TestBencher Pro

User's Manual

www.syncad.com

TestBencher Pro Manual
(rev 6.5A) copyright 1994-1999 SynaptiCAD

Trademarks

- Timing Diagrammer Pro, WaveFormer Pro, TestBencher Pro, VeriLogger Pro, DataSheet Pro, and SynaptiCAD are trademarks of SynaptiCAD Inc.
- Pod-A-Lyzer is a trademark of Boulder Creek Engineering.
- PeakVHDL and PeakFPGA are trademarks of Accolade Design Automation Inc.
- V-System and ModelSim are trademarks of Model Technology Incorporated.
- Viewlogic, Workview, and Viewsim are registered trademarks of Viewlogic Inc.
- Timing Designer and Chronology are registered trademarks of Chronology Corp.
- DesignWorks is a trademark of Capilano Computing.
- Mentor and QuickSim II are registered trademarks of Mentor Graphics Inc.
- OrCAD is a registered trademark of OrCAD.
- PSpice is a registered trademark of MicroSim.
- Windows, Windows NT, and Windows 95/98 are registered trademarks of Microsoft.

All other brand and product names are the trademarks of their respective holders.

Information in this documentation is subject to change without notice and does not represent a commitment on the part of SynaptiCAD. Not all functions listed in manual apply to Timing Diagrammer Pro or WaveFormer Pro. The software and associated documentation is provided under a license agreement and is the property of SynaptiCAD. Copying the software in violation of Federal Copyright Law is a criminal offense. Violators will be prosecuted to the full extent of the law.

No part of this document may be reproduced or transmitted in any manner or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the written permission of SynaptiCAD.

For latest product information and updates contact SynaptiCAD at:
web site: http://www.syncad.com
email: sales@syncad.com
phone: (540)953-3390

Table of Contents

Table of Contents 3

Introduction 7

Chapter 1: A Quick Start to TestBencher Pro 9

Step 1: Pick A Language 9
Step 2: Add the MUT to the Project 9
Step 3: Add Signals to a Transaction Diagram 10
Step 4: Draw First Transaction 10
Step 5: Add Transaction Diagram to the Project 11
Step 6: Add Top-Level Template File 11
Step 7: Instantiate MUT in the Template 11
Step 8: Insert Diagram Calls into Template 12
Step 9: Generate the Test Bench 13
Step 10: Simulate Test Bench 13
Step 11: Finding Out More About TestBencher Pro 14

Chapter 2: Project Functions 15

2.1 Opening, Saving, and Starting New Projects 15
2.2 Project Preferences Dialog 15
2.3 Adding Files and Building the Project 17
2.4 Using the Project Tree Control 18
2.5 ModelSim Integration 19
2.6 Using a Third Party Simulator with TestBencher Projects 20

Chapter 3: Create Transaction Diagrams 21

3.1 Transaction Diagram Objects 21
3.2 Direction and Type of Signals 22
3.3 Driving Bi-directional Signals 22
3.4 Creating Clocked Signals 23
3.5 Creating Continuous Setups and Holds 24
3.6 Variables in Timing Diagrams 25
3.7 Delays and the Delay Properties Dialog 25
3.8 Specifying a Timeout for a Delay 26

Chapter 4: Samples 27

4.1 Point Samples 27
4.2 Window Samples 27
4.3 The Sample Properties Dialog 27
4.4 Self-testing Code from Samples 28
4.5 Writing User-Defined Sample Conditions and Actions 29
4.6 Sample Values as Transaction Outputs 30
4.7 Creating New Sample Types 31

Chapter 5: Markers 33

5.1 Adding a Marker to a Diagram 33
5.2 Absolute and Relative Markers 33
5.3 Time Markers for Specifying End Diagram 34
5.4 Time Markers for Test Bench Loops 34

5.5 HDL Code Markers 35

Chapter 6: Diagram Level Test Bench Settings 37

6.1 The Diagram Level TestBench Settings Dialog 37
6.2 Including External HDL Code Library Files 38
6.3 Enabling/Disabling HDL Code for Parameters 38
6.4 Specifying a Timeout for a Transaction 39

Chapter 7: Editor Functions 41

7.1 Opening, Saving, and Creating New Source Code 41
7.2 Displaying or Finding a Specific Line of Code 41
7.3 Using the Editor Preferences Dialog 42
7.4 Editor Cursor Commands 42
7.5 Using an External Editor 43

Chapter 8: Working with Template Files 45

8.1 Adding the Template to a Project 45
8.2 Instantiating your Model Under Test (MUT) 46
8.3 Using Diagram Calls to Trigger Transactions 47
8.4 Setting State Variables in Diagram Calls 48
8.5 Controlling the Execution Mode of a Transaction 48
8.6 Aborting a Timing Transaction 49
8.7 Adding HDL Code to Template Files 49

Chapter 9: Generating the Test Bench 51

9.1 TestBencher Pro Design Flow 51
9.2 Generate the Test Bench 51
9.3 Errors During Test Bench Generation 52
9.4 Simulating the Test Bench 52
9.5 Test Bench Preferences Dialog (options) 53

Chapter 10: Advanced Techniques 55

10.1 Generating Test Bench-Level Glue Logic 55
10.2 Creating a Global Clock 55
10.3 Loops in Transactions 56

Chapter 11: The Architecture of TestBencher Test Benches. 57

11.1 Files Used to Build a Project 57
11.2 Status Signals and States 57
11.3 Monitoring Transaction Execution 59
11.4 Generated Tasks for Timing Transactions 60
11.5 TestBencher HDL Library Files 60

Appendix A: TestBencher Pro Basic Tutorial 61

1 Setting up the Project 62
2 Creating a TestBencher Pro Timing Diagram 64
3 Creating and Modifying a Template File 69
4 Generate and Examine the Test Bench 73
5 Simulating the Test Bench 74

Appendix B: Performing a Sweep Test (Advanced Tutorial) 77

1 Create a Global Clock Generator with Adjustable Period 79

2 Add Global Clock as an Input to the Read Transaction 80
3 Change Read Transaction to be Cycle Based 80
4 Add Sweepable Delay 82
5 Add a Continuous Setup Checker to the Read Transaction 83
6 Examine the Write Transaction 84
7 Applying Timing Transactions to the Template File 84
8 Adding HDL Code to Sweep the Delay 86
9 Generating the Test Bench and Simulating the Project 88

Appendix C: License Agreement 91

Index 93

Introduction

TestBencher Pro is a self-testing test bench generator for VHDL and Verilog. Test benches generated by TestBencher Pro include input stimulus vectors and extra code that checks simulation output for correctness. The generated test benches are capable of applying different stimulus vectors depending on simulation response so that the test bench functions as a behavioral model of the environment in which the system being tested will operate. Using TestBencher Pro, bus-functional microprocessor interfaces can be modeled with only a few lines of code. TestBencher Pro also provides an automated method for checking large ASIC simulation runs.

The quickest way to learn how to use TestBencher Pro is to work through the tutorials provided in the appendices and to read through *Chapter 1: A Quick Start to TestBencher Pro*. Advanced features such as global clocks, loops in test benches, building glue logic, generating code from samples, and controlling execution using markers can be found in the later chapters. There are also several test bench examples that are located in the \Examples subdirectory of the installation directory.

Additional information regarding the architecture and design flow when working with TestBencher Pro are available at the SynaptiCAD website (**www.syncad.com**).

Chapter 1: A Quick Start to TestBencher Pro

This chapter will cover all the basic steps involved in generating a bus-functional test bench using TestBencher Pro. More detailed information is available in chapters 2-9.

Step 1: Pick A Language

The first step in creating a test bench is to choose the generated language. TestBencher Pro supports Verilog, VHDL 87, and VHDL 93. It is important for VHDL users to choose the VHDL standard that is compatible with your VHDL simulator and your existing model code. As TestBencher receives information on your model code and transaction diagrams it will incrementally generate code, so it is beneficial to choose the language at the beginning of the design. Choose the language for the current project:

- Choose the **Project > Project Settings** menu option to open the *Project Settings* dialog.
- Choose a language from the **Language** drop-down list box and then click **OK** to close the dialog.

To set the default language for all future projects:

- Choose the **Options > Test Bench Preferences** menu option to open the *Test Bench Preferences* dialog.
- Choose a language from the Language drop-down list box and then click **OK** to close the dialog.

Another Project setting that must be set before you begin is the *Simulation Mode*. There are two different modes that TestBencher works in, **Debug Run** and **Auto Run**. The **Debug Run** mode allows you to control when the simulation is run. This is the best setting to use while you are developing your timing diagrams. The **Auto Run** mode runs the simulation every time a signal is modified or added. This mode works best when you are making small changes to a timing diagram, once all of the signals are in place. While designing your timing diagrams, you also want to begin with the **Simulate Project** setting selected while developing your first timing diagram.

To set your Simulation Mode and State:

- If the mode button is set to **Auto Run**, click the button to toggle to **Debug Run**. The mode button is located on the simulation toolbar, below the **Project** menu in TestBencher.
- If the simulation state in the drop down list box is set to **Sim Diagram & Project** , select **Simulate Project** from the drop down menu. The drop down list box is located on the simulation toolbar beneath the **File** menu in TestBencher.

Step 2: Add the MUT to the Project

TestBencher uses a project to control the test bench generation, set the transaction diagrams, and set the template files. The first thing to add to the project is the *model under test*, MUT, files of the design you wish to test. TestBencher will scan these files and copy the top-level signal information to the *Diagram* window. To add a file to the project:

- Right click in the Project window to open the context menu, and select the **Add HDL File(s)** menu option,
- OR, choose the **Project > Add HDL File(s)** menu option.
- Both of these functions open a file dialog. Select the files that you would like to add to the project and click the **Open** button to close the dialog.

Notice that the file names are listed in the project window. The source code can be viewed by double clicking on the file name.

Step 3: Add Signals to a Transaction Diagram

Once the MUT has been added, the test bench signals that interface to the model under test need to be defined.

Add signals and edit the name, direction, and bit width to match the model under test:

- Add signals manually by pressing the **Add Signal** button.
- Double click on the signal name in the signal label window to open the *Signals Properties* dialog.
- Edit the signal **name**, **direction**, and bit width (**MSB** & **LSB**) fields. Signal direction is specified relative to the test bench, not relative to the model under test (e.g., a signal that is an input to a transaction is driven elsewhere in the model). Generally you will want to add signals that match the ports of your MUT, but in more complex transactions you may want to add internal signals that are only used by your transaction and do not directly interface to your MUT.

Once the project is built you can view all the modules, signals, ports, and components in the Verilog files. One module name will be surrounded by brackets <<*name*>>. This is the top-level module for the project. It is the highest-level instantiated component. All sub-modules can be viewed by descending the top-level module's tree.

If the top-level module does not have port signals, the internal signals of the module are displayed in the diagram. If the top-level module has port signals, the output ports are viewed as blue signals and input ports are viewed as black signals. The black input signals can now be edited to create a transaction model.

Step 4: Draw First Transaction

In the next two steps you will draw and add a timing transaction for your test bench project. These steps will be repeated for each timing transaction in your project.

A timing transaction is a timing diagram that represents a reusable interface specification of the bus-functional model that you are creating (e.g., read cycle, write cycle, interrupt cycle). Timing diagrams are created using the built in timing diagram editor and Chapter 3 covers the more advanced features. The bus functional model code is generated from the waveforms, samples, markers, and variables that are contained in the timing diagram.

Specify the following information in each timing diagram:

- The **direction** and **type** (Section 3.2) of each signal is set using the *Signal Properties* dialog box. For example, SIG1 might have a direction of "output" and a type "integer". The direction of transition for **bi-directional** signals (Section 3.3) can be set using the Driven flag in the *Edge* dialog (double click left on the edge to open this dialog box).

The following optional components can be added to each diagram as desired:

- **Variables** (Section 3.6) can be defined for the timing diagram. A variable for a specific data bus gives you the option of using caller-specified state values for the data bus.
- Attach samples to the diagram to generate **self-testing** (Section 4.4) code. This type of code monitors the simulation and will perform as specified by the user (e.g., provide a warning message when the output is not what is expected). Sample actions may be from the library set, or they may be **user-defined** (Section 4.5).
- An **End Diagram Time Marker** (Section 5.3) can be used to indicate the exact end of the timing diagram's execution (this can be used for clock synchronization).
- **Loop Time Markers** (Section 5.4) can be placed in the diagram to indicate looping sections of the timing diagram.

Other optional components will be discussed in more detail later in this manual. You will be able to determine which optional components to add based upon the exact nature of your simulation.

Step 5: Add Transaction Diagram to the Project

After the timing transaction is drawn, save it to a file and add the timing diagram to the project.

Save the timing diagram:

- Choose the **File > Save Timing Diagram** menu option to save the file. This opens the **Save File** dialog. Timing diagram files use a *.tim file extension.

Add the timing diagram to the project:

- Right click in the signal label window and choose **Add Diagram to Project** from the context menu. This puts the timing diagram into the Project window and generates the HDL code for the diagram.

View Generated HDL Source Code:

- Right click on the timing diagram file name in the Project window and choose **Open…** from the context menu. This will open an editor window and display the associated HDL file. The HDL file has the same name as the timing diagram with a *.v extension for Verilog or a *.vhd for VHDL.

At this point you can either design additional transactions by modifying the current timing diagram and following the instructions in **Step 4**, or you can continue with the next few steps and design the top-level test bench. By working with the top-level test bench early in the design you will be able to test individual transactions before constructing the entire bus-functional model.

Step 6: Add Top-Level Template File

This section will explain how to add a Top-level template file. The template file provides the top-level model for your test bench. You will instantiate your model under test and control the sequencing of your diagram transactions from this file. Step 7 will explain how to edit the template for the specific project you are developing.

To add a top-level template file to the project:

- Right click in the Project window and choose **Copy TestBencher Template File** from the context menu. This opens a dialog box with the same name.
- Check the **Add to Project** checkbox in the bottom left of the dialog box to ensure that the template you choose will be copied to the project.
- Enter an existing template file name into the **Original Template** edit box or use the Browse (…) button to find one on the hard drive. TestBencher Pro is shipped with two default template files for each language (found in the `\templatefiles` subdirectory of your TestBencher installation directory). Users can also design their own template files. If this is your first test bench, use **tbench.v** (for Verilog projects) or **tbench.vhd** (for VHDL projects) because it generates the simplest test benches.
- Enter the name for your top-level test bench into the **New Template** edit box.
- Click **OK** to copy the template and add it to the Project window.

Step 7: Instantiate MUT in the Template

Next, you will add a code segment to the Template file that will instantiate the MUT in the test bench. Once the final test bench is generated it will become the top-level module in your design and will be responsible for instantiating the MUT and the timing transactions.

Open the template file and locate the place in the code to instantiate the MUT:

- Double click on the test bench file that was just added to your project. This opens an *Edit* window and displays the source code of the template file.
- Scroll through the template file and locate the following comment:

```
// 1) Instantiate Models Under Test after this comment block:
```

Verilog Users:

If you have already compiled the Verilog MUT, you can optionally use the project window to instantiate your MUT instead of manually typing the instantiation.

- Left click in the template file editor window below the comment shown above.
- Right click on the top level module (module name enclosed in brackets: e.g., `<<< module >>>`) in the Project window to open the pop-up context menu
- Select **Instantiate in TBench**. This will add the instantiation of the selected module into the template file.

An example of instantiation of a MUT in Verilog is:

```
tbsram tbsram( CSB,WRB,ABUS,DATABUS );
```

This line of code will declare your module for use in the project.

VHDL Users:

If you are using VHDL, instantiate the MUT by:

- Type the MUT instantiation into the template file. It will probably look something like this:

```
mutMSB: tbsram port map(CSB, WRB, ABUS, DBUS(15 downto 8));
```

- Next, scroll up in the code to the line

```
-- 1) Declare component for Models Under Test after this comment
```

Define the component portion of your module below this comment block. Use the following example as a model:

```
component tbsram port( CSB : in std_logic;
                       WRB : in std_logic;
                       ABUS : in std_logic_vector(15 downto 0);
                       DATABUS : inout std_logic_vector(7 downto 0)
                      );
end component;
```

Note: Keep the Template file open for use in Step 8.

Step 8: Insert Diagram Calls into Template

This step will show you how to place and sequence the transactions in the test bench.

Use the *Insert Diagram Call* dialog to add timing diagram apply statements to the template file:

- Scroll down in the template file until you find the sequencer code. You will see a comment in the code that looks like:

```
-- 4) Add apply calls for timing diagram transactions after this
```

- Left click in the template file just below this comment. This is the segment of code where timing transactions will be executed.

- Select **Editor > Show Insert Diagram Call Dialog…** This will open the *Insert Diagram Call* dialog. A list of statements will be in the dialog box. Each statement represents one of the timing transactions that you have added to the project.
- Double click on the first **Apply_*TransactionName*** statement to add it to the template file.
- Notice that the Apply statement was inserted in the template file. The *Insert Diagram Call* dialog is a modeless dialog so you can leave it open while you perform other actions. Double clicking on additional Apply statements causes those statements to be added on successive lines.
- If any of the applied transactions contain variables, then edit the template code to provide values for variable names. In the example Apply statement below, a value of 3 is assigned to stateVar.

```
// Apply_verySimpleCyclic(stateVar)

Apply_verySimpleCyclic(3);
```

See *Section 8.3 Using Diagram Calls to Trigger Transactions* for more information about adding diagram calls.

Step 9: Generate the Test Bench

Once the Template file has been successfully edited, you are ready to generate the test bench. To do this:

- Choose **Export > Generate Test Bench** to generate the test bench.

When the test bench is generated, several files are produced. One HDL file is produced for every timing diagram that you added to the project. Another HDL file is produced that represents the test bench itself.

The original template file contained a series of macros that are expanded to generate the HDL code for the test bench. In the template file, macros begin with a dollar sign ($). The macros are used to expand the template file into the HDL code for the test bench file. This will be accomplished through direct textual substitution, resulting in the generated test bench.

The test bench can be generated repetitively, allowing you to make changes to the timing transactions without losing any code that you may have placed in the test bench. It is important to note that any text between the beginning and the ending of the macro comments will be replaced. This means that HDL code that you add should be placed before or after the blocks of code generated by TestBencher (see Section 9.2 for more information).

Two other files will be important during the test bench simulation (Step 10) **-** “verilog.log” and “waveperl.log”. The first file will contain simulator warnings and errors that occur during the simulation. The Report window (lower right hand corner) has a tab on the bottom that is labeled “verilog.log”. Click this tab to view “verilog.log.” The second log file, “waveperl.log,” will report warnings specified by the simulator. The test bench can be set up to record unexpected values, for instance. (Note: other files and reports that are available are listed under the other labeled tabs in the Report window.)

Step 10: Simulate Test Bench

The test bench that you have created can now be simulated by taking all of the files produced by TestBencher Pro and placing them into a simulator. Verilog users can use the information in this section to simulate the test bench using TestBencher Pro.

There are three ways to run the simulation:

- Press the **Run** button (large green triangle) on the simulation button bar, making sure the **Simulate Project** option is selected.
- Choose the **Simulate > Run** menu option, OR

- Press the **<F5>** key.

The waveforms will be displayed in the Diagram window once a project is simulated. If you do not want to continue to watch a particular signal, left click on the signal name in the Diagram window and press the **<Delete>** key.

TestBencher Pro has two simulation modes, **Auto Run** and **Debug Run**. These modes determine when a simulation is performed. The current simulation mode is displayed on the leftmost button on the simulation button bar; this is the *mode button*. In the **Debug Run** simulation mode, simulations are started only when the user presses the Run or Single Step buttons (similar to a standard Verilog simulator). In the **Auto Run** simulation mode, the simulator will automatically run a simulation each time a waveform is added or modified in the Diagram window. This mode makes it easy to quickly test small modules and do bottom-up testing. Press the mode button to toggle between the two simulation modes.

TestBencher Pro also has two simulation states, **Sim Diagram & Project** and **Simulate Project**, that determine what is simulated. The state can be selected by using the drop down list to the left of the mode button on the simulation button bar. The **Sim Diagram & Project** mode indicates that both the diagram waveforms and the Verilog source code will be simulated together. The **Simulate Project** state indicates that Verilog code and the diagram will be simulated independent of one another. The simulation button bar controls the simulation in this state. An additional method of controlling simulation is by use of the **Simulate Diagram** button on the simulation button bar. Waveforms cannot be used as stimuli in the **Simulate Project** state, and so they cannot be used to drive signals. The **Simulate Project** state gives the user the ability to perform timing and analysis on waveforms without incurring the overhead of re-simulating the entire project. This mode is often useful when creating transactions for a test bench.

Step 11: Finding Out More About TestBencher Pro

The quickest way to learn more about using TestBencher is to perform the tutorials provided for the project and to examine the example files provided with the project. Appendices A and B of this manual provide a basic tutorial and a more advanced tutorial, respectively.

Additional example files can be found in the **Examples** subdirectory of the installation directory. Many of these files contain text in the timing diagram that specifies what the timing diagram demonstrates. In addition, there is a **readme.txt** file in the Examples directory that specifies the contents of each subdirectory of the **Examples > VHDL** and **Examples > Verilog** directories.

Chapter 2: Project Functions

TestBencher Pro uses a project to list the files to be simulated, set watches on signals (view signal waveforms), and store simulation options.

2.1 Opening, Saving, and Starting New Projects

Projects are opened and saved using the Project menu options. By default, TestBencher opens with a new untitled project.

To open an existing project:

- Select the **Project > Open HDL Project** menu option. This opens the *Open Project File* dialog where you can select a project file to open.

To save an open project:

- Select the **Project > Save HDL Project** menu option to open a *Save* dialog. By default, project file names have an extension of **HPJ**.

To clear the current project and start a new project:

- Select the **Project > New HDL Project** menu option. You will be asked several questions about saving the files that are currently open and then a new project will be created.

2.2 Project Preferences Dialog

The *Project Settings* dialog controls the simulator run time options. This information is stored inside the project HPJ file. To open the *Project Settings* dialog:

- Choose the **Project > Project Settings** menu option to open the *Project Settings* dialog

There are several options in the *Project Settings* dialog.

- The **Delay Settings** radio buttons determines which delay value is used in **min:typ:max** expressions. This is similar to the **+maxdelays**, **+mindelays**, and **+typdelays** command line simulator options.

The next group of options are the checkboxes to the right of the **Delay Settings** section.

- The **Grab top level signals** check box turns on the automatic monitoring of ports or internal signals in the top-level module.
- The **Show file tree nodes** check box allows file names to be shown in the project tree.
- The **Hide empty lists** check box hides nodes without any leaf nodes. Checking this makes the project tree more readable.

- The **Capture and Show Watched Signals** check box enables the display of waveform results from a simulation run..
- The **Dump Watched Signals** check box will generate a dump file for any watched signals in the diagram. The generated file will have the same name as the .tim file, only with an extension of **.VCD**.
- One setting in this dialog has been reserved for future use – the **Interactive Mode After Compile** checkbox.

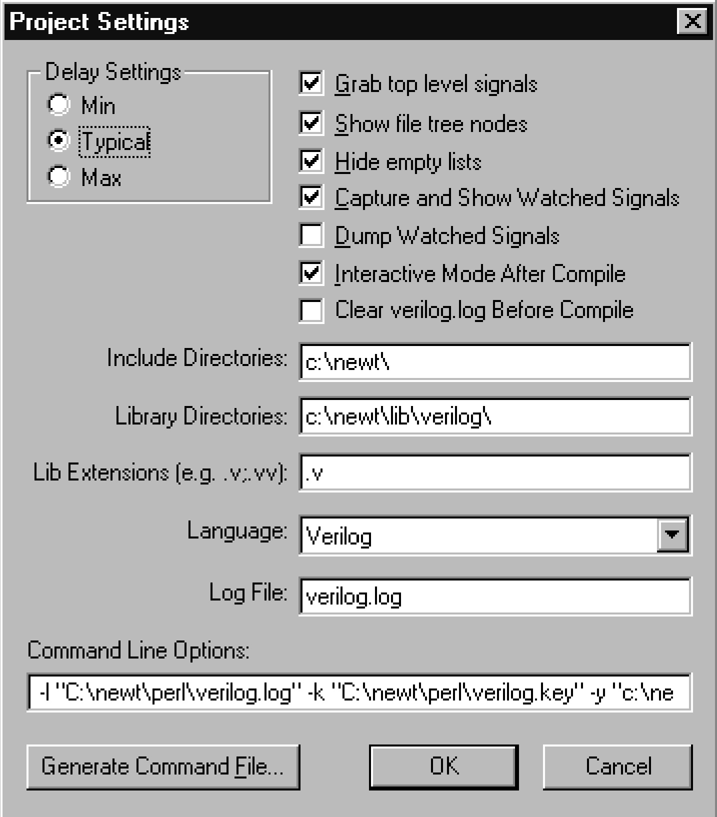


Figure 2.1: Project Settings dialog

Below these two groups are several edit boxes that tell the project where to look for files, the file types that are being used, etc. These options are explained below.

- **Include Directories** edit box specifies the directories where TestBencher searches for files that will be used in the project. The following is a Windows example (Unix users should use the / slashes):

```
C:\design\project;c:\design\library
```

- The **Library Directories** edit box lists the path and directories where TestBencher searches for library files. TestBencher will try to match any undefined modules with the names of the files that have one of the file extensions listed in the **Lib Extensions** edit box. The simulator does not look inside a file unless the undefined module name exactly matches a file name. The simulator does not look at any files unless there are file extensions listed in the **Lib Extensions** edit box. The following is a Windows example (Unix users should use the / slashes):

```
C:\design\project;c:\design\library
```

- The **Lib Extensions** edit box specifies the file name extension used when searching for library files in the library directory. Each library extension should begin with the period character followed by the extension name. Use a semicolon to separate multiple file extensions.

```
.v;.vv
```

- The **Language** setting is controlled by use of a drop down list box. It is important that this setting be properly selected to ensure that the generated test bench is written in the correct language.
- The **Logfile** specifies the name of the log file which receives all the simulation results and information. By default TestBencher uses **verilog.log** file.
- The **Command Line Options** edit box contains the command line equivalents for all the options that are checked in this dialog. When the **Generate Command File** button is pushed, the text contained in the **Command Line Options** edit box along with the list of Verilog files specified in the project window are written to a command file. This file can then be used with the Command Line version of the TestBencher simulator.

