

# SPV Programmer's Guide

# Table of Contents

<b><u>1. INTRODUCTION</u></b>	<b><u>4</u></b>
<b><u>2. GETTING STARTED</u></b>	<b><u>5</u></b>
2.1. CONVENTIONS	5
2.2. SPV APP WIZARD (NOT RELEVANT TO VISUAL EXPRESS USERS)	5
2.2.1. USING THE WIZARD	5
2.2.2. STEP 1: FILE→NEW	5
2.2.3. STEP 2: CHOOSE SPVAPPWIZARD AND SUPPLY A NAME FOR THE PROJECT	6
2.2.4. STEP 3: CHOOSE HDL LANGUAGE, SIMULATOR, AND PROJECT TYPE	6
2.2.5. STEP 4: DIRECTORY SETUP	7
2.2.6. STEP 6: OPTIONALLY LINK TO DPF LIBRARY	8
2.2.7. STEP 6: FINISH AND BUILD	9
2.2.8. SETTING UP THE DEBUGGER	10
2.2.9. STEP 1: RIGHT CLICK ON PROJECT NAME AT LEFT PANE, AND CHOOSE “PROPERTIES” FOR DEBUGGER SETTINGS FORM	10
2.2.10. STEP 1: CHOOSE DEBUG TAB AND FILL IN THE FIELDS	11
2.3. SPVCPPSIM (OPTIONAL)	13
2.3.1. VISUAL C++ 2005	13
2.4. HDL SIMULATION	16
2.5. STUB FILES	21
2.5.1. VERILOG	21
2.5.2. VHDL	22
<b><u>3. SIGNALS AND PROCESSES</u></b>	<b><u>23</u></b>
3.1. FIRST STEPS	23
3.1.1. DRIVING SIGNALS	23
3.2. SIGNALS	26
3.2.1. WHAT IS A SIGNAL?	26
3.2.2. LOGICAL EXPRESSIONS	30
3.3. PROCESSES	31
3.3.1. WHAT IS A PROCESS?	31
3.3.2. PROCESS TYPES	31
3.3.3. EXECUTING PROCESSES	32
<b><u>4. USING BIT VECTORS</u></b>	<b><u>33</u></b>
4.1. OVERVIEW	33
4.1.1. WHAT IS A BIT VECTOR?	33
4.1.2. OTHER BIT VECTOR CLASSES	34
4.2. BIT VECTOR DATA	34
4.2.1. CONSTRUCTION AND INITIALIZATION	34
4.2.2. BIT VECTORS, INTEGERS, AND OPERATORS	37
4.3. DISPLAYING BIT VECTORS	38
4.4. DEBUGGING SUPPORT	39
4.5. ABOUT HEADER FILES	40
4.6. REFERENCE CLASSES	40
4.6.1. SLICE	40

4.6.2.	BOOL	42
<b>4.7.</b>	<b>BIT VECTOR ARRAYS</b>	<b>42</b>
4.7.1.	DEBUGGING BIT VECTOR ARRAYS	44
<b>4.8.</b>	<b>AGGREGATING AND DISPENSING BITS</b>	<b>44</b>
<b><u>5.</u></b>	<b><u>GENERATION</u></b>	<b><u>46</u></b>
<b>5.1.</b>	<b>WHAT IS GENERATION?</b>	<b>46</b>
<b>5.2.</b>	<b>GENERATORS</b>	<b>46</b>
5.2.1.	NON-RANDOM (DETERMINISTIC)	47
5.2.2.	RANDOM	49
5.2.3.	USER DEFINED	51
5.2.4.	COMPOSITE GENERATION	52
5.2.5.	FILE DEFINED GENERATORS	55
<b><u>6.</u></b>	<b><u>COVERAGE</u></b>	<b><u>62</u></b>
<b>6.1.</b>	<b>WHAT IS COVERAGE?</b>	<b>62</b>
6.1.1.	DEFINITION	62
6.1.2.	TYPES OF COVERAGE	62
6.1.3.	COVERAGE CLASSES	62
6.1.4.	ADAPTIVE GENERATION	66
<b><u>7.</u></b>	<b><u>COLLECTING AND COMPARING</u></b>	<b><u>68</u></b>
7.1.1.	COLLECTION PROCESSES	69
7.1.2.	COMPARE	71
7.1.3.	TRANSFER FUNCTION	71
<b><u>8.</u></b>	<b><u>CALLING MATLAB ROUTINES FROM SPV</u></b>	<b><u>72</u></b>
<b>8.1.</b>	<b>SETUP</b>	<b>72</b>
<b>8.2.</b>	<b>TWO WAYS TO CALL</b>	<b>72</b>
<b>8.3.</b>	<b>DPFML INTERFACE CLASSES</b>	<b>73</b>
<b>8.4.</b>	<b>CALLING THE ROUTINES</b>	<b>75</b>
<b>8.5.</b>	<b>MORE USAGE</b>	<b>78</b>

