different command processes, and control several FlexModels simultaneously using a single command stream.
- Allow you to use different command sources. You can send commands to FlexModels using processes in a Verilog or VHDL testbench, a C program, or a VERA testbench. You can switch between the HDL or VERA testbench and a compiled C program as the source for commands.



Note

Multiple command sources are available on simulators that have custom FlexModel integrations. Customers using Direct C Control through the standard SWIFT integration must stick with C. For more information, refer to the [Simulator Configuration for Synopsys Models](#).

FlexModel Structure and Interface

FlexModels use the SWIFT interface for event-based communication with the simulator. FlexModels also use a central Command Core that queues model commands for one or more FlexModels in your design. The Command Core provides high-performance model control without burdening the SWIFT interface. [Figure 1](#) illustrates the FlexModel interface.

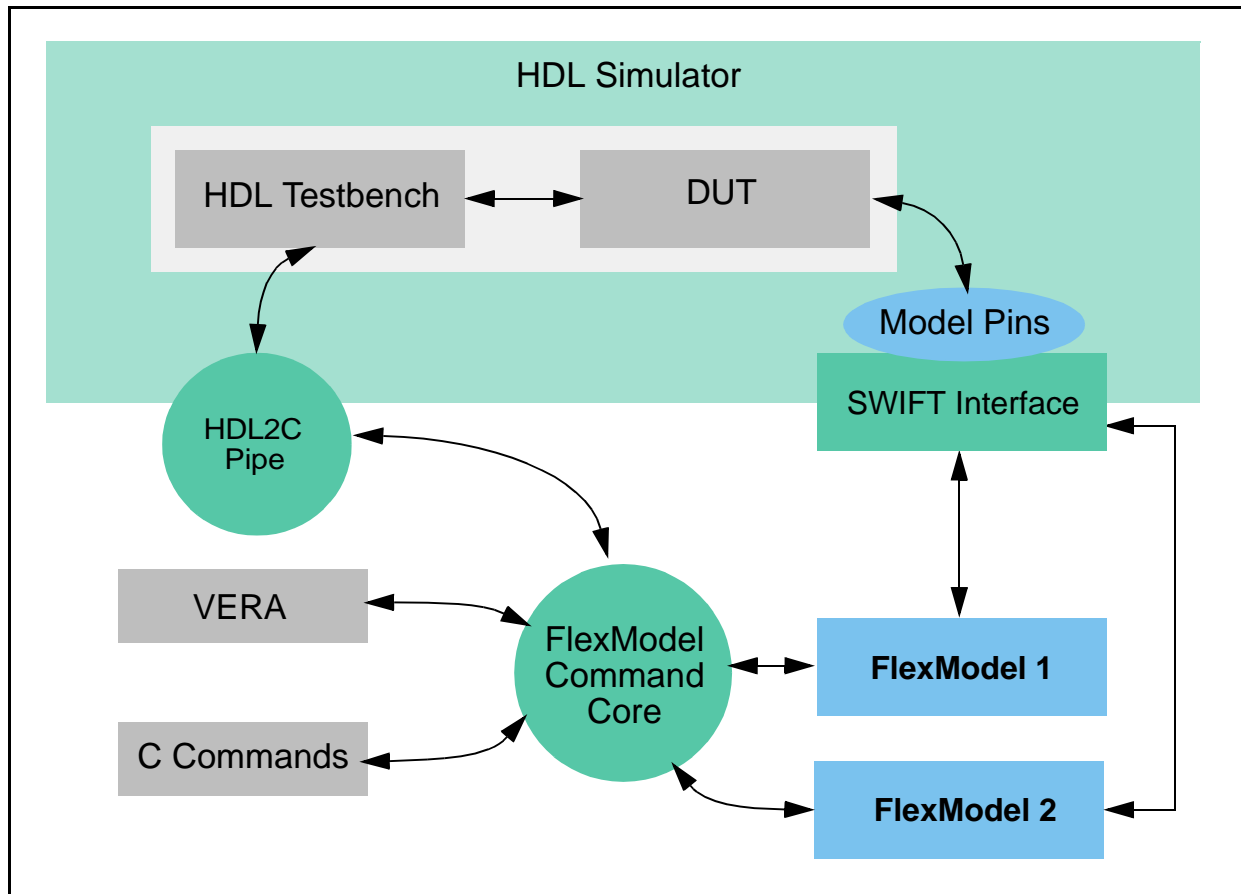


Figure 1: FlexModel Structure and Interface

Installing FlexModels

For FlexModel installation information, refer to the [SmartModel Library Installation Guide](#). This guide contains instructions for installing the models and associated software. For information about supported platforms and simulators, refer to the [SmartModel Library Supported Simulators and Platforms](#).

FlexModel Installation Tree

Figure 2 illustrates the organization of FlexModel files installed in an LMC_HOME tree.

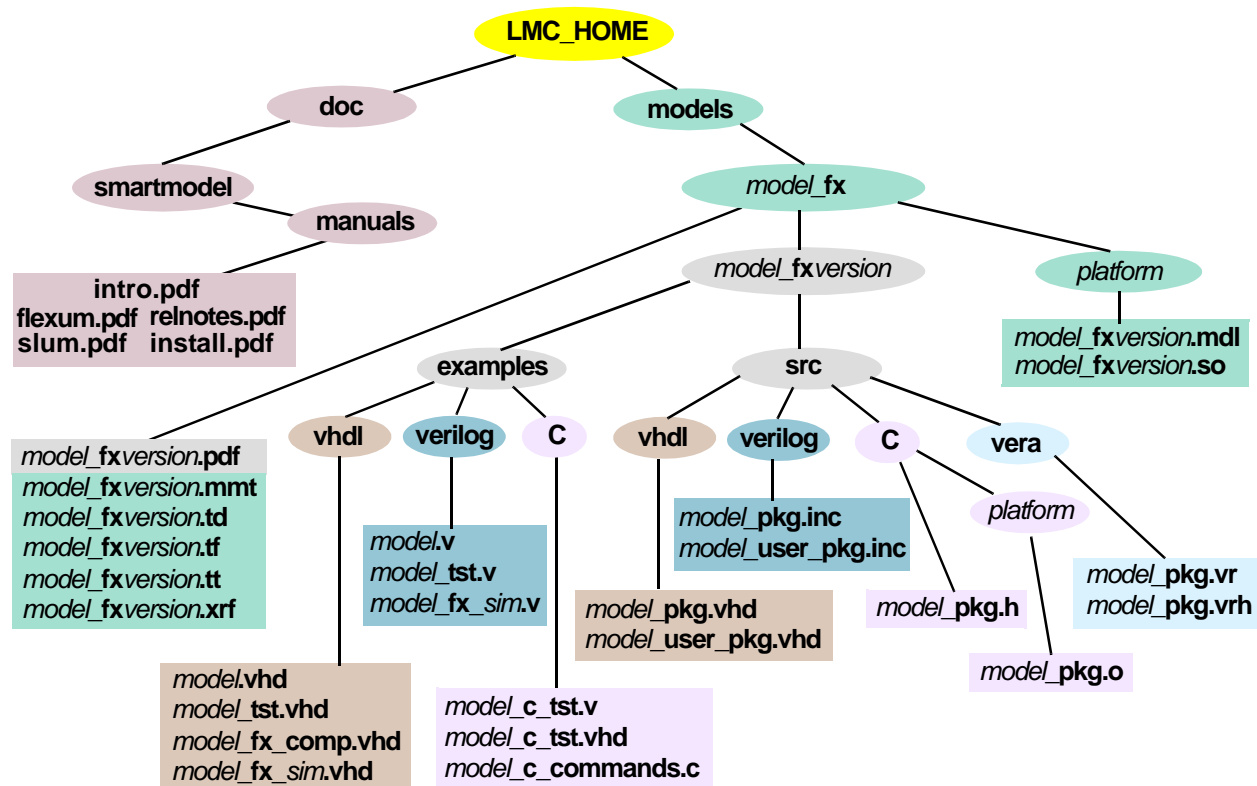


Figure 2: FlexModel Structure in LMC_HOME Tree

FlexModel Licensing

FlexModels use FLEXlm floating licenses to authorize their use, just like other SmartModels. If licensing is not set up, refer to the [SmartModel Library Installation Guide](#) for information about setting up and using the FLEXlm licensing software.

FlexModel Limitations

FlexModels do have some limitations compared to traditional SmartModels:

- FlexModels do not support fault simulation.
- Component-based user-defined timing (UDT) is supported in FlexModels, but instance-based UDT is not.
- SmartModel model logging is not supported for models that have an `_fx` extension. FlexModels of this type have a different model logging mechanism that is described in [“Creating FlexModel Log Files” on page 102](#).
- SmartModel windows are not supported in FlexModels. Instead, use the robust FlexModel command set to access and change internal register information.
- Flexmodels do not support save and restore operations.

2

Using FlexModels

Introduction

This chapter explains how to set up one or more FlexModels and coordinate the flow of FlexModel commands from multiple sources. It also explains how to set up and use FlexModel timing and interrupt service routines. This information is organized in the following sections:

- [“Running flexm_setup” on page 17](#)
- [“Setting Up the Model” on page 18](#)
- [“Using Multiple FlexModel Instances” on page 19](#)
- [“Controlling the FlexModel Command Flow” on page 20](#)
- [“FlexModel Timing” on page 28](#)
- [“FlexModel Interrupts” on page 30](#)

Running flexm_setup

First, run the flexm_setup script to copy the FlexModel's interface files to your working directory. You need to run flexm_setup for each FlexModel you want to use in your design and you must rerun this script after updating your \$LMC_HOME with new or revised FlexModels. This ensures that you pick up the latest package files for the most recent versions of the models.

Syntax

```
flexm_setup [-help] [-dir path] model
```

Argument

model Pathname to the FlexModel you want to set up.

Switches

-help Prints help information.

-d[ir] *path* Copies the contents of the FlexModel's versioned src/verilog and src/vhd directories into *path*/src/verilog and *path*/src/vhdl. The directory specified by *path* must already exist.

Examples

When run without the -dir switch, flexm_setup just prints the name of the versioned directory of the selected model's source files

```
# Lists name of versioned directory containing source files
% flexm_setup mpc860_fx
```

When run with the -dir switch pointing to your working directory, flexm_setup copies over all the versioned package files you need to that working directory.

```
# Creates copy in 'flexmodel' directory of model source files
% mkdir workdir
% flexm_setup -dir workdir mpc860_fx
```

Setting Up the Model

Next, instantiate one or more FlexModels in your design using required SWIFT parameters, as explained in the [Simulator Configuration Guide for Synopsys Models](#). You must allow at least one clock cycle to elapse in your testbench before you issue any FlexModel commands. This allows the FlexModels to initialize. After initialization, FlexModels can accept commands from the testbench. The first FlexModel command for each model instance must always be the `flex_get_inst_handle` command, which returns a unique model instance identifier called the *inst_handle*.

The flex_get_inst_handle Command

The `flex_get_inst_handle` command makes an association between the FlexModelId you used to instantiate the model in your testbench and that specific instance of the model. This is so that you can use more than one FlexModel or multiple instances of the same FlexModel in a design without getting the command streams confused. After you get an *inst_handle*, you use that integer in all subsequent FlexModel commands. It is typically the first required argument.



Note

You do not use the *inst_handle* parameter in VERA Command Mode. See [“Command Syntax Differences in VERA Command Mode”](#) on page 58.

Using Multiple FlexModel Instances

You can have multiple instances of the same FlexModel in the same simulation. If so, you must use separate command streams to avoid conflicts.



Caution

You cannot have multiple command streams (VERA, Verilog, VHDL, or C) sending commands into any one model instance at the same time.

To use more than one instance of a FlexModel in the same simulation, follow these steps. This example uses Verilog syntax. If you are using VHDL or C, make the appropriate substitutions.

1. Qualify all FlexModel commands with their corresponding instance names. For example:

```
// The two instances are called i1 and i2.
mpc740_fx i1(...); The first instance
mpc740_fx i2(...); The second instance

// The command stream for i1.
initial begin
    // command flow for i1 goes here
    i1.mpc740_idle(...);
    // and so forth for i1
end

// The command stream for i2.
initial begin
    // command flow for i2 goes here
    i2.mpc740_idle(...);
```

```
// and so forth for i2  
end
```

2. On the simulator invocation line (Verilog, in this case) add the multi-instance specification to your invocation:

```
+define+mpc740_multi_inst
```

Controlling the FlexModel Command Flow

You can control the flow of FlexModel commands in several ways, as explained in the following sections:

- [“Resetting the Simulation” on page 20](#)
- [“Transferring Control to a C Testbench” on page 21](#)
- [“Using Multiple Command Streams in a C Testbench” on page 21](#)
- [“Using Uncoupled Mode in a C Testbench” on page 22](#)
- [“Burst Transfers” on page 24](#)
- [“Non-pipelined Bus Operations” on page 24](#)
- [“Pipelined Bus Operations” on page 25](#)
- [“Synchronizing the Command Flow” on page 27](#)

Resetting the Simulation

The ability to reset a simulation to an initial state without re-invoking the simulator can save considerable time. Reset is also important for “what if” simulation runs. FlexModels support reset, returning to the state when the simulator was initially invoked.



Attention

Reset is not currently supported on NT. Also, when using FlexModels on Verilog-XL with model logging enabled, some FlexModels may not reset after the third attempt. The workaround is to turn off model logging.

Transferring Control to a C Testbench

You can transfer control from an HDL or VERA testbench to a C testbench using the [flex_run_program](#) command. The model receives all commands from the C testbench before any subsequent model commands in that VHDL process or Verilog block. When you have multiple command streams operating at the same time, there are a few things to keep in mind:

- Non-model commands such as Verilog \$display statements following the [flex_run_program](#) command are processed immediately.
- You cannot issue model request commands (`_req`) in one command source and model result commands (`_rslt`) in another. For example, if you want to make a read request for a FlexModel and then fetch the results, keep both FlexModel commands in either your C or HDL testbench.
- It is best to organize your commands to minimize switching in and out of the same command source. For example, if you want to use a C program for two separate command sequences to be performed at two different points in the simulation, create two separate C command files.
- You cannot have multiple VERA, VHDL, or Verilog processes providing commands to the same model instance.
- You cannot use the [flex_run_program](#) command to switch between different HDL or VERA command sources. For more information on the [flex_run_program](#) command, see “[flex_run_program](#)” on page 72.

Using Multiple Command Streams in a C Testbench

A C testbench can provide commands to more than one model or model instance. This allows two or more models to pass information between each other in proper sequence in a C testbench.

To use multiple command streams in one C testbench, initialize all model instances with the `flex_get_inst_handle` command before issuing the `flex_start_program` command, as shown in the following example:

```
flex_get_inst_handle(InstName1, &id1, &status);  
flex_get_inst_handle(InstName2, &id2, &status);  
flex_start_program(&status);
```



Note

If more than one model instance sends commands from a single C testbench, the mode is automatically set to uncoupled, regardless of the settings used (see methods 1-3 in “[Using Uncoupled Mode in a C Testbench](#)” on page 22).

Using Uncoupled Mode in a C Testbench

Uncoupled mode only affects the C command stream. This applies to C Command Mode on simulators with custom integrations and Direct C Control on simulators with standard integrations. For information on FlexModel simulator integration, refer to the [Simulator Configuration Guide for Synopsys Models](#).

Uncoupled mode is required to enable the use of multiple command streams in complex models with more than one bus (for example). It is also useful when you want to drive more than one instance of the same model or multiple models from a single C testbench.

Coupled mode synchronizes the model with the testbench process that contains model commands, so that the model is prevented from advancing to the next simulation time step when the next command is not available. In uncoupled mode, the model does advance to the next time step, even when the next command is not yet available. In this state, the model continues to poll for new commands, thereby preventing gridlock conditions for multi-model or multi-stream simulations.

FlexModels start up in coupled mode by default. There are three methods of changing the default mode:

1. Using a SWIFT model parameter (for simulators with standard integrations), or using the `flex_run_program` command (for simulators with custom integrations) as shown in the following examples:

Standard Integrations

Using SWIFT parameter in the model instantiation for `_fz` models:

```
defparam u1.FlexModelSrc_command_stream = "path_to_C_file -u |-c"
```

where `command_stream` is the name of the command stream, as defined in the model datasheet. For models with multiple command streams, use two `defparams`, with each one pointing to a unique command stream in the model.

Using SWIFT parameter in the model instantiation for `_fx` models:

```
defparam u1.FlexCFile = "path_to_C_file -u |-c"
```

Custom Integrations

Using `flex_run_program` command from a Verilog or VHDL testbench:

```
flex_run_program("path_to_C_file -u |-c", status);
```

If you specify the `-c` option (default behavior), the C testbench starts model commands in coupled mode. If you specify the `-u` option, the C testbench starts model commands in uncoupled mode. Other options are ignored.

**Note**

The -u switch is useful only if you want to be in uncoupled mode for a single command stream. This is not currently needed for FlexModels, but will enable potential future enhancements.

2. You can also use global FlexModel commands in your C testbench to set and get the coupling mode, using the following syntax:

```
flex_set_coupling_mode(int instance, int coupling_mode, int *status);
flex_get_coupling_mode(int instance, int &coupling_mode, int *status);
```

where *coupling_mode* is one of these two constants:

- FLEX_UNCOUPLED_MODE (sets mode to uncoupled)
- FLEX_FULLY_COUPLED_MODE (sets mode to coupled)

Here are some usage examples:

```
flex_set_coupling_mode (mpc8260_inst1, FLEX_UNCOUPLED_MODE, &status);
flex_get_coupling_mode (mpc8260_inst1, &coupling_mode, &status);
```

3. You can also use the `flex_change_setup` global variable in your C testbench to enable or disable uncoupled mode. This can be handy for interactive use with a C debugger:

```
int flex_change_setup;
```

Set the `flex_change_setup` variable to `FLEX_UNCOUPLED_MODE` or `FLEX_FULLY_COUPLED_MODE` depending on the desired mode of operation. You can use this global variable to interactively modify the simulation setup from within the debugger session. After the initialization sequence is complete, the model checks the value of the `flex_change_setup` variable before executing each command and changes the mode of operation accordingly. Here is an example that uses the `flex_change_setup` global variable to change to uncoupled mode in between bus cycles for the `mpc860_fx` model:

```
mpc860_read(id1, address, tr_attr, FLEX_WAIT_F, &status);
flex_change_setup = FLEX_UNCOUPLED_MODE; /*(set from the debugger)*/
mpc860_write(id1, address, tr_attr, data, FLEX_WAIT_F, &status);
```

**Note**

The `flex_change_setup` command may be used to interactively change other simulation settings in the future, if the need arises.

Burst Transfers

Burst transfers are multiple data transfers caused by a single bus command. Like the devices they model, some FlexModels support burst transfers—check the FlexModel datasheets for supported burst transfer commands and how to use them.