If you work primarily with one language, you probably want to set a default language so that you do not have to reselect the language for each new project. When you have finished making the project settings with the *Project Settings* dialog box, you can change the default language for all future projects by:

- Choosing the **Options > Test Bench Preferences** menu option to open the *Test Bench Preferences* dialog.
- Choosing a language from the **Language** drop-down list box in the *Project Defaults* section of the dialog box and then clicking OK to close the dialog.

2.3 Adding Files and Building the Project

The first step in creating a project is to add timing diagrams to the project window.

If you need to create new timing diagrams, see *Chapter 1: A Quick Start to TestBencher Pro* for a quick start, or *Chapter 3: Create Transaction Diagrams* for a more detailed explanation.

If you have timing diagram files (.tim) already prepared, do the following to add them to the Project window:

- Right click in the Project window to open the context menu, and select the **Add Timing Diagram(s)** menu option,
- OR, choose the **Project > Add Timing Diagram(s)** menu option.
- Both of these functions open a *File* dialog. Select one or more files to add to the project and click the **Open** button to close the dialog.

To add an existing HDL file to the project:

- Right click in the Project window to open the context menu, and select the **Add HDL File(s)** menu option,
- OR, choose the **Project > Add HDL File(s)** menu option.
- Both of these actions open a *File* dialog. Select one or more files to add to the project and click the **Open** button to close the dialog.

Note: If you add a file to a project that has not yet been saved you will be prompted to save the new project. The **Save New Project As** dialog will open. The directory that you save the new project in will be the project directory, and all generated files will be placed in this directory.

Verilog users will be able to build the project for simulation under TestBencher. VHDL users will not be able to build the project within TestBencher, so a hierarchical view of the files will not be possible. When files are first added to the project, you can see the filename but you cannot see a hierarchical view of the modules inside the files. To view the internal modules on the project tree you must first **build** or **run** a simulation. The **build** command compiles the Verilog files and builds the Verilog tree. It does not run a simulation. For large projects, **build** lets you quickly construct the tree without having to wait for a simulation to run.

There are three ways to build a project:

- Press the yellow **Build** button on the simulation button bar,
- Select the **Simulate > Build** menu option,
- OR, press the **<F7>** key

Chapter 8: Working with Template Files gives a detailed explanation of working with Template files. Template files are used to generate the test bench itself once the timing transactions have been added to the project. The Template has a file extension of .v for Verilog users and .vhd for VHDL users. Once all of your timing diagrams have been completed, you will be ready to add your Template file to build the test bench.

To add a top-level template file to the project:

- Right click in the Project window and choose **Copy TestBencher Template File** from the context menu. This opens a dialog box with the same name.
- Check the **Add to Project** checkbox in the bottom left of the dialog box to ensure that the template you choose will be copied to the project.
- Enter an existing template file name into the **Original Template** edit box or use the Browse (…) button to find one on the hard drive. TestBencher Pro is shipped with two default template files for each language. Users can also design their own template file. The **isotbench** test bench generates the waveforms related to the MUT, while the **tbench** generates all of the waveforms that will let you see the exact status of each transaction during the simulation. The **tbench** is the default test bench template file because it provides more information and the file is simpler.
- Enter the name of your top-level project into the **New Template** edit box.
- Click **OK** to copy the template and add it to the Project window.

Once the Template file is in place and edited to include your MUT and timing transactions (*Steps 1 & 8, Chapter 1: A Quick Start to TestBencher Pro*), you can **generate** the test bench by selecting **Export > Generate Test Bench…**

If you are using Verilog, you can **build** the test bench using the **build** or **run** instructions above. After the project is built you can view all the modules, signals, ports, and components in the Verilog files. One module name will be surrounded by brackets <<*name*>>. This is the top-level test bench module for the project. The top-level module is the highest level-instantiated component. All sub-modules can be viewed by descending the top-level module's tree. *Section 9.4: Simulating the Test Bench* discusses project simulation.

2.4 Using the Project Tree Control

The Project Tree control is used to investigate the hierarchical structure of the Verilog components, view source code, and set watches on signals. Each node in the tree has a context sensitive pop-up menu that can be opened by right clicking on the node. The following functions describe the general viewing features of the tree control.

Viewing source code:

- **To open an HDL file:** double click on a file name to open the file in a new editor window.
- **To see the declaration of a signal or component:** double click on the signal or component to jump to the declaration in the HDL source code.

Expanding or hiding branches of a tree:

- Click or to expand or close a node. This will display but not open sub-nodes.
- To completely expand a node and all sub-nodes: right click on a node and choose **Expand Item** from the pop-up menu.

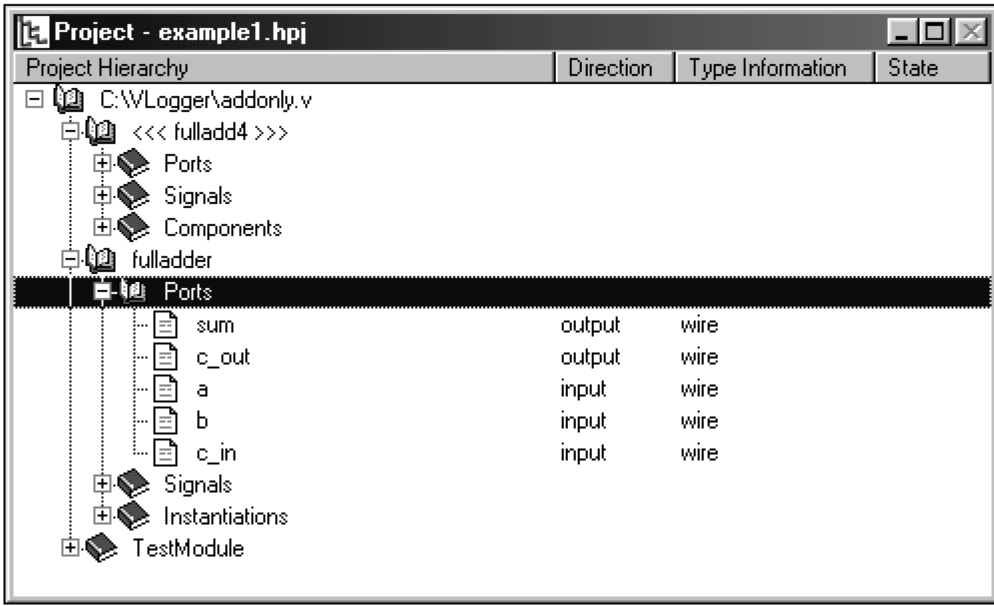


Figure 2.2: Project Tree control window

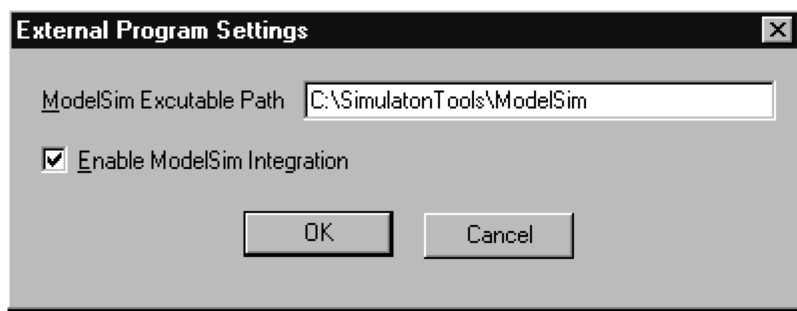
2.5 ModelSim Integration

If you use ModelSim to simulate either your Verilog or your VHDL code, you can enable the ModelSim Integration feature of TestBencher. The integration feature will cause ModelSim to launch with a library built and loaded for the project you are working with in TestBencher. This section describes how to enable ModelSim Integration as well as how to work with it.

Enabling ModelSim Integration

To enable ModelSim Integration:

- Select the **Options > External Program Settings** menu option. This will open the *External Program Settings* dialog.
- Click the **Enable ModelSim Integration** checkbox. Once this checkbox is enabled, the project you are working with (Verilog or VHDL) will be simulated using the ModelSim simulator. Note: If you want to use the Verilog simulator provided with TestBencher Pro, this checkbox can not be checked.
- Enter the path to the executable directory for ModelSim in the **ModelSim Executable Path**. If you are able to execute ModelSim from any command prompt (that is, if the executable path is a part of your path in your account profile), then you do not need to enter an executable path. Otherwise, this should be the directory where the **vlib.exe**, **vsim.exe**, and **vcom.exe** (ModelSim files) are located.



Simulating with ModelSim Integration Enabled

The ModelSim Integration feature uses the project directory for your TestBencher project as the working directory for ModelSim. The ModelSim library associated with the project will have the same name as the project and will be built in a subdirectory of the project directory.

To build the ModelSim library and invoke the ModelSim simulator:

- Follow the normal steps to generating a top-level test bench (see *Chapter 1: A Quick Start to TestBencher Pro* for the steps to test bench generation).

- Click the yellow `TB` button TB on the simulation button bar. This will copy the appropriate SynaptiCAD library files to your project directory and launch the simulator with the '-gui' switch and the '-do' switch. The .do file that will be invoked is the do file generated by TestBencher for the project. The .do file contains the commands to compile each of the appropriate library and source files, including the test bench file, into the project library being created. Upon successful compilation, it will load the `testbench` entity from the top-level test bench.

Once the design is loaded into ModelSim, three windows will be opened automatically – the Structure, Signals, and Wave windows. Additionally, all of the top-level signals from the `testbench` entity and the status signal for each timing transaction (see *Section 11.2: Status Signals and States* for more information regarding status signals) will be placed in the Wave window. These signals will allow you to monitor the MUT and the transactions during simulation.

Behind the Scenes: Files Used for ModelSim Integration

The ModelSim Integration feature creates two extra files (in addition to the normal files generated by TestBencher) in your project directory when you generate the test bench. These files are **<projectName>.bat** (**<projectName>.pl** if you are using a UNIX based system) and **<projectName>.do**, where <projectName> is the name of your project.

The script file (.bat or .pl) file will copy any base library files (such as syncad.v or syncad.vhd) into your project directory. it will then launch the simulator with the directive to call the do command for the generated .do file. These files remain in the directory so that you can execute the .bat file manually at a later time to launch the simulator and load the library without going through the TestBencher interface. Simply go to the project directory and enter the project name at the command line, or double click the .bat file in the project folder if you are working in a Windows environment. If you are working in a UNIX environment, you can execute the Perl script by typing "perl <projectname>.pl" at the command line.

2.6 Using a Third Party Simulator with TestBencher Projects

A command file can be generated for a project that will be simulated using a third party simulator. This file can be used when the third party simulator is invoked from the command line.

To generate the command file:

- Select the **Project > Project Settings** menu option.
- Click the **Generate Command File** button.
- Enter the filename for the command file in the *Filename* edit box. (The file extension will be ".vc" by default.)
- Click the **Save** button to save the file.

To use this file with a third party simulator, you then use the '-f' switch followed by a space and then the filename entered above to simulate the project.

For example, consider a project named test. If the generated command file is named 'test.vc' this file would be used with VeriLogger's command line simulator using:

```
vlogcmd -f test.vc
```

Chapter 3: Create Transaction Diagrams

Transaction diagrams are timing diagrams that represent reusable timing transactions (e.g., read cycle, write cycle, interrupt cycle). This chapter will discuss the components of a timing diagram and how they interact with TestBencher Pro.

When creating a timing diagram you will need to specify the following types of information:

- The **direction** and **type** (*Section 3.2: Direction and Type of Signals*) of each signal using the Signal Properties dialog box. For example SIG1 might have a direction of "output" and a type "integer". For **bi-directional signals** (*Section 3.3: Driving Bi-directional Signals*) the direction of transition can be set using the Driven flag in the Edge dialog (double click left on the edge to open this dialog box).

The following optional components can be added to each diagram as desired.

- **Variables** (*Section 3.6: Variables in Timing Diagrams*) can be defined for the timing diagram. A variable for a specific data bus gives you the option of using caller-specified state values for the data bus.
- Attach **Samples** (*Chapter 4*: Samples) to the diagram to generate **self-testing** (*Section 4.4: Self-testing Code from Samples*) code. This type of code monitors the simulation and will take action as specified by the user (e.g., provide a warning message when the output is not what is expected, or execute another timing diagram, as specified in the sample). Sample actions may be taken from the library set or they may be **user-defined** (*Section 4.5: Writing User-Defined Sample Conditions*).
- An **End Diagram Time Marker** (*Section 5.3: Time Markers for Specifying End Diagram*) can be used to indicate the exact end of the timing diagram's execution (good for clock synchronization).
- **Loop Time Markers** (*Section 5.4*: *Time Markers for Test Bench Loops*) can be placed in the diagram to indicate looping sections of the timing diagram.

You will be able to determine which of the optional components to add based upon the exact nature of your simulation.

3.1 Transaction Diagram Objects

Using the built-in timing diagram editor you will draw timing diagrams that represent reusable timing transactions (e.g., read cycle, write cycle, interrupt cycle). Timing diagrams are drawn using the Signal Button bar (see Figure 3.1). The first group of four buttons (left) are used to add signals, clocks, buses, and spacers to your diagram. The next group of six are used to add objects (i.e., markers and samples) to the diagram. The remaining buttons are used to create the waveforms for the signals.

Figure 3.1: Signal Button Bar

To add a Signal, Clock, Bus or Spacer:

- Left click the appropriate button in the first group of four. This will add the Signal, Clock, Bus or Spacer to the timing diagram.
- If you added a signal, clock or bus, double click the name of the new object to set the properties for that object.

Signals and buses need to have the direction and type specified. They may also need to have waveforms drawn.

To Draw a Waveform:

- Left click the type of state that you want to add in the group of 7 states on the right side of the Signal Button bar.

- Left click in the waveform section of the timing diagram editor to the right of the signal or bus name at the approximate time that you want the state transition to occur. This will place the transition in the waveform. Waveforms are built from left to right.
- Repeat the first two steps until you have completed the signal's waveform.

The other items that can be placed in the timing diagram will be discussed later in this chapter. The Timing Diagram Editor manual provides in depth information for the use of the Timing Diagram editor (an online version of this manual is provided with TestBencher).

3.2 Direction and Type of Signals

The generated test bench provides stimulus and monitors simulation outputs of the circuit that you are designing. In order to do this, the signals that will be exported to the test bench have to match the signals that exist in your designs. If the signals in the test bench are named the same as in your circuit model then the matching will be automatic. If the signal names are different you will have to specify the mapping in MUT instantiation in the template file.

The type and direction of signals must also be compatible between the test bench and the circuit under test. To change the type and direction of signals in the test bench:

- Double left click on the signal name to open the *Signal Properties* dialog box. Make the necessary changes using the drop down list box titled *Direction* and click the OK button to close the dialog.

TestBencher Pro supports special output directions that specify how these signals work when more than one timing diagram tries to drive them. The default direction is **output**.

The following describes direction options and their meanings with respect to the test bench.

- **Shared output -** continues to be driven after a timing diagram finishes. If more than one timing diagram attempts to drive the signal then the last timing diagram to drive wins.
- **Output -** is only driven while that particular timing diagram is active. When the timing diagram finishes, the signal is automatically tri-stated (Note that this setting will only work with data types that support tri-stating).
- **Persistent output -** continues to be driven after a timing diagram finishes. If more than one timing diagram attempts to drive a signal then a simulation conflict will occur. This will be either be resolved by the conflict tables of the simulator or will cause a simulation error to occur.
- **Input -** is what you expect the circuit under test to generate during simulation (these signals are inputs to the timing transactions, driven by the model under test). In the timing diagram, Sample parameters usually end on an input signal, indicating that the input signal should be checked for an expected value at that point on the signal.
- **Inout -** indicates that the signal is bi-directional (see section 3.3). Inout signals contain driven and un-driven signal segments. Driven segments act similar to signals of type **output**.
- **Persistent inout -** indicates that the signal is bi-directional. The driven segments of the signal act like signals of type persistent output.

3.3 Driving Bi-directional Signals

For Bi-directional signals you must choose the inout direction and then indicate which segments on the signal waveform are inputs.

Set the direction of the signal to either **inout** or **persistent inout**:

- Double left click on the signal name to open the *Signal Properties* dialog box. Choose one of the **inout** directions and click the **OK** button to close the dialog.

All the segments on the signal are assumed to have a direction of either output or persistent output and are colored black to indicate their direction. To change a segment to be an input segment (un-driven):

- Double left click on the right-hand edge of the input segment. This opens the *Edge Properties* dialog box.
- Uncheck the **Driven** check box. This indicates that the test bench does not drive the segment to the left of the edge. This segment will be an input to the test bench.
- Click **OK** to close the dialog or use **ALT-N** or **ALT-P** to edit other edges on the same signal. The segment for which you unchecked the driven flag should now be colored blue. If the segment is not blue then make sure that the signal's direction is set to either inout or persistent inout by double clicking on the signal name.

If you are working with a simulation, you will see that TeshBencher creates a second signal for any bi-directional signal you have entered. This created signal and its waveform appear in pink (or red) with the same name as the signal you created. The waveform for this signal will show you exactly what the signal is doing during the course of the simulation.

3.4 Creating Clocked Signals

By default, the events on a signal are delayed relative to the beginning of the timing diagram. However, signals can be made relative to a clock or to another signal's edges. The difference is that instead of edge transitions occurring at an absolute time, as with unclocked signals, the edge transition will occur relative to the edges of the signal (or clock) that they are clocked from. In this manner, the edge transitions use relative time instead of absolute time.

To create a clocked signal:

- Double click the signal name in the signal window to open the *Signal Properties* dialog (see Figure 3.2).
- Select a signal name from the **Clock** drop-down list box.
- Use the **Edge/Leve**l drop-down list box to specify which clock action will cause an event on the signal.
- Click **OK** to close the *Signal Properties* dialog.

The options listed under the **Edge/Level** list box allow you to decide which clock values will affect the signal, where **Neg** and **Pos** refer to the respective edges, **high** and **low** refer to the clock value, etc.

Figure 3.2: Signal Properties dialog

Editing Multiple Signals

Using this feature, you will often want all of the signals in a given timing diagram to be clocked in the same manner. To accomplish this, make sure the *Signal Properties* dialog is closed. Then:

- Select all of the signals you want to be clocked signals by left clicking the signal names in the signal window.
- Right click one of the highlighted signal names and select **Edit Selected Signal(s)** from the pop up menu.

- Select the clock signal name from the **Clock** drop-down list box.
- Specify which clock action will cause an event on the signal using the **Edge/Level** drop-down list box.
- Click **OK** to close the *Signal Properties* dialog.

3.5 Creating Continuous Setups and Holds

Continuous Setups and Holds can be created for any clocked signal in TestBencher Pro. This section will describe how to create a continuous setup or hold that will persist for the duration of the timing transaction execution.

To Create a Continuous Setup or Hold:

- Double left click on a signal name to open the *Signal Properties* dialog.
- Choose the Clocking signal from the **Clock** drop-down list box. The clocking signal can be any clock or signal in the timing diagram.
- Choose the type of edge or level triggering from the **Edge/Level** list box. For a Register circuit, choose **neg** for negative edge triggering, **pos** for positive edge triggering, or **both** for edge triggering. For a Latch circuit choose either low or high level latching.
- The **Set**, **Clear**, and **Clock Enable** are optional signals that model the set, clear, and clock enable lines of the register or latch. If "Not Used" is chosen for a line, then that line is not modeled. These lines can be active low or high and synchronous or asynchronous depending on the settings in the Advanced Register and Latch Controls dialog (see next bullet).
- The **Advanced Register** button opens the *Advanced Register and Latch Controls* dialog (see Figure 3.3) that determines how this individual register is generated. The global defaults can be defined using the **Options > Simulation Preferences** menu. This dialog controls the following options:
  - **Clock to Out**: Describes the delay from the triggering of the clock signal to a change on the output edge.
  - **Setup**: Describes the time for which the input must be stable before the clock-triggering event. If a **min/max** time pair is entered, **Setup** will use the **min** time. Any violations of this setup time will be reported to the simulation log file verilog.log, shown in the report window.
  - **Hold**: Describes the time for which the input must remain stable after the clock-triggering event. If a **min/max** time pair is entered, **Hold** will use the **min** time. Any violations of this hold time will be reported to the simulation log file verilog.log, shown in the report window.
  - **Clock Enable Active Low**: If checked, the clock will be enabled when the clock enable line is low. If unchecked, the clock will be enabled when the clock enable line is high.
  - **Set and Clear Active Low**: If checked, the set and clear lines will control the output when they are low. If unchecked, then the set and clear lines will control the output when they are high.

- **Set and Clear Asynchronous**: If checked, then the set and clear lines will control the output anytime they are active. If unchecked, the model is synchronous and an active set or clear line does not affect the output until the next clock trigger event.

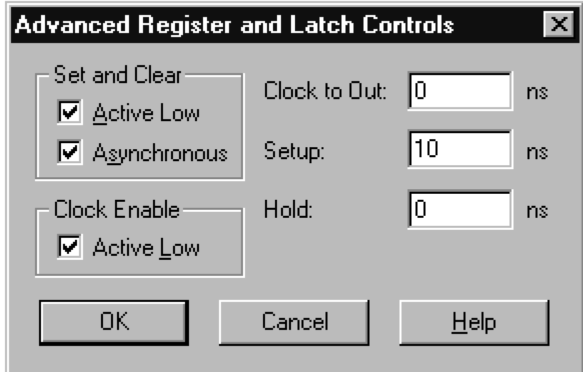


Figure 3.3: Advanced Register and Latch Controls

3.6 Variables in Timing Diagrams

Variables inside timing diagrams can assume a new value each time the diagram is called. This is convenient for timing diagrams that have data and address buses because each time the diagram is called new values can be passed into the timing diagram.

The procedure for specifying a state variable is similar to that of specifying a constant state value for a signal segment. State variables begin with two dollar signs, such as $$name. This name should be entered in place of the constant value for the signal segment. Variables are added to represent the value of a signal segment.

To add a variable signal segment:

- Double click on the segment of the signal in which you want to place the variable to open the *Edit Bus State* dialog.
- Type the variable name into the **Virtual** edit box. For example, $$data might be the name of the variable for the value of data bus.
- Click **OK** to close the dialog box.

You will pass the variable's value in to the timing diagram using the **Apply_***TransactionName* call in the template file. This is covered in *Section 8.3: Using Diagram Calls to Trigger Transactions.*

3.7 Delays and the Delay Properties Dialog

A delay specifies a fixed time between two signal transitions. Both the **min** and **max** values of the *Delay Properties* dialog are used in defining delays. One potential use for delays would be to create a handshaking mechanism between the MUT and a timing transaction.

The HDL code generated for a Delay is a process that will wait for a specific edge transition to occur (the forcing edge). Once this edge has transitioned, then the process will wait for the amount of time specified, and then the delayed transition will occur. The **Enable HDL Code** checkbox in the *Delay Properties* dialog must be enabled for the code to be generated.

To add a Delay to a Timing Diagram:

- Left click the **Delay button** to turn the button red.

- Left click on a transition to select it. For a delay this is the forcing transition. For a setup or hold this is the transition that will be monitored.
- Right click on the second transition to add a parameter between the first and second transitions. For a delay this is the transition that will be moved. For a setup or hold this is the control signal.

The delay will be drawn in the timing diagram between the two specified edges. A default name is assigned to the delay. If you want to change the delay's name, you can do so through the *Delay Properties* dialog.

The *Delay Properties* dialog is used to control the property settings of an individual delay. Open this dialog by:

- Place a delay in the timing dialog.
- Double left click the name of the delay. This will open the *Delay Properties* dialog.

The settings that you need to specify are:

- **Min:** sets a minimum time to be used for the delay.
- **Max:** sets a maximum time to be used for the delay.
- The **Is Apply Subroutine Input** checkbox allows you to generate ports between the test bench and the timing transaction with which to specify the values to use for the min and max settings of the delay.
- The **Enable HDL Code Generation** allows you to turn the code generation on and off without removing the delay from the timing diagram. This checkbox must be checked in order to produce any HDL code for the sample.

Note: The HDL code generation for all delays in a timing diagram can be disabled through the *Diagram-Level Test Bench Settings* dialog. See *Section 6.1: The Diagram Level Test Bench Settings Dialog* for more information on this feature.

The **HDL Code** button will open the *HDL Code For Delay* dialog. This dialog can be used to set a timeout for the delay. This timeout will cause the delay to timeout after the specified time, at which point the **Timeout Action** specified in this dialog will occur. The default **Timeout Action** is that the edge transition will occur.

Using the Min and Max Settings

The *Project Settings* dialog controls how the min and max properties will be used by TestBencher. The **Delay Settings** option in the *Project Settings* has three radio buttons:

- **Min**: if this option is selected, the min value provided in the delay settings will be used for the delay value.
- **Typical**: this option will average the min and max settings for the delay to determine the delay value.
- **Max**: this option will use the max value setting to provide the delay value.

If only one of the two settings has been given a value (min or max), the other setting will internally be given the same value.