# 1. Introduction

The SPV documentation is divided into two main parts, the Programmer's Guide (this document) and the Reference Guide. Whereas the Reference Guide is a dry class-by-class and function-by-function description of the components that make up SPV, the Programmer's Guide is intended to be a tutorial for the uninitiated.

The Programmer's Guide is organized as a logical progression of chapters starting with the basics and progressing towards more advance topics, with examples provided all along the way.

- The first chapter, "Getting Started" describes how to set up a verification project with SPV and takes the first steps in programming SPV.
- "Signals and Processes" delves into how HDL signals can be manipulated from within SPV and how to make C++ code sensitive to changes in the simulation.
- The next chapter, "Bit Vectors" covers the basics of bit vector manipulation.
- The "Generation" chapter shows you how to create complex random and non-random generators for creating verification stimulus.
- The "Coverage" chapter teaches how to use the SPV coverage classes to confirm the thoroughness of the test suite. Also, it shows how to use the coverage database at runtime for more efficient generation.

## 2. Getting Started

This chapter describes the necessary steps required to set up a the beginning of a verification project including the stub files that must be added to the HDL compilation. Most users should use the SpvAppWizard, which supports Active-HDL and ModelSim, that is installed in the the VS 6 environment as part of the SPV install. This, by far, is the easiest way to get started. For other simulators or for SpvSim modeling projects, the steps are described in the sections following the wizard description.

### 2.1. Conventions

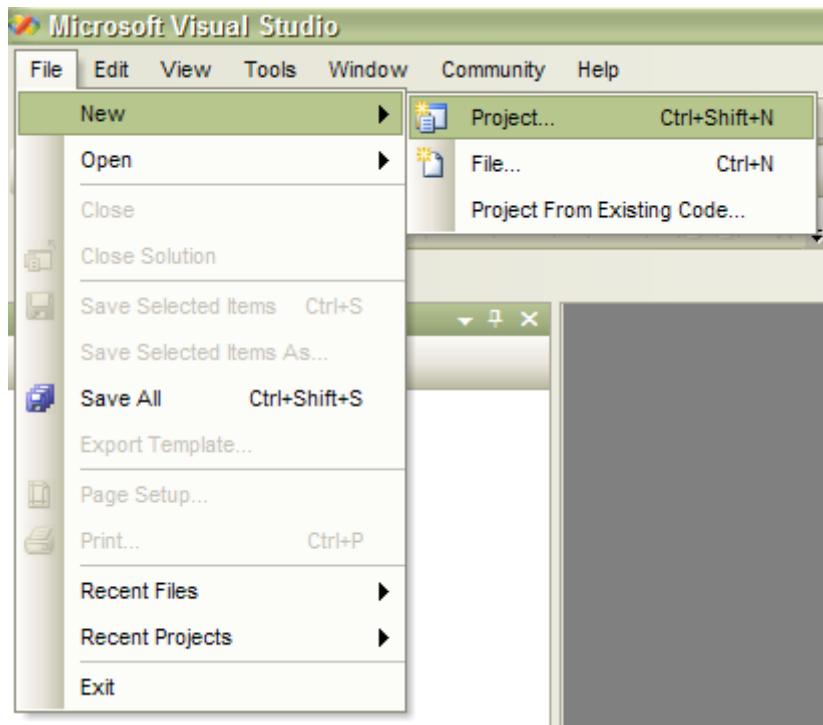
- SPV reserved words, classes, and functions appear in **BOLD**.
- A chain of selections in a menu hierarchy is described with arrows. For example: File→Open.
- Captions of text boxes, buttons, and other GUI elements (except menus) are enclosed in double quotation marks ("").
- In general, words that will have to be replaced by the user with some other text as written in *Italics*.
- Text that must be typed by the user appears in the Ariel font.
- Whenever the word *SpvDirectory* appears in *Italics*, substitute the **SimPlus** installation directory.
- Whenever the word *SpvVersion* appears in *Italics*, substitute the version of **SPV** that you have.
- Whenever the word *HDL* appears in *Italics*, substitute either Verilog or Vhdl, as appropriate to the HDL language you are using.