Non-pipelined Bus Operations

Use non-pipelined bus operations when you want to branch the control program based on the result returned by the model. This means issuing a model result command right after a paired model request command, as shown in the following example:

```

procedure my_read(instance: in integer;
                  address: in BIT_VECTOR (0 to 31);
                  readType: in natural;
                  result: out BIT_VECTOR (0 to 31)
                  ) is
    variable stat: integer;
    begin
        -- Start Read --
        model_read_req(instance, address, readType, FLEX_WAIT_T, status);
        -- If OK --
        if (status > 0) then
            -- Get Read result (tag not needed) --
            model_read_rslt(instance, address, 0, result, status);
        end if;
    end;
end;
```

The following example shows another command sequence that branches according to the result of the returned data:

```

model_read_req(inst, x"00000060", x"0", FLEX_WAIT_T, status);
model_read_rslt (inst, x"00000060", 0, data, status);
if (status = 1 and data(31 downto 0) = x"33334444")
then model_write(inst, mem_write, x"A00000FF", x"1DE4543C",
                  FLEX_WAIT_F, status);
else assert FALSE report "WRONG DATA READ" severity NOTE;
end if;
```

There is a minimum delay of one clock cycle between the completion of a request command and the completion of a corresponding result command. You must precede a result command with a request command.

Pipelined Bus Operations

Bus cycle pipelining occurs when multiple bus operations overlap. Because FlexModels typically divide bus operations between request and result phases or commands, you can pipeline multiple request commands before the result command from the first request is complete. By preloading the model command queue with pipelined bus operations, you can avoid dead cycles and more closely model the behavior of devices that support pipelining. You can then retrieve the results from those reads in any order. This process is illustrated in [Figure 3](#).

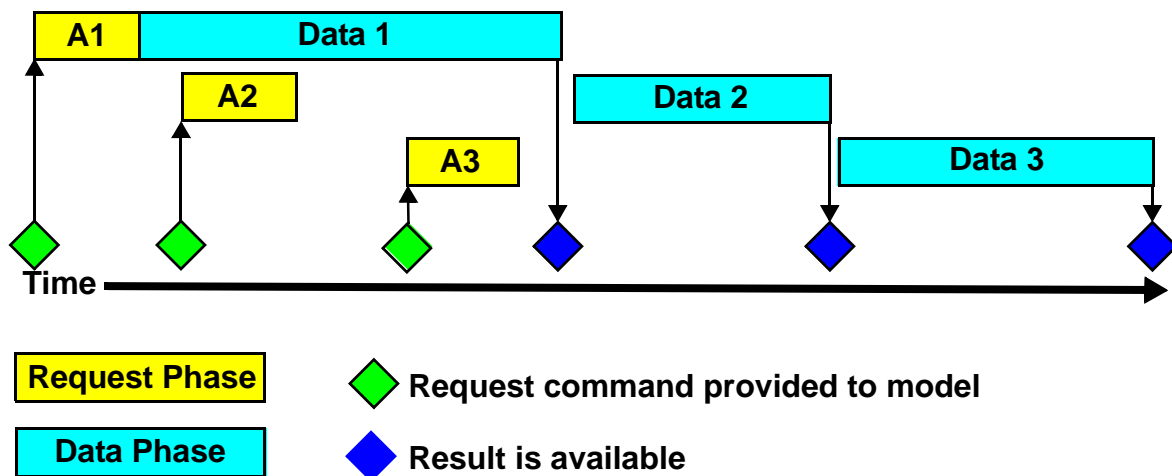


Figure 3: Pipelined Bus Operations

There are two ways to get pipelined bus operations with FlexModels:

- [“Pipelining With `wait_mode` Behavior” on page 25](#)
- [“Pipelining With Delayed Results Checking” on page 26](#)

Pipelining With *wait_mode* Behavior

FlexModel request and result commands work together to retrieve data from the model. Request commands have a *req* suffix and result commands have a *rslt* suffix. Request commands cause the model to post the data and result commands retrieve the results. Testbench operations “wait” or proceed based on how you set the *wait_mode* parameter in the request command. For example:

- If the *wait_mode* in a request command is false (FLEX_WAIT_F), the model immediately proceeds to the next command.
- If the *wait_mode* in a request command is true (FLEX_WAIT_T), the model waits until the command completes before proceeding to the next command.

You can use this wait behavior to pipeline multiple request commands as shown in the following VHDL example.

```
variable data1, data2, data3 : bit_vector (31 downto 0);
-- COMMAND 1:
  model_read_req(inst, X"00000004", X"0", FLEX_WAIT_F, status);
-- COMMAND 2:
  model_read_req(inst, X"00000002", X"0", FLEX_WAIT_F, status);
-- COMMAND 3:
  model_read_req(inst, X"00000000", X"0", FLEX_WAIT_T, status);
  model_read_rslt (inst, X"00000004", 0, data1, status);
  model_read_rslt (inst, X"00000004", 0, data2, status);
  model_read_rslt (inst, X"00000000", 0, data3, status);
```

Commands 1, 2, and 3 are loaded into the model queue immediately because the first two commands have *wait_mode* parameters set to false FLEX_WAIT_F. Commands following 3 are not loaded right away because Command 3 has a *wait_mode* parameter set to true (FLEX_WAIT_T). No further commands are loaded until Command 3 completes. When Command 3 completes, the results commands retrieve the results from the three pipelined read requests.

Pipelining With Delayed Results Checking

Supposed you want to pipeline multiple read commands, and check results in a different order. You can simply invert the order of the result commands as shown in the following example.

```
-- Model Commands are
-- model_read_req(inst, address, wait, status);
-- model_read_rslt(inst, address, result, status);
-- Assume no pipeline reordering

-- read 1
model_read_req(mod1, x"DEADBEEF", FLEX_WAIT_F, tag1);
-- read 2
model_read_req(mod1, x"DEADBEF0", FLEX_WAIT_F, tag2);
-- read 3
model_read_req(mod1, x"DEADBEF1", FLEX_WAIT_F, tag3);
-- result 3
model_read_rslt(mod1, x"DEADBEF1", tag3, data1, stat);
-- result 2
model_read_rslt(mod1, x"DEADBEF0", tag2, data0, stat);
-- result 1
model_read_rslt(mod1, x"DEADBEEF", tag1, data, stat);
```

In this example, the three read requests complete in order, but the read results commands are in reverse order. The model waits until the result 3 command completes (which depends on completion of read 3) before proceeding to the result 2 and result 1 commands, thus producing the pipeline effect.

Synchronizing the Command Flow

To coordinate the behavior of multiple FlexModels in your testbench, use the [flex_synchronize](#) command. Do not use multiple HDL command streams to control a single FlexModel instance. This produces unpredictable model behavior.

The `flex_synchronize` command suspends operations in the model instance identified by the *inst_handle* parameter until the number of instances specified in the *num_instance* parameter execute `flex_synchronize` commands with matching *sync_label* strings. For example, if a FlexModel issues the following command, it suspends all operations until two other model instances execute `flex_synchronize` commands with a matching *sync_label* of “sync1”.

```
flex_synchronize(inst, 3, "sync1" timeout, status);
```

All three models simultaneously execute their next commands one clock cycle after the third model executes this command.

A FlexModel holding for a synchronization cannot recognize any other commands. During this time the model stores exception information in the Command Core exception queue. It is up to the interrupt service routine that you develop to process this exception information after the synchronization occurs. For information on developing interrupt service routines, refer to [“FlexModel Interrupts” on page 30](#).

If a reset occurs, FlexModels execute the reset behavior and either proceed to the next command or resume waiting for the synchronization point if it still hasn't occurred.

If the number of `flex_synchronize` commands with the same *sync_label* does not match the *num_instance* parameter, the Command Core reports an error.

Synchronization Timeouts

If not enough `flex_synchronize` calls are made, several models may get stuck waiting for the last call. To prevent this problem, the `flex_synchronize` command includes a *timeout* value. When a model receives a `flex_synchronize` call, it waits for *timeout* clock cycles before declaring that the synchronization operation is complete. When this happens all other models waiting on the same *sync_label* are allowed to proceed. Subsequent calls using the same *sync_label* return with an error and are ignored. In addition, the same label *sync_label* cannot be used twice. For more information about the `flex_synchronize` command, refer to [“flex_synchronize” on page 77](#).

FlexModel Timing

FlexModels come with standard, component-based timing files just like regular SmartModels. There is a timing file for each model that can accommodate multiple timing versions. By selecting different timing versions for different instances of the same model, you can have these instances behave differently in the design. In addition to these standard timing files, you can create custom, component-based timing files using the SmartModel user-defined timing (UDT) process. UDT is possible because a model's timing file is loaded at simulation startup. For more information on UDT, refer to the [SmartModel Library User's Manual](#).

When you run a FlexModel in timing mode, in general, you are enabling propagation delays, access delays, and timing checks. Bear in mind that FlexModels run up to 40 percent faster in function-only mode, so you may want to set timing mode on only for later simulation runs after functional verification is complete.

Selecting Function-only or Timing Model

By default, FlexModels behave as function-only models. To enable timing mode for a FlexModel, set the FlexTimingMode SWIFT parameter to FLEX_TIMING_MODE_ON (prepend a backtick for Verilog). If you are using Direct C Control, set this parameter to 0 for timing mode off or 1 for timing mode on.

With timing mode on, you can choose the desired timing version for the model by setting the TimingVersion SWIFT parameter. You can also set the timing range (MIN, TYP, or MAX) using the DelayRange SWIFT parameter. For more information about setting FlexModel SWIFT parameters, refer to the [Simulator Configuration Guide for Synopsys Models](#). The following examples enable timing mode on model instance “my_inst_1”.

Verilog Example

Example using SWIFT template generated by host simulator with timing:

```
// Timing-mode instantiation
model
defparam
    u1.FlexModelId = "my_inst_1";
    u1.FlexTimingMode = `FLEX_TIMING_MODE_ON;
    u1.TimingVersion = "timingversion";
    u1.DelayRange = "range";
    u1 ( model_ports );
```

VHDL Example

Example using SWIFT template generated by host simulator with timing:

```
U1: model
    generic map (FlexModelID=> "my_inst_1",
        FlexTimingMode    => FLEX_TIMING_MODE_ON,
        TimingVersion     => "timingversion",
        DelayRange        => "range")
    port map ( model_ports );
```

Controlling Timing Checks and Delays

If you instantiate your FlexModel with timing mode on, you can configure timing checks at runtime using the model-specific *model_set_timing_control* commands. The general syntax for the *model_set_timing_control* commands is:

```
model_set_timing_control(id, timing_parameter, state, status);
```

The complete syntax for these commands and the supported *timing_parameter* values are listed in the individual FlexModel datasheets. The *state* parameter takes one of two predefined constants:

- FLEX_ENABLE—Enables timing for the specified parameter.
- FLEX_DISABLE—Disables timing for the specified parameter.

The following examples show how to use *model_set_timing_control* commands to configure timing checks for the tms320c6201_fx FlexModel. The first command initializes timing with all timing and access delays turned on. Then, specific commands turn off all setup checks, and one specific hold check.

Verilog Example

```
// Timing previously enabled with FlexTimingMode parameter for inst
// Turn off all setup timing checks
tms320c6201_set_timing_control(inst, `TMS320C6201_SETUP,
                                `FLEX_DISABLE, status);

// Turn off the hold check from CLKOUT(1h) to INT7(ha)
tms320c6201_set_timing_control(inst,
                                `TMS320C6201_TH_CLKOUT1_LH_INT7_HA, `FLEX_DISABLE, status);
```

VHDL Example

```
-- Timing previously enabled with FlexTimingMode parameter for inst
-- Turn off all setup timing checks
tms320c6201_set_timing_control(inst,TMS320C6201_SETUP,FLEX_DISABLE,
                               status);

-- Turn off the hold check from CLKOUT(1h) to INT7(ha)
tms320c6201_set_timing_control(inst,TMS320C6201_TH_CLKOUT1_LH_INT7_HA,
                               FLEX_DISABLE, status);
```

FlexModel Interrupts

Most FlexModels support interrupts of various types based on the physical devices they model. For information on the specific interrupt types supported by individual FlexModels, refer to the model datasheets. This chapter explains how interrupts are detected and serviced by FlexModels and how to write interrupt routines in VHDL, Verilog, VERA, and C.

Interrupt Service Routines

If you want a FlexModel to respond to interrupts, you must write an interrupt service routine that specifies how the model handles interrupts of different priorities. Check the example testbenches that come with all FlexModels. Many of them have basic interrupt service routines that you can copy and modify as needed based on how you want to control FlexModel interrupts in your own testbench:

- VHDL — \$LMC_HOME/models/*modellmodelversion*/examples/vhdl/*model_tst.vhd*
- Verilog — \$LMC_HOME/models/*modellmodelversion*/examples/verilog/*model_tst.v*
- C — \$LMC_HOME/models/*modellmodelversion*/examples/C/*model_c_commands.c*

You can also use the example interrupt service routines documented in this chapter as starting points:

- [“Developing HDL Interrupt Routines” on page 32](#)
- [“Developing C Interrupt Routines” on page 34](#)
- [“Developing VERA Interrupt Routines” on page 36](#)

Detecting and Servicing Interrupts

Interrupts can only be detected while the HDL or C command source is allowing simulation time to advance. FlexModel interrupts are level sensitive—the models check for and detect interrupts only on rising clock edges. When a FlexModel detects a supported interrupt signal asserted, it queues the servicing request. To make a FlexModel detect an asynchronous interrupt, latch the value so that the model can “see” the interrupt on the next rising clock edge.

While responding to an interrupt, a FlexModel can detect another interrupt and call the interrupt service routine again as long as simulation time has advanced at least one clock cycle. If the new interrupt has a higher priority than the one currently being serviced, the model bumps the lower-priority interrupt and completes the processing for the higher-priority interrupt before returning to and finishing up the processing for the lower-priority interrupt. You can nest interrupt processing this way with as many different interrupt priorities as you want. Resets always have the highest priority and cannot be masked.

FlexModels complete any previously started bus cycle before servicing interrupts as specified in your interrupt service routine. For example, consider a command stream with nine bus commands. Interrupt detection and servicing might proceed as shown in [Figure 4](#).

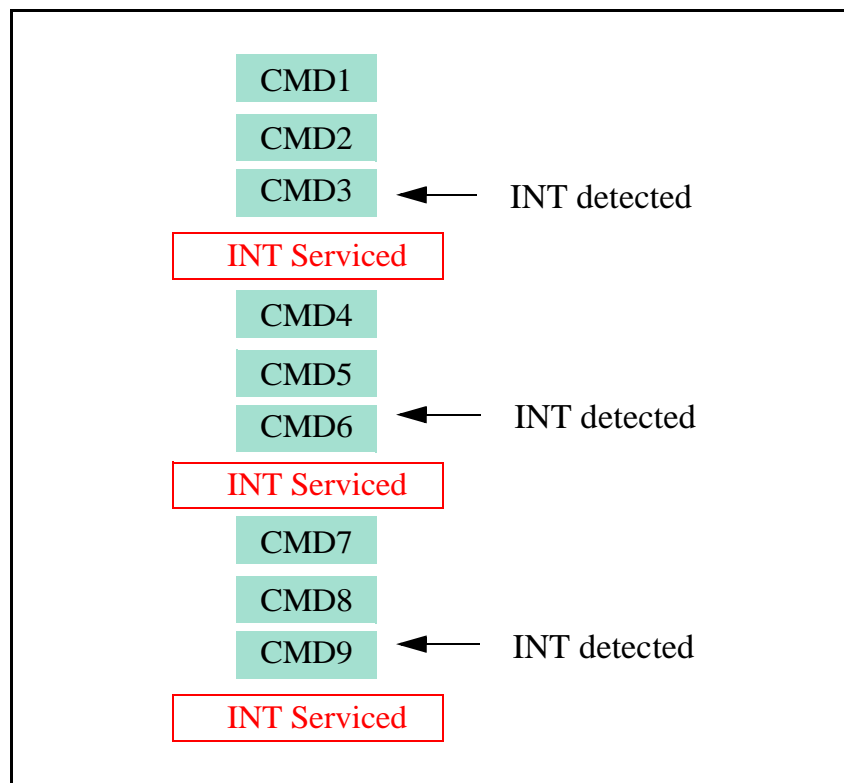


Figure 4: Interrupt Detection and Servicing

Once an interrupt is detected and reported, FlexModels wait until the interrupt is deasserted and reasserted before recognizing the same interrupt again.

Using Multiple Models

You can have more than one FlexModel or multiple instances of the same FlexModel in one HDL testbench. If so, create separate interrupt service routines for each model instance. However, you cannot have more than one FlexModel or multiple instances of the same FlexModel in the same C program. With C, you need to create separate model instantiations (Direct C Control) or separate programs that you launch with the [flex_run_program](#) command (HDL Command Mode). Each process should have its own interrupt service routine.

Interrupt Detection with C Testbenches

To allow simulation time to advance while your C testbench is running and thus enable interrupt servicing, set the *wait_mode* parameter of the last model bus cycle command to true (FLEX_WAIT_T). As an alternative, you can use the [flex_wait](#) command to achieve the same effect. Otherwise the C program will simply run to completion before any interrupts can be serviced by the model.

Note that interrupt service routines in C testbenches can use result (*model_rslt*) commands and model commands with the *wait_mode* parameter set to true (FLEX_WAIT_T). These are not allowed in HDL Command Mode.

When a FLEX_WAIT_T command is executing in a C interrupt service routine, other interrupts of higher priorities can be serviced.

Developing HDL Interrupt Routines

To develop a VHDL or Verilog interrupt service routine, follow these steps:

1. Define a signal or reg in your testbench called *MyInt* or something similar.
2. Use the [flex_define_intr_signal](#) command to register this signal with the FlexModel. You only need one testbench signal regardless of the number of interrupt types the model supports. FlexModels toggle this signal when any supported interrupt pin is asserted.
3. Write process (*MyInt*) or always (*MyInt*) blocks for VHDL or Verilog, respectively. You need one block for each interrupt priority you want to support.

The first command in each process/always block must be a *model_get_intr_priority* command. This command returns a positive integer—higher numbers indicate higher interrupt priorities. Each process/always block should check the returned interrupt priority. Only one will match and execute its service routine.

All model commands in an interrupt service routine must be contained within paired *model_begin_intr* and *model_end_intr* commands that both specify the same model instance handle and interrupt priority. In addition, all interrupt commands contained between the *model_begin_intr* and *model_end_intr* commands must have their *wait_mode* parameters set to false (FLEX_WAIT_F). You cannot set *wait_mode* to true (FLEX_WAIT_T) or use results (*model_rslt*) commands in HDL interrupt routines.