3.8 Specifying a Timeout for a Delay

Delays can be given a default action in the event of a timeout to ensure that the edge transition occurs. This timeout can be set using the **HDL Code** button in the *Delay Properties* dialog. When this button is clicked, the *HDL Code for Delay* dialog opens.

The two items to be specified in this dialog are the **Timeout** value and the **Timeout Action**. The **Timeout** value is the time that a delay process will wait for the forcing edge transition to occur. If a timeout is specified, then the HDL code will wait for the forcing edge transition or for the amount of time specified, whichever comes first. The **Timeout Action** will determine the action that will be taken if the timeout occurs first. The default action is that the edge transition will occur.

Chapter 4: Samples

Sample parameters generate self-testing code in the test bench. Samples are normally used to monitor the signal values coming back from the model under test. Samples can test the value of a signal at a specific time or at a time relative to an event on another signal. The value that is the result of a sample can be exported to the top-level module. This could be used, for instance, to provide an input value for a state variable in another timing transaction or to determine if a specific timing transaction is to be executed or not.

4.1 Point Samples

Point samples are used to test or report the value of a given signal at a given time. The time selected for the sample to occur may be absolute or relative.

Absolute Samples

Absolute samples are set to sample at an absolute time. By doing this, the time that the sample is taken will not vary unless you specifically move it. To create an absolute sample, right click on the signal to be sampled to place the absolute sample.

Input Relative Samples

Samples that are set to sample within a set time after a diagram transition are called relative samples. Relative samples are created by left clicking on the relative edge and right clicking on a signal to be sampled. If a relative sample is relative to an input transition, the sample is called an Input-Relative Sample. When diagram execution gets to the beginning time of an input edge that has an Input-Relative Sample attached to it, the diagram suspends execution until the input transition occurs. This allows a sample to occur an exact time after an input transition, even though the time of the input transition may not be determined until simulation time. A relative sample is clocked off the edge of another signal, and so will move with that edge.

4.2 Window Samples

Window samples are used to test the value of a signal over an interval of time. This feature is useful for testing that the value of a given signal does not change over a specified time frame, for reporting the value of the signal as it changes during a specified time, or for verifying that the signal goes through a specified sequence of states.

Windowed samples may also be absolute or relative (see *Section 4.1: Point Samples* for more information about absolute and relative samples).

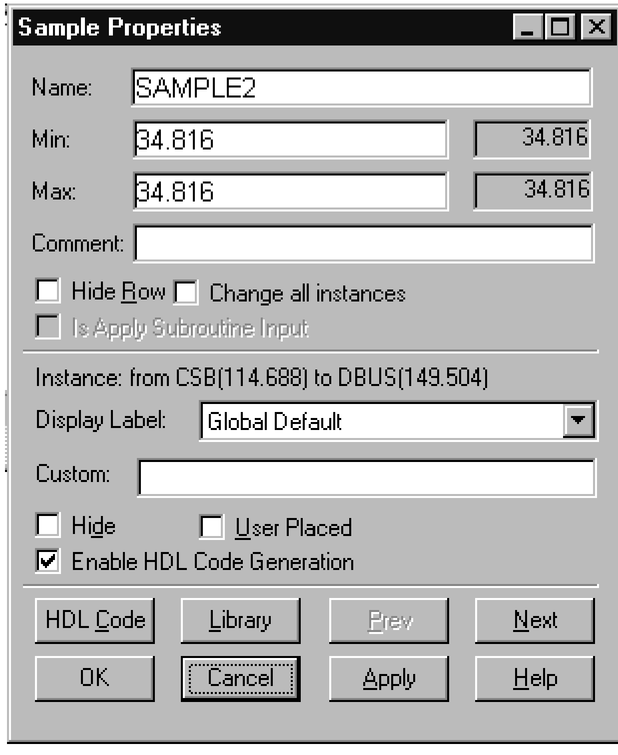
To create a windowed sample:

- Left click the sample button [Sample] on the *Simulation Button Bar*.
- If you want the sample to be relative, left click the edge that you want the sample to be relative to.
- Right click on the signal to be sampled. Note that you can place the sample at either the beginning time that you want the window sample to use or the ending time.
- Double click the sample name to open the *Sample Properties* dialog.
- Type the beginning sample time into the **Min:** edit box of the *Sample Properties* dialog.
- Type the ending sample time into the **Max:** edit box of the *Sample Properties* dialog.

4.3 The Sample Properties Dialog

The *Sample Properties* dialog is used to set the properties of a given sample. This dialog is discussed more thoroughly in the Timing Diagram Editor Help. This section will give a brief overview of the various options in the dialog.

- The **Enable HDL Code Generation** checkbox must be checked for any code to be generated for the sample.
- The **HDL Code** button will be discussed in *Section 4.4: Self-testing Code From Samples*.
- The **Min:** and **Max:** edit boxes are used to specify the beginning and ending times for a sample window.
- The **Display Label** drop down menu allows you to select what text will be displayed for the sample.
- The **Library** button will open the parameter library. This library can be used to modify certain properties. By double left clicking on a field in the library, the property dialog for the parameter indicated will open so that you can modify the value of that field.
- The **Prev** and **Next** buttons can be used to move from one parameter to the next (forwards or backwards on the parameter list).



The HDL code generation of Samples can be enabled and disabled using the diagram level test bench settings (see *Section 6.3: Enabling/Disabling HDL Code for Parameters*).

4.4 Self-testing Code from Samples

Sample parameters generate self-testing code in the test bench. Generally samples will end on a blue signal that is defined with a direction of input. The blue **input** signals are inputs to the transaction and are driven externally. In the timing diagrams they represent what output the user expects to see from the simulation.

To add samples to the timing diagram, depress the **Sample** button and use the left and right mouse buttons to add samples (See *Section 4.7: Creating New Sample Types*).

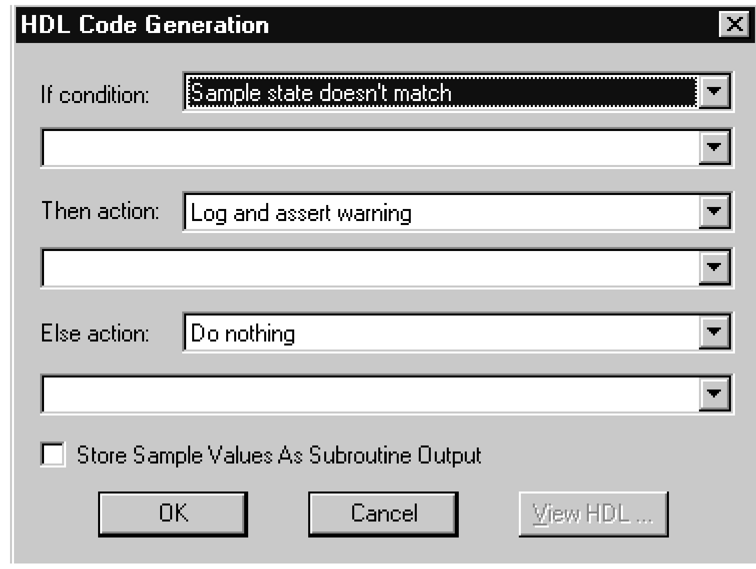
To choose the type of HDL code to be generated for a sample:

- Right click the segment of the signal that you want to place the **Sample** on.
- Double left click on the sample name to open the *Sample Properties* dialog. You can edit the sample window and start time using this dialog.
- Left click on the **HDL Code** button in the lower left hand side of the dialog. This opens up the *HDL Code Generation* dialog.
- Choose the type of condition to test for from the **IF Condition** drop down list box.
- Choose the action to take if the condition is true from the **THEN Action** drop down list box.
- Choose the action to take if the condition is false from the **ELSE Action** drop down list box.
- If you want the results of the sample to be exported to the top level module, check the **Store Sample Values As Subroutine Output** checkbox.

Below each listbox is an edit box that can be used to specify a User-defined action or condition. *Section 4.5: Writing User-Defined Samples Conditions and Actions* describes this in further detail.

Some of the Sample conditions and actions are as follows:

If conditions:



Sample State doesn't match: indicates that the state within the sample window should match the state that occurs during simulation.

User-defined condition: lets the user enter the exact VHDL or Verilog code for the condition into the edit box below the if conditions drop-down list box.

Actions for then and else blocks:

Do nothing: take no action if this branch is executed.

Log and Assert Warning: logs a warning to a file, and also prints a message to the screen.

Assert Warning: prints a message to the screen

End Diagram (set status to Done): Ends execution of this particular timing diagram. The Test Bench will continues to execute as if this timing diagram had normally ended.

Pause Simulation(Verilog only): Stops the entire simulation. Note that this option is not available for VHDL.

User-defined (action enter below): lets the user directly enter VHDL or Verilog code for the action into the edit box below the action drop-down list box. See *Section 4.5: Writing User-Defined Sample Conditions and Actions* for more information.

Note: The user can add additional sample conditions and actions by editing the sample script. See *Section 4.7: Creating New Sample Types*.

4.5 Writing User-Defined Sample Conditions and Actions

You can add language-specific (e.g. Verilog or VHDL) sample conditions to your diagrams to test for the occurrence of complex events using a User-Defined Condition. Typically this code will access or change the value of signals in the diagram at the sample time, but it is frequently desirable to define a sample condition in terms of previously executed samples. For example, you might wish to execute an action if three different previous samples have all been true. This can be accomplished by writing HDL code that accesses the values of variables which store information about previously executed samples.

User-defined **Sample** conditions and actions are entered in the *HDL Code Generation* dialog box with the following steps:

- Select the user-defined option for the **Condition**, **Then Action**, or **Else Action**, using the drop down list box provided.
- Left click in the text box immediately below the option you have set to be user-defined. Note: if you are using a condition or action that you have previously defined, use the drop down feature of this box and select the appropriate option.
- Type in the condition or action that you have defined.

- When you have finished defining/selecting the conditions and actions, close the *HDL Code Generation* Dialog by clicking **OK**.

Two sample variables are automatically generated in a test bench for each sample: a **Boolean variable** that indicates whether the sample condition was true or false and a **sample state variable** that contains the value of the sampled signal at the sample time. The Boolean variable is named as the sample name with _Flag appended to it and the sample state value is the sample name. For example, a sample named SAMPLE0 would create a Boolean variable named **SAMPLE0_Flag** and a state variable named **SAMPLE0**.

Below is an example of a user-defined sample condition written in VHDL that would cause a sample's then action to execute if three previous samples named SAMPLE0, SAMPLE1, and SAMPLE2 have all been true:

```
SAMPLE0_Flag and SAMPLE1_Flag and SAMPLE2_Flag
```

Sample variables can be used in the following places:

- the **condition of the current sample** can reference the sample's *state value variable*. The *Boolean result variable* of the sample cannot be referenced here, because the result of the condition cannot be buried inside the condition.
- **actions of the current sample** can reference both the sample's *state value* and *Boolean result* variables.
- **conditions and actions of samples that occur after the current sample** can use either the *state value* or *Boolean result* of the current sample. These are usually triggered from a User-specified action or condition.

Note: Sample variables are local to the diagram containing the sample and can only be referenced by HDL code contained in the diagram.

Example

Assume you have a diagram with three Samples (SAMPLE0, SAMPLE1, and SAMPLE2) where the first two samples test the values of two signals. To make SAMPLE2 true if both SAMPLE0 and SAMPLE1 are true, you would enter the following User-Defined Condition into the edit box below the IF condition:

SAMPLE0_Flag and SAMPLE1_Flag are Boolean variables inside the diagram that are given values when their respective Samples execute. Two other variables, SAMPLE0 and SAMPLE1, are also created by TestBencher. These variables hold the value (at the sampling time) of the signals being sampled and can also be used in User-Defined Conditions. For example, to test that the value sampled by SAMPLE0 is equal to the value sampled by SAMPLE1, enter the following User-Defined Condition for SAMPLE2:

Note: the types of the signals sampled by SAMPLE0 and SAMPLE1 must be the same, or you will get a type mismatch error when you compile your test bench.

4.6 Sample Values as Transaction Outputs

Sample values can be exported from a timing transaction to the top-level test bench. This allows you to add code to the top level test bench that will alter the execution sequence of timing transactions, for instance, based upon the sampled value.

To export a sample value from a timing transaction:

- Double click the sample to be exported. This will open the *Sample Properties* dialog.
- Click the **HDL Code** button to open the *HDL Code Generation* dialog.
- Check the **Store Sample Value as Subroutine Output** checkbox.

This will add the port for the sample value to the port list in both the timing transaction code and the top-level test bench. You will be able to reference the sample value signal directly at the top level test bench. The name used to access the value would be of the form:

```
TransactionName_SampleName
```

The following example (using the transactions found in Tutorial 5) shows how the sample value could be used to determine whether or not to call a certain transaction:

```
Apply_Tbread ('hF0, 'hAE); // stored sample outputs = Tbread_SAMPLE0
if (Tbread_SAMPLE0 === 'hAE)
   Apply_Tbwrite ('hF0, 'hAF)
```

4.7 Creating New Sample Types

Most of the time you will be using HDL code to specify the actions of Samples. There may, however, be instances when the use of Perl script is preferred. These scripts would define what HDL code is generated for the Sample. This section will explain how to add a user defined Perl script as a Sample type.

There are two steps to adding a new type of action or trigger condition for Samples:

- Adding the Perl subroutine to perform the code generation.
- Adding the subroutine name to TestBencher Pro so that the sample action or condition will be listed in the *Sample HDL* dialog box.

With TestBencher closed, write the Perl condition or action script:

- Open the **Sample.pm** file.
- Find the comment block that says:

```
Conditional Subroutine : UserDefinedConditionalSubroutine
```

or

```
Action Subroutine : UserDefinedActionSubroutine
```

- Copy the skeleton subroutine shown below the comment block.
- Rename the subroutine to represent your new sample type.
- Write your new subroutine. The $hdlCode variable must be set. This string will be the HDL code that is generated for the sample. $miscInfo is an optional variable that can be used for more code, such as a display or assert statement if desired.

Note: You can always examine the other functions in this file to help write your function.

- Save **sample.pm**

Next we will add the new condition or action type to the appropriate *HDL Code Generation* dialog drop down menu. Adding the new routine name to TestBencher Pro:

- In TestBencher Pro choose the **Export > Add or Execute Script** menu item to open a dialog of the same name.
- Choose the **Conditional** or **Action** radio button in accordance with the type of script you are adding.

- Type the name of the routine into the **Script Name & Command Line** edit box, and type a description into the **Menu Description** edit box.
- Click the **Add** button to add the script.
- Click **OK** to close the dialog box.

Chapter 5: Markers

Markers can be added to timing diagrams to specify specific actions to be taken by the transaction during execution. These actions can include signifying the end of a transaction, creating loops in the transaction, and inserting HDL code calling a subroutine into the transaction.

5.1 Adding a Marker to a Diagram

Every marker type provided by TestBencher Pro is placed on the diagram in the same manner. The property settings for the marker are used to specify the different types of markers and the portion of the diagram that the marker is attached to.

To place a marker in a diagram:

- Left click the **Maker** button [Marker] on the *Signal Button Bar*.
- Right click in the diagram area to place the marker. Note: This will place an absolute marker. This type of marker is discussed in *Section 5.2: Absolute and Relative Markers*.

Once you have placed the marker in the diagram area, double click on the marker line to open the *Marker Properties* dialog. The options in this dialog are used to specify how the marker will behave and what kind of action, if any, it will cause. The following sections will refer to this dialog.

The code being generated for a marker can be easily disabled using the Enable Marker HDL Code checkbox (see *Section 6.3: Enabling/Disabling HDL Code Generation for Parameters*) of the *Diagram Level Test Bench Settings* dialog. This allows you to generate your HDL code without extra code for the markers. By doing this, you will not have to remove the markers from the diagram for a clean code generation after model verification.

5.2 Absolute and Relative Markers

As with Samples (see *Chapter 4: Samples*), a marker can be absolute or relative. An absolute marker is attached to a specific time, while a relative marker is attached to a specific edge. Relative markers will move with an edge (i.e., if attached to a clock and the frequency of the clock changes), while an absolute marker will always remain at the time it was placed.

To add an absolute marker:

- Double click the marker line to open the *Marker Properties* dialog.
- Select the **Attach to time** radio button.
- Type the time you want the marker to be attached to in the **Attach to time** edit box.
- Click the **OK** button to close the dialog.

To add a relative marker:

- Left click on the marker button on the *Signal Button Bar*.
- Left click on the edge of the signal that you want the marker attached to. The edge will be highlighted with a green bar.
- Right click to place the marker relative to the selected edge.

5.3 Time Markers for Specifying End Diagram

TestBencher Pro can use a special type of time marker called an **End Marker** to indicate the execution end of a timing diagram. If there are no End Diagram markers then the longest non-clock signal will determine the end on the timing diagram. If there is more than one End Diagram Marker then the earliest one will determine the end of the timing diagram.

End Diagram Markers are especially useful for syncing up multiple timing diagrams that share the same clock. For example, it is convenient to place an End Marker at the exact ending transition of a clock cycle. This allows the timing diagram to be called several times in a row without getting an irregular clock period at the transition between the diagrams. (Note that this is only an issue when global clocks are not being used.)

To add time markers to the timing diagram, depress the **Marker** button and use the right mouse button to add a time marker.

To modify a time marker to be an end time marker:

- Double left click on the **marker line** (not on the marker name) to open the *Edit Time Marker* dialog.
- Choose **End Diagram** from the **Marker Type** drop-down list box.
- If the Marker is not located at the exact location you want it to be, then check the **Attach to Time** radio button and type the exact time into the edit box to the right of the radio button.
- Click the **OK** button to close the dialog. Note: if you want to name the marker, you may do so in the *Edit Time Marker* dialog by typing the name in the text box at the top of the dialog.

5.4 Time Markers for Test Bench Loops

TestBencher Pro can generate a test bench that loops continuously over a sequence of test vectors either forever or until a defined condition is met. These loops are set up using three types of Time Markers:

Loop Start: sets the beginning point for the loop and defines an exit condition if there is one.

Loop End: defines the ending point for the loop sequence.

Exit Loop When: can be placed between a Loop Start and a Loop End marker to allow a loop to be exited in mid-execution. On exiting the test vectors resume immediately after the End Loop Marker.

Every Loop Start marker must be matched with a Loop End Marker. Loops may be nested. The inner loop must be exited first. The loop is exited when the marker condition fails, or is reached, depending on which marker it is. For instance, an exit loop condition may depend on the value of a particular signal in the transaction. The conditions entered for each of the three loop markers must be a Boolean condition.

Note: Relative looping markers must be placed on the same edge transition type (positive or negative) of the same signal in order to avoid a compile error. The safest way to avoid a compile error is to ensure that the two markers are not relative. If you want the markers to be relative, make sure that they are attached to the same signal and that both are attached to either a rising or a falling edge of the signal.

To modify a time marker to be a loop marker:

- Double left click on the **marker line** (not on the marker name) to open the *Edit Time Marker* dialog.
- Choose one of the Loop types from the **Marker Type** drop-down list box.
- Click the **OK** button to close the dialog.
- Enter the Boolean condition appropriate for the **Marker Type** in the *Condition* text box below the **Marker Type** drop down list.

5.5 HDL Code Markers

HDL code markers are used to call user defined subroutines or library subroutines (see *Section 6.2: Including External HDL Code Library Files* for more information about using libraries with TestBencher).

To add an HDL Code marker:

- Add a marker to the diagram. (Left click the marker button on the *Signal Button Bar*, then right click in the diagram to place the marker.)
- Double click the marker line to open the *Marker Properties* dialog.
- Select **HDL Code** in the **Marker Type:** drop-down list box.
- Specify the subroutine name and any necessary parameters in the **HDL Code:** edit box.

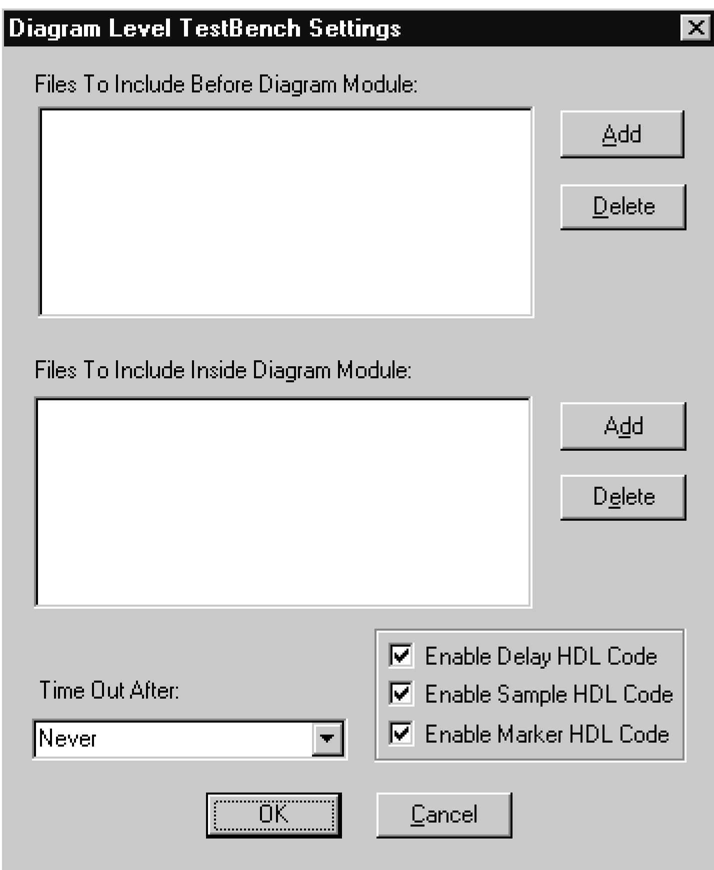
Note that in order for this marker type to work properly, you must ensure that the subroutine is defined and that the proper parameters are used. If the subroutine is in a library, then the library file must be included for the diagram using the diagram level test bench settings (see *Chapter6:Diagram Level Test Bench Settings*). If you want to define the subroutine for the current project only, then you can define it in the top-level test bench. You can define the subroutine in the diagram code, once it is generated, but if you change the diagram, then the code will be over-written, so this is not recommended

Chapter 6: Diagram Level Test Bench Settings

Certain properties can be set for each diagram within a project. These settings will help to determine exactly what code is generated for each diagram.

6.1 The Diagram Level TestBench Settings Dialog

Diagram level settings can be made individually for each diagram in a project. These settings are made through the *Diagram Level TestBench Settings* dialog.



To open this dialog:

- Open the diagram for which you will be changing the settings.
- Select the **Options > Diagram Level Test Bench Settings…** menu option. This will open the *Diagram Level TestBench Settings* dialog.

The fields of this dialog will be referred to in the remaining sections of this chapter

6.2 Including External HDL Code Library Files

External library files can include standard packages of subroutines or subroutines that are user-defined. These files can be included on the diagram level through the *Diagram Level TestBench Settings* dialog. HDL libraries can be included at one of two levels – either for use by the entire diagram (included before the diagram module), or inside of the diagram module.

Verilog Users:

Files that are selected to be included at the diagram level will be included using the `` `include `` statement. Any file that is included before the diagram module will be listed at the top of the file. Files that are included as a part of the diagram module will be listed after the module declaration and the register declarations. Note that the full path will be specified in the diagram. This will allow files that are not included in the library path to be included in the diagram.

VHDL Users:

Files that are selected to be included at the diagram level will be copied into the diagram code. This will ensure that the desired package is placed in the project while avoiding file parsing. Files that are included before the diagram module will be placed after the `library` and `use` statements at the top of the generated code (before the architecture definition). Files that are included inside of the diagram module will be listed inside of the architecture definition, after the `use` statements. Beginning and ending comment blocks will be placed around the file inclusion.

To Add an HDL Code Library File to the Diagram:

- Open the *Diagram Level TestBench Settings* dialog. (Select **Options > Diagram Level Test Bench Settings**)
- Determine if the file to be included will be placed before the diagram module or inside of the diagram module.
- Click the **Add** button to the right of the appropriate list box (the labels for the list boxes indicate which will hold the file to include before the diagram module and which will hold the files to include inside of the diagram module). This will open the *Add HDL Source File(s)* dialog.
- Use the *Add HDL Source File(s)* dialog to select the file(s) you will include.
- Click the **Open** button to add the file to the diagram and close the dialog box.
- Click the **OK** button to close the *Diagram Level TestBench Settings* dialog.

Although the code generation for Verilog and VHDL will treat the file lists from this dialog differently, the file selection process for the languages is the same in this dialog.

6.3 Enabling/Disabling HDL Code for Parameters

The *Diagram Level TestBench Settings* dialog has three check boxes that will allow you to toggle whether sample, marker and delay HDL Code generation is enabled.

Enable Delay HDL Code
Enable Sample HDL Code
Enable Marker HDL Code

If a situation arises in which you want the HDL code generation for samples, markers, delays, or all of them to be turned off, make sure that the appropriate check box (shown above) is *not* checked.

A Note About Disabling Sample HDL Code Generation:

If you have selected one or more samples to be an output of a timing transaction, you will not want to disable the HDL code generation for samples unless you hard code constant values within the diagram.

6.4 Specifying a Timeout for a Transaction

A transaction timeout will aid in the prevention of an endless wait condition occurring in a timing transaction. The time selected as a timeout duration is measured in diagram lengths for an individual diagram. For instance, a diagram whose entire execution should be complete in 150ns could have a timeout duration of 150ns, 300ns, 450ns, etc. This setting is selected in the *Diagram Level TestBench Settings* dialog. The default timeout length is set to **Never**, so the diagram will never timeout unless you specify a timeout duration.