### 2.2. SPV App Wizard (Not relevant to Visual Express Users)

Application Wizard is disabled in Visual Express 2005, but it is possible to receive the preped demo from costumer support.

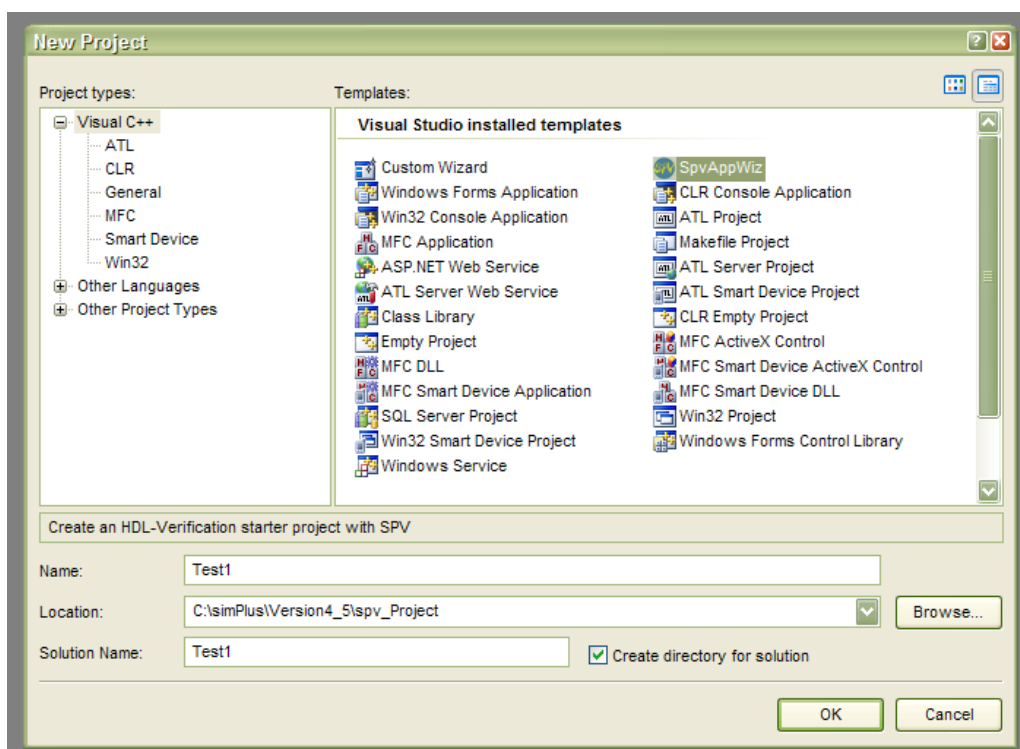