Example HDL Interrupt Routine

The following VHDL example interrupt service routine for the tms320c6201_fx model can handle two interrupts (INT1 and INT2), where INT2 has a higher priority than INT1. For Verilog, replace the process statements with always blocks.

```
architecture
    signal model_int_signal : std_logic;
begin

    process
        flex_define_intr_signal(inst,"TOP/tms320c6201_int_signal",status);
        ....
    end
    process (tms320c6201_int_signal) --PROCESS 1
        tms320c6201_get_intr_priority(inst, priority, status);
        if (priority = 1)
            tms320c6201_begin_intr(inst, priority, status);
            --interrupt service routine for priority 1
            tms320c6201_end_intr(inst, priority, status);
        end
    end

    process (tms320c6201_int_signal) --PROCESS 2
        tms320c6201_get_intr_priority(inst, priority, status);
        if (priority = 2)
            tms320c6201_begin_intr(inst, priority, status);
            --interrupt service routine for priority 2
            tms320c6201_end_intr(inst, priority, status);
        end
    end
end
```



Attention

With Scirocco 2000.02, you can only define interrupt signals at the top level in your HDL testbench without specifying any hierarchy. For example, to make the above code sample work with Scirocco 2000.02, change “TOP/tms320c6201_int_signal” to “tms320c6201_int_signal”. With Scirocco 2000.06 and above, use a full path delimited by colons (“:TOP:tms320c6201_int_signal”).

Scenario 1—INT1 occurs before INT2

To understand how this interrupt service routine works, consider the case where the FlexModel receives the INT1 interrupt before the INT2 interrupt. The execution sequence proceeds as follows:

1. Model samples the INT1 signal asserted and toggles the *model_int_signal* in the HDL testbench.
2. This starts both process 1 and process 2.
3. The *model_get_intr_priority* command executes and returns the priority as “1” so process 2 exits and process 1 starts executing its commands. Simulation time continues to advance as model commands in the interrupt routine are executed.
4. Now the model samples the INT2 signal asserted and again toggles the *model_int_signal* signal in the HDL testbench.
5. This starts process 1 and process 2 again.
6. The *model_get_intr_priority* command executes and returns the priority as “2” so process 1 exits and process 2 starts executing its commands.
7. At this point the model stops executing commands for INT1 and begins processing commands for INT2, because it has a higher priority.
8. When the model finishes executing all commands for INT2, it goes back and finishes executing commands for INT1.

Scenario 2—INT2 occurs before INT1

In the other case, where the model samples INT2 and begins processing commands from process 2 before a lower-priority INT1 interrupt occurs, the model finishes processing all commands for INT2 before servicing the lower-priority INT1.

Developing C Interrupt Routines

To develop C interrupt service routines for use with Direct C Control or C Command Mode, follow these steps:

1. Define a function in the C testbench or program for the interrupt handler.
2. Register this function with the model using the [flex_switch_intr_control](#) command. This function is called by the model whenever it samples a supported interrupt signal asserted.

Note that you only need one interrupt function in your C testbench to handle all interrupts of any type for that model instance. Attempts to register more than one interrupt function for the same model instance result in an error.

3. The first command in the interrupt function must be a *model_get_intr_priority* command. This command returns a positive integer—higher numbers indicate higher interrupt priorities. You can decode this priority using case statements and then begin the processing appropriate for that interrupt priority level.
4. Enclose all model commands for a particular model instance and interrupt priority in between paired *model_begin_intr* and *model_end_intr* commands with the same model instance handle and interrupt priority.

Example C Interrupt Routine

For an extensive example of a C interrupt routine, refer to [“C Testbench Example” on page 95](#). The following example is smaller in scope, but illustrates the basic structure required.

```
#include "flexmodel_pkg.h"
#include "model_pkg.h" /* Interrupt Function Prototype */
void my_intr_handler();
int id; /* NOTE: id is global so it is visible in the interrupt routine */
main() {
    int status;
    char *Inst1;
    /* Begin Initialization Sequence */
    flex_get_inst_handle(Inst1, &id, &status);
    flex_start_program(&status)
    /* Register interrupt function with command core, only after this
       command has been executed will the function be called (when an
       interrupt occurs) */
    flex_define_intr_function(id, my_intr_handler, &status);
    . . . . . /* Continue model/generic commands */
    /* Verify C testbench is still running while HDL interrupts occur */
    void my_intr_handler() {
        int valid, id, priority;
        /* Get the Model Id and Priority for the interrupt that occurred */
        model_get_intr_id_priority(id, &valid, &priority, &status);
        switch(priority) {
            case 1 :
                model_begin_intr(id,priority,&status);
                /* Issue commands HERE for priority 1 */
                model_end_intr(id,priority,&status);
            case 2 : /* Issue commands HERE for priority 2 */
                break;
            default :
                break;
        } /* end switch (priority) */
        . . . . .
    }
}
```

Developing VERA Interrupt Routines

To use interrupts in VERA Command Mode, follow the same procedures described in [“Developing HDL Interrupt Routines” on page 32](#). However, do not include the interrupt signal in the VERA testbench. Instead, define the interrupt signal in the top-level VHDL or Verilog testbench, just as if you were in HDL Command Mode.

Note that in VERA, as in HDL, you cannot use model commands with the *wait_mode* parameter set to true (FLEX_WAIT_T) or result (*model_rslt*) commands within interrupt routines.

Defining the Interrupt Signal

When you define the interrupt signal in the VERA testbench, you must pass in the full path to the signal in the top-level VHDL testbench or reg in the top-level Verilog testbench. The following example shows how to define an interrupt signal in a top-level Verilog testbench:

```
#include <vera_defines.vrh>
#include "flexmodel_pkg.vrh"
#include "model_pkg.vrh"

program my_test
{
  // Create an instance of the model class.
  ModelFx model = new("modelInstName", "u1.CLK");
  // Define the Intr Signal
  // NOTE : here INTR_SIGNAL is the name of a reg in the top
  //         level verilog testbench and we pass the full path
  //         to the interrupt signal.
  model.define_intr_signal("model_test_top.INTR_SIGNAL", status);
}
```

Monitoring the Interrupt Signal

The following example illustrates one way to monitor the interrupt signal in a top-level VHDL testbench. For other methods of determining when the interrupt signal has been toggled in the Verilog or VHDL testbench, and for more information on VERA syntax, refer to the *Vera Verification System User's Manual*.

```
#include <vera_defines.vrh>
#include "flexmodel_pkg.vrh"
#include "model_pkg.vrh"
// Create A VERA Port data type.
port my_port { intrSignal; }
program my_test
{
  // Create an instance of the model class.
```

```

    ModelFx model = new("modelInstName", "u1/CLK");
    // Create a Variable of type my_port
    // Give it a null bind
    my_port intrPort = new;
    // Make a connection to the interrupt signal in the
    // top level VHDL testbench
    signal_connect(intrPort.$intrSignal, "model_test_top/INTR_SIGNAL",
"dir=input itype=PSAMPLE");
    // Define the Intr Signal
    model.define_intr_signal("model_test_top/INTR_SIGNAL", status);
    fork
    {
        // Interrupt Routine For Interrupt Priority 1
        integer priority, valid_f, status;
        while (1)
        {
            @ ( intrPort.$intrSignal );
            model.get_intr_priority(valid_f, priority, status);
            if ( priority == 1 )
            {
                printf ("***** DETECTED EXCEPTION PRIORITY 1 \n");
                model.begin_intr(1, status);
                // Send Commands For Priority 1 here.
                model.end_intr(1,status);
            }
        }
    }
    {
        // Interrupt Routine For Interrupt Priority 2
        integer priority, valid_f, status;
        while (1)
        {
            @ ( intrPort.$intrSignal );
            model.get_intr_priority(valid_f, priority, status);
            if ( priority == 2 )
            {
                printf ("***** DETECTED EXCEPTION PRIORITY 2 \n");
                model.begin_intr(2, status);
                // Send Commands For Priority 1 here.
                model.end_intr(2,status);
            }
        }
    }
    join // End fork
} // End program my_test

```

3

FlexModel Command Modes

Introduction

If you are using a simulator with a custom FlexModel integration, you can issue FlexModel commands from HDL, VERA, or C. Otherwise, with the standard SWIFT integration, you use Direct C Control. For information about configuring FlexModels in your simulator with both standard and custom integrations, refer to the [Simulator Configuration Guide for Synopsys Models](#). This chapter explains how to use FlexModels with the different command modes:

- [“Using HDL Command Mode” on page 39](#)
- [“Using C Command Mode” on page 43](#)
- [“Using VERA Command Mode” on page 47](#)

Using HDL Command Mode

In HDL Command Mode, FlexModels execute VHDL or Verilog commands contained in the top-level system testbench file. HDL Command Mode gives you control over the model that is tightly integrated with events in the simulation. In this mode, you can generate command results, create test sequences that loop or branch on command results, and synchronize the command flow of several models in a testbench.

When you use HDL Command Mode, the Command Core queues model commands and executes them in the order received. Multiple commands can be active simultaneously (waiting for results) if the model supports pipelining.

To use HDL Command Mode, instantiate a FlexModel in your testbench and then create a command process for the model that includes the FlexModel commands that you want the model to execute. Here is a VHDL example of a command process for a FlexModel executing in HDL Command Mode:

```

CMD_STREAM: process
begin
    wait for CLK_PERIOD;
    assert (false) report "loading commands" severity NOTE;
    model_configure(inst,cls_code,X"112233",status);  -- Class Code
    model_configure(inst,dev_id,X"4500",status);
    model_idle(inst, 5, FLEX_WAIT_F,status);
    model_read_req(inst, X"00000004", X"0", FLEX_WAIT_F, status);
    flex_print_msg (inst, "This is a read_req test", status);
    assert (false) report "end of commands" severity NOTE;
    wait;
end process CMD_STREAM;

```

Do not use multiple HDL command streams to control a single FlexModel instance; this produces unpredictable model behavior.

In HDL Command Mode, FlexModel commands are executed as procedure calls (VHDL) or task calls (Verilog) at the testbench level. The testbench continues issuing commands to the models until it encounters one of the following:

- **A result command or a command containing a *wait_mode* parameter set to true.** This causes the testbench to wait for the model to complete the command before issuing the next command.
- **A *flex_run_program* command.** This transfers control to a C testbench. Subsequent model commands in the HDL testbench are processed only after all model command in the C testbench have completed.
- **A *flex_synchronize* command.** This causes the testbench to suspend command delivery to one or more models until the specified number of models execute corresponding *flex_synchronize* commands. (See [“Using HDL Command Mode” on page 39.](#))



Note

In Verilog, when you use the *model_read_rslt* command or any FLEX_WAIT_T command simultaneously from two instances of the same model, you get corrupted results. This is because Verilog tasks have a static scope within a module.

VHDL Control

Model commands are VHDL procedures delivered in model-specific packages. These Synopsys-provided FlexModel packages include the procedures and constants needed to call the model from the testbench.

You must specify an instance-specific model identifier by setting a VHDL generic. Model instances that do not have unique identifiers cause the Command Core to issue an error. If there is only one model instance, the default ID value is zero.

In HDL Command Mode, you need to define a unique interrupt signal for each interrupt used by each model instance. A particular model instance may also require multiple interrupt service routines. See [“Developing HDL Interrupt Routines” on page 32](#) for an example model interrupt service routine.

Verilog Control

The Verilog control mechanism closely mirrors that of the VHDL implementation. You must include Synopsys-provided, model-specific Verilog source files to make the FlexModel tasks available to the testbench.

For information on getting FlexModels set up with their HDL package files, refer to the [Simulator Configuration Guide for Synopsys Models](#).

HDL Control Between Model and Testbench

If you are using a simulator with a custom FlexModel integration, individual FlexModels come with a set of HDL procedures or tasks that can be invoked from the HDL testbench. These procedures communicate with the Command Core using the same HDL-to-C mechanism that the models use. The HDL testbench and the model do not attempt to access the Command Core at the same time, preventing conflicts. FlexModels interact with the Command Core on falling clock edges, while the HDL testbench procedures or tasks use the rising edges.

Figure 5 illustrates a simple read_req/read_rslt pair from the HDL testbench without pipelining. The testbench and model activity are synchronized to the rising edges of the clock, but the interaction between the FlexModel and the Command Core only occurs on the falling edge. The user sees a single clock cycle delay before the first command starts, and a one-cycle delay before the results of the operation are available to the testbench.

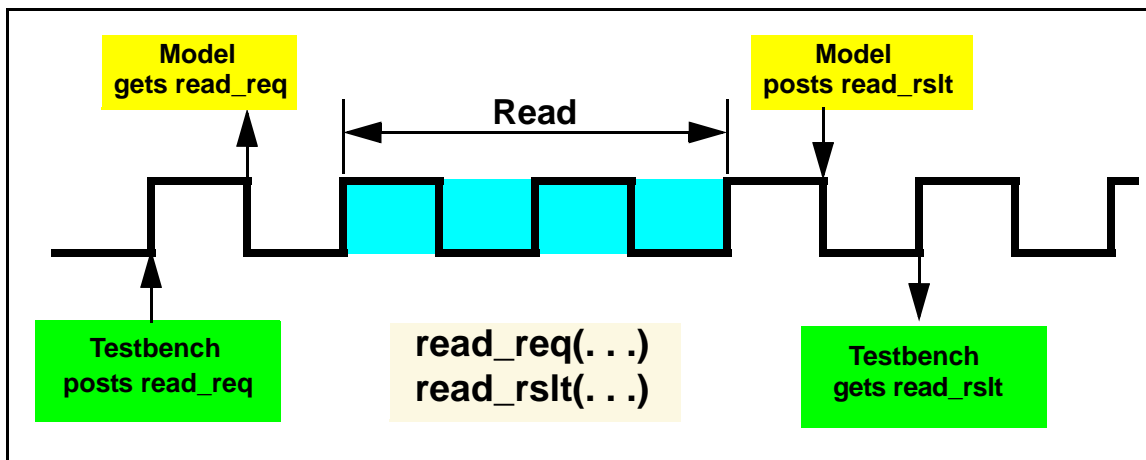


Figure 5: Read_req/read_rslt Pair for Testbench

Figure 6 shows how multiple model state commands can occur in a single clock cycle.

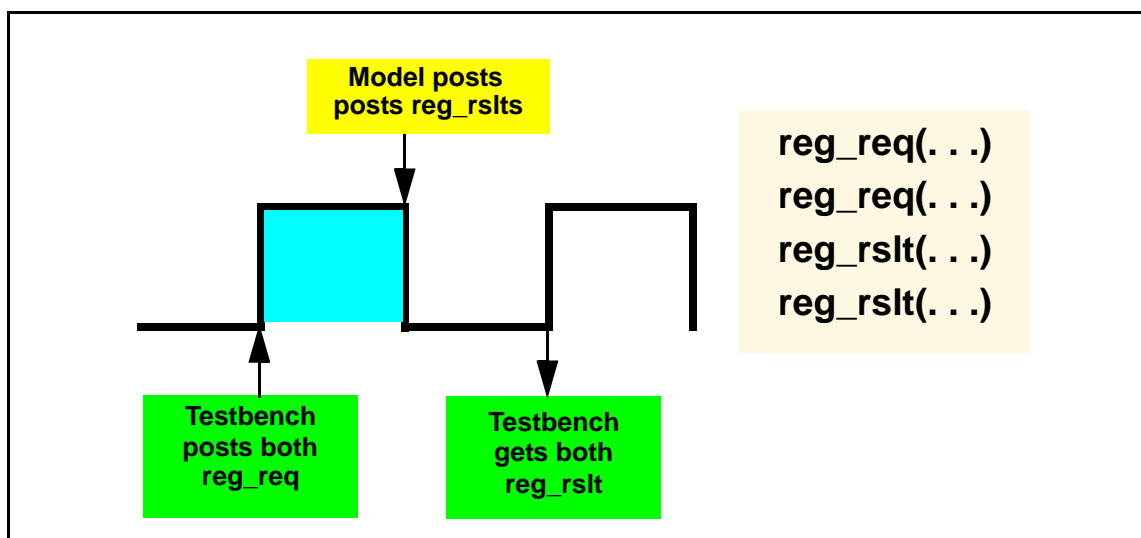


Figure 6: Multiple Commands within a Single Clock Cycle

The mechanism illustrated in Figure 6 guarantees that the model is in a stable state when the register values are posted.

Using C Command Mode

The following description of C Command Mode applies just to simulators with custom FlexModel integrations. Customers with standard SWIFT integrations can also issue commands from a C program, but use a different method called Direct C Control. For more information, refer to the [Simulator Configuration Guide for Synopsys Models](#).

In C Command Mode, FlexModels execute commands contained in an external C program. C Command Mode is efficient because you don't have to recompile your simulator when you make changes to the model control program. Note that the C programming language does not provide for concurrency or recognize the notion of simulation time. For information on other limitations to be aware of when using C Command Mode, refer to ["Pipelined Bus Operations"](#) on page 25.

Figure 7 shows how to enable C Command Mode using the `flex_run_program` command to call an external C program from the HDL testbench.

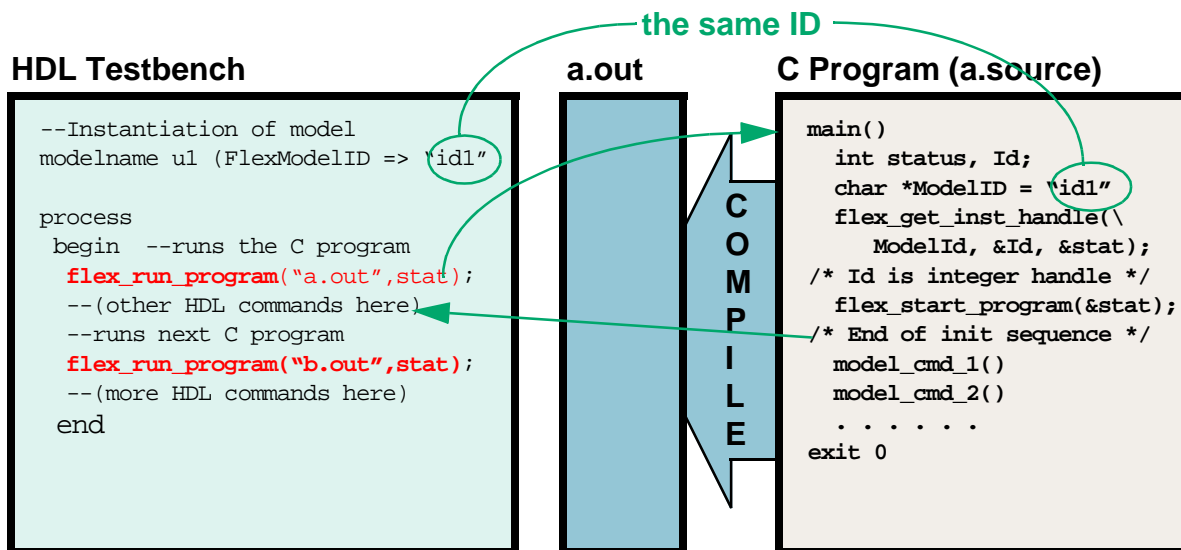


Figure 7: Accessing a C Testbench from HDL

In Figure 7, control returns from the C program back to the HDL testbench after the "init" sequence completes, immediately following execution of the `flex_start_program` command. All model commands in the C program are executed by the model before any subsequent model commands in that VHDL process or Verilog block.