To set the timeout duration for a given diagram:

- Open the *Diagram Level TestBench Settings* dialog. (Select **Options > Diagram level Test Bench Settings**)
- Select the desired timeout duration from the **Time Out After:** drop-down listbox.
- Note that if you do not want a timeout to ever occur for this diagram, select **Never** from the listbox.

Chapter 7: Editor Functions

TestBencher Template files are accessed through the editor windows. The editor windows are integrated parts of the simulation environment. Double clicking in the *Project Tree*, *Errors*, or *Breakpoints* windows will open an editor and display the relevant source code. The editor windows are also used to display the current execution line for **single-step** debugging.

All editor windows provide color-syntax editing, search, single-click breakpoint placement, goto lines, and font control. The simulator automatically recognizes when a file is modified in an editor window, and will warn you when it needs to be saved.

7.1 Opening, Saving, and Creating New Source Code

TestBencher source code files are opened and saved using the **Editor** menu options. When TestBencher starts a new simulation, it checks for any unsaved files and automatically prompts you to save them.

To open an existing source code file use one of the following methods:

- Double click on the *filename* in the *Project Tree* control,
- OR, choose the **Editor > Open HDL file** menu option.

To create a new source code file:

- Select the **Editor > New HDL file** menu option.

To save an open source code file:

- Select the **Editor > Save HDL file** menu option to open a *Save* dialog. By default, TestBencher file names have an extension of **.v or .vhd**.

To close the editor window and save the source code:

- Choose the **Editor > Close** menu option. If the file has been altered, you will be prompted to save the file.

7.2 Displaying or Finding a Specific Line of Code

Most TestBencher display windows are linked directly to an editor window, making it easy to view relevant source code.

Below is a list of windows and buttons that can be used to jump directly to a particular line of code.

- The *Errors* tab in the *Report* window displays compilation errors. Double click on an error to open an editor and display the line where the error was found.
- The *Breakpoints* tab in the *Report* window displays all the breakpoints in the current project. Double click on a breakpoint to open an editor and display the line where the breakpoint is located.
- The **Goto** button, the magnifying glass on the simulation bar, opens an editor starting at the last line executed. This button is only active during a simulation.
- The *Project Tree* control displays all signals, ports, and modules used in the project.

There are also several ways to search for line numbers or character strings in a file. Use the following keyboard combinations inside an active editor window to locate source code.

To move to a specific line in your code:

- Press **<Ctrl> + G** to open the *Jump To* dialog. Enter the line number to view.

To search for a character string:

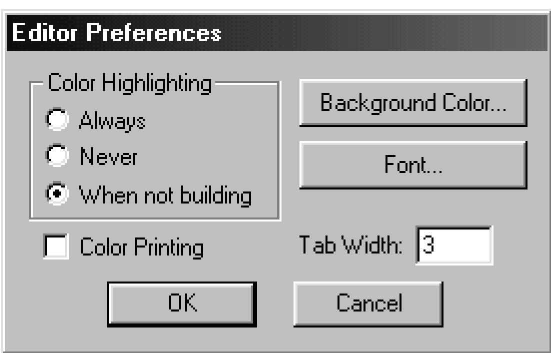
- Press **<Ctrl>+F** to open the *Search* dialog. Enter the character string to locate.
- To perform another search, press the **<F3>** key.

7.3 Using the Editor Preferences Dialog

The *Editor Preferences* dialog controls the editor options. This information is stored inside the project **HPJ** file.

To open the *Editor Preferences* dialog:

- Select the **Editor > Editor Preferences** menu option.



There are several sections in the *Editor Preferences* dialog.

- The **Color Highlighting** radio buttons determine when color syntax editing is active. By default, the **When not building** option is selected so that the color syntax editing does not slow the build time of large projects.
- The **Color Printing** checkbox prints the source code in color. If unchecked, all code is printed in black.
- The **Background Color** button opens the Color dialog and lets you set the background color of the editor windows.
- The **Font** button opens the Font dialog. In the Font dialog you can set the font type, size, and color of the text in the edit windows.
- The **Tab Width** edit box sets the number of spaces that a tab character will generate. The default setting is 3 spaces, or it can be set to match the tab width of an external editor.

7.4 Editor Cursor Commands

The Report Window displays are full-featured editor windows. Listed below are the keyboard and mouse commands supported by the editor window.

Key	Purpose
Left/right arrow keys	Moves the cursor one space left or right
Up/down arrow keys	Moves the cursor one line up or down
Page Up	Moves the cursor one page up
Page Down	Moves the cursor one page down
Home	Move to the beginning of the current line
End	Move to the end of the current line
Ctrl+Up	Move to the beginning of the previous line
Ctrl+Down	Move to the beginning of the next line
Ctrl+Right	Move to the start of the next word
Ctrl+Left	Move to the start of the previous word

Ctrl+Page Up	Move to beginning of file
Ctrl+Page Down	Move to end of file
F1	Opens this help file
F4	Print from window
Shift+F4	Print options
F5	Search for a word or phrase
Ctrl+F5	Insert blank line before current line
F6	Replace
Shift+F9	Delete current line
Alt+J	Join lines (removes the next carriage return)
Alt+C	Block copy (duplicates selected text, and inserts it after selection)
Alt+Backspace	Undo
Alt+3	Protected text toggle (selected text cannot be modified while protected)
Alt+7	Change the color of the selected text
Ctrl+T	Tab
Ctrl+A	Select all
Ctrl+X	Cut
Ctrl+C	Copy
Ctrl+V	Paste
Ctrl+Z	Undo
Ctrl+Y	Redo
Ctrl+F	Search
Alt+S	Find in files (multiple file search)
Ctrl+G	Jump to line #

7.5 Using an External Editor

External editors can be used with TestBencher. If you use an external editor, make sure it is configured to detect when other programs externally modify a file. While simulating and debugging in TestBencher, you will want to use the internal editors to make quick fixes to the code so you can continue simulating. If your editor does not detect that you have modified a file, it may overwrite your fixes.

Chapter 8: Working with Template Files

When TestBencher creates a test bench, it generates the top-level control file of the test bench from a user-specified template file. TestBencher ships with several generic template files. When you first open a template file, you will find skeletal code, commenting that explains what each section of the template file will do, and substitution macros that are expand to HDL code when the test bench is generated (see *Understanding TestBencher Macros* below). The elements that are common to most test benches are included in the template file macros. The template file also contains the code that handles the sequencer process for the test bench. The sequencer process contains diagram calls that control the sequence in which diagram transactions execute.

The generic Verilog template files are named **isotbench.v** and **tbench.v**. The VHDL template files are called **isotbench.vhd** and **tbench.vhd**. You can modify these files to add additional VHDL or Verilog code that you would like to include in all the test benches you generate. It is recommended that you read *Section 8.7: Adding HDL Code to Template Files* before adding your own HDL code to template files.

Understanding TestBencher Macros

TestBencher generates the test bench code by use of macros found in the template file. Macros are used to generate top-level code that varies for each test bench. These macros consist of a pair of commented keywords. An example of a macro keyword pair is:

```
-- $ComponentInstantiationsForAllDiagrams
-- End $ComponentInstantiationsForAllDiagrams
```

During test bench generation, these commented lines will expand to include the component instantiations for the diagrams contained in the project. An example expansion would be:

```
-- $ComponentInstantiationsForAllDiagrams
tbglobal_clock_0 : tbglobal_clock port map (SweepTest2.tbglobal_clock.tb_trigger,
                                            SweepTest2.tbglobal_clock.tb_status,
                                            GClock);
-- End $ComponentInstantiationsForAllDiagrams
```

For this example, there is one component instantiation that is generated from the timing diagrams in the project.

Note: Every time the test bench is generated, the code contained between the two lines of the macro keyword pair is overwritten. This is important to note because you will need to make sure that any HDL code that you add to the template file (or test bench) is added before or after a macro and not between the two lines that comprise the macro.

This chapter describes how to add a template file to the project, add calls to the transactions, and how to instantiate the model under test.

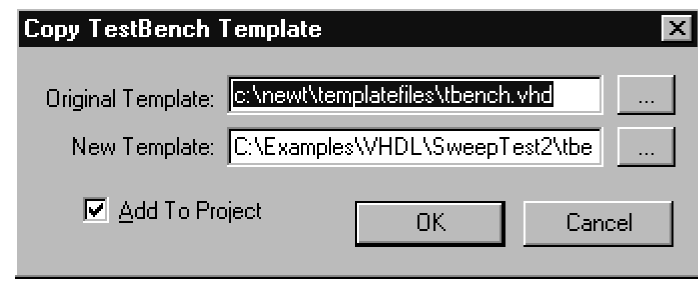
8.1 Adding the Template to a Project

Each project must have at least one template file in order to generate a test bench.

To create a top-level template file for a project:

- Right click in the Project window and choose **Copy TestBencher Template File** from the context menu. This opens a dialog box with the same name
- Check the **Add to Project** checkbox in the bottom left of the dialog box to ensure that the template you choose will be included in the project.

- Choose the template file that is appropriate for you. Be sure that the template file matches the language that you are working with (e.g., **isotbench.v** or **tbench.v** for Verilog (*.v)). .



TestBencher Pro is shipped with several default template files for each language and users can design their own. The **isotbench** template file generates the waveforms related to the MUT, while the **tbench** template file generates all of the waveforms that will let you see the exact status of each transaction during the simulation. The **tbench** is the default test bench template file because it provides more information and the file is simpler.

- Enter the existing template file name for the template file that you have chosen in the **Original Template** edit box or use the Browse (…) button to find one on the hard drive.
- Enter a file name for your template file (this file will become your test bench) in the **New Template** edit box. You can also use this edit box to specify the full directory path for the template file.
- Click **OK** to copy the template and add it to the Project window. Notice that the template file appears in the project window.

This template file will have comments in the code that will help you find the proper places to instantiate your MUT (see *Section 8.2: Instantiating Your Model Under Test (MUT)*) and add your diagram calls (see *Section 8.3: Using Diagram Calls to Trigger Transactions*).

8.2 Instantiating your Model Under Test (MUT)

The instantiation of the MUT serves to connect the top-level test bench to the Model Under Test. Open the template file and locate the place to instantiate the MUT in the code:

- Double click on the template file in the project window. This opens an *Edit* window and displays the source code of the template file.
- Scroll through the template file and locate the comment that contains the following text:

```
Instantiate Models Under Test after this comment block:
```

Verilog Users:

For Verilog MUTs, if you have already compiled your MUT, you can optionally use the project window to instantiate the MUT instead of manually typing the instantiation.

- Move the cursor to the place in the template file where you want to instantiate the MUT.
- Next, in the Project Window right click the MUT and select **Instantiate in TBench** from the pop-up context menu. This will insert the MUT instantiation code in the template file beginning at the point where you placed the cursor.

An example of instantiation of a MUT in Verilog is:

```
MyModelUnderTest mut1(CSB, WRB, ABUS, DBUS[15:8]);
```

VHDL Users:

If you are using VHDL, do the following to instantiate the MUT:

- Type the MUT instantiation into the template file. It will probably look something like this:

```
mutMSB: tbsram port map(CSB, WRB, ABUS(7 downto 0),
```

```
                        DBUS(7 downto 0));
```

- Next, scroll up in the code to the line

```
-- 3) Declare component for Models Under Test after this comment
```

Below this comment block is where you define the component portion of your module. Use the following example as a model:

```
component tbsram port( CSB : in std_logic;
                        WRB : in std_logic;
                        ABUS : in std_logic_vector(7 downto 0);
                        DATABUS : inout std_logic_vector(7 downto 0)
                      );
end component;
```

8.3 Using Diagram Calls to Trigger Transactions

Diagram Calls allow you to execute timing transactions from the test bench using 'Apply' function calls. Timing transactions can be executed in different manners, for example with non-concurrent single-run sequencing or with concurrent looping. *Section 8.5: Controlling the Execution Mode of a Transaction* will give you more information regarding the execution modes of timing transactions.

Apply diagram calls are made in one or more sequencer processes which dictate the order in which transactions take place. By default a top level template file contains one sequencer process, but you can add more if you have sets of transactions that need to execute completely asynchronously.

Inserting a Diagram Call

Use the *Insert Diagram Call* dialog to add timing diagram apply statements to the template file by:

- Scroll down in the HDL code until you find the sequencer code. You will see a comment in the code that looks like:

```
Add apply calls for timing diagram transactions after this comment
```

- Left click in the template file just below this comment. This is the segment of code where timing transactions will be executed.
- Select the **Editor > Insert Diagram Calls…** menu option to open the *Insert Diagram Calls* dialog. This dialog contains a list of available 'Apply' statements, one for each diagram in the project. If you have not added any diagrams to your project the list will be empty.
- Choose either **Run Once** or **Loop Forever** for the **Run Mode**, and choose either **Wait for Completion** or **Concurrent Apply** for the **Wait Mode**. These modes are set using the appropriate radio button for the desired option (see *Section 8.5: Controlling the Execution Mode of a Transaction* for more information about execution modes).
- Double click on an **Apply_*TransactionName*** statement to insert a call into the template file.

Note: The *Insert Diagram Calls* dialog is a modeless dialog so you can leave it open while you perform other actions. Double clicking on additional Apply statements causes those statements to be added below the last statement added.

- If any of the applied transactions contain variables, then edit the template code to provide values for variable names (see *Section 8.4: Setting State Variables in Diagram Calls* for more details). In the example Apply statement below, a value of three is assigned to `stateVar`.

```
// Apply_verySimpleCyclic(stateVar);
Apply_verySimpleCyclic(3);
```

8.4 Setting State Variables in Diagram Calls

Input signals and state variables in a diagram can be mapped to constant data values in a Diagram Call. Replace the variable name with the desired value in the diagram call to map the constant data value to the variable.

For example, adding a call to the timing transaction Tbread (used in Tutorial 5) would place the following lines of code in the test bench:

```
// Apply_Tbread(addr, data);
Apply_Tbread(addr, data);
```

In order to map the values, replace `addr` and `data` with the desired constants in the function call:

```
// Apply_Tbread(addr, data);
Apply_Tbread('hF0, 'hAE);
```

8.5 Controlling the Execution Mode of a Transaction

There are two modes that can be set for each transaction, **Run Mode** and **Wait Mode**. These two modes are used to control how transactions execute. They are generally used to create global clocks, persistent glue logic, or setup and hold checkers that operate for the duration of a test bench. The Wait Mode is also useful for simultaneously starting several transactions.

The two modes are set in the Apply statement that starts a transaction. The *Insert Diagram Call* dialog is used to specify the settings for the two modes (or you can manually type the appropriate apply call in your template file). The particular settings that are chosen will determine exactly which subroutine for the transaction will be called.

The example in the image below would be used to include the timing transaction `Tbread` using a **Run Mode** of **Loop Forever** and a **Wait Mode** of **Concurrent Apply**.

Run Mode

There are two modes that the Run Mode can be set to:

Run Once: This mode sets a transaction to run once. Any repetition of the transaction in this case must come from one of the other two methods of looping (see *Section 10.3: Loops in Transactions*). This is the default setting for the Run Mode. Nothing needs to be entered in the Apply statement for this mode to take affect.

Loop Forever: Used to create persistent logic. This mode causes the transaction to loop continuously until the simulation is complete.

Wait Mode

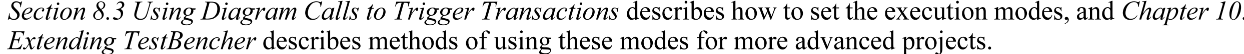
Wait Mode also has two modes:

Wait for Completion: This mode will cause other transactions to wait until this transaction has completed before allowing other transactions to begin execution.

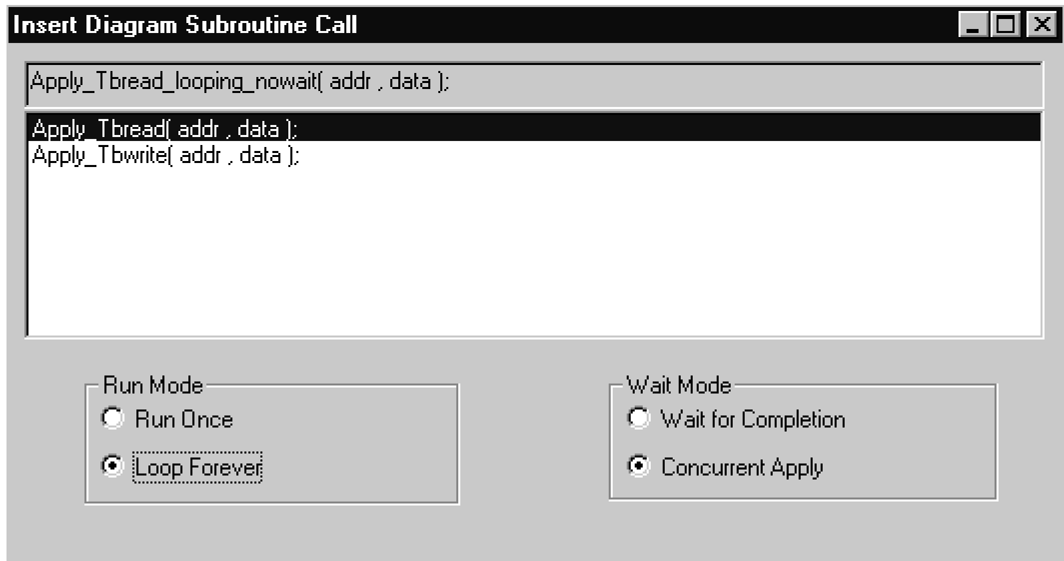
Concurrent Apply: This mode will allow other transactions to run while the current transaction is executed. This mode is ideal for persistent logic.

The Apply function call that would appear in your template file using the example in the image below would look like this:

```
//Apply_Tbread_looping_nowait(addr, data);
Apply_Tbread_looping_nowait(addr, data);
```



Section 8.3 Using Diagram Calls to Trigger Transactions describes how to set the execution modes, and *Chapter 10: Extending TestBencher* describes methods of using these modes for more advanced projects.



8.6 Aborting a Timing Transaction

The Abort Call

The **Abort_***TransactionName* diagram call is used to end a transaction that was applied using the 'Loop Forever' Run Mode. This is useful for ending a looping transaction conditionally, or for ending simulations that contain global clocks, for instance. The subroutine is provided to allow you to prevent an endless loop where one is not desired, and to provide a method of ending the simulation once all other transactions have completed. Simply place a call to this subroutine for the appropriate transaction after the other diagram calls have been placed in the test bench file.

The example below demonstrates the use of the Abort call to end a simulation that employs a global clock:

```
// Apply_GlobalClock_looping_nowait;
Apply_GlobalClock_looping_nowait;
// Apply_Counter;
Apply_Counter;

Abort_GlobalClock;
```

In the example a global clock is set to execute for the duration of the simulation. The second transaction call would cause a counter transaction to execute. Once the counter has completed its execution, the global clock is aborted, ending the simulation.

8.7 Adding HDL Code to Template Files

HDL code can be added to the template files for each project, or you create your own custom template file. Before making any changes to an original copy of a template file provided with TestBencher, it is highly recommended that you make a back-up copy. It is also important to remember that when the test bench code is generated the macros that expand will overwrite any code that is between the keyword pair that comprises the macro (see *Understanding TestBencher Macros* in the introduction to this chapter).

The most common place to add HDL code to a template file or test bench would be in the sequencer process. This process is the area where diagram calls are placed in the template file (see *Section 8.3: Using Diagram Calls to Trigger Transactions*). This allows you to place conditions, for example, on whether or not a diagram call is made. An example of adding a loop at the test bench level using HDL code can be found in the advanced TestBencher Tutorial: *Performing a Sweep Test*. In the following example from the tutorial, a **for-loop** is added to the sequencer process so that a write cycle followed by a read cycle is applied to the model under test six times. A variable is declared for the loop counter, and the **for-loop** is placed around two diagram calls as seen in this simple Verilog code segment:

```
// Sequencer Process
real delay0; // delay0 will serve as the index and the delay value
initial
begin
  for (delay0 = 32.0; delay0 > 5.0; delay0 = delay0 - 5.0)
  begin
    // Apply_Tbwrite( addr , data , $realtobits(delay0_min) );
    Apply_Tbwrite( 'hF0 , 'hAE , $realtobits(delay0) );

    // Apply_tbread( addr , data , $realtobits(delay0_min) );
    Apply_tbread( 'hF0 , 'hAE , $realtobits(delay0));
  end
  Abort_tbglobal_clock;

$fclose(logfile);
end
```

Chapter 9: Generating the Test Bench

This chapter discusses how to generate the top-level test bench from the template file, errors during test bench generation and simulation of the test bench. It also discusses how to set default preferences for the top-level test bench. These default preferences will allow you to set the default language, for instance, for future sessions with TestBencher.

9.1 TestBencher Pro Design Flow

There are three main steps to creating a test bench with TestBencher Pro.

- **Draw a series of timing diagrams** that represent reusable timing transactions (e.g., read cycle, write cycle, interrupt cycle) using the built-in timing diagram editor. See *Chapter 3: Create Transaction Diagrams* for more information.
- **Copy and Edit a Template File** that calls the timing diagrams and describes the interface to the model under test. The diagram calls (see *Section 8.3: Using Diagram Calls to Trigger Transactions*) in the template file control the sequence in which the timing diagram transactions are applied to the simulation model.
- **Generate test bench** using the **Export > Generate Test Bench** menu item. The generated test bench (see *Section 9.2: Generate the Test Bench*) contains a component for each timing diagram and a sequencer component that triggers each timing diagram in the order specified by the template file. The generated test bench uses only standard VHDL/Verilog constructs so it can be used with all compliant VHDL/Verilog simulators.

9.2 Generate the Test Bench

Once the timing diagrams and the template file are finished, TestBencher Pro will generate the test model by expanding macros in the project's template file.

To Generate the Test Bench:

- Choose the **Export > Generate Test Bench...** menu option. You will be prompted to save any files that are untitled.

Before TestBencher generates the top-level test bench, it verifies that the HDL files that were generated for each of the timing diagrams in the project are up-to-date. The diagram HDL files are first created when the timing diagrams are added to the project and are updated each time thereafter when the diagram is saved. Once TestBencher has verified that these files are current, it will generate the top level test bench by expanding a series of macros in the template file.

The template file contains a series of macros that will be expanded during the test bench generation. These macros will be comments in the language that you are working with to ensure that they do not interfere with the HDL code. The name of the macro being expanded will begin with a dollar sign ("`$`"). An example of a macro is:

```
// $SignalsForAllDiagrams
// End $SignalsForAllDiagrams
```

A comparison of the original template file with the generated test bench will show that macros in the template file have been expanded in the generated file. For example, the macro above would expand to the following code segment for the sram model in Tutorial 5:

```
// $SignalsForAllDiagrams
wire  CSB;
wire [7:0] ABUS;
wire  WRB;
wire [7:0] DBUS;
// End $SignalsForAllDiagrams
```

Each time the test bench is generated the HDL code contained within the macro statement is deleted and regenerated. Any HDL code outside of the macro statements will not be affected. If you want to add your own HDL code to a template file you will need to make sure that it does not appear within a macro statement so that it does not get over-written. A couple of examples for code that you may wish to add would be a loop for a specific diagram call or a randomizer. Any changes that you might make that would affect the code within the macros (e.g., changing the size of a wire declaration in the above example) should be handled at the diagram level. This will ensure that the port size for the signal matches in the timing transaction and in the top-level test bench.

9.3 Errors During Test Bench Generation

Errors and warnings that may occur during test bench generation will be reported in the test bench log file, "`waveperl.log`". During file generation, errors and warnings are encountered as they occur. At the end of the test bench generation, the log file will report a list of the total number of warnings and errors that have occurred. The messages that are logged for each error and warning will be as detailed as possible, offering potential explanations for problems where possible.

The "`waveperl.log`" file will be displayed in the Report window. This window has a tab on the bottom that is labeled "waveperl.log". Click this tab to view the log file. (Note: other files and reports that are available are listed under the other labeled tabs in the Report window.) It is important to examine the log file after each test bench generation to see if there were any problems during the code generation.

Possible errors include illegal syntax in a template file and type mismatches between diagram port connections and top level test bench signals. TestBencher will also warn you of potential problems, such as two distinct signals within the same timing diagram using identical names but different directions (this may not always be a problem, but if the signal is being used as an output of the diagram, it may cause confusion).

9.4 Simulating the Test Bench

The test bench that you have created can now be simulated by taking all of the files produced by TestBencher Pro and placing them into a simulator. Verilog users can use the information in this section to simulate the test bench using TestBencher Pro.

Files Needed for Test Bench Simulation:

You will need the following files to simulate your generated test bench:

- The HDL files for the MUT.
- One HDL code file for each timing transaction.
- One top-level HDL test bench file (the expanded template file).

Using ModelSim to Simulate a Verilog or VHDL Test Bench

If you use ModelSim to simulate your projects, please refer to *Section 2.5: ModelSim Integration* for more information.

To Simulate a VHDL Test Bench (other than ModelSim), you must ensure that the generated test bench and any models you have written are placed into your VHDL work directory. Generally you will want to configure TestBencher to generate the test bench files directly into your work directory so that you can simulate the files immediately without having to copy them manually (or you can write a Unix script file or DOS batch file to perform this action).

Note: TestBencher assumes that your VHDL compiler includes pre-written libraries for packages std_logic_1164, std_logic_textio, and std_logic_arith. If you do not have these library files, you will need to acquire them. (SynaptiCAD can provide these files – contact our sales department.) Analyze (compile) these files and run your simulator.

To Build and Simulate a Verilog Test Bench Using TestBencher Pro:

TestBencher contains a full Verilog simulator that you can use to run your simulation.

Build the simulation by:

- Press the yellow `TB` button (the first of the simulation buttons to the right of the **Debug Run** button).