#### 2.2.1. Using the wizard

#### 2.2.2. Step 1: File→New



### 2.2.3. Step 2: Choose SpvAppWizard and supply a name for the project

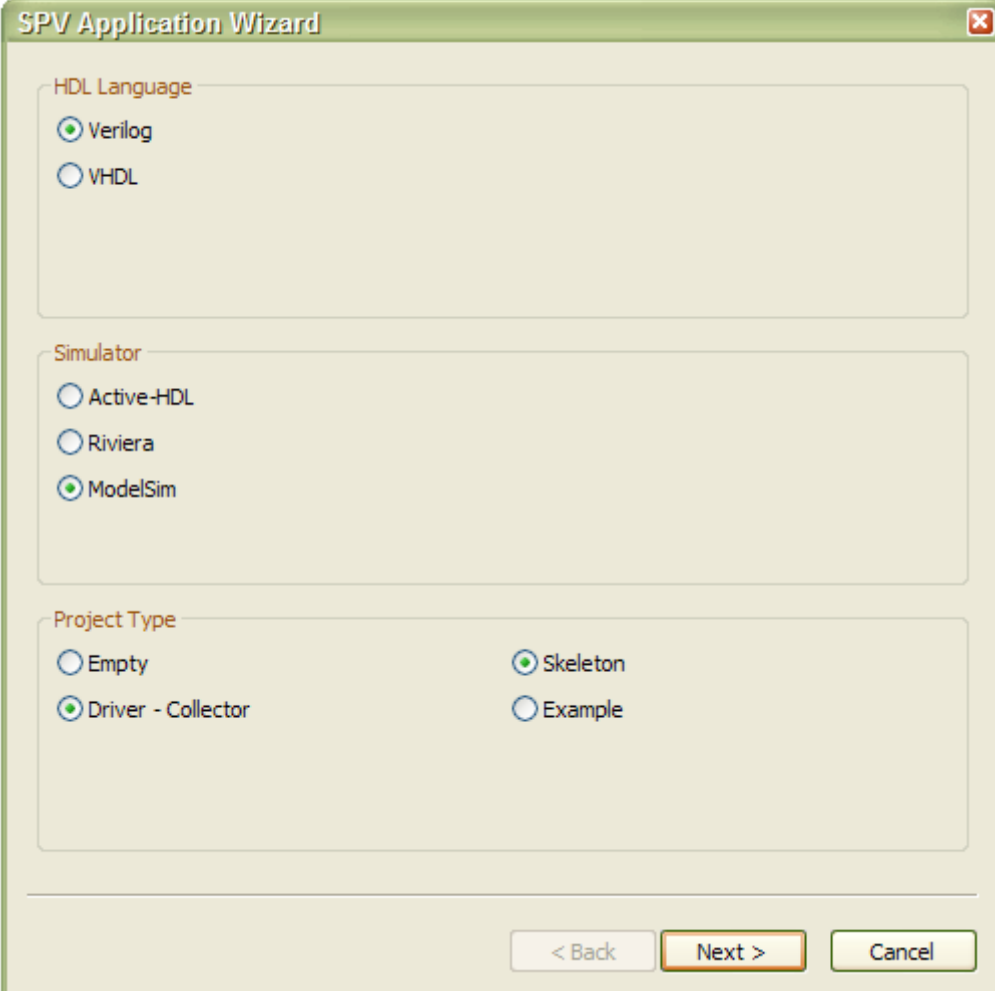
This example uses “Test1” as the project name.