Note

You cannot have multiple VHDL processes or Verilog blocks providing commands to the same model instance.

Keep in mind that integers in HDL are integers in C. In C Command Mode, `std_logic_vectors` and `bit_vectors` used as input values to functions are represented using the `FLEX_VEC_CONST` type, while return values are represented using the `FLEX_VEC` type. Also, C functions with return values require you to pass in the address. For information about creating and using `FLEX_VEC` vectors for use with FlexModel commands, refer to [“FlexModel C Testbench Interface” on page 81](#).

To use C Command Mode, refer to the following procedures:

1. [“Creating an External C File” on page 44](#)
2. [“Compiling an External C File” on page 45](#)
3. [“Switching Control to an External C Program” on page 47](#)

Creating an External C File

Create the external C file according to the following procedure:

1. Include the two Synopsys-provided header files:
 - **`flexmodel_pkg.h`** This file contains the function prototypes for the generic FlexModel functions.
Location: `$LMC_HOME/sim/C/src`
 - **`model_pkg.h`** This file contains model-specific function prototypes and constants that make the commands easier to use.
Location: `$LMC_HOME/models/model_fx/model_fxversion/src/C`
2. Initialize the C program using the [flex_get_inst_handle](#) and [flex_start_program](#) commands, as shown in the following example:

```
main()
{
    int status, Id;
    char *ModelID = "idl" /* Must be same as in VHDL testbench */
    flex_get_inst_handle( ModelID, &Id, &status);
    /* Id is the returned integer handle */
    flex_start_program(&status); /* End of initialization sequence */
}
```

This next example adds a definition for an interrupt function.

```
/* This is in the C testbench */
void my_intr_function()
{
    main() {
        int status, Id;
        char *ModelID = "idl";

        flex_get_inst_handle(ModelID, &Id, &status);
        /* Exiting initialization phase */
    }
}
```

```
flex_start_program(&status);
/* Registering my_intr_function next */
flex_define_intr_function(Id, my_intr_function, &status);
```

Common Errors to Avoid

Here's an example of what not to do. You cannot issue a `flex_start_program` command until you obtain a model instance handle using the `flex_get_inst_handle` command.

```
main() {
    int status, Id;
    char *ModelId = "idl";

    flex_start_program(&status);
    /** Error: flex_start_program before getting instance handles ***/
```

Another common error is to issue model commands before the initialization sequence is complete, as shown in the following example.

```
main() {
    int status, Id;
    char *ModelId = "idl";

    flex_get_inst_handle(ModelId, &Id, &status);
    model_write(Id, Addr, Data, &status);
    /** Error: issuing model command before end of initialization ***/
```

Compiling an External C File

The compile line you use differs based on your platform. Note that these examples include creation of a working directory (*workdir*) and running `flexm_setup`:

- a. On HP-UX, you need to link in the -LBSD library as shown in the following example:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% /bin/c89 -o executable_name
your_C_file.c
workdir/src/C/hp700/model_pkg.o
$LMC_HOME/lib/hp700.lib/flexmodel_pkg.o
-I$LMC_HOME/sim/C/src
-Iworkdir/src/C
-lBSD
```

- b. On Solaris, you need to link in the `-lsocket` library as shown in the following example:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% cc -o executable_name
your_C_file.c
workdir/src/C/solaris/model_pkg.o
${LMC_HOME}/lib/sun4Solaris.lib/flexmodel_pkg.o
-I${LMC_HOME}/sim/C/src
-Iworkdir/src/C
-lsocket
```

- c. AIX:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% /bin/cc -o executable_name
your_C_file.c
workdir/src/C/ibmrs/model_pkg.o
${LMC_HOME}/lib/ibmrs.lib/flexmodel_pkg.o
-Iworkdir/src/C
-I${LMC_HOME}/sim/C/src
-ldl
```

- d. Linux:

```
% mkdir workdir
% flexm_setup -dir workdir model_fx
% egcs -o executable_name
your_C_file.c
workdir/src/C/x86_linux/model_pkg.o
${LMC_HOME}/lib/x86_linux.lib/flexmodel_pkg.o
-Iworkdir/src/C
-I${LMC_HOME}/sim/C/src
```

- e. On NT, you need to link in a Windows socket library as shown in the following example.

```
> md workdir
> flexm_setup -dir workdir model_fx
> cl -O2 -MD -DMSC -DWIN32 -Feexecutable_name
your_C_file.c
workdir\src\C\pcnt\model_pkg.obj
%LMC_HOME%\lib\pcnt.lib\flexmodel_pkg.obj
-I%LMC_HOME%\sim\C\src
-Iworkdir\src\C
wssock32.lib
```

**Note**

The entire compilation expression must appear on the same line. The NT example was tested using Microsoft's Visual C++ compiler v5.0.

Switching Control to an External C Program

You switch model control to an external C program using the `flex_run_program` command in your HDL testbench. The following example shows a FlexModel command process that executes an external C program.

```
CMD_STREAM : process
begin
    wait for CLK_PERIOD;
    assert(false) report "Running C Program" severity NOTE;
    flex_run_program("a.out", status);
    assert(false) report "Finished Running C Program" severity NOTE;
    wait;
end process CMD_STREAM;
```

Using VERA Command Mode

FlexModels come with an object-oriented VERA command interface that lets you control them from a VERA testbench. When you use FlexModel from VERA, you get all the benefits of the powerful VERA verification language. In VERA Command Mode, you can use any FlexModel command or feature available in HDL Command Mode. The FlexModel-to-VERA command interface is a direct connection to the C part of the hybrid HDL/C FlexModel architecture. Because the connection is not through the simulator PLI/FLI, it runs faster.

VERA Command Mode syntax differs slightly from that of HDL Command Mode. For more information, see [“Command Syntax Differences in VERA Command Mode” on page 58](#).

The following sections document how to use VERA with FlexModels. For general information about using VERA, refer to the *Vera Verification System User's Manual*.

FlexModel VERA Classes

VERA is an object-oriented language. The FlexModel VERA command interface uses the inheritance feature to construct a model class hierarchy. At the top of the hierarchy is a general model class. Other model classes inherit from this general class. [Figure 8](#) shows the model hierarchy.

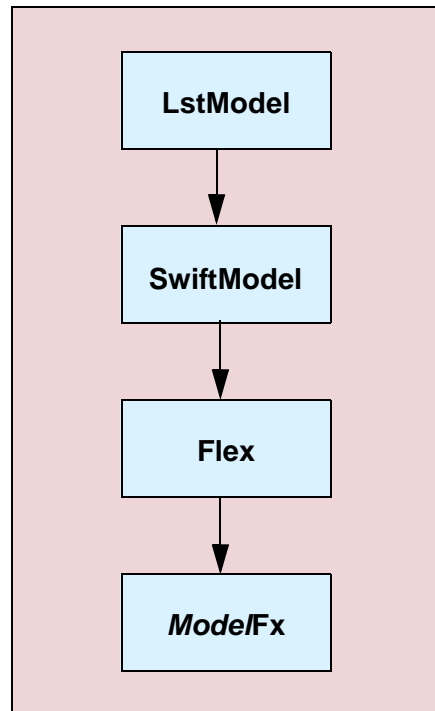


Figure 8: VERA Model Class Hierarchy

The *LstModel*, *SwiftModel*, and *Flex* classes are abstract or virtual classes. These classes cannot be instantiated directly in VERA testbenches. Only an instance of a *ModelFx* class can be created in a VERA testbench.

The commands used to control FlexModels are public methods of the *ModelFx* class. You can send FlexModel commands from VERA to the model only through an instance of the *ModelFx* class. Global FlexModel commands (see [“Global FlexModel Command Descriptions” on page 60](#)) must also be sent through an instance of the *ModelFx* class. The *ModelFx* class automatically inherits any new features that are added to the *LstModel* and *SwiftModel* classes.

VERA Files in the LMC_HOME Tree

[Table 1](#) describes the VERA files installed in your LMC_HOME tree.

Table 1: VERA Files in the LMC_HOME Directory

File Name	Location	Description
lstmodel.vrh	\$LMC_HOME/sim/vera/src	External class declaration for LstModel class.
swiftmodel.vrh	\$LMC_HOME/sim/vera/src	External class declaration for SwiftModel class.
flexmodel_pkg.vrh	\$LMC_HOME/sim/vera/src	External class declaration for the Flex class.
model_pkg.vrh	\$LMC_HOME/models/model_fx/model_fxversion/src/vera	External class declaration for the model-specific <i>ModelFx</i> class.
lstmodel.vr	\$LMC_HOME/sim/vera/src	Source file for the LstModel class.
swiftmodel.vr	\$LMC_HOME/sim/vera/src	Source file for the SwiftModel class.
flexmodel_pkg.vr	\$LMC_HOME/sim/vera/src	Source file for the Flex class.
model_pkg.vr	\$LMC_HOME/models/model_fx/model_fxversion/src/vera	VERA source file for the model-specific <i>ModelFx</i> class.

The *ModelFx* Class Constructor

The constructor for a *ModelFx* class expects two string arguments, the FlexModel instance name and the clock signal.

The first argument, the FlexModel instance name, is the string instance name given to the FlexModel in the top-level Verilog or VHDL testbench. The constructor uses this argument to get an instance handle for the FlexModel. If the instance name passed is invalid, the model issues an error and sets a flag in the class indicating the severity of the error. (For information on accessing the error status, see [“Accessing the Current Error Status”](#) on page 51.)

The second argument is the full path to the clock signal to be used in FlexModel commands. This clock signal is used within commands that have associated wait behavior. VERA creates a dynamic bind to this clock signal within the constructor using VERA's signal_connect feature.

Because you are using VERA's signal_connect function, you must use the -x switch with Verilog-XL at runtime, or use -P \$VERA_HOME/lib/vera_pli_dyn.tab for VCS at runtime. For more information about the signal_connect function, refer to the *Vera Verification System User's Manual*.

**Note**

If VERA cannot find the clock signal in the design, it issues a runtime error.

If you call the constructor at the same time you create the *ModelFx* object, the constructor returns at the next positive edge of the clock signal passed in. This delay is necessary because the testbench cannot obtain the FlexModel's instance handle until at least one clock period has elapsed. If you create the model object and call the new function after some time has elapsed, the constructor returns immediately.

Examples with Top-level Testbenches

The following two VERA testbench examples show a VERA testbench paired with a Verilog testbench and a VERA testbench paired with a VHDL testbench. Note that in the Verilog example, the model's constructor advances to the next positive edge of top.U1.CLK (the clock signal passed in to the constructor) before returning. In the VHDL example, however, the model's constructor returns immediately, because the testbench has already waited for one clock.

Example: VERA Testbench Paired with Verilog Testbench

Verilog Testbench

```
module top;
    .
    .
    .
    myfxmodel U1 (.CLK(CLK), .RST(RST));
    defparam
        U1.FlexModelId = "my_model";
```

VERA Testbench

```
program model_test {
    ModelFx    inst1 = new("my_model", "top.U1.CLK");
```

Example: VERA Testbench Paired with Top-level VHDL Testbench

VHDL Testbench

```
entity top if end top;
architecture test of top is
    .
    .
    .
    U1 : myfxmodel
        generic map (FlexModelId => "my_model")
        port map (
            CLK => CLK,
            RST => RST
        );
```

VERA Testbench

```
program model_test {
    ModelFx    inst1;
    repeat (1) @ posedge CLOCK;
    inst1 = new("my_model", "top/U1/CLK");
```

Accessing the Current Error Status

When an error occurs within the model object, it prints an error message to standard error. The model object saves the error message and the severity of the error. There are three possible severity levels:

- FLEX_VERA_NOERROR—no errors
- FLEX_VERA_WARNING—warnings
- FLEX_VERA_FATAL—fatal errors

You can use one of two methods to access the current error status:

- showStatus()—Returns the present error severity level.
- showErrors()—Prints any errors to standard out.

Example: Accessing Current Error Status

The following example shows to get the current error status from a VERA testbench.

```
program model_test
{
    ModelFx    inst1;
    repeat (1) @ posedge CLOCK;
    inst1 = new("my_model", "top/U1/CLK");

    if ( inst1.showStatus() == FLEX_VERA_FATAL ) {
        inst1.showErrors();
        // Take suitable action
    }
    else {
        // No fatal errors, proceed.
    }
} // program model_test
```

FlexModel Logging from the VERA Class

When FlexModel logging is turned on, the VERA class creates a file and logs the versions of the objects. This file is used by Customer Support for debugging purposes. This file is named *model_instance_name.versions*. For example, if the instance name for the tms320c6201 is inst1, then the file created is named tms320c6201_inst1.versions. For more information on FlexModel logging, see [“Reporting Problems” on page 101](#).

4

FlexModel Command Reference

Introduction

This chapter explains the different types of FlexModel commands and their common elements and provides a complete command reference for the global FlexModel commands. This information is presented in the following major sections:

- [“Model-Specific and Global Commands” on page 53](#)
- [“About the Commands” on page 54](#)
- [“Global FlexModel Command Descriptions” on page 60](#)

Model-Specific and Global Commands

You use FlexModels by issuing commands to model instances in your testbench. There are two basic kinds of FlexModel commands, as shown in [Table 2](#).

Table 2: FlexModel Command Types

Command Type	Used To	How to Identify	Where Documented
Model-specific	Exercise processor or bus protocol functions.	Command prefix equals the model name. For example: mpc860_write	Individual model datasheets.
Global	Control program flow or handle general housekeeping functions.	Command prefix is flex. For example: flex_get_inst_handle	In this manual. Refer to “Global FlexModel Command Descriptions” on page 60 later in this chapter.

Model-specific and global commands can generally be used in all FlexModel command modes, including HDL Command Mode, VERA Command Mode, C Command Mode, and Direct C Control. In addition, FlexModels support a set of C functions and operators for use with model commands when working in C Command Mode or with Direct C Control. These C functions also have the “flex” prefix. For details on the supported C functions and operators, refer to [“FlexModel Command Reference” on page 53](#).

**Note**

In VERA Command Mode, you use the *modelObject* prefix instead of the *model* prefix. For more information, refer to [“Command Syntax Differences in VERA Command Mode” on page 58](#).

About the Commands

FlexModel commands are built with a common underlying architecture to aid readability. For example, we already saw that model-specific commands are easy to identify because they all have the model name as their prefix. And of course, the command names are intended to describe the functions performed. Understanding the meaning of other command components can help improve your productivity working with FlexModels, so let's take a look at the key features of FlexModel commands.

Bus and Zero-Cycle Commands

Not all FlexModel commands generate bus cycles. For example, commands that check or modify model characteristics are called zero-cycle commands. You can execute multiple zero-cycle commands without advancing simulator time. Commands that generate bus cycles (like *model_write*) generally take at least one clock cycle to execute.

The *inst_handle* Parameter

Each FlexModel instance in your design needs a unique identifier called an *inst_handle*, which you obtain using the [flex_get_inst_handle](#) command. After you get an *inst_handle*, you use that value in the *inst_handle* parameter of all subsequent FlexModel commands. For more information, refer to [“Setting Up the Model” on page 18](#). Note that you do not use the *inst_handle* parameter in VERA Command Mode. See [“Command Syntax Differences in VERA Command Mode” on page 58](#).

The *req* and *rslt* Command Suffixes

The *req* and *rslt* command suffixes are used with request and result commands, respectively. You combine result commands with corresponding request commands to retrieve data from FlexModels. Request commands direct the model to post the data and result commands retrieve the results. For more information on how to use request and result commands, refer to [“*Pipelining With wait_mode Behavior*” on page 25](#).

Command Result Identifiers

You use command result identifiers with result commands to access data posted by request commands. There are two types of result identifiers: command tags and *addr* parameters. In many cases you use the integer returned in the *status* parameter of the request command as the *cmd_tag* argument of the paired result command. With other result commands you use the *addr* parameter returned by a paired request command to specify the starting address for the data to retrieve. The idea in both cases is to uniquely identify the data you want to retrieve by using values returned by a preceding model request command.

When you use an *addr* parameter with a result command and more than one request command for the same address has posted, the result command returns data for the first request command received by the model before returning data for the second result command received, and so on. For example:

```
read_req(00000000); // request data from address "00000000", and the
                    data is "a"
.... // some other stuff or just time delay, during which time the
data at "00000000" changed to "b"
read_req(00000000); // request data from address "00000000", and the
                    data is "b"
read_rslt(00000000,return_data); // get the data requested back to the
testbench
print(return_data); // the data printed is "a"
read_rslt(00000000,return_data); // get the data requested back to the
testbench
print(return_data); // the data printed is "b"
```

To avoid this behavior, use command tag result identifiers whenever possible. Refer to the command reference sections of the individual model datasheets for more information about the supported model request and result commands and the result identifiers available with each one.

The *wait_mode* Parameter

Many FlexModel commands allow you to specify a *wait_mode* parameter. If you set this parameter to true (FLEX_WAIT_T), the model pauses until the command completes. If you set the *wait_mode* parameter to false (FLEX_WAIT_F), the model proceeds directly to the next command in the queue without waiting for the first command to complete.

The *status* Parameter

All FlexModel commands return a *status* parameter. Depending on the command type, this parameter can convey two different types of information.

- **Type 1:** Commands that do not return results to the control process. For this type, the models return a *status* of 1 if the command completes successfully.
- **Type 2:** Commands that return results to the control process. For this type, the models return a positive integer in the *status* parameter if the command completes successfully. This integer increments by one with each new command of this type so that you can use the *status* value as a tag to uniquely identify the results you want with results commands, as explained in [“Command Result Identifiers” on page 55](#).

For commands of either type that do not complete successfully, FlexModels return a *status* of 0 or a negative integer. Negative integers indicate specific error types that you can look up in [Table 3](#).

Table 3: Status Parameter Error Codes

Error Code	Description
Fatal Errors: Status value range -100 through -199	
-100	An instance of one model was passed to a command for another model type.
-101	A command associated with an uninitialized model type was received.
-102	The model instance name was not mapped to an instance handle (with flex_get_inst_handle).
-103	The system ran out of memory.
-104	An attempt was made to define an instance for a second time.
-105	An attempt was made to access an undefined model instance.
-106	A C program exited with fatal errors.
Internal Errors: Status value range -200 through -299	
Contact Customer Support. See “Getting Help” on page 11	

Table 3: Status Parameter Error Codes (Continued)

Error Code	Description
User Errors: Status value range -300 through -399	
-300	Contact Customer Support. See “Getting Help” on page 11 .
-301	Contact Customer Support. See “Getting Help” on page 11 .
-302	Cannot open file to read.
-303	The flex_clear_queue command was called with an invalid queue-initialize number.
-304	An interrupt command with a priority less than zero was received. Interrupt priorities start at zero.
-305	An attempt was made to access an uninitialized data queue.
-306	An attempt was made to access an uninitialized command queue.
-307	An attempt was made to access an uninitialized active command queue.
-308	An attempt was made to access an uninitialized exception queue.
-309	A flex_synchronize command was received with a NULL <i>sync_tag</i> string.
-310	Two flex_synchronize commands were received—they had the same <i>sync_tags</i> but different <i>sync_totals</i> . See “flex_synchronize” on page 77 for the correct syntax.
-311	A second attempt was made to assign a <i>sync_tag</i> to a model instance that already had one.
-312	A <i>sync_tag</i> associated with too many flex_synchronize commands was received.
-313	The executable file specified by the flex_run_program command could not be found.

Note that FlexModel C functions return a *status* of 1 when they complete successfully and do not return at all on fatal errors.

Command Syntax Differences in VERA Command Mode

In VERA Command Mode, the model functions are called through a model object; therefore, the model name is not a part of the command name in VERA. So a command that would look like this in HDL Command Mode:

```
model_read_req();
```

would look like this in VERA Command Mode:

```
modelObject.read_req();
```

The *modelObject* is an instance of the *ModelFx* class that you create in your VERA testbench.

The FlexModel class encapsulates the *inst_handle* value; therefore, the *inst_handle* argument is not required in FlexModel commands from VERA. So a command that would look like this in HDL Command Mode:

```
model_read_req(inst_handle, address, data, wait_mode, status);
```

would look like this in VERA Command Mode:

```
modelObject.read_req(address, data, wait_mode, status);
```

Global FlexModel Commands

Global FlexModel commands are available to all FlexModels. They either perform supervisory functions (switching command sources, handling interrupts, printing messages) or operate globally on all models (synchronizing models, clearing queues, and enabling tagging). The prefix “flex” is common to all of these commands.

[Table 4](#) lists the global FlexModel commands. Some commands are available only in specific command modes, as shown in the “Command Mode” column. The commands are described in detail in [“Global FlexModel Command Descriptions” on page 60](#).

Table 4: Global FlexModel Command Summary

Command Name	Command Mode	Description
flex_clear_queue	All	Clears the queues for the model.
flex_define_intr_function	C	Defines a C interrupt function for the model.
flex_define_intr_signal	HDL, VERA	Defines the testbench interrupt signal for the model.
flex_get_cmd_status	All	Checks the status of a model command.
flex_get_coupling_mode	C	Checks the coupling mode for the model.
flex_get_inst_handle	All	Gets an <i>inst_handle</i> for the model.
flex_print_msg	All	Prints a message.
flex_run_program	HDL, VERA	Switches control to a compiled C program.
flex_set_coupling_mode	C	Sets the coupling mode for the model
flex_start_program	C	Signals the Command Core that the testbench is done getting model instance handles and is beginning to send model commands.
flex_switch_intr_control	C	Switches model interrupt control to HDL.
flex_synchronize	All	Synchronizes the model with other models in the testbench.
flex_wait	C	Causes the model to wait for a specified number of clock cycles

Global FlexModel Command Descriptions

The following pages describe the global FlexModel commands. Model-specific commands are described in the individual FlexModel datasheets.

flex_clear_queue

Clear the command queue. Used in all command modes.

Syntax

flex_clear_queue (*inst_handle*, *queue_select*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>queue_select</i>	Specify one of the following constants: FLEX_ALL_QUEUES — Clear all queues FLEX_CMD_QUEUE — Clear only the command queue FLEX_RSLT_QUEUE — Clear only the result queue
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information. For more information on command status, see “The status Parameter” on page 56 .

Description

The flex_clear_queue command clears the queue(s) for the specified model instance. It executes immediately and overrides any commands that are in wait mode except in C Command Mode, where there is no concurrency.

Prototypes

C

```
void flex_clear_queue (  
    const int      inst_handle,  
    const int      queue_select,  
    int            *status);
```

VHDL

```
procedure flex_clear_queue (  
    inst_handle      : in  integer;  
    queue_select     : in  integer;  
    status           : out integer );
```

Verilog

```
task flex_clear_queue;  
    input    [31:0]  inst_handle;  
    input    [31:0]  queue_select;  
    output   [31:0]  status;
```

VERA

```
task clear_queue (  
    integer        queue_select,  
    var integer    status);
```

Examples

The following examples clear just the command queue for the model instance specified by the “inst” *inst_handle*.

```
// Verilog Example  
flex_clear_queue(inst, `FLEX_CMD_QUEUE, status);  
  
-- VHDL Example  
flex_clear_queue(inst, FLEX_CMD_QUEUE, status);  
  
/* C Example */  
flex_clear_queue(inst, FLEX_CMD_QUEUE, &status);  
  
// VERA Example  
model_object.clear_queue(FLEX_CMD_QUEUE, status);
```

flex_define_intr_function

Defines a C interrupt function. Used only in C Command Mode.

Syntax

```
flex_define_intr_function (inst_handle, my_function, status);
```

Parameters

<i>inst_handle</i>	The model instance for which interrupts are to be controlled from the C testbench.
<i>my_function</i>	A pointer to the C interrupt function. This function must return void, and does not take any arguments.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully.

Description

The `flex_define_intr_function` command specifies to the Command Core which function to call if an interrupt occurs. This command only works in C Command Mode. For HDL Command Mode, use the equivalent [flex_define_intr_signal](#) command documented on [page 63](#).

Prototype

C

```
void flex_define_intr_function(
    const int      inst_handle,
    FLEX_FUNC      my_function,
    int            *status );
```

Example

```
/* C Example: Function prototype */
void my_intr_handler();
main() {
    int status;
    flex_define_intr_function(id, my_intr_handler, &status);
}
/* Defined interrupt function */
void my_intr_handler() {
    . . . . /* Handler routine commands go HERE */
}
```

flex_define_intr_signal

Defines an interrupt signal in an HDL or VERA testbench. Not used in C Command Mode (instead, see [“flex_switch_intr_control” on page 76](#)).

Syntax

```
flex_define_intr_signal (inst_handle, “sig_name”, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
“ <i>sig_name</i> ”	A name for the interrupt signal you want to define. This is a signal that you define in your testbench.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Description

The flex_define_intr_signal command defines a signal in an HDL or VERA testbench. In VHDL the “*sig_name*” is a signal. In Verilog, it is a register. In both cases, the “*sig_name*” must specify the full path to the signal. FlexModels toggle this signal when they detect interrupts, thus starting the interrupt service routines tied to that signal.

Prototypes

VHDL

```
procedure flex_define_intr_signal (  
    inst_handle      : in integer;  
    sig_name         : in string;  
    status           : out integer );
```

Verilog

```
task flex_define_intr_signal;  
    input    [31:0] inst_handle;  
    input    [8*`FLEX_CHARMAXCNT:1] sig_name;  
    output   [31:0] status;
```

VERA

```
task define_intr_signal (  
    string          sig_name,  
    var integer     status);
```

Examples

-- VHDL Example

```
architecture.....  
    signal int_signal:std_logic;  
    ....  
begin  
    process  
        flex_define_intr_signal(inst,"top/int_signal", status);
```

// Verilog Example

```
module example  
    reg int_signal;  
    ....  
    initial  
        begin  
            flex_define_intr_signal(inst, "top.int_signal", status);
```

// VERA Example

```
model_object.define_intr_signal("top.int_signal", status);
```


flex_get_cmd_status

Checks the status of a command in the model's queue. Used in all command modes.

Syntax

```
flex_get_cmd_status (inst_handle, cmd_tag, valid_f, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>cmd_tag</i>	An integer that identifies the command in the command queue. This is usually the returned <i>status</i> of the command.
<i>valid_f</i>	A boolean returned value (1 = valid, 0 = invalid) that indicates whether the specified <i>cmd_tag</i> represents a valid command in the queue.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Description

Given a model *inst_handle* and *cmd_tag*, the flex_get_cmd_status command returns the *valid_f* true if the specified command is active or pending.

Prototypes

C

```
void flex_get_cmd_status (
    const int      inst_handle,
    const int      cmd_tag,
    int            *valid_f,
    int            *status);
```

VHDL

```
procedure flex_get_cmd_status (
    inst_handle      : in    integer;
    cmd_tag          : in    integer;
    valid_f          : out   boolean;
    status           : out   integer );
```

Verilog

```
task flex_get_cmd_status;
    input    [31:0] inst_handle;
    input    [31:0] cmd_tag;
    output    valid_f;
    output    [31:0] status;
```

VERA

```
task get_cmd_status (
    integer    cmd_tag,
    var integer valid_f,
    var integer status);
```

Examples

The following examples return *valid_f* true because the preceding specified commands are valid.

```
-- VHDL Example
arm7tdmi_read_req(inst, addr1, 0, FLEX_WAIT_F, tag1);
flex_get_cmd_status(inst, tag1, valid_f, status);

// Verilog Example
arm7tdmi_read_req(inst, addr1, 0, `FLEX_WAIT_F, tag1);
flex_get_cmd_status(inst, tag1, valid_f, status);

/* C Example */
arm7tdmi_read_req(inst, addr1, 0, FLEX_WAIT_F, &tag1);
flex_get_cmd_status(inst, tag1, &valid_f, &status);

// VERA Example
model_object.read_req(addr1, 0, `FLEX_WAIT_F, tag1);
model_object.get_cmd_status(tag1, valid_f, status);
```

This last command returns *valid_f* false, because tag2 did not get assigned to any command yet, and thus is not valid.

```
flex_get_cmd_status(inst, tag2, valid_f, status);
```

flex_get_coupling_mode

Checks the coupling mode for a model while in C Command Mode.

Syntax

```
flex_get_coupling_mode (inst_handle, coupling_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>coupling_mode</i>	The command returns a <i>coupling_mode</i> value: FLEX_UNCOUPLED_MODE FLEX_FULLY_COUPLE_MODE
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Description

Given a model *inst_handle*, the flex_get_coupling_mode command returns the *coupling_mode* for the model. FlexModels start up in coupled mode by default.

Prototype

C

```
void flex_get_coupling_mode (  
    const int    inst_handle,  
    const int    coupling_mode,  
    int          *status);
```

Example

The following example returns the *coupling_mode* for the *mpc8260_inst1* model instance.

```
/* C Example */  
flex_get_coupling_mode (mpc8260_inst1, &coupling_mode, &status);
```

flex_get_inst_handle

Returns a unique instance handle for the model. Not used in VERA Command Mode.

Syntax

```
flex_get_inst_handle (InstName / instance / ModelInstName, inst_handle, status);
```

Parameters

InstName (C), *instance* (VHDL), *ModelInstName* (Verilog)

The unique instance name specified as the SWIFT FlexModelID parameter when the model is instantiated.

inst_handle

An integer value used as a unique model instance identifier. This value must be used in all subsequent FlexModel commands for this model instance.

status

A *status* of 1 means the command completed successfully. A *status* less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see [Table 3 on page 56](#)).

Description

The flex_get_inst_handle command returns a unique instance handle for use in all subsequent FlexModel commands. This must be the first command issued for each FlexModel instance in your design. This command can be used in HDL Command Mode or C Command Mode, but not in VERA Command Mode (see [“Command Syntax Differences in VERA Command Mode” on page 58](#)).

In VERA Command Mode, you do not need to use this command because the instance handle is automatically issued when an instance of the model's class is created.

Prototypes

C

```
void flex_get_inst_handle(  
    const char*    InstName,  
    int            *inst_handle,  
    int            *status);
```

VHDL

```
procedure flex_get_inst_handle (
    instance      : in    string;
    inst_handle   : inout integer;
    status        : out   integer );
```

Verilog

```
task flex_get_inst_handle;
    input    [`FLEX_CHARMAXCNT*8:1] ModelInstName;
    output   [31:0] inst_handle;
    output   [31:0] status;
```

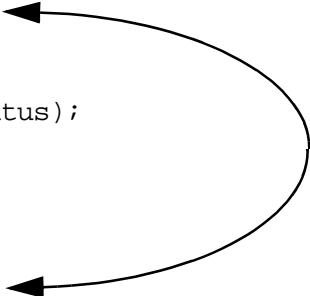
Examples

The following examples return a unique instance handle to the variable “tms_1_handle”:

```
-- VHDL Example
flex_get_inst_handle("tms_1", tms_1_handle, status);
tms320c6201_idle(tms_1_handle, 2, FLEX_WAIT_T, status);

// Verilog Example
flex_get_inst_handle(ModelInstName, tms_1_handle, status);
tms320c6201_idle(tms_1_handle, 2, `FLEX_WAIT_T, status);

/* C Example */
HDL Testbench
--Instantiation of instance "tms_1"
model ul(FlexModelId => "tms_1")
process
begin
    flex_run_program("a.out", status);
end
C Testbench
main() {
    int id, status;
    char *Inst = "tms_1";
    flex_get_inst_handle(Inst, &id, &status);
    flex_start_program(&status);
}
```



Same
Name

flex_print_msg

Prints a message to the screen. Used in all command modes.

Syntax

```
flex_print_msg (inst_handle, "text", status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>"text"</i>	A literal string that specifies the message to be output; must be enclosed in quotation marks.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Description

The flex_print_msg command prints the specified *"text"* to the screen.

Prototypes

C

```
void flex_print_msg (
    const int      inst_handle,
    const char     *text,
    int            *status);
```

VHDL

```
procedure flex_print_msg (
    inst_handle      : in  integer;
    text             : in  string;
    status           : out integer );
```

Verilog

```
task flex_print_msg;
    input    [31:0] inst_handle;
    input    [8*`FLEX_CHARMAXCNT:1] text;
    output   [31:0] status;
```

VERA

```
task print_msg (  
    string      text,  
    var integer status);
```

Examples

The following examples produce output formatted as shown below, where *time* is the current simulation time:

```
time ns:  INSTANCE inst_name    NOTE: This is a test  
  
-- VHDL Example  
flex_print_msg(inst, "This is a test", status);  
  
// Verilog Example  
flex_print_msg(inst, "This is a test", status);  
  
/* C Example */  
flex_print_msg(inst, "This is a test", &status);  
  
// VERA Example  
model_object.print_msg("This is a test", status);
```

flex_run_program

Transfers control to a C program. Used in HDL and VERA command modes.

Syntax

```
flex_run_program ("filename", status);
```

Parameters

<i>"filename"</i>	The <i>"filename"</i> of a compiled C program; must be enclosed in quotation marks.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Description

The flex_run_program command switches control to the *"filename"* compiled C program. The model receives all commands from the C program before any subsequent HDL commands in that VHDL process or Verilog always block.



Note

You cannot have multiple VHDL processes or Verilog always blocks providing commands to the same model instance.

Prototypes

VHDL

```
procedure flex_run_program (
    filename      : in string;
    status        : out integer );
```

Verilog

```
task flex_run_program;
    input    [8*(`FLEX_CHARMAXCNT-2):1] filename;
    output   [31:0] status;
```

VERA

```
task run_program (
    input    filename,
    var integer status);
```


Examples

The following examples all switch control to a compiled C program named myprogramfile.

```
-- VHDL Example
flex_run_program("/proj/asic23/myprogramfile", status);

// Verilog Example
flex_run_program("/proj/asic23/myprogramfile", status);

// VERA Example
model_object.run_program("/proj/asic23/myprogramfile", status);
```

flex_set_coupling_mode

Sets the coupling mode for a model while in C Command Mode.

Syntax

```
flex_set_coupling_mode (inst_handle, coupling_mode, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle returned by the flex_get_inst_handle command.
<i>coupling_mode</i>	Specify the <i>coupling_mode</i> using one of the following two constants: FLEX_UNCOUPLED_MODE—sets mode to uncoupled FLEX_FULLY_COUPLE_MODE—sets mode to coupled
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Description

Given a model *inst_handle*, the flex_set_coupling_mode command sets the *coupling_mode* for the model. FlexModels start up in coupled mode by default.

Prototype

C

```
void flex_set_coupling_mode (  
    const int      inst_handle,  
    const int      coupling_mode,  
    int            *status);
```

Example

The following example sets the *coupling_mode* to uncoupled for the mpc8260_inst1 model instance.

```
/* C Example */  
flex_set_coupling_mode (mpc8260_inst1, FLEX_UNCOUPLED_MODE, &status);
```

flex_start_program

Start a C program for a FlexModel. Used only in C Command Mode.

Syntax

flex_start_program (*status*)

Parameter

<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully.
---------------	---

Description

The flex_start_program command signals to the Command Core that the C testbench has obtained all the model instance handles needed and is ready to send model commands. You must run the flex_get_inst_handle command to retrieve the model instance handle before issuing the flex_run_program command. Also, you cannot send other commands to the model until after you run flex_start_program. In summary, use the commands in this order:

1. flex_get_inst_handle
2. flex_start_program
3. Other FlexModel commands

Prototype

C

```
void flex_start_program(  
    int      *status );
```

Example

```
/* C Example */  
main() {  
    int status;  
    int id;  
    flex_start_program(&status)  
    /* Now you can issue model commands */
```

flex_switch_intr_control

Switches interrupt control to an HDL testbench. Used only in C Command Mode.

Syntax

```
flex_switch_intr_control (inst_handle, status);
```

Parameters

<i>inst_handle</i>	An integer instance handle for the model instance under C control.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully.

Description

The `flex_switch_intr_control` command switches interrupt control for the specified model instance from C Command Mode to HDL Command Mode.

Prototype

C

```
void flex_switch_intr_control(
    const int      inst_handle,
    int            *status);
```

Example

```
/* C Example */
void my_intr_function()
main() {
    int status;
    int id
    char *inst = "1"
    flex_get_inst_handle(Inst, &id, &status);
    flex_start_program(&status);
    flex_define_intr_function(id, my_intr_function, &status);
    . . . .
    /* Now switch interrupt control to HDL*/
    flex_switch_intr_control(id, &status);
```

flex_synchronize

Synchronize the operation of two or more FlexModels. Used in all command modes.

Syntax

flex_synchronize (*inst_handle*, *sync_total*, *sync_tag*, *sync_timeout*, *status*);

Parameters

<i>inst_handle</i>	An integer instance handle returned by the <code>flex_get_inst_handle</code> command.
<i>sync_total</i>	A positive integer that specifies the number of model instances to synchronize with.
<i>sync_tag</i>	A text string that uniquely identifies the synchronization (for example, <code>sync1</code>).
<i>sync_timeout</i>	If the <i>sync_timeout</i> number of clock cycles elapses before the model receives the <i>sync_total</i> number of matching synchronize commands, the command times out.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully. Negative integers provide error code information (see Table 3 on page 56).