There are three ways to run the simulation within TestBencher:

- Press the **Run** button (large green triangle) on the simulation button bar, making sure the **Sim Diagram & Project** option is selected.
- Choose the **Simulate > Run** menu option, OR
- Press the **<F5>** key.

Once a project is simulated the waveforms will be displayed in the Diagram window. If you do not want to continue to watch a particular signal, left click on the signal name in the Diagram window and press the **<Delete>** key. A log will be maintained during the simulation ("verilog.log"). This log is shown in the report window. Any notes, warnings and errors reported by the simulator will appear in this log.

To Simulate a Verilog Test Bench Using Another Simulator, follow the standard procedure for your simulator using the files listed above. You may also need to add the appropriate wavelib library file to the files to simulate if your diagrams used the Signal Properties dialog to create register or latch components.

9.5 Test Bench Preferences Dialog (options)

The *Test Bench Preferences* dialog sets up defaults for how test bench projects and timing diagrams are created (WaveFormer Pro also uses the options in the Signal HDL Defaults section of this dialog).

To Open the Test Bench Preferences Dialog:

- Choose the **Options > Test Bench Preferences** menu item.

Signal HDL Defaults Section (also used by WaveFormer Pro): When code is generated for a signal the test bench must instantiate a signal with a specific direction and type. Each signal's direction and type can be individually set in the *Signal Properties* dialog by double clicking on the signal name.

Direction: The default is **shared output**.

VHDL Type: The default is **std_logic**.

Verilog Type: The default is **wire**.

Project Defaults Section

Language: TestBencher Pro supports Verilog, VHDL87 and VHDL93. The default is **Verilog**.

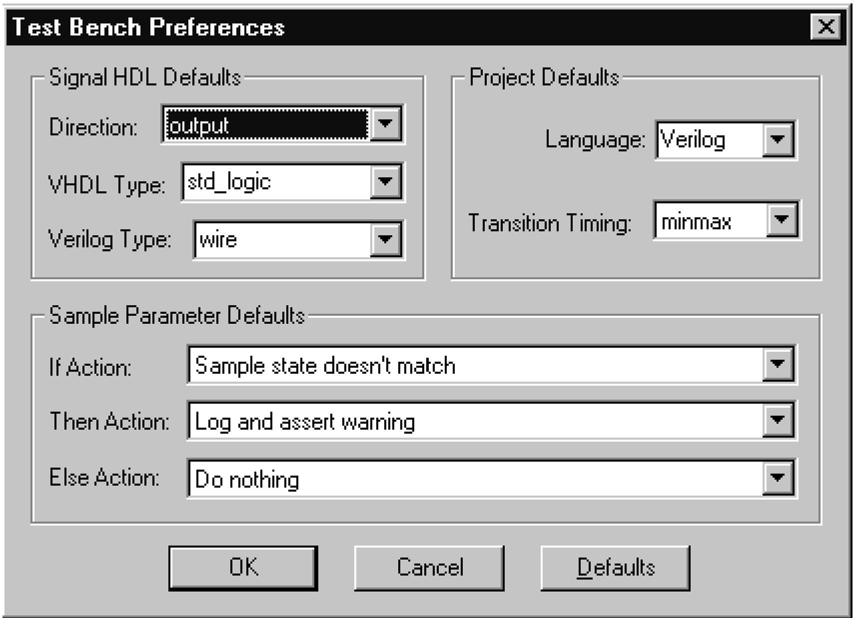
Transition Timing: When a signal edge has an uncertainty region caused by delay, TestBencher Pro uses either the minimum or maximum transition to generate the test bench. The default is maximum transition timing.

Sample Parameter Defaults Section: Samples generate the self-testing code in the test bench. The If Action is the condition that will be monitored during execution. If the condition is true then the "Then Action" will be executed. If the condition is false then the "Else Action" will be executed. When no action should be taken choose "Do nothing" as the action choice.

If Action: The default is **Sample State does not match**.

Then Action: The default is **Log and Assert Warning**.

Else Action: The default is **Do nothing**.



Chapter 10: Advanced Techniques

This chapter will discuss a few advanced techniques that can be used with TestBencher Pro. The first section discusses test bench level glue logic; that is, logic that will persist throughout the simulation. The next section will describe how to create and use a global clock that will run continuously and how to abort the clock once the rest of the test bench has finished execution. The last topic of the chapter describes the various methods of creating looping transactions using TestBencher.

10.1 Generating Test Bench-Level Glue Logic

TestBencher can generate HDL code for persistent glue logic that will operate throughout the entire simulation. One example of when this is useful when you need to model the boolean equations for a bus arbiter that interacts with your model under test. Persistent glue logic is created by adding boolean equations to a diagram (see *Section 12.1: Generating Waveforms from Boolean Equations* in the Timing Diagrammer manual) and applying the diagram as a looping, concurrent transaction so that it runs for the entire duration of the test bench.

For example, if you want to verify that a setup time is met between a data signal and a clock signal for the entire duration of the test bench, you would want to create this sort of glue logic. Assume the model under test generates the data signal, so its edge transition times are not known until runtime (the clock could be generated by the test bench or by the MUT). To verify that the setup time is met, you can create a persistent flip-flop that is only used to verify the setup time. The data signal generated by the MUT needs to be an input to the diagram containing the persistent flip-flop so that the flip-flop can monitor the data signal's transitions. The clock must either be generated by the diagram or it must be an input to the diagram. The continuous setup and hold is then created between these two signals as described in *Section 3.5: Creating Continuous Setups and Holds*.

10.2 Creating a Global Clock

Global clocks are often used to provide a method of synchronization for multiple diagram transactions and to drive the MUT. This type of clock can be created in TestBencher by setting up a diagram with a clock that will run continuously throughout the simulation while other transactions are running. These two things can be accomplished by use of the mode settings in Apply statements.

For a timing transaction containing a global clock the Run Mode should be set to 'Loop Forever' to ensure that the transaction loops for the duration of the simulation. The Wait Mode should be set to 'Concurrent Apply' to ensure that other transactions will not wait for the clock's execution to complete before they start (since the clock will not complete until the test bench finishes execution). It is important that the Wait Mode is set properly or the only transaction that will ever run is the diagram containing the global clock.

When creating a global clock you will need to have one transaction (the clock generator) that has the global clock as an outpu), while the other transactions treat the clock as an input. The clock should have the same name in each of the diagrams, ensuring that the global clock is used for the input of the other transactions.

To Create a Global Clock:

- Open the timing diagram you want the clock to be in (select the **File > Open Timing Diagram** menu option). If you want the global clock to be in a diagram by itself, then open a new timing diagram (select the **File > New Timing Diagram** menu option).
- Add a clock signal to the diagram and specify the clock settings.
- Open the *Signal Properties* dialog (double click the clock name).
- Specify the **direction** as 'output' and specify the name of the global clock.
- Close the *Signal properties* dialog (click the **OK** button).
- Save the timing diagram (select the **File > Save Timing Diagram** menu option).

Next, in each of the timing diagrams that will use the global clock as an input to the transaction do the following:

- Open the timing diagram (select the **File > Open Timing Diagram** menu option).
- Add a clock to the diagram.
- Open the *Signal Properties* dialog (double click the clock name).
- Set the **direction** of the clock as 'input' and set the **name** to be the same as the global clock created above.
- Close the *Signal Properties* dialog (click the **OK** button).
- Save the timing diagram (select the **File > Save Timing Diagram** menu option).

When the top-level test bench is built, the output of the global clock will become an input to the timing transactions that have specified the global clock as an input.

Ending the Execution of a Global Clock

Each transaction has an Abort_*TransactionName* subroutine that is created. The purpose of this subroutine is to provide an easy method of ending a simulation that uses transactions such as a global clock. Once you have entered the appropriate Diagram calls in sequence, you can call **Abort_GlobalClock**, for example (assuming that the global clock is in a transaction named GlobalClock), to end the simulation. The clock will run until all other transactions have completed, and stop when the abort subroutine is called.

10.3 Loops in Transactions

There are three methods of creating looping transactions. The first is to create a transaction that will loop continuously until the simulation is complete (all other transactions have stopped execution) using the **Run Mode** and **Wait Mode**. The second method is to use looping time markers (see *Section 5.4: Time Markers for Test Bench Loops*) to create a conditional loop. The last method of creating looping for a timing diagram is to use HDL code directly in one of the source files (.v or .vhd).

Below is an example using Verilog code to create a loop that will cause an entire transaction to repeat ten times. Note that this code segment would be located in the template or test bench file:

```
LoopControlValue = 0;
while(LoopControlValue < 10)
  begin
    Apply_tbwrite('hF0, LoopControlValue)
    LoopControlValue = LoopControlValue + 1;
  end
```

In this example, the value of the loop control variable will be written to the memory address `'hF0`. The loop will be repeated ten times, each time changing the value that is being written to memory. The while construct could also be placed in one of the timing diagram code files to force a particular segment of a transaction to repeat.

The second two methods mentioned can be used to force the entire transaction to loop or to have portions of the transaction loop.

Chapter 11: The Architecture of TestBencher Test Benches

This chapter will provide a more in depth discussion of the files generated by TestBencher, as well as the signals used by the top level test bench. This discussion will provide a more thorough understanding of how to use the test bench and project files generated to ensure that your projects are successfully tested and validated.

11.1 Files Used to Build a Project

TestBencher builds several files for use in a project. The files are generated as follows:

- The diagram-level timing transaction files (one for each timing diagram in the project).
- The top-level test bench is generated within the project's top-level template file. It is used to control the sequencing and interaction of the timing transactions.
- Tasks are generated for each timing transaction during test bench generation (in VHDL this is a separate file, in Verilog the tasks are included directly into the top-level test bench file).

Diagram Level Generated Code

The diagram level timing transaction files are generated each time a timing diagram is added to a project. The file is then regenerated when that timing diagram is saved (while it is still a part of the project). The code that is generated at this level will provide the code that is executed for a particular timing transaction.

VHDL Specific Task Generation

An additional file is generated for VHDL projects. This file contains the tasks that are generated for each diagram. A package is created, named 'tb_vhdlProjectName_tasks,' (where vhdlProjectName is the name you chose for your project file) that contains the task definitions. This package is stored in a file by the same name, with the extension '.vhd'. You will need to ensure that this file is placed in your work directory when you simulate your project with your VHDL simulator.

11.2 Status Signals and States

Each timing transaction, as well as the top-level test bench, will be controlled through a series of status signals that TestBencher creates. This section will discuss what signals are generated and monitored at each level, as well as how the states of the status signals are used to control the simulation.

The top-level test bench monitors and controls each transaction using two signals - a status and a trigger. These two signals are generated by TestBencher and placed in the port list for the transaction. The trigger is used to fire the transaction while the status tells the top level when the transaction has completed.

In the same way that the top-level test bench monitors a status for each transaction, each diagram monitors status signals for each sequence and process within the transaction. This way the transaction knows when an end diagram occurs, for instance.

State Transitions for the Diagram Status Signal

The diagram status signal is triggered by the diagram trigger at the top level. The trigger is used to begin the transaction. When the top-level test bench sets the diagram trigger to ONCE or LOOPING the diagram status is set to that value (see Figure 1). The diagram status is then set to DONE when either the diagram trigger is set to ABORT (by the Abort_DiagramName task), or when one of the internal statuses is set to ABORT or TIMEOUT (VHDL only). The

last way that the diagram status can be set to DONE is when all of the internal statuses have transitioned to the DONE state. This occurs when the status is in ONCE mode, not LOOPING. Figure 1 demonstrates the state transitions of the diagram status, as well as what causes each state transition.

Figure 1: State transitions for the DiagramName_status

State Transitions for Internal Sequence and Process Status Signals

Each timing transaction can have one or more sequences or processes that are used to control the waveforms during transaction execution. These sequences and processes will execute concurrently during the timing transaction. Each of these sequences and processes will have its own status signal that is used internal to the transaction (referred to as **internal status signals**). These are used by the transaction to monitor each sequence for an ABORT or TIMEOUT to occur, as well as to determine when every sequence and process has completed execution.

When the diagram status changes to a ONCE or LOOPING state, the internal status signals will transition to the equivalent state (i.e., if the diagram status is ONCE, the internal status signals transition to the ONCE state). If one of these sequences or processes encounters an End Diagram feature or a time out, then the status signal for that sequence or process is set to ABORT or TIMEOUT. This causes the diagram status to move to the DONE state. As soon as the diagram status moves to the DONE state, all of the internal status signals will move to the DONE state. Figure 2 demonstrates the state transitions of the internal status signals, as well as the stimuli for state transitions.

Note: There are a few slight differences in the implementation of VHDL and Verilog. VHDL has TIMEOUT as a separate state that the signals can be given. In Verilog, when a time out occurs, the status is set to ABORT. Also, while VHDL constantly monitors the diagram status to determine when a time out or abort has occurred, Verilog has an 'End_Transaction' task that is called. These implementation differences do not change the basic flow that is used.

The Status Signal Names

At the top-level test bench these signals will be named 'DiagramName_tb_status' and 'DiagramName_tb_trigger.' For example, Tbread.tim (see tutorial 5) would have two signals in the top-level test bench, 'Tbread_tb_status' and 'Tbread_tb_trigger.' These signals will refer to the diagram trigger and status. Since the top-level test bench does not monitor the individual status signals for sequences and processes internal to the transaction, it does not have signals that represent them.

At the transaction level the diagram status signal has the name 'tb_status' and the diagram trigger is named 'tb_trigger.'

The internal status signal names have a naming scheme that will ensure that each status signal is both unique and easily recognizable. The name for each status will begin with ‘tb_status.’ The base name of the sequence or process that is controlled by an individual status signal is then appended to ‘tb_status.’ For example, the ‘Unclocked_Sequence’ in Tbread.tim (from Tutorial 5) will have a status signal that is named ‘tb_status_Unclocked.’

11.3 Monitoring Transaction Execution

The best method of monitoring transaction execution is by monitoring the status signals described in *Section 11.2: Status Signals and States*. Each of the four states that a status signal can enter will help determine exactly what each process of each transaction is doing. In addition to these signals, you can also monitor the states of the signals in your simulation. This section will describe how to watch signals during a simulation and will explain the naming convention used for these watched signals.

The top-level test bench names the ports of each transaction according to the following naming scheme:

```
testbench.Port(OrSignal)Name
ex.: testbench.WRB
```

If you are working with Verilog, you will be able to simulate your model with TestBencher. The rest of this section will discuss how the signals on various levels of your model can be accessed and watched during a simulation.

Using TestBencher’s Simulator to Monitor Transaction Execution (Verilog)

TestBencher provides a **watch** signal. This allows you to *watch* the values of signals during a simulation. When you build the top-level test bench, some of these signals are automatically placed into a new timing diagram. The two different test benches will generate different sets of signals. You can experiment with the two and determine which you prefer to use.

You can add signals to be watched by using the Project Tree in the *Project Window*. Open the <<<testbench>>> module of the project tree. This will show you two levels in the hierarchy: the **Signals** and the **Components**. The **Signals** contains the signals that are found in the top-level test bench. The **Components** of the test bench are the timing transactions and the MUT. Right clicking on the **Components** item within the test bench module will open a context menu that will give you the options to **Watch Component** and **Watch Component and Sub-component**.

For each item found in the test bench module, the option to **Watch Component**, **Watch Signals**, or **Watch Connection** will appear, depending on where you are in the hierarchy. By selecting this option, you can add watch signals to a timing diagram.

It is important to understand the naming scheme used when you add watch signals to help you monitor the simulation. All signals begin with the module name, in this case ‘testbench.’ Following the module name is the component name. For example, in the diagram shown above, selecting **Watch Component** from the context menu would add a series of signals that begin with ‘testbench.Tbread.’ Following the component name is the name of the signal that is being watched. Tutorial 5 has a diagram named **Tbread.tim** with a signal named **WRB**. If this signal were added, it would have the name testbench.Tbread.WRB. If we were watching the *signal* for Tbread’s WRB from the top level test bench, then the name would be testbench.WRB.

Note: The difference in the naming conventions is the key to whether you are looking at the signal within the transaction or the value being exported to the top-level. The signal that is in the transaction has the component name and the signal name, while the top-level signal uses just the signal name.

11.4 Generated Tasks for Timing Transactions

Five tasks are generated for each timing transaction. Four of these tasks are used to control the execution modes of the transaction. The fifth is used to abort a transaction. This section will discuss the four tasks used to control the modes of execution, and then the abort task.

Tasks to Control the Modes of Execution (Apply_ Tasks)

The **Run Mode** and the **Wait Mode** (discussed in *Section 8.5: Controlling the Execution Mode of a Transaction*) areused to control the execution of a diagram. For instance, they help to determine if a transaction will run once or continuously. The settings for these modes correspond to the name of the task you will call. A diagram that is set to run continuously will have the word '_looping' appended to the task name. Also, if the transaction can be run concurrently with other transactions, '_nowait' will be appended to the task name. As an example:

```
Apply_GlobalClock_looping_nowait();
```

would be the appropriate call to begin a global clock. The '_looping' will ensure that the clock runs continuously, and the '_nowait' will ensure that other transactions will not wait for the global clock to end before they begin.

Each task name from the generated tasks reflects the behavior with which the transaction will be executed.

The **Run Mode** (see *Section 8.5: Controlling the Execution Mode of a Transaction*) determines if a timing transaction executes once or continuously. A transaction that runs once will have a status of DONE while it waits to be triggered again. A transaction that is applied with a Run Mode of LOOPING will run in a continuous looping cycle until it is aborted (through a sample, for instance) or until it is ended from the test bench level.

The **Wait Mode** (see *Section 8.5: Controlling the Execution Mode of a Transaction*) determines if the timing transactions run sequentially or simultaneously. Simultaneous execution allows you to add items such as a global clock to your project. Specifically, a timing transaction that is executed in NOWAIT mode means that the execution of the transaction will not cause the next transaction to wait. The next transaction will begin execution as soon as the first begins. However, if this mode is not applied, then the default WAITING mode is used. This forces the transactions to run sequentially.

The Abort Transaction Task

The Abort task is used to force a transaction to end from the top level. This task could be used to end a transaction that is set to run continuously, for instance. Calling this task causes the diagram trigger to be set to ABORT. Once this happens, all processes within the transaction act as though an end diagram marker had occurred. Execution of the transaction will be clean and immediate with this task.

11.5 TestBencher HDL Library Files

TestBencher uses two library files, one for Verilog and one for VHDL. Both files contain values that are used as constants. In addition, the VHDL library file contains a package that will be used by the files generated by TestBencher. The two library files are **syncad.v**, for Verilog, and **syncad.vhd** for VHDL. You may want to copy these files into the library path for your simulator. If you are using TestBencher for simulations, you will not need to move the file. These two files are installed in the same directory as the executable file for TestBencher (C:\Vlogger by default).

The VHDL Library File

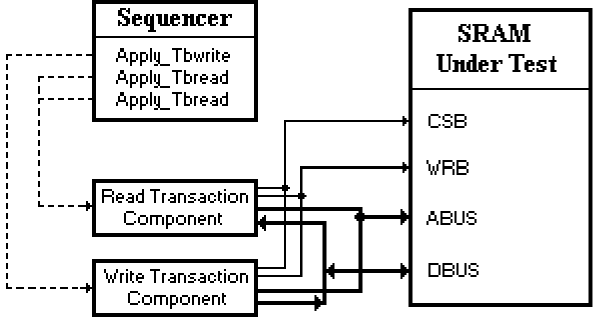
The library file for VHDL contains a package titled **TBDefinitions**. This package contains type definitions, functions and procedures specific to the code generated by TestBencher. It also contains a resolver process for the type Tstatus, defined in the package. The functions contained will detect clock edges, as well as various conversion functions. These items were placed in the library file so that they could be used by both the timing transactions and the test bench.

Appendix A: TestBencher Pro Basic Tutorial

In this tutorial we will generate a test bench to test an SRAM model. Please note that you must have a full version license or a 30 day evaluation license file to perform the actions in this tutorial. You can obtain a license file by visiting our website (**www.syncad.com/license.htm**) or by contacting our sales department. The figure below shows a block diagram of the test bench that we will create around the SRAM.

We will perform the following steps in this tutorial:

1. Create a project for the test bench. This project will ultimately contain a timing diagram file for each bus transaction we model, a top level test bench template file that controls the sequencing of the bus transactions, and a file containing the SRAM model under test.
2. Modify the timing diagram that describes the read cycle bus transaction and examine the generated HDL code.
3. Copy and modify the top-level template file used to generate the top level component of the test bench. We will instantiate the SRAM under test in this component and make subroutine calls to control the execution order of the timing diagram transactions.
4. Generate a top-level test bench and examine the resulting HDL code.
5. (Optional) Simulate the model and test bench using a Verilog or VHDL simulator. This step assumes you have available a 3rd party HDL simulator (Note for Verilog Users: TestBencher includes a Verilog simulator that can be used to perform this step. See **Help > VeriLogger Getting Starting** for more details on using the internal Verilog simulator). The simulation will demonstrate what happens when a sample triggers on a false condition.



Schematic of circuit that the tutorial test bench will exercise

This tutorial does not cover any of TestBencher Pro's drawing features or the language independent bus format. These features are covered in the first three tutorials (**Basic Drawing and Timing Analysis, Simulation, Waveform Generation, and Parameters**, and **Advanced HDL Stimulus Generation**). *If you are new to SynaptiCAD's timing diagram editing environment then you should do the tutorials listed above first*. This tutorial only discusses how to create multi-diagram self-testing test benches.

This tutorial uses two timing diagrams: **tbread.tim** (copied from **tbread_orig.tim**) and **tbwrite.tim**. Two versions of the SRAM model under test (MUT) are provided in files named **tbsram.v** (Verilog) and **tbsram.vhd** (VHDL). Work with the MUT that is appropriate for you. TestBencher uses a top-level template file to create the test bench. This file will be added to your project, then modified to include the timing transactions and MUT instantiation for this project.

Preparing your working environment for this tutorial:

If do not have read/write access to the TestBencher Pro installation directory then you must copy the tutorial files to a new directory with that you have read/write permission. If you can read and write to the TestBencher Pro directory then skip down to *Step 1: Setting up the Project*.

To set up a project directory:

1. Create a new directory or folder called **myExamples**. This directory should be in a path that your account has full access to.
2. Copy the following files from the **Examples** directory located in the TestBencher installation directory into the new **myExamples** directory:

   tbread_orig.tim

   tbwrite.tim

   tbsram.v or **tbsram.vhd** (.v for Verilog users, .vhd for VHDL users)
3. Copy the following files from the TestBencher installation directory into the new myExamples directory:

   If you are working with VHDL: **wavelib.vhd** and **syncad.vhd**.

   If you are working with Verilog: **wavelib.v** and **syncad.v**

When the tutorial refers to the **Examples** directory, use this new project directory (**myExamples**). Using a directory that is in your working path on the network will ensure that TestBencher has permission to write the files that it generates for you.

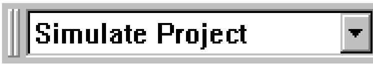
1 Setting up the Project

TestBencher Pro uses a project to save the options and organize the files that will be used to generate the test bench. In this section you will adjust the project settings, change the project name, and add the timing diagram files and the model under test files.

1.1 Set Simulation Properties

The Simulation properties affect when and how TestBencher will perform simulations. During the initial design phase you will want to turn off the interactive simulation features by setting the following options on the Simulation button bar at the top of the screen:

1. **Simulate Project** should be selected as the Simulation State. If **Sim Project & Diagram** is set, use the *Simulation State* drop down box to change the setting.

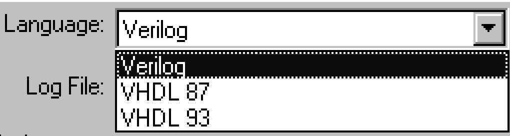


2. Make sure that the Simulation Mode is set to **Debug Run**. (Click the *Simulation Mode* button if **Auto Run** is the currently selected mode.)

1.2 Choose a Test Bench Language

Set the language for your project using the *Project Settings* dialog:

1. Select the **Project > Project Settings...** menu option to bring up the *Project Settings* dialog.
2. Choose the appropriate language from the **Language** drop down box and press **OK** to close the dialog. (Notice that this setting can also be changed on the Simulation button bar.)



ModelSim Integration

If you are going use ModelSim to simulate your projects, you can use the new ModelSim Integration feature to auto-build the project library and launch the simulator with the design loaded. As a reminder, TestBencher does provide a Verilog simulator, but the ModelSim Integration can be used with both dialects of VHDL or Verilog.

To enable ModelSim Integration:

1. Select the **Options > External Program Settings** menu option. This will open the *External Program Settings* dialog.
2. Check the **Enable ModelSim Integration** checkbox. This will disable TestBencher's Verilog simulator and enable the ModelSim Integration feature.
3. Enter the path to the ModelSim executable files in the **ModelSim Executable Path** edit box. This should be the directory that contains the ModelSim executable files (**vlib.exe**, **vsim.exe**, **vcom.exe** and **vlog.exe**).
4. Click **OK** to close the dialog and enable the ModelSim Integration feature.

1.3 Save the Project to a specific file name

When TestBencher Pro is opened, it automatically begins a new project named. Save the current **untitled.hpj** to a project named **sramtest.hpj**.

To Save a new project:

1. Choose the **Project > Save HDL Project** menu option. This will open the *Save Project As* dialog.
2. Select the **Examples** directory.
3. Type in the name **sramtest.hpj** in the *File name* edit box.
4. Click the **Save** button to save the project.