### 2.2.4. Step 3: Choose HDL language, simulator, and project type

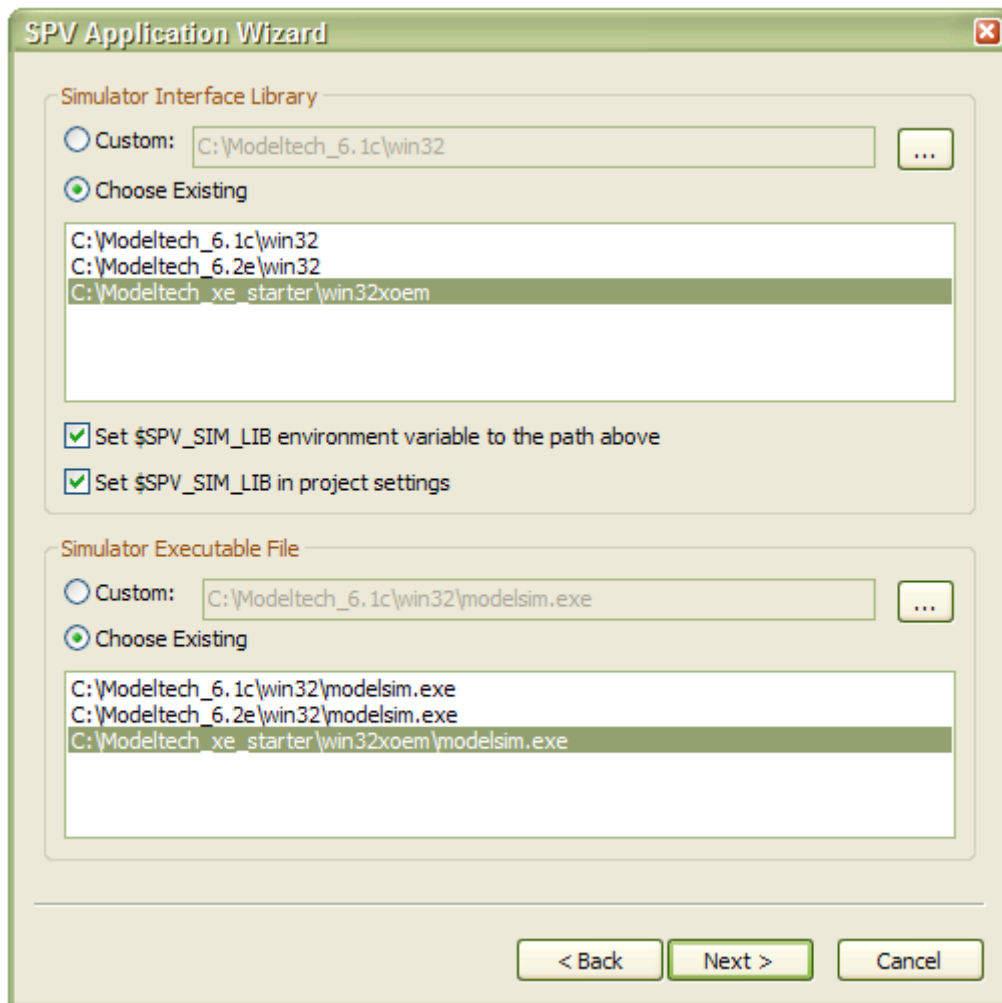
The “Empty” project creates just the SPV entry point function and no more. The “Driver-Collector” project (which is what we will use in this book)

creates a somewhat more developed structure. This structure includes a driver/generator to fabricate and insert stimuli and a collector/comparator to monitor the DUT and compare the output to the expected values. Press the “?” button for a schematic diagram.  
(The examples in the this book were created using the Wizard with the default choices.)

The image shows a Windows-style dialog box titled "SPV Application Wizard". It has a green title bar with a close button (X) in the top right corner. The dialog is divided into three sections, each with a title and a group of radio buttons. The first section is titled "HDL Language" and contains two radio buttons: "Verilog" (which is selected, indicated by a green dot) and "VHDL". The second section is titled "Simulator" and contains three radio buttons: "Active-HDL", "Riviera", and "ModelSim" (which is selected, indicated by a green dot). The third section is titled "Project Type" and contains four radio buttons arranged in two columns: "Empty", "Driver - Collector" (selected, indicated by a green dot), "Skeleton" (selected, indicated by a green dot), and "Example". At the bottom of the dialog, there are three buttons: "< Back", "Next >" (highlighted with an orange border), and "Cancel".