Definition

The `flex_synchronize` command suspends command execution for the model instance named in the *inst_handle* parameter until *sync_total* number of synchronize commands with matching *sync_tag* parameters have been executed by other models in the testbench.

Prototypes

C

```
void flex_synchronize (  
    const int      inst_handle,  
    const int      sync_total,  
    const char     *sync_tag,  
    const int      sync_timeout,  
    int            *status );
```

VHDL

```

procedure flex_synchronize (
    inst_handle      : in    integer;
    sync_total       : in    integer;
    sync_tag         : in    string;
    sync_timeout     : in    integer;
    status           : out   integer );

```

Verilog

```

task flex_synchronize;
    input    [31:0] inst_handle;
    input    [31:0] sync_total;
    input    [8*`FLEX_CHARMAXCNT:1] sync_tag;
    input    [31:1] sync_timeout;
    output   [31:0] status;

```

VERA

```

task synchronize (
    integer      sync_total,
    string       sync_tag,
    integer      sync_timeout,
    var integer  status);

```

Examples

In the following example, the `flex_synchronize` in command (4) causes command execution to halt for instance 1. Command execution resumes when a matching *sync_label* “sync1” has been identified. In this case command (5) carries the identical *sync_label* “sync1”. Command (6) starts after commands (1) (2) and (3) have been completed. (FLEX_WAIT_F is a predefined constant. For more information, refer to [“The wait_mode Parameter” on page 56.](#))

```

-- VHDL Example
(1) arm7tdmi_read_req(inst1, config_read,X"00000004",1, X"0", X"00000000", 0,
    FLEX_FALSE, FLEX_WAIT_F, status);
(2) arm7tdmi_read_req(inst2, config_read,X"00000004",1, X"0", X"00000000", 0,
    FLEX_FALSE, FLEX_WAIT_F, status);
(3) arm7tdmi_read_req(inst2, config_read,X"00000004",1, X"0", X"00000000", 0,
    FLEX_FALSE, FLEX_WAIT_F, status);
    -- Synchronize instance 1 with 2 instances with the sync label "sync1"
    -- identical to wait_on ("sync1");
(4) flex_synchronize (inst1, 2, "sync1", 0, status);
    -- Synchronize instance 2 with 2 instances with the sync label "sync1"
    -- identical to trigger ("sync1");
(5) flex_synchronize (inst2,2,"sync1", 0, status);

```

flex_wait

Temporarily halts command execution in a C testbench.

Syntax

```
flex_wait (clock_cycles, status);
```

Parameters

<i>clock_cycles</i>	The number of clock periods to halt the C testbench.
<i>status</i>	A <i>status</i> of 1 means the command completed successfully. A <i>status</i> less than or equal to 0 means the command did not complete successfully.

Description

The `flex_wait` command halts execution in the C testbench for the specified number of *clock_cycles*. This means that the next command in the queue will only be seen by the model after the specified number of clock cycles have elapsed.

Prototype

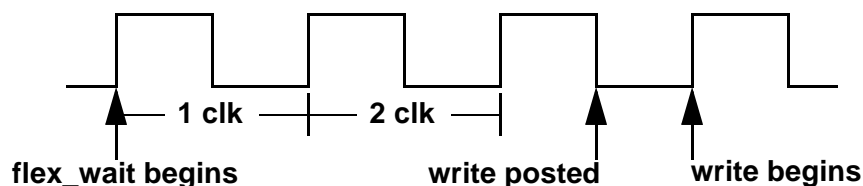
C

```
void flex_wait (
    const int    clock_cycles,
    int          *status );
```

Example

```
/* C Example */
main() {
    int nstatus
    flex_wait(2, &nstatus); /* Wait for 2 clock cycles */
    model_write(id, addr, data, &nstatus); /* Seen 3 clock cycles later */
}
```

The following diagram shows the timing cycles for this example:



5

FlexModel C Testbench Interface

Introduction

This chapter explains how to define and manipulate FLEX_VEC vectors using the FlexModel C functions and operators (provided in ANSI-compliant include files). This information is organized in the following sections:

- [“Creating FLEX_VEC Vectors” on page 82](#)
- [“FLEX_VEC Lexical Rules” on page 83](#)
- [“FLEX_VEC Error Handling” on page 84](#)
- [“FLEX_VEC Command Descriptions” on page 85](#)
- [“C Testbench Example” on page 95](#)

What Are FLEX_VEC Vectors?

Before you can use the C versions of the model-specific commands documented in the individual FlexModel datasheets, you must define the required variables or vectors using the FLEX_DEFINE command described in this chapter. For example, *model_read* commands typically require you to specify an address variable (*addr* or something like that). You can use the FLEX_DEFINE function to create the data structure in C for that *addr* variable and then issue the model-specific FlexModel command to exercise the model. Data structures created with FLEX_DEFINE are called FLEX_VEC vectors.

This definition process is necessary because C does not provide variables that are handy for manipulating vectors such as the 32-bit data or address buses needed to work with processor models, for example. Also, although C does provide many operators for manipulating integers and strings, those operators do not work with the FLEX_VEC vectors you create for use with FlexModel commands. So, FlexModels come with comparable FlexModel C operators that work with the FLEX_VEC vectors you create.

Creating FLEX_VEC Vectors

You create FLEX_VEC vectors using either the FLEX_DEFINE command or the FLEX_VEC_SIZEOF command. Use the FLEX_DEFINE command for vectors that only need to be used in the local scope of the function. If you need to create FLEX_VEC vectors dynamically with a global scope use the FLEX_VEC_SIZEOF command.

FLEX_DEFINE

The FLEX_DEFINE command creates a FLEX_VEC vector named *vecName* that is *vecSize* bits wide, with an initial value of *initVal*. You must specify a vector string literal or the FLEX_NULL_VEC macro in the *initVal* argument. Use FLEX_DEFINE at the top of the current scope before any functions are called.

Syntax

```
FLEX_DEFINE (vecName, vecSize, initVal);
```

Example

The following example creates a FLEX_VEC called *addr* with space for 64 bits.

```
FLEX_DEFINE (addr, 64, "haaaabbbbccccdddd");
```

FLEX_VEC_SIZEOF

To dynamically create FLEX_VEC vectors, use the FLEX_VEC_SIZEOF macro. You can calculate the *bitcnt* on the fly based on other operations in your C testbench. The example that follows contains the function declaration and assignment in one line of code, which creates a FLEX_VEC with a local scope. If you want the FLEX_VEC to have a global scope, put your function declaration outside of the subroutine where you make the variable assignment.

Syntax

```
FLEX_VEC_SIZEOF (int bitcnt);
```

Example

```
FLEX_VEC dynVec64 = (FLEX_VEC)malloc(FLEX_VEC_SIZEOF(64));
```

FLEX_VEC Lexical Rules

The following lexical and semantic rules apply to FLEX_VEC vectors:

- Vector values must be either string literals or objects of type FLEX_VEC created with FLEX_DEFINE.
- Values are truncated on the left side to fit the size of the receiving variable. For example, if you assign “haf” to a 4-bit wide vector the result is “hf”.
- VHDL 9-state values are mapped to 4-state values as shown in [Table 5](#). Therefore, FLEX_VEC vectors do not represent signal strength levels.

Table 5: VHDL 9-State to 4-State Conversion

9-state	4-state
(0, L)	0
(1, H)	1
(U, X, W, -)	X
(Z)	Z

- For integer variables., use the FLEX_INT 32-bit unsigned data type.
- All functions other than the comparison functions have a return type of void.

Vector Strings

Vector strings can be in hexadecimal or binary format:

```
"h[0-9a-fA-FxXzZ] +" /* hexadecimal */
"b[01hHlLuUwWxXzZ-] +" /* binary */
```

where []+ means one or more occurrences of the characters within the brackets. Illegal characters are silently converted to Xs. Here are some examples:

```
"h01234"      /* Hexadecimal literal */
"b011011"     /* Binary literal */
"01234"       /* Illegal vector literal. Missing prefix 'h' */
"b0JM11011"   /* Illegal char in binary vector -> "b0xx11011" */
"b01LHUXW-Z"  /* 9-state to 4-state -> "b0101xxxxz" */
```

Assigning Literals to FLEX_VEC Constants

You can assign string literals to FLEX_VEC constants, as shown in the following examples.

```
const FLEX_VEC    addrIncr;  
    /* Assign a vector value */  
    addrIncr = "h4";  
    /* Assign a different vector value */  
    addrIncr = "hffffeeee";
```

If you assign a literal to a FLEX_VEC vector instead of a FLEX_VEC constant you lose the memory allocation for the vector. To assign a literal to a FLEX_VEC vector created by the FLEX_DEFINE command, use the flex_assign operators documented on [page 85](#).

Note that arguments of type const FLEX_VEC do not have any allocated storage, since FLEX_DEFINE has not been used. Therefore, they can only be used as input values, not for result values.

FLEX_VEC Error Handling

The FLEX_VEC commands documented in [FLEX_VEC Command Descriptions](#) do not return error status. Instead, they increment internal error, warning, and note message counters. To retrieve the current counts, use the flex_errors(), flex_warnings(), and flex_notes() commands.

Using incorrect command syntax or violating any of the [FLEX_VEC Lexical Rules](#) will result in an error. Most error types generate informative error messages on your screen.

You can check the error counts as often as you want, but checking error status only at critical points in your testbench will result in a more readable coding style. You may want to run the following commands at the end of your C testbench to ensure that the program executed as expected.

- flex_errors ()
- flex_warnings ()
- flex_notes ()

The following example shows how to use the flex_fprintf command to print the values of the three internal counters:

```
flex_fprintf(stderr, "Status: %d error(s), %d warning(s), %d note(s)\n",  
             flex_errors(), flex_warnings(), flex_notes());
```

FLEX_VEC Command Descriptions

Following are descriptions of the FLEX_VEC commands.

flex_assign

The flex_assign command assigns the *vec2* value to *vec1*. For example:

```
void flex_assign(FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 = vec2 */
```

flex_assign_int

The flex_assign_int command assigns the *i* integer value to *vec1*. For example:

```
void flex_assign_int(FLEX_VEC vec1, FLEX_INT i) /* vec1 = i; */
```

flex_assign_int_array

The flex_assign_int_array command assigns an integer array to *vec1* using *count* number of integers from *intArray[]*. The 0th element of *intArray[]* is treated as the left-most number and the (*count*-1)th element is treated as the right-most number.

Syntax

```
void flex_assign_int_array(FLEX_VEC vec1, unsigned int count, FLEX_INT
intArray[])
```

For example:

```
FLEX_INT intArray[] = {0xffffeeee, 0xddddcccc, 0xbbbbaaaa, 0x99998888};
FLEX_DEFINE(bigBus, 128, FLEX_NULL_VEC);
FLEX_DEFINE(halfAsBigBus, 64, FLEX_NULL_VEC);
/* Assign the whole value from the intArray to bigBus */
flex_assign_int_array(bigBus, 4, intArray);
/* bigBus == "hffffeeeedddccccbbbbbbaaaa99998888" */
/* Try to assign the whole value from the intArray to halfAsBigBus */
flex_assign_int_array(halfAsBigBus, 4, intArray);
/* halfAsBigBus == "hbbbbaaaa99998888", truncated from left */
/* Assign the first two elements from the intArray to halfAsBigBus */
flex_assign_int_array(halfAsBigBus, 2, intArray);
/* halfAsBigBus == "hffffeeeedddcccc", takes the first two elements*/
```

flex_assign_int_list

The flex_assign_int_list command assigns an integer list to *vec1* using *count* number of FLEX_INT values from *lhInt* to *rhInt*. For example:

```
void flex_assign_int_list(FLEX_VEC vec1, unsigned int count, FLEX_INT
lhInt, FLEX_INT rhInt);
```

flex_incr

The `flex_incr` command increments the *incrVec* vector and puts the result in *result*. For example:

```
/* vec += incrVec */  
void flex_incr(FLEX_VEC result, const FLEX_VEC incrVec)
```

flex_decr

The `flex_decr` command decrements the *decrVec* vector and puts the result in *result*. For example:

```
/* vec -= decrVec */  
void flex_decr(FLEX_VEC result, const FLEX_VEC decrVec)
```

flex_add

The `flex_add` command adds *vec1* and *vec2* and puts the result in *result*. For example:

```
/* result = vec1 + vec2 */  
void flex_add (FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC  
vec2)
```

flex_sub

The `flex_sub` command subtracts *vec2* from *vec1* and puts the result in *result*. For example:

```
/* result = vec1 - vec2 */  
void flex_sub (FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC  
vec2)
```

flex_eq

The `flex_eq` command returns true if *vec1* is equal to *vec2*. For example:

```
int flex_eq (const FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 == vec2  
*/
```

flex_ne

The `flex_ne` command returns true if *vec1* is not equal to *vec2*. For example:

```
int flex_ne (const FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 != vec2  
*/
```

flex_lt

The `flex_lt` command returns true if *vec1* is less than *vec2*. For example:

```
int flex_lt (const FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 < vec2 */
```

flex_lte

The `flex_lte` command returns true if *vec1* is less than or equal to *vec2*. For example:

```
int flex_lte(const FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 <= vec2 */
```

flex_gt

The `flex_gt` command returns true if *vec1* is greater than *vec2*. For example:

```
int flex_gt (const FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 > vec2 */
```

flex_gte

The `flex_gte` command returns true if *vec1* is greater than or equal to *vec2*. For example:

```
int flex_gte(const FLEX_VEC vec1, const FLEX_VEC vec2) /* vec1 >= vec2 */
```

flex_slice_le

The `flex_slice_le` command copies a bit slice from the *fromVec* vector to the *result* vector. The “le” stands for little-endian—this operator copies from the 0th bit in the *fromVec* vector, starting with the right-most bit. Note that truncation, if any, still occurs on the left side. If you specify a *lhIdx* less than the *rhIdx*, the bits are reversed in the *result* vector.

Syntax

```
void flex_slice_le(FLEX_VEC result, const FLEX_VEC fromVec, unsigned int lhIdx, unsigned int rhIdx)
```

For example:

```
FLEX_DEFINE(data8, 8, "h0");
/* Little-endian */
void flex_slice_le(data8, "b0110100100010111", 11, 4);
/* no bit reversal, data8 == "b10010001" */
void flex_slice_le(data8, "b0110100100010111", 4, 11);
/* bit reversal, data8 == "b10001001" */
```

flex_slice_be

The `flex_slice_be` command copies a bit slice from the *fromVec* vector to the *result* vector. The “be” stands for big-endian—this operator copies from the 0th bit in the *fromVec* vector, starting with the left-most bit. Truncation, if any, occurs on the left side. If you specify *lhIdx* greater than *rhIdx*, the bits are reversed in the *result* vector. Here's the syntax:

```
void flex_slice_be(FLEX_VEC result, const FLEX_VEC fromVec, unsigned int
lhIdx, unsigned int rhIdx)
```

For example:

```
FLEX_DEFINE(data8, 8, "h0");
/* Big-endian */
void flex_slice_be(data8, "b0110100100010111", 4, 11);
/* no bit reversal, data8 == "b10010001" */
void flex_slice_be(data8, "b0110100100010111", 11, 4);
/* bit reversal data8 == "b10001001" */
```

flex_slice_le_offset

The `flex_slice_le_offset` command does a little-endian copy of a bit slice from *fromVec* to *result* starting with the specified offset of *resultOffsetIdx* bits in the *result* vector. Truncation, if any, occurs on the left side. If you specify a *lhIdx* less than the *rhIdx*, the bits are reversed in the *result* vector. Here's the syntax:

```
void flex_slice_le_offset(FLEX_VEC result, unsigned int resultOffsetIdx,
const FLEX_VEC fromVec, unsigned int lhIdx, unsigned int rhIdx)
```

For example:

```
FLEX_DEFINE(rs16, 16, "b1110111111110111");
/* Little-endian */
flex_slice_le_offset(rs16, 4, "b0110100100010111", 11, 4);
/* no bit reversal, rs16 == "b1110100100010111", middle 8-bits get
changed the others unchanged */
flex_assign(rs16, "b1110111111110111"); /* Reinitialize */
flex_slice_le_offset(rs16, "b0110100100010111", 4, 11);
/* bit reversal, rs16 == "b1110100010010111" , middle 8-bits get
changed the others unchanged */
```


flex_slice_be_offset

The `flex_slice_be_offset` operator does a big-endian copy of a bit slice from *fromVec* to *result* starting with the specified offset of *resultOffsetIdx* bit in the *result* vector. Truncation, if any, occurs on the left side. If you specify *lhIdx* greater than *rhIdx*, the bits are reversed in the *result* vector. Here's the syntax:

```
void flex_slice_be_offset(FLEX_VEC result, unsigned int resultOffsetIdx,
    const FLEX_VEC fromVec, unsigned int lhIdx, unsigned int rhIdx)
```

For example:

```
FLEX_DEFINE(rslt16, 16, "b1110111111110111");
/* Big-endian */
flex_assign(rslt16, "b1110111111110111"); /* Reinitialize */
flex_slice_be_offset(rslt16, "b0110100100010111", 4, 11);
/* no bit reversal, rslt16 == "b1110100100010111" */
flex_assign(rslt16, "b1110111111110111"); /* Reinitialize */
flex_slice_be_offset(rslt16, "b0110100100010111", 11, 4);
/* bit reversal, rslt16 == "b1110100010010111" */
```

flex_rshift

The `flex_rshift` command shifts the *vec* vector *shiftCnt* bits to the right and puts the result in *result*. Truncation, if any, is determined by the length of the *result* vector. Empty bit positions are set to zeros. Here is the syntax:

```
/* result = vec >> shiftCnt */
void flex_rshift(FLEX_VEC result, const FLEX_VEC vec, unsigned int
    shiftCnt)
```

For example:

```
FLEX_DEFINE(rslt8, 8, "h0");
flex_lshift(rslt8, "hf", 4);
/* equivalent C: rslt8 = 0xf << 4 rslt8 == "hf0" */
```

flex_lshift

The `flex_lshift` command shifts the *vec* vector *shiftCnt* bits to the left and puts the result in *result*. Truncation, if any, is determined by the length of the *result* vector. Empty bit positions are set to zeros. Here is the syntax:

```
/* result = vec << shiftCnt */
void flex_lshift(FLEX_VEC result, const FLEX_VEC vec, unsigned int
    shiftCnt)
```