1.4 Add the MUT to the Project

The first file you will add to the project will be the model under test or MUT. The MUT is added to the project in order to give TestBencher access to the port information so that you will not have to set the type, direction and vector size of signals by hand. For this tutorial the MUT is the SRAM model contained in **tbsram.v** for Verilog users and **tbsram.vhd** for VHDL users. Add the MUT to the project:

1. Choose the **Project > Add HDL File(s)...** menu option. This will open the *Add Files...* dialog.
2. If you are working with Verilog, select the file **tbsram.v**. For VHDL, select the file **tbsram.vhd**. These files are located in the Examples directory.
3. Click **Open** to close this dialog and add the file to the project.

1.5 Add tbwrite.tim to the Project

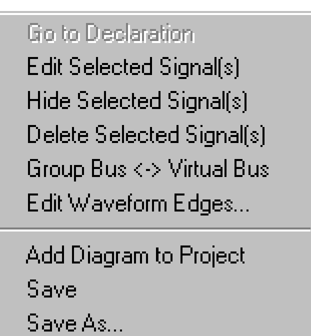
To make the tutorial a little shorter we have already completed the timing diagram that represents the write cycle. In this section we will add the write cycle timing diagram using the Project menu function:

1. Choose the **Project > Add Timing Diagram(s)...** menu option to open the *Add Files...* dialog.
2. Select the file **tbwrite.tim** (located in the Examples directory) and click **Open** to add the file to the project.

1.6 Add tbread.tim to the Project

We have also partially completed the read cycle timing diagram and saved it in a file called **tbread_orig.tim**. You will be editing the read cycle diagram during this tutorial. So that the tutorial may be completed again, we will first copy the original into a file called **tbread.tim** and then add the copy in to the project:

1. Choose the **File > Open Timing Diagram...** menu option to open the *Open File* dialog.
2. Select **tbread_orig.tim** (located in the Examples directory) and **Open** button to load the timing diagram into the *Diagram* window.
3. Select the **File > Save Timing Diagram As...** menu option to open the *Save As* dialog.
4. Type **tbread.tim** in the *File Name* edit box, then click **Save** to save the file under the new name.
5. Since the timing diagram is already loaded in TestBencher we can use the right click context menu to add the diagram to the project. Right click on top of one of the signal names and choose the **Add Diagram to Project** menu option.



1.7 View the source code generated by tbread.tim

Each time a timing diagram is saved or added to the project, a new transaction model is generated for the timing diagram. You can watch how adding objects to the **tbread.tim** changes the generated code by periodically saving during the tutorial and looking at the generated code.

To view the source code:

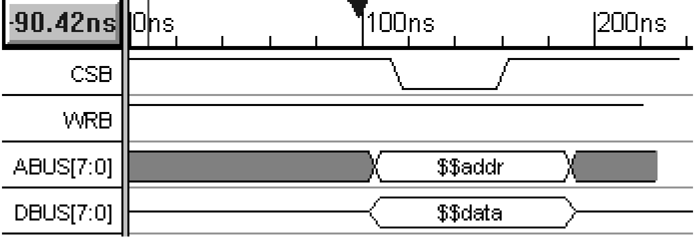
1. Right click the filename **tbread.tim** in the Project window to open the project pop-up menu.
2. Select the **Open C:\Examples\tbread.v** (or **tbread.vhd**) menu option. Keep this window open so that we can refer to the code periodically during the diagram construction.

There should now be three files in the project: **tbread.tim**, **tbwrite.tim** and the **tbsram** HDL file. To complete the testbench we will also be adding a top-level template file in section 4 which is responsible for instantiating the MUT and sequencing the timing transactions.

2 Creating a TestBencher Pro Timing Diagram

TestBencher Pro uses timing diagrams that represent reusable bus transactions and a template file to generate the test bench. In this tutorial there are 2 timing diagrams, **tbread.tim** and **tbwrite.tim**, that represent the read and write cycles used in testing the memory module. The write cycle is a completed timing diagram. The read cycle, **tbread.tim**, has the basic waveforms and delays that you would normally have in a timing diagram, but it lacks the additional information that TestBencher Pro needs to generate a test bench. First, we will specify the type, direction, and vector size of each signal. Next we will add additional information to bi-directional signals to indicate which segments are input and which are output (driven). Then we will add variables to the diagram so that values can be passed into the address and data busses. Finally, we will add a sample that will test the operation of the SRAM in the model.

The diagram window has the diagram for **tbread.tim**. The diagram should look like this:



2.1 Setting the type, direction, and vector size for signals

Normally, the type, direction and vector size of signals will be acquired from the MUT. We will manually set these signal properties here so that you can see how the code is generated. Set the type, direction, and vector size for a group of signals in tbread.tim:

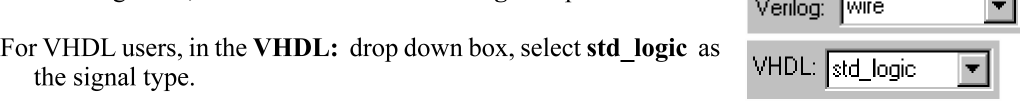
1. Left click the **CSB** and **WRB** signal names to select the group of signals.
2. **Right click** on one of the highlighted signal names to open the signal pop-up menu.
3. Select **Edit Selected Signal(s)...** to open the *Signal Properties* dialog.

   Notice that the signal name edit box is gray. This is because multiple signals are being edited. Any property setting that you make now will affect all of the highlighted signals.

4. Make sure that the **Export Signal** check box is checked. This indicates that this signal will be exported as part of the test bench. This is the default state of a new signal.
5. Select **output** from the **Direction** drop down box. The direction of *output* means that these signals are outputs of the test bench, and thus an input to a model under test (your circuit).
6. Next, you will set the signal type according to your language.

   - For Verilog users, select **wire** from the **Verilog:** drop down box.

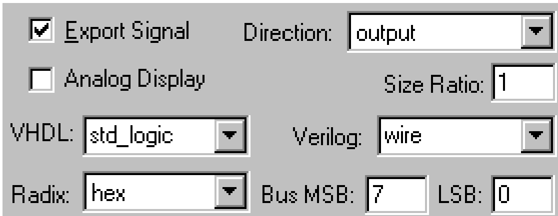
   - For VHDL users, in the **VHDL:** drop down box, select **std_logic** as the signal type.



7. Click **OK** to close the *Signal Properties* dialog. (You can left click anywhere in the drawing environment to un-highlight the signals.)

Setting type, direction and vector size for individual signals:

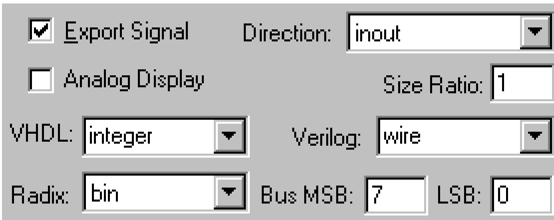
The property settings for **ABUS** will match those in the figure below. You can open the *Signal Properties* dialog for **ABUS** to verify this by double clicking the signal name in the Signal window.



Tip: You can click the **Next** button to go to the next signal on the timing diagram if you have the Signal Properties dialog open.

1. Double click the signal name **DBUS**. This will bring up the *Signal Properties* dialog with **DBUS** in the *Name* box at the top of the dialog.

2. Set the properties for **DBUS**: check the **Export Signal** check box, and select the direction as **inout**. A direction of **inout** means that **DBUS** is a *bi-directional signal*. Sometimes the test bench will be driving this signal, other times the model under test will be driving this bus. Later in this tutorial we will specify which sections are driven by the test bench and which are not.



3. Select the VHDL type as **std_logic** or the Verilog type as **wire**, depending on which language you are working with. Set the vector size of the bus to 8 bits. Enter **7** into the *Bus MSB* edit box, and **0** into the *LSB* edit box.
4. Click the OK button to close the dialog.

   Remember you can save the timing diagram at any point and the code in the tbread editor window will be automatically updated. At this point, it will have the port definitions generated at the top of the code.

2.2 Clearing the Driven flag for input transitions of an inout (bi-directional) signal

Because the signal *DBUS* has a direction of **inout**, we must differentiate between input segments and output segments. By default all signal segments have a direction of output and are colored black. Input segments will be colored blue.

To set the direction of the segments in **DBUS**:

1. Double left click on the **first edge** of signal *DBUS* (located at about 105 ns). This will open the *Edge Properties* dialog. Position the dialog so that you can see both the *DBUS* signal and the dialog at the same time.

   Notice that the **Driven (Export to HDL)** check box is checked. Driven (Export to HDL) This means that the signal segment to the left of the selected edge is an *output* and is colored black. We want this segment to be exported as a signal assignment, so leave it checked.

2. Click the **Next** button to move to the next edge on *DBUS* (which is at about 190 ns). Notice that the Green selection bar moved to the edge on the right side of the valid segment.

3. **Uncheck** the **Driven (Export to HDL)** check box. Driven (Export to HDL) This will indicate that the valid section to the left of the selected edge is an input segment that will be driven by the model under test.
4. Click the **Next** button to move to the last edge on *DBUS* (at about 242 ns). Notice that the valid section has turned Blue, and that the Green selection bar moved to the last edge on the signal.
5. Make sure that the **Driven (Export to HDL)** check box is checked.
6. Click **OK** to close the dialog. The segments of *DBUS* should now be colored black-blue-black.

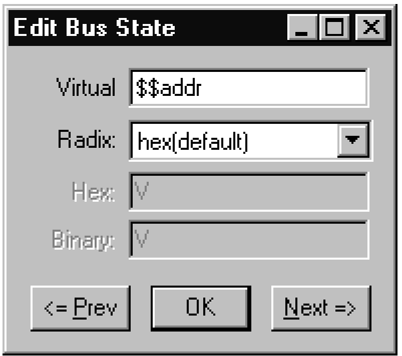
2.3 Adding a Parameterized State Value

Next we will add variables to the timing diagram so that values for the address and data buses may be passed into the test bench transaction. We call these variables "Parameterized state values" because their function and use in the test bench resembles the way parameters are passed in to a C subroutine. In *tbread.tim* we will add two variables: one for the value on the address bus, and one for the value on the data bus. Later when we modify the template file we will pass values into these variables.

Add the address and data state variables to the diagram:

1. Left click on the valid segment in the center of the *ABUS* signal to select the segment. The selected segment has a green box drawn around it.

2. Click on the **HEX** button HEX on the button bar. This opens the *Edit Bus* dialog.

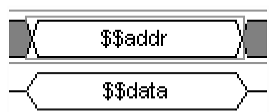


3. Type **$$addr** into the **Virtual** edit box. The "$$" in front of the variable name indicates that this is a parameterized state variable. (Note: Leave the dialog open in this step).

Note: If the "$$" is missing, TestBencher Pro will assume that this is the value of the address rather than a variable that will assume a value at a later time.

4. Double click on the blue valid segment in the center of *DBUS* to move the focus of the *Edit Bus State* dialog to the new segment.

5. Enter **$$data** *Virtual* edit box, then click **OK** to close the dialog.



6. Select the **File > Save Timing Diagram** menu option to save the diagram. Notice that the code in the **tbread.v** (or **tbread.vhd**) editor window has been updated.

Note: This segment of Verilog code from the tbread.v shows that DBUS is tristated z when entering the blue section of the diagram (beginning at 105 ns) even though the variable $$data has been assigned to this segment. This signal will not be driven by this diagram; $$data will be used to capture the value of the valid segment. Later, we will use this value in a sample. Since ABUS is an output signal, $$addr is the value that will be driven by the bus.

```
#105            //AbsoluteTime 105
ABUS = addr;
DBUS_driver = 8'bzzzzzzzz;
```

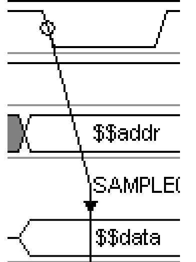
Note: VHDL users will be able to find similar code in the tbread.vhd file.

We will work more with the variables when we modify the template file.

2.4 Adding a Sample Parameter

Next we will add a Sample parameter to the timing diagram. A Sample parameter generates self-testing, test bench code.

To add a sample:



1. Left click on the **Sample** button [Sample] on the button bar. This makes the right mouse button add sample parameters.

2. Left click on the **first falling edge** of signal *CSB* (at about 110 ns) to select the edge.

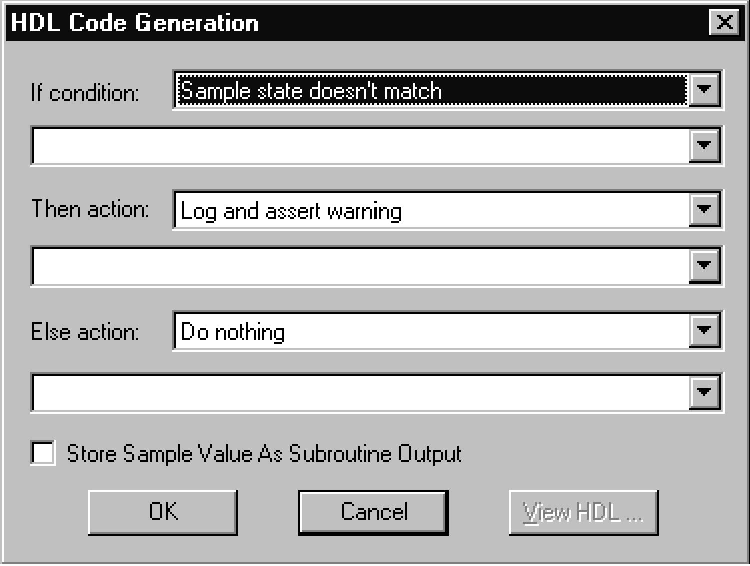
3. **Right click** inside the **blue valid segment** on *DBUS*. This adds a sample parameter.

Next we will determine the type of code that SAMPLE0 will generate:

1. Double left click on the **SAMPLE0** name in the drawing window. This will open the *Sample Properties* dialog.

2. Click the **HDL Code** button [HDL Code] in the bottom left hand corner of the dialog to open the *HDL Code Generation* dialog. Samples generate if-then-else code in a test bench.

3. Select **Sample state doesn't match** for the *IF condition*. This means the test bench will compare the actual value on the data bus with the value in $$data at the sample time during simulation.



4. Select **Log and assert warning** for the *THEN action*. This means that if the values don't match, a warning will be written to the testbencher log file and a warning will be asserted.

5. Select **Do nothing** for the *ELSE action*. If the value on the data bus matches the value of $$data, then the circuit is working properly and we do not wish to take any action.

6. Click the **OK** button to close the *HDL Code Generation* dialog. Click **OK** to close the *Sample Properties* dialog as well.

7. Save the timing diagram and look for the code that was generated for **Sample0** in the **tbread.v** (or **tbread.vhd**) editor window.

   The Verilog code for **Sample0** will look like this:

```
SAMPLE0 = DBUS;
SAMPLE0_Flag = ! (DBUS === data);
if (SAMPLE0_Flag)
   begin
   $fdisplay(testbench.logfile,"TestBencher> In %m at %t: On DBUS,
           expected %8b: detected %8b",$time,data, SAMPLE0);
   $display("TestBencher> In %m at %t: On DBUS,
           expected %8b: detected %8b",$time,data,SAMPLE0);
   end
```

   The VHDL code for **Sample0** will look like this:

```
SAMPLE0 <= DBUS;
SAMPLE0_Flag <= not (DBUS = data);
if (not (DBUS = data)) then
  tbLog("On DBUS, expected " &toString("data") &": detected "
         &toString(DBUS), WARNING);
```

```
    assert FALSE
    report "On DBUS, expected " &toString("data") &": detected "
    &toString(DBUS)
    severity WARNING;
  end if;
```

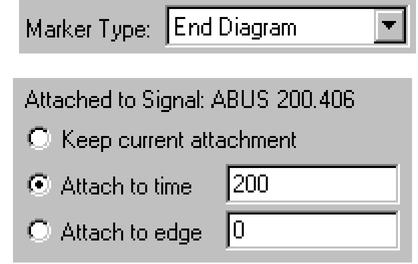
Note: The formatting for these sample codes may vary according to your version of TestBencher.

2.5 Adding an End Diagram Marker

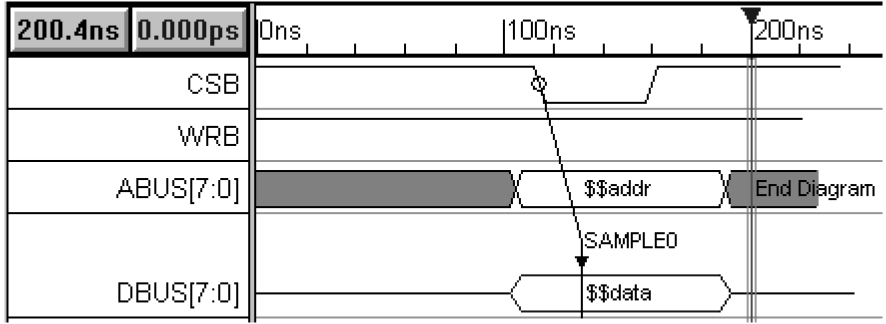
Now we will add an **End Diagram Marker** to *tbread.tim* to indicate exactly where the timing diagram ends.

To add the End Diagram Marker:

1. Left click on the **Marker** button [Marker] on the button bar.
2. **Right click** near the time **200 ns**. This draws a marker line.
3. Open the *Edit Time Marker* dialog by double clicking on the marker line.
4. Select a *Marker Type* of **End Diagram** from the drop down list box. This designates this marker as the end of the timing diagram.
5. Check the **Attach to time** radio button and enter **200** into the edit box next to the radio button. This means that the marker will be attached to the absolute time 200ns.
6. Click **OK** to close the dialog.
7. Save the timing diagram by selecting the **File > Save Timing Diagram** menu option.



At this point in the tutorial your tbread.tim should look exactly like the figure below.



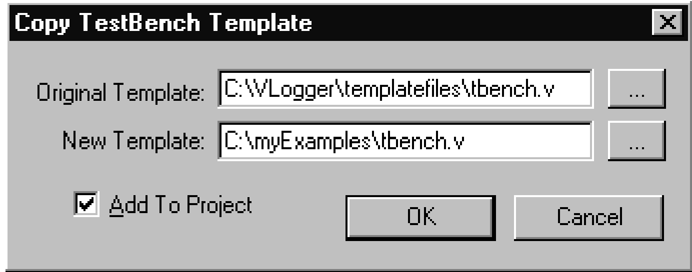
3 Creating and Modifying a Template File

In this section we will copy and modify the top level template file that will be used to generate the test bench. The template file has all of the code that TestBencher needs to generate a test bench except the information specific to your model. We will modify the template file so that it includes the MUT instantiation and the Apply statements that will control the sequencing of the timing transactions.

3.1 Add the top-level template file

To add the top-level template:

1. Choose the **Project > Copy TestBencher Template File...** menu option to open the *Copy TestBench Template* dialog.



2. In the **Original Template** edit box, choose either **tbench.v** for Verilog or **tbench.vhd** for VHDL.
3. In the **New Template** edit box, change the file name to **tbench0.v** for Verilog or **tbench0.vhd** for VHDL. The default path should be the same as the path for the project.
4. Make sure the **Add To Project** check box is checked so that TestBencher automatically adds the file to your project
5. Click **OK** to copy the template file and add it to your project

   Note: the template file you have just added to the project will become the test bench code file. Upon successful generation of a test bench, macros in the template file are expanded and replaced with the generated code. If you wish to make modifications to the project and generate a new test bench, you will be able to do so directly from this file. This will be discussed further in Section 4.

3.2 Instatiating the MUT

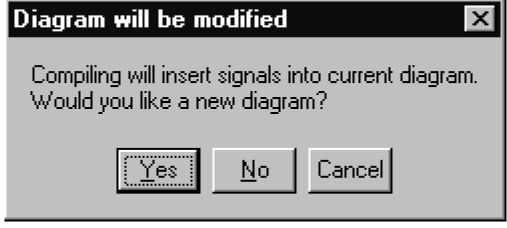
Next, we will instantiate the model under test, MUT, in the template file. In the final code generation, the test bench will be the top level component. It will contain the instantiations of the MUT and the timing diagram models. This section is broken into two distinct segments - one for Verilog users and one for VHDL users. Double left click on the template filename **tbench0** in the project window. This will open the HDL code in an editor window. Next, move to the segment of this section that is appropriate for you.

Verilog Users:

Automatically instantiate the MUT using the Project window features:

1. Click the yellow build button. This will build the MUT.

The dialog shown to the right will open; it is asking if you want a new diagram. Click **Yes**. Whenever a file is built, the timing diagram will have signals added to it. To preserve the original copy of your timing diagram, always have TestBencher create a new diagram when you receive this prompt.



Note: This timing diagram is generated so that you can have the signals prepared when you are ready to generate waveforms for a transaction that will work with the MUT, or that will act as an input to the MUT. In this case, we do not need to do either of these things. To clear the diagram for the top-level test bench, select **File > New Timing Diagram**, or select the signals and press <delete>.

2. Notice that in the Project window a `<<tbsram>>` is attached to the tbsram HDL file. Expanding this tree will display all the models in the tbsram file. If your model did not generate the `<<tbsram>>` then change your simulation preferences by choosing the **Options >Simulation Preferences** menu. Check the **Auto-create test bench and tree** check box. Rebuild to see the expanded tree.

3. Double click on the **tbench0** filename in the Project window to open an editor window with the template code displayed.

4. Locate the comment in the template file editor window that indicates where to place the MUT instantiation. You use <CTRL>-F to search for the following text:

```
// 1) Instantiate Models Under Test after this comment block:
```

5. Left click once below this comment block in the template file editor window so that the blinking cursor is on the line where you want to instantiate the MUT.

6. In the Project window, right click on the top-level module (**<<<tbsram>>>**) to open the module pop-up menu:

   Notice that the pop-up menu is different for the MUT than it is for the timing diagrams.

Watch Component
Watch Component and Sub-component
Instantiate in TBench
Add HDL File(s)...
Add Timing Diagram(s)...
Expand Item
Copy TestBencher Template File...

7. Select **Instantiate in TBench** from the pop-up menu. Notice that this has automatically added the MUT instantiation to the template file where your cursor was located.

8. Right click in the template file editor window and select **Save** from the pop-up menu. If you would prefer to instantiate your MUTs manually, use the instructions for VHDL users as a basis. Skip now to Section 3.3.

VHDL Users:

Next we will instantiate and declare the MUT for VHDL users:

1. Locate the comment in the template file editor window that indicates where to place the MUT instantiation.

```
-- 2) Instantiate Models Under Test after this comment block:
```

2. Type the following MUT instantiation into the template file below the comment above:

```
mutMSB: tbsram port map(CSB,WRB,ABUS(7 downto 0),DBUS(7 downto 0));
```

   (Note that this can be entered on one line.)

3. Next, scroll up in the code to the line

```
-- 1) Declare component for Models Under Test after this comment
```

   Below this comment block is where you define the component portion of your module. You can either type in the following code for the component definition or locate the necessary code (from the entity definition) in the **tbread.tim** editor window and copy it. Either way you choose, you will need to have these lines in the template file:

```
component tbsram port(
     CSB: in std_logic;
     WRB: in std_logic;
     ABUS: in std_logic_vector(7 downto 0);
     DBUS: inout std_logic_vector(7 downto 0));
end component;
```

4. Right click in the template file editor window and select **Save** to save the template file.

Now you are ready to Apply your timing transactions to the template file.

3.3 Applying Timing Transactions to the Template File

Apply statements will represent the sequence in which the transactions will be carried out. Each Apply statement is representative of one timing transaction. By entering the Apply statements in the template file you are defining the execution order of the timing transactions.

1. Scroll down in the template file until you find the comment block that has this line:

   ```
   Add apply calls for timing diagrams transactions after this comment
   ```

   (This comment is almost at the very end of the file.)

2. Left click in the **tbench0** editor window below this comment so that the blinking cursor is in the place where you want to add the apply statements.

3. Select the **Editor > Insert Diagram Calls...** menu option to open the *Insert Diagram Subroutine Call* dialog. (If you prefer, you can right click in the template file editor window and select *Insert Diagram Calls* from the pop-up menu.)

4. Double left click the `Apply_tbwrite( addr , data );` statement. This will add an **Apply** statement for the **tbwrite** transaction to your template file.

5. Next, double left click twice the Apply statement for **tbread** to add two more statements to the template file. Note: VHDL diagram calls will have an extra parameter that will indicate the Project name (e.g. `Apply_tbread(sramtest, addr, data);`).

6. Close the *Insert Diagram Subroutine Call* dialog.

7. Look at the **tbench0** editor window. You will see the Apply statements that have been added in the order that you selected them. Notice that there are currently no values for the *addr* and *data* variables. We will enter those values next.

Providing values for variables in timing transactions:

1. Left click in the **tbench0** editor window and move the cursor to the Apply statements. Each Apply statement is appears twice, once as a comment and once as a line of code. The comment serves a reminder of the order of the parameters.

2. Next edit the Apply code lines and replace the state variable names with actual variables that will be passed into the timing diagrams.

   For Verilog Type:

   ```
   Apply_tbwrite('hF0, 'hAE);
   Apply_tbread('hF0, 'hAE);
   Apply_tbread('hF0, 'hEE);
   ```

   For VHDL Type:

   ```
   Apply_tbwrite(sramtest, x"F0", x"AE");
   Apply_tbread(sramtest, x"F0", x"AE");
   Apply_tbread(sramtest, x"F0", x"EE");
   ```

   Note: Some VHDL 87 compilers will not automatically convert the hex value to a vector string. You can type-cast these values as unsigned to force the conversion.

3. Notice that these statements represent writing the hex value AE to memory cell F0. The **tbread** Apply statements then will read the value from the same memory cell. The data values provided in the second two Apply statements will be used to compare with the actual value. The first call to tbread will expect to find a value of hex AE in the address F0. The second call to tbread will expect to find the hex value EE instead.