#### 2.2.5. Step 4: Directory setup

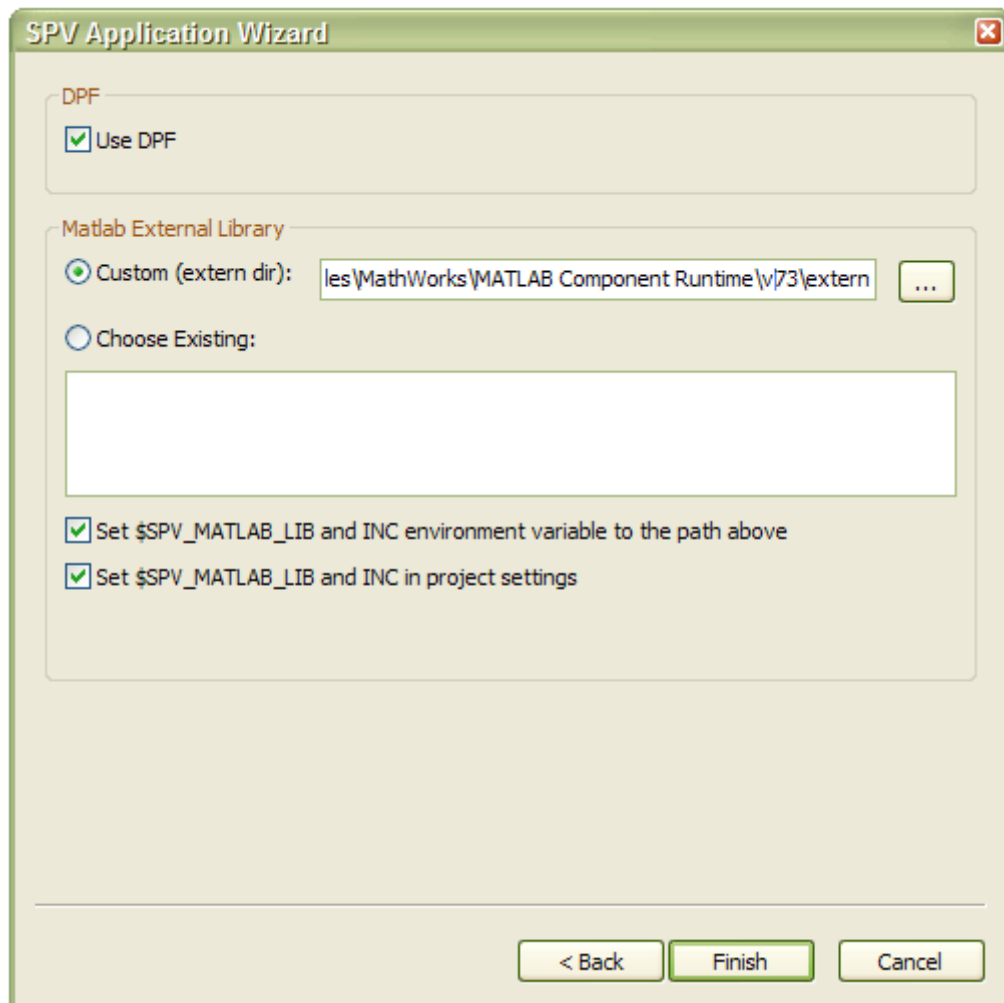
This form searches for installed simulators and displays the directories it has found for the interface libraries (for linking) and the simulator executable (for compiling the sample HDL). The Wizard will create or modify an environment variables, \$SPV\_SIM\_LIB, based on these choices, as long as the “Set \$SPV\_SIM\_LIB environment variable to the path above” is set. If you have changed the simulator directory from the last execution of the wizard, or this is the first time the wizard is run, then after Step 5, below, you will have to shut down the MSVC 6 environment and restart it so that the new environment variable will take effect.