For example:

```
FLEX_DEFINE(rslt8, 8, "h0");
```

```
flex_lshift(rslt8, "hf", 4);
/* equivalent C: rslt8 = 0xf << 4 rslt8 == "hf0" */
```

flex_rrot

The `flex_rrot` command rotates the *vec* vector *shiftCnt* bits to the right and puts the result in *result*. The rotation point is determined by the size of the *result* vector. Here is the syntax:.

```
/* result = right rotate vec by shiftCnt */
void flex_rrot (FLEX_VEC result, const FLEX_VEC vec, unsigned int
shiftCnt)
```

For example:

```
FLEX_DEFINE(rslt8, 8, "h0");
flex_rrot(rslt8, rslt8, 5);
/* left rotate a rslt8 by 5-bits rslt8 == "b00010110" rslt8 == "h07" */
```

flex_lrot

The `flex_lrot` command rotates the *vec* vector *shiftCnt* bits to the left and puts the result in *result*. The rotation point is determined by the size of the *result* vector. Here is the syntax:.

```
/* result = left rotate vec by shiftCnt */
void flex_lrot (FLEX_VEC result, const FLEX_VEC vec, unsigned int
shiftCnt)
```

For example:

```
FLEX_DEFINE(rslt8, 8, "h0");
flex_lrot(rslt8, "b101100", 4);
/* left rotate a 6-bit vector into rslt8 rslt8 == "b11000010" */
```

flex_not

The `flex_not` command does a bitwise not operation on *vec* and puts the result in *result*. Here is the syntax:

```
/* result = ~vec */
void flex_not(FLEX_VEC result, const FLEX_VEC vec)
```

flex_or

The `flex_or` command does a bitwise or operation on *vec1* and *vec2* and puts the result in *result*. Here is the syntax:

```
/* result = vec1 | vec2 */
void flex_or(FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC vec2)
```

flex_and

The `flex_and` command does a bitwise and operation on *vec1* and *vec2* and puts the result in *result*. Here is the syntax:

```
/* result = vec1 & vec2 */
void flex_and (FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC
vec2)
```

flex_nor

The `flex_nor` command does a bitwise nor operation on *vec1* and *vec2* and puts the result in *result*. Here is the syntax:

```
/* result = ~(vec1 | vec2) */
void flex_nor (FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC
vec2)
```

flex_nand

The `flex_nand` command does a bitwise nand operation on *vec1* and *vec2* and puts the result in *result*. Here is the syntax:

```
/* result = ~(vec1 & vec2) */
void flex_nand(FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC
vec2)
```

flex_xor

The `flex_xor` command does a bitwise xor operation on *vec1* and *vec2* and puts the result in *result*. Here is the syntax:

```
/* result = vec1 ^ vec2 */
void flex_xor (FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC
vec2)
```

flex_xnor

The `flex_xnor` command does a bitwise xnor operation on *vec1* and *vec2* and puts the result in *result*. Here is the syntax:

```
/* result = ~(vec1 ^ vec2) */
void flex_xnor(FLEX_VEC result, const FLEX_VEC vec1, const FLEX_VEC
vec2)
```

flex_to_int

The `flex_to_int` command extracts the right-most 32-bits from *vec* and puts them in the `FLEX_INT` pointed to by *i*. Here is the syntax:

```
void flex_to_int(const FLEX_VEC vec, FLEX_INT* i)
```

For example:

```
FLEX_INT myInt = 0;
int i;
FLEX_DEFINE(data128, 128, "h0");

flex_assign_int_list(data128, 4, 0x8889999, 0xaaabbbb, 0xccddd,
                      0xeeffff);

/* data128 == "h088899990aaabbbb00ccddd0eeffff" */
/* Read the rightmost int from data128. Extracts 0x0eefff */
/* into myInt with a warning about the fact that data128    */
/* wider than a single FLEX_INT */
flex_to_int(data128, &myInt);
/* myInt == 0x0eefff */
```

flex_to_int_array

The `flex_to_int_array` command extracts *count* number of 32-bit integers from *vec* and puts them in the *ia[]* array. If *count* is 0 the entire contents of *vec* are extracted. The *int** pointed to by *count* is set to the number of integers extracted. The right-most 32 bits in *vec* are put in the last array element and the left-most bits are placed in the 0th array element. Make sure that the receiving array is large enough to hold all the integers in *vec*. If *count* is higher than the number of integers in *vec*, its value is changed to the actual number of integers extracted. Here is the syntax:

```
void flex_to_int_array(const FLEX_VEC vec, unsigned int* count, FLEX_INT
ia[])
```

For example:

```
FLEX_INT myInt = 0;
unsigned int count;
int i;
FLEX_INT ia[4] = { 0, 0, 0, 0 };
FLEX_INT i1, i2, i3, i4;
FLEX_DEFINE(data128, 128, "h0");

flex_assign_int_list(data128, 4, 0x8889999, 0xaaabbbb, 0xccddd,
                      0xeeffff);

/** Using flex_to_int_array */
count = 0;
flex_to_int_array(data128, &count, ia);
/* ia[0] == 0x08889999 */
```

```

/* ia[2] == 0x0aaabbbb */
/* ia[3] == 0x00cccd dd */
/* ia[4] == 0x0eeeffff */
/* count == 4 */

/* Reset ia */
ia[0] = ia[1] = ia[2] = ia[3] = 0;
count = 8;
flex_to_int_array(data128, &count, ia);
/* Issues a warning about only reading 4 FLEX_INTs */
/* ia[0] == 0x08889999 */
/* ia[2] == 0x0aaabbbb */
/* ia[3] == 0x00cccd dd */
/* ia[4] == 0x0eeeffff */
/* count == 4 */

/* Reset ia */
ia[0] = ia[1] = ia[2] = ia[3] = 0;
count = 2;
flex_to_int_array(data128, &count, ia);
/* Issues a warning about only reading the 2 rightmost FLEX_INTs */
/* while the actual vector is 4 FLEX_INTs wide */
/* ia[0] == 0x00cccd dd */
/* ia[1] == 0x0eeeffff */
/* ia[2] == 0 */
/* ia[3] == 0 */
/* count == 2 */

```

flex_to_int_list

The `flex_to_int_list` command extracts *count* number of 32-bit integers from *vec* and puts them into a list with the right-most bits going to *lhInt* and the left-most bits to *rhInt*.

```

void flex_to_int_list (const FLEX_VEC vec, unsigned int* count,
    FLEX_INT* lhInt, ... , FLEX_INT* rhInt);

```

For example:

```

FLEX_INT myInt = 0;
unsigned int count;
int i;
FLEX_INT ia[4] = { 0, 0, 0, 0 };
FLEX_INT i1, i2, i3, i4;
FLEX_DEFINE(data128, 128, "h0");

flex_assign_int_list(data128, 4, 0x8889999, 0xaaabbbb, 0xcccd dd,
    0xeeeffff);

/** Using flex_to_int_list */
count = 0;
flex_to_int_list(data128, &count, &i1, &i2, &i3, &i4);

```

```

/* i0 == 0x08889999 */
/* i2 == 0x0aaabbbb */
/* i3 == 0x00ccddd */
/* i4 == 0x0eeeffff */
/* count == 4 */

/* Reset FLEX_INTs */
i0 = i1 = i2 = i3 = 0;
count = 8;
flex_to_int_list(data128, &count, &i1, &i2, &i3, &i4);
/* Issues a warning about only reading 4 FLEX_INTs */
/* i0 == 0x08889999 */
/* i2 == 0x0aaabbbb */
/* i3 == 0x00ccddd */
/* i4 == 0x0eeeffff */
/* count == 4 */

/* Reset FLEX_INTs */
i0 = i1 = i2 = i3 = 0;
count = 2;
flex_to_int_list(data128, &count, &i1, &i2, &i3, &i4);
/* Issues a warning about only reading the 2 rightmost FLEX_INTs */
/* while the actual vector is 4 FLEX_INTs wide */
/* i0 == 0x00ccddd */
/* i1 == 0x0eeeffff */
/* i2 == 0 */
/* i3 == 0 */
/** count == 2 */

```

flex_iprintf

The `flex_iprintf` command works just like the ANSI C `printf` utility. You can use `flex_iprintf` to print a string to the simulator transcript. For example:

```
void flex_iprintf(int instHandle, const char* formatStr, ...);
```

The *instHandle* must be a valid model instance handle obtained with the `flex_get_inst_handle` command. The maximum string length is 255 characters.

flex_fprintf

The `flex_fprintf` command works just like the ANSI C `fprintf` utility. You can use `flex_fprintf` to print a string to a file. For example:

```
void flex_fprintf(FILE* fp, const char* formatStr, ...);
```

In this example, *fp* must point to a `FILE*` open for output. The maximum string length is 255 characters.

flex_sprintf

The `flex_sprintf` command works just like the ANSI C `sprintf` utility. You can use `flex_sprintf` to print a string to a buffer. For example:

```
void flex_sprintf(char* buf, const char* formatStr, ...);
```

In this example, *buf* must be a character array large enough to hold the resulting string. The maximum string length is 255 characters.

The %H and %B Conversions

These print functions work just like the ANSI C `printf` utility. In addition, they support `%H` and a `%B` formatting conversions that you can use to print a `FLEX_VEC` vectors or `const FLEX_VEC` literals. These conversions support the same formatting features as the C “`%s`” conversion. The `%H` and `%B` conversions print vectors without the “`h`” or a “`b`” prefixes. For example, you could print the contents of different variables to standard error as follows:

```
flex_fprintf(stderr, "Extracted data128(h%H) into \n i1(%#x), i2(%#x),\n i3(%#x), i4(%#x)\n", data128, i1, i2, i3, i4);
```

This produces output that looks like the following:

```
"Extracted data128(h0x088899990aaabbbb00cccd00eeefffff) into\n i1(0x8889999), i2(0xaaabbbb), i3(0xcccd00), i4(0xeeefffff)"
```

C Testbench Example

The following C testbench example illustrates how to use the `FLEX_VEC` vectors described in this chapter to set up and process interrupts, and to perform a variety of general FlexModel functions.

```
#include "model_pkg.h"
#include "flexmodel_pkg.h"
#define MODEL_ADDRBUS_WIDTH 32
#define MODEL_DATABUS_WIDTH 32

/* Interrupt Function */
void my_intr_function();

/* Global Id define, so that it is visible in the intr function */
int nId;

void
main()
{
    int          nStatus, i;
    int          tag1, tag2;
```

```

char          *sInstName = "1";

/* Define four FLEX_VEC type arrays and initialize them with
 * a NULL vector, these vectors have actual storage and will
 * be used to get the returned results from result commands */
FLEX_DEFINE ( ret_data, MODEL_DATABUS_WIDTH, FLEX_NULL_VEC);
FLEX_DEFINE ( act_data, MODEL_DATABUS_WIDTH, FLEX_NULL_VEC);
FLEX_DEFINE ( ADDRESS,  MODEL_ADDRBUS_WIDTH, FLEX_NULL_VEC);
FLEX_DEFINE ( DATA,    MODEL_DATABUS_WIDTH, FLEX_NULL_VEC);

/* Defining some FLEX_VECs. One very important point to note
 * here is that since a FLEX_DEFINE has not been used,
 * there is no actual storage for these vectors and they can
 * ONLY be used as input values and not for result values. */
const FLEX_VEC      BADADDR;
const FLEX_VEC      ADDR_INCR;
const FLEX_VEC      DATA_INCR;

/* Get the instance handle */
flex_get_inst_handle( sInstName, &nId, &nStatus );

/* Issue a start program, indicating end of initialization */
flex_start_program(&nStatus);

/*****
    End of Initialization, Now commands can be sent
    *****/
/* Define interrupt function with the command core, For more
 * information on this refer to the section on Interrupts */
flex_define_intr_function(nId, my_intr_function, &nStatus);

/* Using flex_fprintf to print a debug message */
flex_fprintf(stderr, "Beginning my C Command Stream\n");

/*****
    Test 1 : Do a read, and verify the results.
    Desc    : Demonstrates passing of addresses/data to commands
    *****/
/* Issue a model_read and pass address directly to command */
model_read_req(nId, "b000011111111111111111111111111110000",
FLEX_WAIT_T, &nStatus);
/* Read the results, ret_data is the array we defined earlier
 * using FLEX_DEFINE. ( Note : This time pass in a hex address ) */
model_read_rslt(nId, "h0ffffff0", 0, ret_data, &nStatus);
/* Use flex_eq to compare the results */
if ( ! flex_eq ( "b10101010101010101010101010101010", ret_data ) )
    flex_fprintf(stderr, "Test 1 Failure : Mismatch Found\n");

/*****

```



```

Test 2 : Do a read, and verify the results.
Desc   : Demonstrates storing addresses as vectors and then
        passing these vectors into commands.
*****/
/* Issue a model_read, using flex_assign to store the address
 * in ADDRESS, and then pass this ADDRESS to the read command */
flex_assign(ADDRESS, "b0000111111111111111111111111111110000");
model_read_req(nId, ADDRESS, FLEX_WAIT_T, &nStatus);
/* Read the results, ret_data is the array we defined earlier
   using FLEX_DEFINE, use the same ADDRESS array defined earlier */
model_read_rslt(nId, ADDRESS, 0, ret_data, &nStatus);
/* Use flex_assign to store the result in the array we defined
 * earlier using FLEX_DEFINE (Note : Passing a binary address) */
flex_assign( act_data, "b10101010101010101010101010101010");
/* Use flex_eq to compare the results */
if ( ! flex_eq ( act_data, ret_data ) )
    flex_fprintf(stderr, "Test 2 Failure : Mismatch Found\n");

/*****
Test 3 : Perform multiple Writes, looping through the address
Desc   : Demonstrates using the vector operations provided to
        loop, compare e.t.c The while loop below behaves as follows
        (i) It loops as long as data is less than a certain data
        (ii) It breaks out of loop if address exceeded value
        (iii) If address is equal to a value it skips that address
        (iv) Otherwise it does a write, increments the address and data.
*****/
/* Setup the start address, data, bad address and increments */
flex_assign(ADDRESS, "h0000ff00");
flex_assign(DATA, "h00000000");
BADADDR    = "h0000ff40";
ADDR_INCR  = "h4"; /* Increment address = 4 bytes */
DATA_INCR  = "b1"; /* Increment data = 1 */

while ( flex_lte(DATA, "h0000ffff") ) {
    /* Check if we have exceeded the address space */
    if ( flex_gte ( ADDRESS, "h0000ffff" ) )
        break;
    /* Check if address is same as the address we wish to avoid */
    if ( flex_eq(ADDRESS, BADADDR ) )
        continue;
    /* Else do a write */
    model_write(nId, ADDRESS, DATA, &nStatus);
    /* Increment the address and data */
    flex_incr(ADDRESS, ADDR_INCR);
    flex_incr(DATA, DATA_INCR);
}

/*****

```

```

    Test 4 : Wait for 5 Clks to expire and then synchronize
            with the HDL testbench.
    Desc   : Demonstrates using flex_wait and synchronize
    *****/

/* Stop sending commands for 5 clks */
flex_wait(5, &nStatus);

/* Synchronize this instance with another instance which is
 * being controlled from HDL, both instances are synchronizing
 * on the tag "SYN_2" and we are going to wait for 12 clks for
 * the synchronize to complete */
flex_synchronize(nId, 2, "SYN_2", 12, &nStatus);

/*****
    Test 5 : Stop the C Testbench as we are expecting interrupts
            to occur, so we need to keep the C Testbench running
            and then switch out of C interrupt mode.
    Desc   : Demonstrates using flex_wait and
flex_switch_intr_control.
    *****/

/* Call flex_wait and tell it to pause for 50 clock cycles,
 * We expect all interrupts to be over by this time */
flex_wait(50, &nStatus);

/* Indicate that now interrupts for instance with id = nId needs
 * to be controlled from HDL.
 * NOTE : This automatically happens once the testbench exits. */
flex_switch_intr_control(nId, &nStatus);

/*****
    Test 6 : Use the slice operations to get parts of a vector.
    Desc   : Demonstrates use of the slice operations.
            1) The following loop reads some data from memory
               ( read_req and read_rslt )
            2) Uses the last 8 bits of this data as the
               increment address.
    *****/
flex_assign(ADDRESS, "hffff0000");
for ( i = 0; i < 16; i++ ) {
    /* A temporary FLEX_VEC with storage */
    FLEX_DEFINE(SMALL_ADDRESS, 8, FLEX_NULL_VEC);

    model_read_req(nId, ADDRESS, FLEX_WAIT_F, &nStatus);
    model_read_rslt(nId, ADDRESS, ret_data, &nStatus);

    /* Get bits 24 to 31 ( last 8 ) from ret_data and
     * save them in SMALL_ADDRESS */

```

```

        flex_slice_le(SMALL_ADDRESS, ret_data, 24, 31);

        flex_incr(ADDRESS, SMALL_ADDRESS);
    }

    /* DONE Exit the C Testbench */
    exit(0);
}

/*****
                                INTERRUPT HANDLER
*****/
void
my_intr_function()
{
    int nValid, nPriority, nStatus;
    /* Id used here is the global variable which was
     * assigned when we obtained the instance handle */
    model_get_intr_priority(nId, &nValid, &nPriority, &nStatus);

    /* Use flex_fprintf to print the priority */
    flex_fprintf(stderr, "Priority = %D\n", nPriority);

    switch (nPriority) {
        case 1:
            model_begin_intr(nId,nPriority,&status)
            /* Place commands HERE for priority 1. Commands must be placed
             between begin and end intr commands.*/
            model_end_intr(nId,nPriority,&status);
        case 2:
            model_begin_intr(nId,nPriority,&status)
            /* Place commands HERE for priority 2. Commands must be placed
             between begin and end intr commands.*/
            model_end_intr(nId,nPriority,&status);
        default
            printf("Unknown priority\n");
    }
}

```

A

Reporting Problems

Introduction

This chapter explains how to run diagnostics, create FlexModel log files, and send debug information to Customer Support, in the following major sections:

- “Model Versions and History” on page 101
- “Running FlexModel Diagnostics” on page 102
- “Creating FlexModel Log Files” on page 102
- “Sending the Log Files to Customer Support” on page 105

For FlexModels that end with an “_fz” extension, refer to the *SmartModel Library User's Manual* for the applicable model logging procedures. The logging mechanism described in this chapter applies only to models that end with an “_fx” extension.

Model Versions and History

If you believe a FlexModel is not working correctly, first verify the version number of the model you are working with by using the Browser tool (\$LMC_HOME/bin/sl_browser) to access the model datasheet. The History and Version Addendum located at the back of all FlexModel datasheets lists the model's MDL version number. You can then compare reported fixes for subsequent versions of that model by reading the model history section in the latest datasheet. The latest FlexModel datasheets are available via the Model Directory on the Web:

<http://www.synopsys.com/products/lm/modelDir.html>

For more information on model history, refer to the *SmartModel Library User's Manual*.