4. Save your template file by right clicking in the template file editor window and selecting **Save** from the pop-up menu.

You are now ready to generate your test bench.

4 Generate and Examine the Test Bench

At this point we have created all the timing diagrams and edited a template file that describes the sequencing of the diagrams in the test bench. To generate the test bench:

1. Select the **Export > Generate Test Bench...** menu option. This will use the template file to generate the code for the test bench.

   Note: If you are working in Verilog, you will be prompted to save the current diagram, which is untitled.tim. You can save it to tbtutorial5.tim so that it can be used to hold signals from the testbench later in the tutorial.

Notice that the template editor window was updated when the test bench was generated. The file now contains the test bench HDL code. You can scroll through this code to examine each section of the test bench as it is discussed. The top-level test bench is comprised of two modules: the test bench and the test bench driver. These modules will be discussed in the next few sections.

4.1 The Test Bench Module

The top level module of the generated test bench code is the testbench module. This module contains the signal definitions for all diagrams, the MUT instantiation, and component instantiations for the diagrams.

To view the testbench module:

1. Right click in the test bench editor window. This will bring up the editor pop-up menu.
2. Select the **Find** menu option. This will open the *Search* dialog.
3. Enter `module testbench` for Verilog or `entity testbench` for VHDL in the *Find What* edit box.
4. Click **Find**. The editor locates the test in the test bench editor window and highlights it. Below this text you can see the signal definitions and the MUT instantiation (you may need to scroll down).
5. Close the *Search* dialog.

   Note: Compare the code of the template file to that of the generated test bench. You will be able see by examining the comments of the test bench code which segments were produced from the template file. Every *macro* in the template file begins with a '$' and is related to one of the commented code blocks in the generated test bench. The comments in the test bench tell you which macro created each block of code.

This module also contains several code segments that drive the test bench, the instantiations of the timing transactions, and a sequencer that controls the ordering of the transactions.

The instantiations of the timing transactions are handled by a code segment produced by the macro "ComponentInstantiationsForAllDiagrams" (You can search for this term, or a portion of it, to locate the comment in the text that comes before these instantiations). In this case, the two transactions being instantiated are tbread and tbwrite.

4.2 Event Driven Transactions

Each of the timing transactions in the test bench, as well as the test bench itself, are assigned two control signals called **tb_trigger** and **tb_status**. They are used to provide the event driven aspect of the test bench. The type of event that might drive the test bench is a specific point in time within the simulation being reached, the rise or fall of the clock for clocked signals, or an edge occurrence within a signal. Each of the HDL files for the timing transactions (and the test bench) contain processes that wait for these signals to be triggered, apply the appropriate stimulus, and then wait for them to be triggered again. Within each timing diagram each clocked sequence has a status variable that is used in

much the same way. Once the trigger for the transaction is fired, the statuses are set and the sequence execution begins. Once all of the sequences have completed execution, the internal diagram status is set to the DONE state, and all sequence wait for the trigger to fire again.

5 Simulating the Test Bench

In this section we will review the necessary files for test bench simulation and examine the generated test bench. And describe the processes necessary to simulate the testbench.

These files are required for test bench simulation:

1. The HDL files for the MUT (your original model code). For this tutorial either **tbsram.v** or **tbsram.vhd** is the MUT file.
2. One HDL code file for each timing transaction. These files are named after the timing diagram names. For this tutorial you will have either **tbread.v** and **tbwrite.v**, or **tbread.vhd** and **tbwrite.vhd** depending on language.
3. One top-level HDL test bench file (this is the expanded template file). For this tutorial it is named **tbench0.v** or **tbench0.vhd**.
4. For VHDL, there are two additional package files you must include in your work directory: **syncad.vhd** which contains generic types and tasks used by TestBencher and a project-specific package file called **tb_sramtest_tasks.vhd**, where *sramtest* is the name of the project.

TestBencher Pro has a Verilog simulator built-in. If you wish to use this simulator, use the instructions in section 5.1. If you are working with VHDL or wish to use a different Verilog simulator, follow the instructions in section 5.2.

5.1 Simulating the Project with TestBencher's Built in Verilog Simulator

Since TestBencher Pro contains a full Verilog simulator we will quickly run the Verilog simulation:

1. Click the yellow TB test bench compile button. This will cause several signals to be added to the timing diagram. These signals can be used to monitor various portions of the test bench.

   Note: Examine the signal names of the test bench signals in tbtutorial5.tim. The signal names are generated based upon the level within the test bench that they are monitoring. For instance, `testbench.Tbread_tb_status[1:0]` represents the diagram status signal for Tbread. This is what the test bench 'sees' from it's side of the port. If we were watching from the Tbread side of the port, the signal name would be `testbench.Tbread.tb_status[1:0]`. The top-level signals that we are the most interested in are `testbench.CBS`, `testbench.WRB`, `testbench.ABUS[7:0]`, and `testbench.DBUS[7:0]`.

2. Click on the green simulation run button to simulate the project.

View the simulation waveforms, and notice the following chain of events:

1. The **tbwrite** diagram is executed - writing data to the SRAM.
2. The **tbread** diagram is executed twice - the first time reading the expected value from the SRAM, and second time reading an unexpected value.
3. The unexpected value during the second execution of **tbread** causes a warning to be asserted and also writes a warning message to the log file verilog.log.

5.2 Simulating the Project Using a Third Party Simulator

Simulating using the ModelSim Integration Feature:

If you are using the ModelSim integration feature, click the yellow tb button TB to autolaunch ModelSim and build the Project libraries. See Section 2.5 of the TestBencher manual (or on-line help) for more information about ModelSim integration.

Simulating with Other Third Party Simulators:

To simulate a VHDL test bench, you will need a total of seven files in the project directory: **wavelib.vhd**, **syncad.vhd**, **tb_sramtest_tasks.vhd**, **tbread.vhd**, **tbwrite.vhd**, **tbsram.vhd**, and **tbench0.vhd**. (Note: the first two files will be needed for all of the test benches you generate using TestBencher. You may want to add them to the library directory of your preferred VHDL simulator). Verilog users will need the following files: **syncad.v**, **tbread.v**, **tbwrite.v**, **tbsram.v**, and **tbench0.v**.

If your preferred VHDL simulator is not already running, please launch it at this time. Create a project in the VHDL simulator and add the seven files listed above to the project. These files will be located in the **Examples** directory.

> Note: if you chose not to create a project directory at the beginning of this tutorial, then you will need to copy these files into a project directory that your simulator can use at this time. The first two files listed will be located in the installation directory of TestBencher, while the other five are located in the **Examples** directory of the TestBencher installation directory.

You will want to be able to view the output of the simulation. If your simulator will automatically create a file for this purpose, simulate the project and allow the file to be created. If not, then create a waveform file and add the signals from the top-level testbench to the file. Simulate the project.

The status signals show the execution state of each diagram and are useful for debugging your test bench. The default initialization for each status signal will be ABORT. Each status signal will then initialize to DONE prior to triggering. When a diagram begins executing the status signal will be set to ONCE and when the diagram finishes executing the status signal will be set to DONE. The status flag for a diagram applied with a run mode of LOOPING will be set to LOOPING when it executes instead of ONCE to indicate that the diagram will re-execute whenever it reaches its end point. The Apply functions that are generated for each transaction indicate which runMode and waitMode are being used. (See the TestBencher manual for more information about these two modes.)

View the simulation waveforms, and notice the following chain of events:

1. The **tbwrite** diagram is executed - writing data to the SRAM.
2. The **tbread** diagram is executed twice - the first time reading the expected value from the SRAM, and second time reading an unexpected value.
3. The unexpected value during the second execution of **tbread** causes a warning to be asserted and also writes a warning message to the simulation log file.

Congratulations! You have now completed the introduction tutorial for TestBencher Pro. The next TestBencher Pro tutorial discusses the use of a global clock, a sweepable delay and a continuous setup checker. For more information regarding the features available with TestBencher (such as adding library files to a diagram, using markers to create loops in a diagram, and advanced uses for other diagram components) please see the online manual for TestBencher, available through the **Help > TestBench Generation Help** menu option.

Additional example files can be found in the **Examples** subdirectory of the installation directory. Many of these files contain text in the timing diagram that specifies what the timing diagram demonstrates. In addition, there is a **readme.txt** file in the Examples directory that specifies the contents of each subdirectory of the **Examples > VHDL** and **Examples > Verilog** directories.

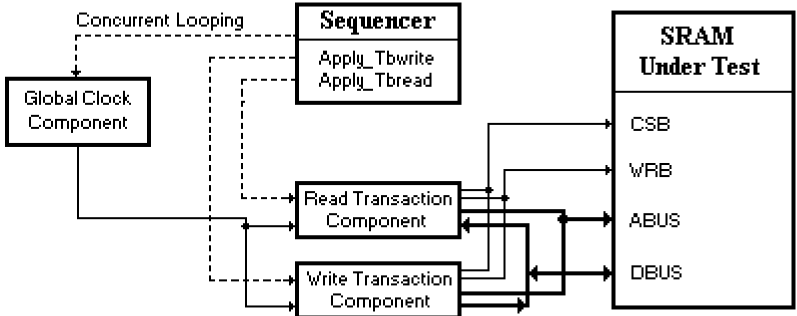
Appendix B: Performing a Sweep Test (Advanced Tutorial)

This tutorial will generate a test bench that cycles through read/write transactions on an SRAM model. It will sweep an input edge to the SRAM model until a setup failure is generated. The SRAM model is the same model that was used in the TestBencher Basic Tutorial.

This tutorial will demonstrate several advanced capabilities of TestBencher:

1. Create a global clock for a test bench using a concurrent apply call.
2. Generate stimulus vectors as a function of a clocking signal.
3. Use delays to control the timing of an edge transition (making an edge transition relative to a waveform generated outside a transaction).
4. The use of continuous setups as protocol checkers
5. Place HDL code at the top level to control timing transaction sequencing and to pass timing values to a transaction at simulation run-time. This allows programmatic and randomized timing of clocks and edge transitions.

This tutorial picks up where the TestBencher Basic Tutorial left off. Please note that you must have a full version license or a 30 day evaluation license file to perform the actions in this tutorial. You can obtain a license file by visiting our website (**www.syncad.com/license.htm**) or by contacting our sales department.



Schematic of circuit that the tutorial test bench will exercise

This tutorial does not cover any of TestBencher Pro's drawing features or the language independent bus format. These features are covered in the first three tutorials (**Basic Drawing and Timing Analysis, Simulation, Waveform Generation, and Parameters**, and **Advanced HDL Stimulus Generation**). *If you are new to SynaptiCAD's timing diagram editing environment then you should do the tutorials listed above first*. Also, this tutorial will begin with a project that has already been created. It is assumed that you are able to create TestBencher projects. If you are not familiar with creating test benches using TestBencher Pro, it is recommended that you perform the **TestBencher Basic Tutorial** before completing this tutorial.

This tutorial uses three timing diagrams: **tbread.tim** (copied from **tbread_orig.tim**), **tbwrite.tim**, and **tbglobal_clock.tim** (created during the tutorial). Two versions of the SRAM sweep test project are provided, named **SweepTest1.hpj** and **SweepTest2.hpj**. The first version generates a cycle-based test bench, the second generates a timing-based test bench. As with the TestBencher Basic Tutorial, the only two files in the project that you will need to modify (aside from the creation of the global clock) are **tbread.tim** and the top-level template file.

Note: We provide two copies of the read diagram for the possibility of the tutorial being performed more than

once. The file "tbread_orig.tim" contains the diagram in its original state, so it can be copied and renamed to "tbread.tim".

Preparing your working environment for this tutorial:

If do not have read/write access to the TestBencher Pro installation directory then you must copy the tutorial files to a new directory to which you have read/write permission. If you can read and write to the TestBencher Pro directory then skip down to *Open the Tutorial Project*.

To set up a project directory:

1. Create a new directory or folder called SweepTest1. This directory should be in a location in your path to which your account has full read/write access.
2. Copy the following files from the Examples\Verilog\SweepTest1 or Examples\VHDL\SweepTest1 directory located in the TestBencher installation directory into the new SweepTest1 directory:

   - SweepTest1.hpj

   - tbread.tim

   - tbwrite.tim

   - tbsram.v or **tbsram.vhd** (.v for Verilog users, .vhd for VHDL users)
3. Copy the following files from the TestBencher installation directory into the new **SweepTest1** directory:

   - If you are working with VHDL: wavelib.vhd and syncad.vhd.

   - If you are working with Verilog: wavelib.v and syncad.v

When the tutorial refers to the **SweepTest1** directory, use this new project directory.

ModelSim Integration

If you are going use ModelSim to simulate your projects, you can use the new ModelSim Integration feature to auto-build the project library and launch the simulator with the design loaded (otherwise, skip ahead to *Open the Project*). As a reminder, TestBencher does provide a Verilog simulator, but the ModelSim Integration can be used with both dialects of VHDL or Verilog.

To enable ModelSim Integration:

1. Select the **Options > External program Settings** menu option. This will open the *External Program Settings* dialog.
2. Check the **Enable ModelSim Integration** checkbox. This will disable TestBencher's Verilog simulator and enable the ModelSim Integration feature.
3. Enter the path to the ModelSim executable files in the **ModelSim Executable Path** edit box. This should be the directory that contains the ModelSim executable files (**vlib.exe**, **vsim.exe**, **vcom.exe** and **vlog.exe**).
4. Click **OK** to close the dialog and enable the ModelSim Integration feature.

Open the Tutorial Project:

1. Select the **Project > Open HDL Project** menu option to open the *Open Project File* dialog.
2. Move to either the new **SweepTest1**, **Examples\Verilog\SweepTest1** or **Examples\VHDL\SweepTest1** directory using the browsing features of the dialog.
3. Select **SweepTest1.hpj** file name and click the **Open File** button to load the project into TestBencher. Four files should be displayed in the project window: *tbread.tim*, *tbwrite.tim*, the *tbench1* template file, and the *tbsram* model file.

1 Create a Global Clock Generator with Adjustable Period

A global clock generator can be used to synchronize several timing transactions within a test bench. Typically this is done by using one diagram that runs continuously and concurrently to generate the clock (the clock signal is an output in this diagram). In the synchronized timing transactions, an input clock with the same name as the global clock is added to each diagram. Inside the top-level test bench, TestBencher will automatically connect the global output clock to each of the input clocks. In this section we will create a timing diagram and add a global clock with an adjustable period. The period will be specified in the top-level template file later in the tutorial.

1.1 Create a new timing diagram

1. Select the **File > New Timing Diagram** menu option. This may cause a dialog to open asking if you want to save the current timing diagram. If you have made changes that you want to keep, select **Yes**. Otherwise select **No**. A new timing diagram now opens in the timing diagram editor.

1.2 Create a Free Parameter to control the clock period

Next, add a free parameter which will become the variable that will allow the clock period to be changed by the top-level test bench.

1. In the Parameter window, click the **Add Free Parameter** button. This will add a new free parameter to the Parameter window. [Add Free Parameter]
2. Double click on the name of the new free parameter (**F0**) to open the *Free Parameter Properties* dialog.
3. Type **Period** in the *Name* edit box.
4. Type in the value **100** in the *Min* edit box. The value that you specify for the *Min* property will have no affect on the value of the free parameter in the generated testbench for this tutorial. However, it is necessary to enter a value here so that when we use the free parameter in the clock settings there is something valid to be evaluated for the period.
5. Check the **Is Apply Subroutine Input** checkbox to make this a runtime controllable parameter. This makes the parameter an input to the diagram call (Apply call) in the top-level test bench. If this checkbox were not checked, then the generated code would use the min and max values of the parameter.
6. Make sure the **Enable HDL Code Generation** checkbox is checked. Without this there will be no HDL code generated for the parameter. This feature allows you to turn off code generation for the parameter without removing the parameter from the diagram.
7. Click **OK** to close the *Free Parameter Properties* dialog.

1.3 Add a Clock with an Adjustable Period

Next we will add the global clock. Since this diagram will generate the clock for the entire simulation, we must make sure that the direction of the clock is output and that the clock is exported from the timing transaction. We will use the free parameter created above to allow us to set the period of the clock from the top-level test bench.

To add a clock to the diagram and specify the period:

1. Click the **Add Clock** button [Add Clock] on the signal button bar to open the *Clock Properties* dialog.
2. Type **GClock** in the *Name* edit box.
3. Type **Period** in the *Period Formula* edit box. This will specify the free parameter that we just created as the value of the period for the clock. Note that we could also enter an equation here that included the free parameter as a term, but in this case we just want to perform a direct substitution of the value of the free parameter.
4. Click **OK** to close the *Clock Properties* dialog and add the clock to the diagram.

5. By default, new clocks and signals have a direction of **output** and have their **export signal** check boxes checked. If you want to verify these settings, double click on the clock name to open the *Signal Properties* dialog and look at dialog settings.

1.4 Add Global Clock Generator to the Project

1. Right click on clock name to open a pop-up context menu.
2. Select **Add Diagram to Project** from the menu to open the *Save As* dialog.
3. Type the name **tbglobal_clock.tim** in the *Filename* edit box.
4. Click the **Save** button to save the timing diagram and close the dialog. Notice that the timing diagram is now a part of the project.

   Note: GClock is the only signal in this diagram. In many instances, this is how you will want the global clock to be incorporated into your project because you will want the clock to run for the entire duration of the simulation. During this simulation there are no other processes that will need to run for the entire duration of the simulation.

2 Add Global Clock as an Input to the Read Transaction

Next add the global clock as an input to the read transaction. We will use the global clock to synchronize the output of stimulus from the read cycle.

To add the global clock to the read transaction:

1. In the Project window, double click the **tbread.tim** file name to load the timing diagram in the Timing Diagram Editor window.
2. Click the **Add Clock** button on the button bar above the signal names in the Timing Diagram Editor window. This will open the *Clock Properties* dialog.
3. Type in the name **GClock** in the *Name* edit box. Note this is the same name as the global clock we defined in the clock generator diagram (Capitalization counts).
4. Click **OK** to close the *Clock Properties* dialog.
5. Double click the name **GClock** that was just added to the diagram to open the *Signal Properties* dialog.
6. Set the direction to **input**. Since GClock is located in another transaction, we have to input its value into this transaction.
7. Click **OK** to apply the changes and close the *Signal Properties* dialog. Notice that the waveform for the clock is now blue to indicate that it is an input clock.

3 Change Read Transaction to be Cycle Based

By default, TestBencher will generate time-based transactions which means that an event within a diagram will generate an event within the HDL code at a specific time relative to the beginning of the transaction. However when generating test benches for cycle-based simulators the events in a diagram must be synchronized with a particular clock signal. To change a diagram from time-based to cycle-based, all you have to do is make the signals reference a clock signal. Below is an example of the unclocked code that is generated for the read transaction now, and a sample of the cycle-based code that will be generated once you perform the steps in this section.

Verilog - Unclocked Code Sample	**Verilog - Same edge transitions, off pos edge of GClock:**

#100 //AbsoluteTime 110
ABUS = addr;
#120 //AbsoluteTime 210
ABUS = 8'bxxxxxxxx;
	@ (posedge GClock)
	#10.001 //AbsoluteTime 110
	ABUS = addr;
	@ (posedge GClock)
	#10.001 //AbsoluteTime 210
	ABUS = 8'bxxxxxxxx;

VHDL - Unclocked Code Sample	**VHDL - Same edge transitions, off pos edge of GClock**:

wait until (tb_status = DONE)
for 110 ns;
--AbsoluteTime 110 ns
if (tb_status = DONE) then
exit;
end if;
ABUS <= addr;
DBUS <= "ZZZZZZZZ";
wait until (tb_status = DONE)
for 100 ns;
--AbsoluteTime 210 ns
if (tb_status = DONE) then
exit;
end if;
CSB_Unclocked <= '1';
ABUS <= "XXXXXXXX";
	posEdge(tb_status, GClock);
	wait until (tb_status = DONE)
	for 10.001 ns;
	--AbsoluteTime 110 ns
	if (tb_status = DONE) then
	exit;
	end if;
	ABUS <= addr;
	DBUS <= "ZZZZZZZZ";
	posEdge(tb_status, GClock);
	wait until (tb_status = DONE)
	for 10.001 ns;
	--AbsoluteTime 210 ns
	if (tb_status = DONE) then
	exit;
	end if;
	CSB_GClock_pos <= '1';
	ABUS <= "XXXXXXXX"

Table 1: HDL Code Generated for Clocked and Unclocked Samples

Set the *Clock* property of each Signal to indicate Cycle-based code generation:

1. Select **CSB**, **WRB**, **ABUS** and **DBUS** at the same time. You can do this by either clicking each individual signal name or by clicking the top signal name and dragging across the other signal names.
2. Right click over one of the highlighted signal names to open the pop-up context menu.
3. Select **Edit Selected Signal(s)** from the context menu to open the *Signal Properties* dialog.
4. Choose **GClock** from the drop-down list for the *Clock* property. Any signal or clock in the diagram can be chosen as the clocking signal, in this case we want to use the global clock.
5. Select **pos** from the drop-down list for the *Edge/Level* setting to indicate which edge of the clock to use as the sample edge.
6. Click **OK** to close the *Signal Properties* dialog.

We will also need to change the **End Diagram Marker** so that it will also be attached to **GClock**. Otherwise, the marker would end the timing transaction at a specific time instead of a clock edge.

To change the attachment of the marker to a clock edge:

1. Double click the **End Diagram Marker** to open the *Edit Time Marker* dialog.
2. Select the **Attach to Edge** radio button.

3. Click **OK** to close the dialog and enter a special edge attachment mode. As you move the cursor a green bar will jump around to the closest edge.
4. Left click the third negative edge transition of **GClock** (at approximately 250ns). This will attach the marker to that clock edge.

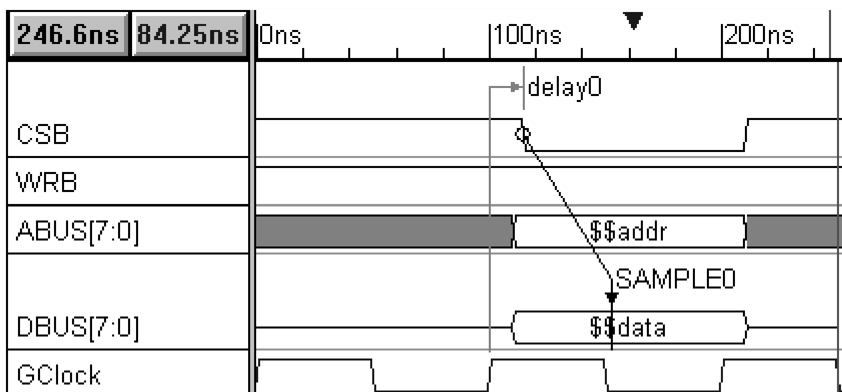
4 Add Sweepable Delay

For this test bench we are sweeping the edge of CSB to determine the range of times over which the circuit will operate properly. In order to sweep an edge's transition time, we must first add a delay to the timing diagram. The value for this delay will also be an Apply Subroutine Input so that we will be able to set the delay value from the top-level test bench. Since the test bench will be in control of the value of the delay, it will be able to change the value of the delay for each execution of the timing transaction, thereby creating the sweep effect.

To add the delay to the read transaction:

1. Left click the **Delay** button Delay on the button bar. The text of the button will turn red when it is selected.
2. Left click the rising edge of the **GClock** signal located at 100ns. This will mark the edge that the delay starts on.
3. Right click the falling edge of the **CSB** signal located at approximately 115ns. This will place the delay in the diagram. The edge that the delay ends on (on signal **CSB**) is the edge that will be delayed. Since GClock is an input to the transaction, the actual time of this edge transition will be determined at run-time by the frequency of GClock, and will be delayed D0 time from the second rising edge of GClock that occurs after the read transaction is started.
4. Double click the delay (**D0**) to open the *Delay Properties* dialog.
5. Check the **Is Apply Subroutine Input** checkbox. Just as with the free parameter we added, this will allow the values of the delay to be controlled from the top-level test bench. Otherwise, the values in the properties dialog would be used.
6. Make sure the **Enable HDL Code Generation** checkbox is checked. This will allow the HDL code for the delay to be generated.
7. Type the name **delay0** in the *Name* edit box.
8. Click **OK** to close the *Delay Properties* dialog.

The portion of the timing diagram that contains the delay should look like this:



We did not bother to enter values for the **min** and **max** properties of the delay because these values will be controlled from the top-level test bench.

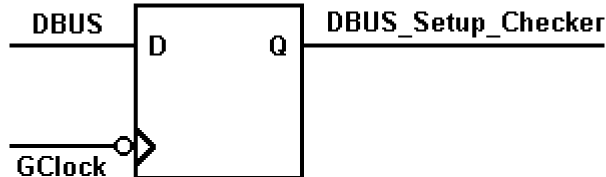
5 Add a Continuous Setup Checker to the Read Transaction

There are two types of setups that can be created using TestBencher. The setup checker we will create is a continuous setup. This type will verify the setup at every edge specified during the timing transaction (every negative edge of **GClock** in this case). The second type of setup, a point setup, can be added using the **setup** button on the button bar in the Timing Diagram Editor window. This type of timing constraint verifies the timing between two specific edges in the diagram to ensure that the signal of interest is stable for the required amount of time.

A continuous setup checker will be used to determine the point of failure in the test of this model. We will specify a specific amount of time for which the value of **DBUS** must be stable before each negative edge of the global clock. Since we are going to sweep the transition timing for the CSB signal (which controls when the read is performed), at some point during the simulation this test will fail, providing the point of failure for the model.

We use a flip-flop (shown in **Figure 1**) to model the continuous setup checker. In order to create the setup checker (the flip-flop), we will need to create a new signal. This signal will be simulated and it will be kept internal to the diagram.

Figure 1: Clock Model used in this tutorial



To add a continuous setup checker to the read transaction:

1. Click the **Add Signal** button on the button bar.
2. Double click the name of the new signal (**SIG0**) to open the *Signal Properties* dialog box.
3. Type **DBUS_Setup_Checker** in the *Name* edit box.
4. Select the **Simulate** radio button so that the signal will be continuously simulated.
5. Set the *Clock* to **GClock** and the *Edge/Level* to **neg**. This will ensure that the continuous setup checker validates at every negative edge of the global clock.
6. Type **DBUS** in the *Boolean Equation* edit box. This setup checker will be used to test DBUS, so that is the signal whose value we will use in the boolean equation.
7. Set the *direction* of the signal to be **internal**. This will make sure that this signal is kept internal to the read component; in this simulation, there is no reason for any other transaction to make use of it.
8. Set the *MSB* value to **7** and the *LSB* value to **0**. In order to verify the correct value, the checker must be of the same bus size as the signal it is testing.
9. Click the **Advanced Register** button to open the *Advanced Register and Latch Controls* dialog.
10. Type the value **10** in the *Setup* edit box of this dialog. This defines the amount of time that the signal being checked must be stable in order to prevent a failure.
11. Click **OK** to close the *Advanced Register and Latch Controls* dialog.
12. Click **OK** to close the *Signal Properties* dialog.
13. Select the **File > Save Timing Diagram** menu option to save the changes made to the timing diagram.

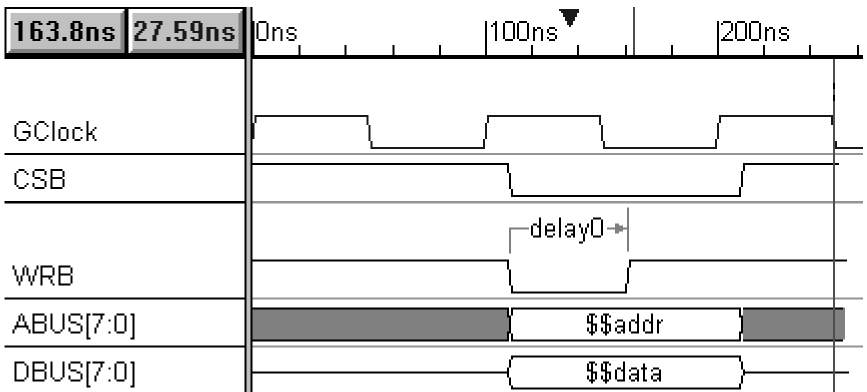
6 Examine the Write Transaction

The write transaction has already been completed for this tutorial, however we will examine the diagram as a review of the types of changes that need to be made to a timing diagram.

To open the write transaction:

1. In the Project window, double click on the **tbwrite.tim** filename.

The timing diagram will look like this:



Verify that the following statements are true:

1. The direction for **GClock** is 'input' which is indicated by a blue signal color. (The direction can be changed using the *Signal Properties* dialog.)
2. Double click **delay0** to open the *Delay Properties* dialog. The **Is Apply Subroutine Input** and the **Enable HDL Code Generation** checkboxes are checked. Remember that the first checkbox makes the min/max timing values of **delay0** into inputs to the Apply call for this transaction, while the second checkbox allows the HDL code to be generated for the delay (as described for the free parameter in Step 1). Click **OK** to close the dialog.
3. Select all signals except **GClock**. Right click over the highlighted signal names and select **Edit Selected Signal(s)** from the pop-up context menu. Make sure that the selected *Clock* is **GClock** and that the *Edge/Level* setting is set to **pos** in the *Signal Properties* dialog. These settings indicate that tbwrite will generate cycle-based code. Click **OK** to close the *Signal Properties* dialog.

7 Applying Timing Transactions to the Template File

The Template file that generates the top-level test bench has been partially completed and added to the project. You will need to add Apply statements that will sequence the timing transactions. By entering the Apply statements in the template file you are defining the execution order of the timing transactions in the final test bench.

First we will add the global clock and set it up to run continuously looping, and concurrently with the other timing transactions. Then we will add the diagram calls for the write transaction and the read transaction. The read and write transactions will be set to run once in a blocking mode so that timing diagrams applied after will have to wait for the current transaction to complete.

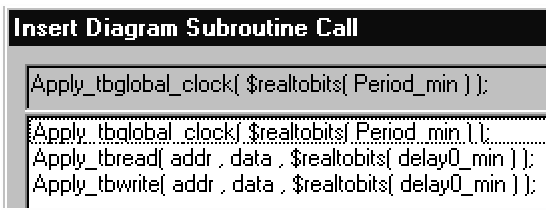
Apply timing transactions

1. In the Project window, double click the **tbench1** template file. This will cause an editor window to open and display the template code.
2. Scroll down in the template file until you find the comment block that has this line:

```
Add apply calls for timing diagram transactions after this comment
```

(This comment is almost at the very end of the file.)

3. Left click in the **tbench1** editor window below this comment so that the blinking cursor is in the place where you want to add the apply statements.
4. Right click in the same place and choose **Insert Diagram Calls...** from the pop-up menu to open the *Insert Diagram Subroutine* dialog. (If you prefer, you can choose the **Editor > Insert Diagram Calls...** main menu function.)



   Notice that there are parameters listed in the tbread and tbwrite subroutine calls. These parameters allow you to provide the input values for the delay min setting and bus values for the data and address buses. The parameter in the global clock subroutine allows you to provide input values for the period of the clock.

5. Choose the **Loop Forever** radio button for the *Run Mode* and the **Concurrent Apply** radio button for the *Wait Mode*. These settings will cause the clock to run for the entire simulation, with other transactions running concurrently.
6. Double left click the **Apply_tbglobal_clock(...)** statement. This will place a diagram call in the template file that looks like:

   For Verilog:

```
// Apply_tbglobal_clock_looping_nowait( $realtobits(Period_min) );
Apply_tbglobal_clock_looping_nowait( $realtobits(Period_min) );
```

   For VHDL:

```
// Apply_tbglobal_clock_looping_nowait(SweepTest1,
//                                     Period_min_real );
Apply_tbglobal_clock_looping_nowait(SweepTest1,Period_min_real );
```

7. Now set the *Run Mode* back to **Run Once** and the *Wait Mode* back to **Wait for Completion**.
8. Double left click the **Apply_tbwrite(...)** statement. This will add an Apply statement for the **tbwrite** transaction to your template file below the apply statement for the global clock.
9. Double left click the **Apply_tbread(...)** statement to add a diagram call for the read transaction.
10. Click the **Close** button to close the *Insert Diagram Subroutine Call* dialog.

The template file now has three diagram subroutine calls listed. We will be using a constant value for the clock period in this simulation, so we will modify the parameters to the global clock diagram call.

11. In the tbench1 editor window, replace the two parameter names in the global clock call with the value '100'. This will provide a period of 100ns for the clock.

```
// Apply_tbglobal_clock ( $realtobits(Period.min) );
Apply_tbglobal_clock( $realtobits(100.0) );    // for Verilog


Apply_tbglobal_clock( SweepTest1 , 100.0 );     -- for VHDL
```

We will edit the read and write apply calls in the next section, when we add HDL code to the sequencer process to generate the changing values for the delay.

8 Adding HDL Code to Sweep the Delay

The diagram subroutine calls were placed in the Sequencer portion of the test bench. The sequencer controls the order and manner in which the timing transactions are executed. In this case, we want to add a **for-loop** around the diagram calls that will change the values of the delays so that the sweep effect is created.

8.1 Add a Variable to the Sequencer

In order to create the **for-loop**, we need a variable to store the current value that we are applying. This variable can then be changed with each iteration of the **for-loop**.

To add the variable:

1. Locate the comment in the template code that says:

   ```
   Sequencer Process
   ```

   (This comment will be just a little bit above the diagram calls that we just inserted.)

2. Insert one of the two variable declarations shown in bold text below (the lines of text that will surround the declaration are provided for the sake of clarity).

   For Verilog (italic code is already in the file):

   ```
   // Sequencer Process
   real delay0; // delay0 will serve as the index and the delay value
   initial
   ```

   For VHDL (italic code is already in the file):

   ```
   process
   variable delay0 : real; -- delay0 will hold the delay value
   begin
   ```

8.2 Add a For-Loop and Input Values to the Diagram Calls

Since we have specified that the global clock will be applied continuously throughout the simulation, we do not want to include it in the **for-loop**. Both the write and the read transactions will be included in the for-loop, so that we can repeat their execution until the point of failure is found.

> Note: You may have noticed that the delays in the read and write transaction have the same name. The two delays are completely independent of one another since they reside in different timing diagrams. TestBencher will not try to relate the two delays. We have given the two delays the same name because (for the sake of simplicity) we are going to provide the same input values for the two delays.

Skip to the section below for the language you are working with, then modify the sequencer code so that it matches what is below. After that is done, we will discuss the changes that were made and what they are doing.

Verilog Users:

Modify the template file so that it matches the following HDL code (code in italic is already present):

```
for (delay0 = 32.0; delay0 > 5.0; delay0 = delay0 - 5.0)
begin
// Apply_Tbwrite( addr , data , $realtobits(delay0_min) );
Apply_Tbwrite( 'hF0 , 'hAE , $realtobits(delay0) );
```

```
// Apply_tbread( addr , data , $realtobits(delay0_min) );
Apply_tbread( 'hF0 , 'hAE , $realtobits(delay0));
end
```

VHDL '93 Users:

Modify the template file so that it matches the following HDL code (code in italic is already present):

```
for i in 0 to 5 loop
delay0 := real (32 - (i * 5));
-- Apply_Tbwrite( SweepTest1 , addr , data , delay0_min_real );
Apply_Tbwrite( SweepTest1 , x"F0" , x"AE" , delay0 );

-- Apply_tbread( SweepTest1 , addr , data , delay0_min_real );
Apply_tbread( SweepTest1 , x"F0" , x"AE" , delay0 );
end loop;
```

VHDL '87 Users:

Modify the template file so that it matches the following HDL code (code in italic is already present):

```
for i in 0 to 5 loop
delay0 := real (32 - (i * 5));
-- Apply_Tbwrite( SweepTest1 , addr , data , delay0_min_real );
Apply_Tbwrite( SweepTest1 , "11110000" , "10101110" , delay0 );

-- Apply_tbread( SweepTest1 , addr , data , delay0_min_real );
Apply_tbread( SweepTest1 , "11110000" , "10101110" , delay0 );
end loop;
```

Evaluating the HDL Code

The **for-loop** that we have entered will repeat 6 times. The input value for the delays will be reduced by five with each iteration, producing the sweep effect. Two timing errors will be reported. The first will be in the first loop and will be caused by the fact that the falling edge of CSB does not set up properly before the next falling edge of the clock (this happens because the delay is too long). The second error will be reported by the continuous setup checker during the final iteration. This error will report a message for each bit of DBUS, for a total of eight messages.

The loop that we have created will cause the read and write transactions to iterate once for each iteration of the loop unconditionally.

8.3 Ending the Simulation from the Test Bench

Since we have applied the global clock to run in a continuous loop, the simulation will never end unless we manually end the execution of the global clock once the other transactions have completed. Normally you would have to end the simulation manually through the simulator for this sort of case. We have provided a special **Abort** call for each timing transaction that you can use to end the execution of that transaction. To make use of this, add the following line to the HDL code after the end of the **for-loop**:

For Verilog:

```
Abort_tbglobal_clock;
```

For VHDL:

```
Abort_tbglobal_clock(SweepTest1);
```

This subroutine call will end the execution of the global clock. This will ensure that the simulation will end without the need for manual interference.

8.4 Save the Template File

Save your template file by right clicking in the template file editor window and selecting **Save** from the pop-up menu.

9 Generating the Test Bench and Simulating the Project

Before simulating the project, we must generate the HDL code for the top-level test bench. To generate this code:

1. Select the **Export > Generate Test Bench** menu option. This will expand the macros in the template file, with the generated HDL code for the top-level test bench in the file.

Simulating the Project with TestBencher's Built in Verilog Simulator

To simulate the project:

1. Click the yellow TB test bench compile button.
2. Click on the green simulation run button to simulate the project. This will run the simulation.

Simulating the Project Using the ModelSim Integration Feature

If you are using the ModelSim integration feature, click the yellow tb button TB to autolaunch ModelSim and build the Project libraries. See Section 2.5 of the TestBencher manual (or on-line help) for more information about ModelSim integration.

Simulating the Project Using Another Third Party Simulator

The files that you will need are:

1. The HDL files for the MUT (your original model code). For this tutorial either **tbsram.v** or **tbsram.vhd** is the MUT file.
2. One HDL code file for each timing transaction. These files are named after the timing diagram names. For this tutorial you will have either **tbglobal_clock.v**, **tbread.v** and **tbwrite.v**, or **tbglobal_clock.vhd**, **tbread.vhd** and **tbwrite.vhd** depending on language.
3. One top-level HDL test bench file (this is the expanded template file). For this tutorial it is named tbench1.v or tbench1.vhd.
4. The **wavelib_inertial.v** or **wavelib.vhd** library file (found in the TestBencher installation directory).
5. For VHDL, there are two additional package files you must include in your work directory: **syncad.vhd** which contains generic types and tasks used by TestBencher and a project-specific package file called **tb_SweepTest1_tasks.vhd**, where SweepTest1 is the name of the project.

This completes the *Performing a Sweep Test* tutorial. During the course of this tutorial we have discussed:

1. Creating a global clock and synchronizing transaction stimulus with the clock.
2. Creating a free parameter for use as an input to a timing transaction.
3. Creating a delay whose values are an input to the timing transaction.
4. Controlling the execution mode of a timing transaction.
5. Adding HDL code to the sequencer process of the template file.

6. Aborting a continuously executing transaction from the top-level test bench.

Another Method of Creating a Sweep Test

An example project has been created for you to examine that demonstrates another method of performing a sweep test. The second method uses a timing-based test bench instead of the cycle-based test bench used in this tutorial. In the second example, edge transitions are made relative to the clock edges using delays. The method that you choose to use will depend upon the specific model that you are testing.

Finding Out More

Additional example files can be found in the **Examples** subdirectory of the installation directory. Many of these files contain text in the timing diagram that specifies what the timing diagram demonstrates. In addition, there is a **readme.txt** file in the Examples directory that specifies the contents of each subdirectory of the **Examples > VHDL** and **Examples > Verilog** directories.

Appendix C: License Agreement

SynaptiCAD

TestBencher Pro - DataSheet Pro - WaveFormer Pro - Timing Diagrammer Pro - VeriLogger Pro

Software License Agreement

- Read Before Use -

Please read and understand this license.

Note: Throughout this agreement, the word Software refers to the software product that you have licensed from SynaptiCAD.

You have purchased a license to use one of the following products: **TestBencher Pro, DataSheet Pro, WaveFormer Pro**, **VeriLogger Pro**, or **Timing Diagrammer Pro** software. The software is owned and remains the property of SynaptiCAD, is protected by international copyrights, and is transferred to the original purchaser and any subsequent owner of the Software media for their use only on the license terms set forth below. Opening the packaging for **TestBencher Pro, DataSheet Pro, WaveFormer Pro, VeriLogger Pro,** or **Timing Diagrammer Pro** and/or using either **TestBencher Pro, DataSheet Pro, WaveFormer Pro, VeriLogger Pro,** or **Timing Diagrammer Pro** indicates your acceptance of these terms. If you do not agree to all of the terms and conditions, return the unopened Software and manuals immediately for a full refund.

Use of the Software

- **SynaptiCAD** grants the original purchaser ("Licensee") the limited rights to possess and use the Software and User's Manual ("Software") for its intended purpose. Licensee agrees that the Software will be used solely for Licensee's internal purposes and that the Software will be installed on a single computer only. If the Software is installed on a networked system, or on a computer connected to a file server or other system that physically allows shared access to the Software, Licensee agrees to provide technical or procedural methods to prevent use of the Software by more than one user.
- One machine-readable copy of the Software may be made for BACKUP PURPOSES ONLY, and the copy shall display all proprietary notices, and be labeled externally to show that the backup copy is the property of SynaptiCAD, and that use is subject to this License.
- Use of the Software by any department, agency or other entity of the U.S. Federal Government is limited by the terms of the below "Rider for Governmental Entity Users."
- Licensee may transfer its rights under this License, provided that the party to whom such rights are transferred agrees to the terms and conditions of this License, and written notice is provided to SynaptiCAD. Upon such transfer, Licensee must transfer or destroy all copies of the Software.
- Except as expressly provided in this License, Licensee may not modify, reverse engineer, decompile, disassemble, distribute, sub-license, sell, rent, lease, give or in any other way transfer, by any means or in any medium, including telecommunications, the Software. Licensee will use its best efforts and take all reasonable steps to protect the Software from unauthorized use, copying or dissemination, and will maintain all proprietary notices intact.

LIMITED WARRANTY **SynaptiCAD** warrants the Software media to be free of defects in workmanship for a period of ninety days from the purchase. During this period, SynaptiCAD will replace at no cost any such media returned to SynaptiCAD, postage prepaid. This service is SynaptiCAD's sole liability under this warranty.

DISCLAIMER LICENSE FEES FOR THE SOFTWARE DO NOT INCLUDE ANY CONSIDERATION FOR ASSUMPTION OF RISK BY SYNAPTICAD, AND SYNAPTICAD DISCLAIMS ANY AND ALL LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR OPERATION OR INABILITY TO USE THE SOFTWARE, EVEN IF ANY OF THESE PARTIES HAVE BEEN ADVISED OF THE POSSIBILITIES OF SUCH DAMAGES. FURTHERMORE, LICENSEE IN-

DEMNIFIES AND AGREES TO HOLD SYNAPTICAD HARMLESS FROM SUCH CLAIMS. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY THE LICENSEE. THE WARRANTIES EXPRESSED IN THIS LICENSE ARE THE ONLY WARRANTIES MADE BY SYNAPTICAD AND ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND OF FITNESS FOR A PARTICULAR PURPOSE. THIS WARRANTY GIVES YOU SPECIFIED LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM JURISDICTION TO JURISDICTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF WARRANTIES, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Term

- This license is effective as of the time Licensee receives the Software, and shall continue in effect until Licensee ceases all use of the Software and returns or destroys all copies thereof, or until automatically terminated upon failure of Licensee to comply with any of the terms of this License.

General

- This License is the complete and exclusive statement of the parties' agreement. Should any provision of this license be held to be invalid by any court of competent jurisdiction, that provision will be enforced to the extent permissible, and the remainder of the License shall nonetheless remain in full force and effect. This License shall be controlled by the laws of the State of Virginia, and the United States of America.

Rider For U.S. Governmental Entity Users

This is a rider to **TestBencher Pro/ DataSheet Pro/ VeriLogger Pro/ WaveFormer Pro/ Timing Diagrammer Pro** SOFTWARE LICENSE AGREEMENT ("License") and shall take precedence over the License where a conflict occurs.

1. The Software was: developed at private expense (no portion was developed with government funds) and is a trade secret of SynaptiCAD and its licensor for all purposes of the Freedom of Information Act; is "commercial computer software" subject to limited utilization as provided in any contract between the vendor and the government entity; and in all respects is proprietary data belonging solely to SynaptiCAD and its licensor.
2. For units of the DoD, the Software is sold only with "Restricted Rights" as that term is defined in the DoD Supplement to DFAR 252.227-7013 (b)(3)(ii), and use, duplication or disclosure is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Manufacturer: SynaptiCAD, PO Box 10608, Blacksburg, Va 24062-0608 USA.
3. If the Software was acquired under a GSA Schedule, the Government has agreed to refrain from changing or removing any insignia or lettering from the Software or Documentation or from producing copies of manuals or disks (except for backup purposes) and: (1) Title to and ownership of the Software and Documentation and any reproductions thereof shall remain with SynaptiCAD and its licensor; (2) use of the Software shall be limited to the facility for which it is acquired; and (3) if the use of the Software is discontinued at the original installation location and the Government wishes to use it at another location, it may do so by giving prior written notice to SynaptiCAD, specifying the new location site and class of computer.
4. Government personnel using the Software, other than under the DoD contract or GSA Schedule, are hereby on notice that use of the Software is subject to restrictions that are the same or similar to those specified above.

Index

A

Abort Call 49
- ending global clock execution 56

Absolute Samples 27
Adding
- diagrams to project 11
- HDL file(s) 9
- signals to diagram 10
- template file to project 11

Advanced Register and Latch Controls 25
Apply Statements
- see diagram calls 12, 47

Attach to Time 33
Auto Run 9

B

Bi-directional Signals 10
Breakpoints 41
Building Project 17
- files used 57

Bus
- adding 21

C

Clocks 23
- adding 21

Compilation Errors
- finding 41

Continuous Setups and Holds 24
Copy TestBencher Template File 11, 45
Creating New Sample Types 31
Customization
- editor preferences 42
- language 9
- project settings 9
- template files 49
- test bench preferences 9

D

Debug Run 9
Delay Settings
- project default 15

Delays 25
- adding to timing transaction 25
- HDL code generation 26
- properties 25, 26
- timeout 26

Design Flow 51
Diagram Calls 12, 47
- aborting 49
- inserting 12, 47
- modifying 12
- setting state variables 48

Diagram Level Test Bench Settings 37
- dialog 37
- HDL code generation 39
- timeout 39

Direction
- inout 22
- input 22
- of signals 22
- output 22
- persistent inout 22
- persistent output 22
- shared output 22

E

Edge/Level 23
Editing Multiple Signals 23
Editor Preferences 42
- background color 42
- color highlighting 42
- color printing 42
- font 42
- tab width 42

Editor Window 42
End Diagram Marker 10
Errors
- test bench generation 52

Execution Mode
- controlling 48

Execution Modes 47, 56
External Editors 43
External HDL Code Library Files 38

G

Generating the Test Bench 51
Global Clocks
- creating 55
- ending 56

Glue Logic
- test bench-level 55

Goto Button 41

H

HDL Code Generation
- enabling/disabling 38
- for timing diagrams 57
- markers 33
- samples 28

HDL Files
- closing 41
- goto specific line number 41
- opening 41
- saving 41

HDL Library Files 60
Holds
- continuous 24

I

Include Directories
- library directories 16
- project settings 16

Include Files 38

J

Jump To Dialog 41

L

Language 9
- default 9
- setting default 17

Libraries
- directories 16
- library extensions 16
- samples 28
- using HDL code libraries 38

Looping Markers 10

Loops
- in test benches 34
- in transactions 56

M

Macros
- in the template file 45

Markers 33
- absolute 33
- adding to timing diagrams 33
- end diagram 10, 34
- HDL code generation 33
- looping 10, 34
- relative 33
- that execute HDL code 35

Model Under Test
- see MUT 46

ModelSim Integration 19
- enabling 19
- simulation using 20
- specifying executable path 19

MUT 46
- adding to project 9
- example of instantiation 46
- in template file 46
- instantiation 11, 46

P

Point Samples 27

Project Settings 9
- capture and show watched signals 16
- delay settings 15
- dump watched signals 16
- grab top level signals 15
- hide empty lists 15
- include directories 16
- interactive mode after compile 16
- language 16
- library directories 16

Project Tree 18, 41

Projects
- adding diagrams 11
- adding files 9, 17
- adding HDL files 17
- adding MUT 9
- adding the template file 18
- adding timing diagrams 17
- building 17
- creating 15
- default settings 53
- language 9
- opening 15
- preferences 9, 15
- project settings dialog 9
- project tree control 18
- saving 15

R

Relative Samples 27

Run Mode 47, 56

S

Sample Actions
- based on conditions 29
- user-defined 29

Sample Conditions
- if conditions 29
- user-defined 29

Samples 27
- absolute 27
- as transaction outputs 30
- creating new types 31
- example of user-defined 30
- HDL code generation 28
- point 27
- properties dialog 27
- relative 27
- self-testing 10
- self-testing code 28
- user-defined 10
- user-defined actions 29
- window 27

Search Dialog 41

Self-Testing Code 10
- samples 28

Setups
- continuous 24

Signal Properties 23
- advanced register 24
- bit width 10
- direction 10
- LSB 10
- MSB 10
- name 10
- type 10

Signals
- adding 21
- adding to diagram 10
- bi-directional 10, 22
- clock-driven 23

direction 22
see Signal Properties 10
Sim Diagram & Project 9
Simulate Diagram Button 14
Simulate Project 9
Simulation
test bench 52
Simulation Mode
auto run 9
debug run 9
Simulation States
Sim Diagram & Project 9
Simulate Project 9
Source Files
see HDL files 41
State Transitions
for internal sequence 58
of diagram status signals 57
State Variables 48
Status Signals 57
and state transitions 57
internal 58
process 58
Status States 57

T

Tasks
generation 60
Template Files 45
adding diagram calls 47
adding HDL code 49
adding to a project 45
copying 11, 18, 45
customization 49
MUT instantiation 11, 46
Test Bench
generating 51
generation 13, 18
generation errors 52
preferences 9
project defaults 53
simulating Verilog 14
simulation 52
Third Party Simulation
ModelSim integration 19
Third Party Simulators 20
Timeouts
for delays 26
timing diagram 39
Timing Diagrams
components 21
creating 21
drawing 21
loops in 56
timeout 39
variables 25
Timing Transactions
generated tasks 60
Transactions
aborting 49
monitoring execution 59

U

Use Files 38
User-defined Samples 10

V

Variables 10, 25
defining state 25
state 10, 25
VHDL
task generation 57
Virtual Edit Box 25

W

Wait Mode 47, 56
Watches 59
component 59
connection 59
setting 59
Signals 59
Waveforms
drawing 21
Window Samples 27
defining 27