You can contact Customer Support to request the latest version of any model. For details on how to get in touch with us, refer to “Getting Help” on page 11.

Running FlexModel Diagnostics

It is possible that the model behavior you are seeing is caused by a faulty installation or from using an older version of a FlexModel. If you do call Customer Support, and assuming there is no immediate solution to your problem, you will most likely be asked to run the swiftcheck diagnostic tool to verify the model version that you are using and check your environment. For information on how to run swiftcheck, refer to [Checking SmartModel Installation Integrity](#) in the *SmartModel Library User's Manual*. This tool produces a swiftcheck.out file. Send this file to Customer Support along with the other model logging files, as described in [“Sending the Log Files to Customer Support”](#) on [page 105](#).

Creating FlexModel Log Files

To create the FlexModel log files needed by Customer Support to debug model problems, follow these steps:



Attention

For FlexModels that end with an “_fz” extension, refer to the [SmartModel Library User's Manual](#) for the applicable model logging procedures. The logging mechanism described in this procedure applies only to models that end with an “_fx” extension.

1. For each FlexModel in your testbench, use the model-specific *model_set_msg_level* commands to set the message levels all the way up.
2. In the directory where you run your simulation, use the UNIX touch command or create an empty file that conforms to the following syntax. The entire string must be in uppercase.

MODEL_INSTANCE.LOG_MODE

where:

MODEL is the model name without the _fx.

INSTANCE is the instance name specified in the FlexModelId SWIFT parameter.

Typical FlexModel instantiations look like the following examples. Note the value of the FlexModelId generic or defparam—that is what you use in the ***INSTNAME*** portion of the model logging file name.

VHDL:

```
U1 : MPC860
```

```
    generic map (FlexModelId => "Model_id_1",
                 FlexTimingMode => FLEX_TIMING_MODE_OFF,
                 TimingVersion => "MPC860-25",
                 DelayRange    => "MAX")
```

VERILOG

```
defparam u1.FlexModelId = "Model_id_1";
```

For this example, the logging file name would be:

MPC860_MODEL_ID_1.LOG_MODE

1. Rerun your simulation so that the models can record their activity in the following log files:

- pin events — `pin.model_instance.log`
- trace messages — `msg.model_instance.log`
- model commands — `cmd.model_instance.log`

For example, assuming model logging is enabled for an `mpc860_fx` model instance with a `FlexModelId` of `"Model_Id_1"`, the model generates the following files:

- `cmd.mpc860_Model_Id_1.log`
- `pin.mpc860_Model_Id_1.log`
- `msg.mpc860_Model_Id_1.log`

Command Logging

Model commands are logged as shown in the following example

```
t:150
mpc860_idle(inst, 1, 'FLEX_WAIT_F, status)
```

This indicates that at simulation time 150, the `"inst"` of the `mpc860_fx` model executed an idle command for one clock cycle.

Note that command logs always show the *wait_mode* parameter as false (`FLEX_WAIT_F`), even if the command was issued with the *wait_mode* set to true (`FLEX_WAIT_T`).

Commands from a testbench that communicate directly with the Command Core are not logged. For example, the `mpc860_read_rslt` command does not get logged, since it is only accessing results information.

Stimulus Logging

Model stimulus is logged in a file named:

model_logger.v

The logger contains a process/always block which is sensitive to all the input pins, output pins, and bidirectional pins. This process/always block is only invoked when logging is enabled. The stimulus logging format is described in [Table 6](#).

Table 6: Stimulus Logging Format

Entries in File	Description
-timeformat: <i>units:precision</i>	This entry is printed once at the top of the file. It lists the time units and precision. These values are needed to recreate reported model behavior.
t: <i>time_value</i>	Time in units that pins were logged.
p <i>pin_number</i> i <i>input_value</i>	Entry for input pin.
p <i>pin_number</i> o <i>output_value</i> r <i>resolved_value</i>	Entry for output pin.
p <i>pin_number</i> b <i>bidir_value</i> r <i>resolved_value</i>	Entry for bidirectional pin.

Stimulus Log Example

Here is an example of a stimulus log file:

```
-timeformat:ns:1
t:0
p0
i1
p1
i0
p6
oz
rz
p10
bz
rz
t:60
p6
oz
rz
```


[illegible]

Notice that for input pins, only the value of the pin is recorded (i). For output pins, both the value that the model is driving onto the pin (o) and the resolved value (r) are recorded. For bidirectional pins, both the value the model is driving (b) and the resolved value (r) are recorded. Thus, contentions for output and bidirectional pins can be caught.

Message Logging

Here is an example of a message log file:

```

420 NS      INSTANCE 1. Idle State
450 NS      INSTANCE 1. arm7tdmi_write (00000F0C);
450 NS      INSTANCE 1. T1 State
480 NS      INSTANCE 1. T2 State
480 NS      INSTANCE 1. Writing Data: Address = 00000F0C
480 NS      Size = 4, Data = 98765432
510 NS      INSTANCE 1. Idle State
540 NS      INSTANCE 1. arm7tdmi_read_req (00000F00);
540 NS      INSTANCE 1. T1 State
570 NS      INSTANCE 1. T2 State
600 NS      INSTANCE 1. Latching Data: Address = 00000F00
600 NS      Size = 4, Data = 3C3C3C3C

```

Sending the Log Files to Customer Support

After you rerun your simulation to generate the model log files, tar those files up along with the swiftcheck.out file you created as described in [“Running FlexModel Diagnostics” on page 102](#). Then zip the tarball up using gzip and send the zipped log files to [Customer Support](#) as an e-mail attachment. Include your call number if you have one and a description of the problem in the body of your message.

Index

A

About This Manual [9](#)

AIX

 compiling C files [46](#)

B

bit_vectors [44](#)

Branching [20](#)

Burst transfers [25](#)

C

C Command Mode

 compiling C file [45](#)

 concurrency [43](#)

 creating C file [44](#)

 errors example [45](#)

 example [95](#)

 initialization example [44](#)

 interrupt example [34](#)

 interrupts explanation [32](#)

 interrupts, using [32](#)

 simulation time [43](#)

 switching to C [47](#)

 using [43](#)

C Command Stream

 coupled mode [22](#)

 multiple command streams [21](#)

 uncoupled mode [22](#)

C interrupt function [62](#)

C program

 compiling [45](#)

 running [72](#)

 switching to [47](#), [72](#)

Command Core [14](#), [39](#), [41](#)

Command Interface [17](#), [39](#), [53](#)

 command modes [17](#), [39](#)

 logging commands [103](#)

 organization [21](#)

Command Mode

 HDL defined [39](#)

 using HDL [20](#), [39](#)

Command Sequencing [24](#)

Command Suffixes

 req [55](#)

 rslt [55](#)

Command Syntax

 FLEX commands [59](#)

 model commands [60](#)

 result identifiers [55](#)

 status parameter [56](#)

 wait flag [56](#)

Command Types

 request [25](#)

 result [24](#), [25](#), [55](#)

Commands

 flex_clear_queue [60](#)

 flex_define_intr_function [76](#)

 flex_define_intr_signal [63](#)

 flex_get_cmd_status [65](#)

 flex_get_coupling_mode [67](#)

 flex_get_inst_handle [68](#)

 flex_print_msg [70](#)

 flex_run_program [72](#)

 flex_set_coupling_mode [74](#)

 flex_start_program [75](#)

 flex_switch_intr_control [76](#)

 flex_synchronize [77](#)

 flex_wait [79](#)

 result identifiers [55](#)

Comments?

 reporting doc suggestions [12](#)

Compiling

 external C program [45](#)

Compiling C files

 AIX [46](#)

 HP-UX [45](#)

 Intel NT [47](#)

 Linux [46](#)

 NT [46](#)

 Solaris [46](#)

Constants

FLEX_ALL_QUEUES 60
 FLEX_CMD_QUEUE 60
 FLEX_COUPLED_MODE 23
 FLEX_DEFINE 83
 FLEX_DISABLE 29
 FLEX_ENABLE 29
 FLEX_INT 83
 FLEX_RSLT_QUEUE 60
 FLEX_TIMING_MODE_ON 28
 FLEX_UNCOUPLED_MODE 23
 FLEX_VEC 44, 83
 FLEX_VEC_CONST 44
 FLEX_WAIT_F 56
 FLEX_WAIT_T 56
 MAX 28
 MIN 28
 TYP 28

Controlling command flow 20

Conventions

command syntax 11
 system-generated text 10
 UNIX prompt 10
 user input 10
 variables 11

D

DelayRange 28

Direct C Control

compiling C files 45
 restrictions 13

Documentation

online 15

E

Errors

synchronize command 27
 timeout 27

Examples

branching on result 24
 C Command Mode errors 45
 C Command Mode example 95
 C Command Mode interrupt 34
 C initialization 44

logging files 103
 message logging 105
 non-pipelined transfers 24
 stimulus logging 104
 switching to C program 47
 Verilog timing 28
 Verilog timing setup 29
 VHDL command mode 39
 VHDL interrupt 32
 VHDL timing 29
 VHDL timing setup 30
 wait_mode 26

F

FLEX Commands

command descriptions 60
 command summary 59

flex_add 86
 FLEX_ALL_QUEUES 60
 flex_and 91
 flex_assign 85
 flex_assign_int 85
 flex_assign_int_array 85
 flex_assign_int_list 85
 flex_change_setup 23
 flex_clear_queue 60
 FLEX_CMD_QUEUE 60
 FLEX_COUPLED_MODE 23
 flex_decr 86
 FLEX_DEFINE 83
 flex_define 82
 flex_define_intr_function 34, 76
 flex_define_intr_signal 32, 62, 63
 FLEX_DISABLE 29
 FLEX_ENABLE 29
 flex_eq 86
 flex_errors 84
 flex_fprintf 94
 flex_get_cmd_status 65
 flex_get_coupling_mode 67
 flex_get_inst_handle 68
 flex_gt 87
 flex_gte 87

- [flex_incr](#) [86](#)
- [FLEX_INT](#) [83](#)
- [flex_iprintf](#) [94](#)
- [flex_lrot](#) [90](#)
- [flex_lshift](#) [89](#)
- [flex_lt](#) [87](#)
- [flex_lte](#) [87](#)
- [flex_nand](#) [91](#)
- [flex_ne](#) [86](#)
- [flex_nor](#) [91](#)
- [flex_not](#) [90](#)
- [flex_notes](#) [84](#)
- [flex_or](#) [90](#)
- [flex_print_msg](#) [70](#)
- [flex_rrot](#) [90](#)
- [flex_rshift](#) [89](#)
- [FLEX_RSLT_QUEUE](#) [60](#)
- [flex_run_program](#) [21](#), [25](#), [40](#), [72](#)
- [flex_set_coupling_mode](#) [74](#)
- [flex_slice_be](#) [88](#)
- [flex_slice_be_offset](#) [89](#)
- [flex_slice_le](#) [87](#)
- [flex_slice_le_offset](#) [88](#)
- [flex_sprintf](#) [95](#)
- [flex_start_program](#) [43](#), [44](#), [75](#)
- [flex_sub](#) [86](#)
- [flex_switch_intr_control](#) [76](#)
- [flex_synchronize](#) [27](#), [40](#), [77](#)
- [FLEX_TIMING_MODE_ON](#) [28](#)
- [flex_to_int](#) [92](#)
- [flex_to_int_array](#) [92](#)
- [flex_to_int_list](#) [93](#)
- [FLEX_UNCOUPLED_MODE](#) [23](#)
- [FLEX_VEC](#) [44](#), [82](#), [83](#)
- [FLEX_VEC_CONST](#) [44](#)
- [flex_vec_sizeof](#) [82](#)
- [flex_wait](#) [32](#), [79](#)
- [FLEX_WAIT_F](#) [33](#), [56](#), [103](#)
- [FLEX_WAIT_T](#) [32](#), [56](#), [103](#)
- [flex_warnings](#) [84](#)
- [flex_xnor](#) [91](#)
- [flex_xor](#) [91](#)

- [FLEXIm license](#) [15](#)
- [flexm_setup](#) [17](#)
- [FlexModel](#)
 - [block diagram](#) [14](#)
 - [Command Core](#) [14](#), [39](#)
 - [command interface](#) [17](#), [39](#), [53](#)
 - [controlling command flow](#) [20](#)
 - [features](#) [13](#)
 - [initialization](#) [18](#)
 - [licensing](#) [15](#)
 - [limitations](#) [15](#)
 - [structure](#) [15](#)
- [flexmodel_pkg.h](#) [44](#)
- [FlexModels](#)
 - [Command Core](#) [41](#)
- [Functions](#) [62](#)
- [Functions C-mode](#)
 - [flex_add](#) [86](#)
 - [flex_and](#) [91](#)
 - [flex_assign](#) [85](#)
 - [flex_assign_int](#) [85](#)
 - [flex_assign_int_array](#) [85](#)
 - [flex_assign_int_list](#) [85](#)
 - [flex_decr](#) [86](#)
 - [flex_define](#) [82](#)
 - [flex_eq](#) [86](#)
 - [flex_errors](#) [84](#)
 - [flex_fprintf](#) [94](#)
 - [flex_gt](#) [87](#)
 - [flex_gte](#) [87](#)
 - [flex_incr](#) [86](#)
 - [flex_iprintf](#) [94](#)
 - [flex_lrot](#) [90](#)
 - [flex_lshift](#) [89](#)
 - [flex_lt](#) [87](#)
 - [flex_lte](#) [87](#)
 - [flex_nand](#) [91](#)
 - [flex_ne](#) [86](#)
 - [flex_nor](#) [91](#)
 - [flex_not](#) [90](#)
 - [flex_notes](#) [84](#)
 - [flex_or](#) [90](#)
 - [flex_rrot](#) [90](#)
 - [flex_rshift](#) [89](#)
 - [flex_slice_be](#) [88](#)

flex_slice_be_offset 89
 flex_slice_le 87
 flex_slice_le_offset 88
 flex_sprintf 95
 flex_sub 86
 flex_to_int 92
 flex_to_int_array 92
 flex_to_int_list 93
 FLEX_VEC 82
 flex_warnings 84
 flex_xnor 91
 flex_xor 91

H

HDL Command Mode 20, 39
 HDL Control
 Command Core timing 41
 HDL-C mechanism 41
 multiple state commands 42
 timing diagram 42
 Header files 44
 Help
 how to get 11
 HP-UX
 compiling C files 45

I

Initialization 18
 inst_handle 27, 55, 68
 Install Process 14
 Integers
 in C and HDL 44
 Interrupt Commands
 and FLEX_WAIT_F 33
 and FLEX_WAIT_T 32
 model_begin_intr 33
 model_end_intr 33
 model_get_intr_priority 32, 35
 Interrupt Service Routine 30
 definition 30
 invocation 31
 priority-specific 30
 process/always block 32

Interrupts
 C Command Mode description 32
 C Command Mode setup 32
 clock edges 31
 detection 31
 handler synchronization 27
 nesting of 32
 reset 31
 VHDL control 41
 VHDL example 32
 Interrupts and Exceptions 30

L

Licensing 15
 Linux
 compiling C files 46
 LMC_HOME tree 15
 Logging 101
 bidirectional pins 105
 cmd.model_instname.log 103
 command format 103
 commands 103
 commands not logged 103
 enabling for instance 102
 log file examples 103
 message logfile 105
 messages 105
 model_logger.v file 104
 msg.model_instname.log 103
 pin.model_instname.log 103
 stimulus 104
 stimulus example 104
 strategy 102

M

MAX 28
 MIN 28
 Model Logging 102
 model_begin_interrupt 35
 model_end_interrupt 35
 model_pkg.h 44
 model_set_timing_control 29
 Multiple command sources 27

Multiple Command Streams
in C testbench [21](#)

N

Non-pipelined transfers
example [24](#)

NT

compiling C files [46](#)
num_instance parameter [27](#)

P

Parameters

DelayRange [28](#)
inst_handle [55](#)
num_instance [27](#)
sig_name [63](#)
status [56](#)
sync_label [27](#)
sync_tag [77](#)
sync_timeout [77](#)
sync_total [77](#)
TimingVersion [28](#)
valid_f [65](#)
wait flag [56](#)
wait_mode [40](#)

Pipelined bus operations [25](#)

Pipelining

delayed result checking [27](#), [39](#)
phase diagram [25](#)
request phase [25](#)
results phase [25](#)

Preface [9](#)

Propagation delays [28](#)

R

Related documents [9](#)

Request commands [25](#)

Reset [20](#), [27](#)

Result command [24](#)

Results

commands [25](#), [55](#)
delayed checking [27](#), [39](#)
from commands [25](#)

model state [25](#)

Results phase [25](#)

S

SLC

Synopsys Common Licensing [12](#)

SmartModel

Browser tool [9](#)

Solaris

compiling C files [46](#)

Status Parameter

C Command Mode [57](#)

definition [56](#)

types of information [56](#)

std_logic_vectors [44](#)

Suspending command execution [77](#)

SWIFT Interface [14](#)

Switching

to a C program [47](#)

Switching command sources [25](#)

Symbol Conventions [10](#)

sync_label parameter [27](#)

Synchronizing command flow [27](#), [39](#)

Synopsys Common Licensing [12](#)

T

Tag

checking [65](#), [67](#), [74](#)

Timing

access delays [29](#)

Access delays [28](#)

checks

checks for timing [28](#)

controlling behavior [28](#)

controlling messages [29](#)

custom timing [28](#)

function-only [28](#)

introduction [28](#)

propagation delays [28](#)

relationships [28](#)

UDT [28](#)

user-defined [28](#)

Verilog example [29](#)

- VHDL example [30](#)
- TimingVersion [28](#)
- Troubleshooting [101](#)
 - message log [105](#)
 - message logging [105](#)
 - sending a log file [102](#)
 - stimulus logging [104](#)
 - trace messages [105](#)
- TYP [28](#)
- Typographical conventions [10](#)

U

- User-defined timing (UDT) [28](#)

V

- Variables
 - flex_change_setup [23](#)
- vector representation [81](#)
- Vector Representation in C [81](#)
 - 9-state to 4-state [83](#)
 - void [83](#)
- VERA Command Mode
 - class constructor [49](#)
 - files in LMC_HOME [49](#)
 - testbench examples [50](#)
 - using [47](#)
 - VERA classes [48](#)
- VERA interrupt routines [36](#)
- Verilog
 - task calls [40](#)
 - timing example [29](#)
 - timing setup [28](#)
- Verilog control [41](#)
- VHDL
 - procedure calls [40](#)
 - testbench example [39](#)
 - timing example [30](#)
 - timing setup [29](#)
- VHDL control [41](#)
- Visual C++ [47](#)

W

- wait Flag [56](#)
- Wait in C Command Mode [79](#)
 - timing diagram [79](#)
- wait_mode example [26](#)
- wait_mode parameter [40](#)
- Websites
 - Synopsys [12](#)