### 2.2.6. Step 6: Optionally link to DPF library

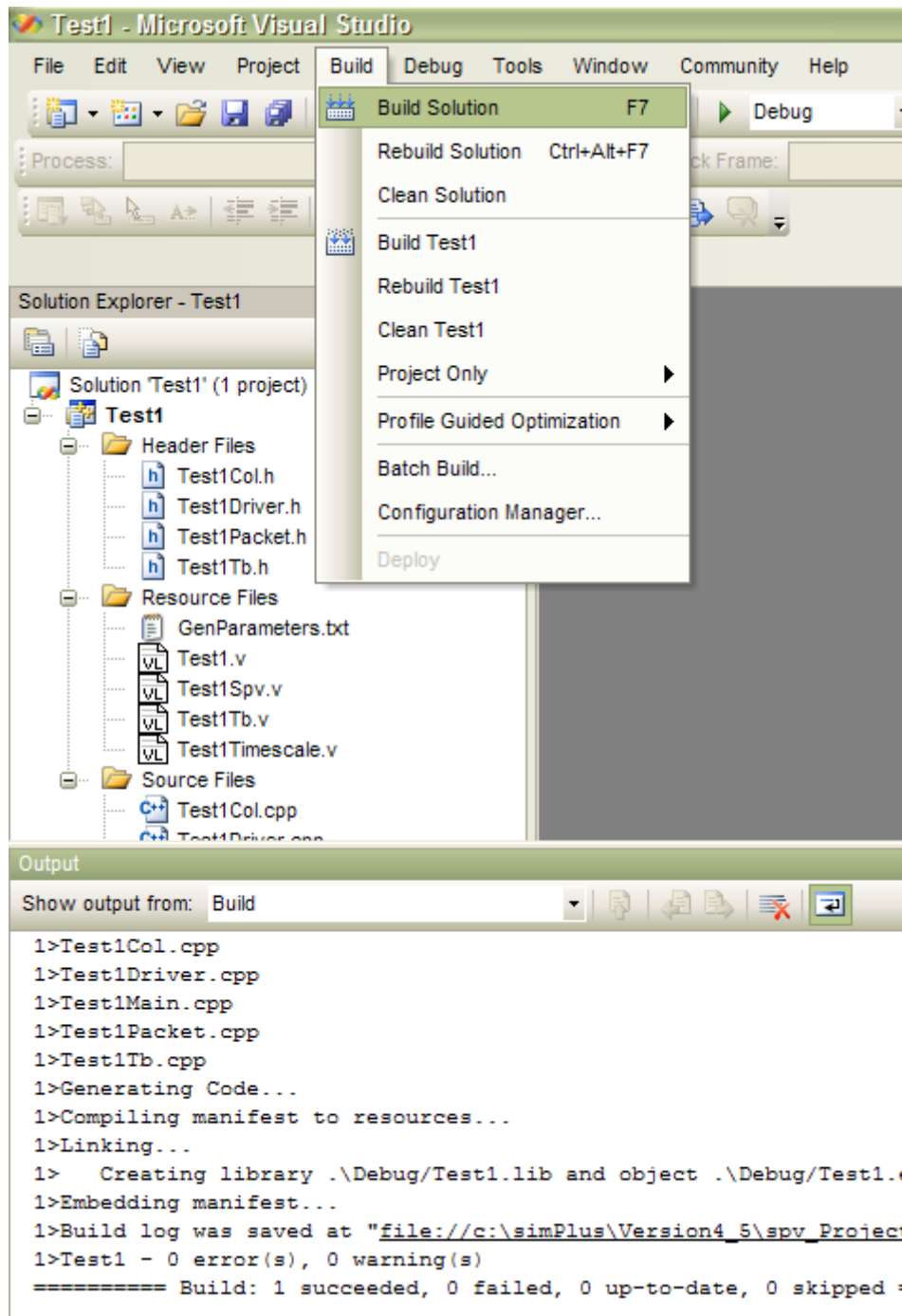
This form gives you the option of link in the DPF library to your project. The main feature here reason to do so is the Matlab interface classes that you can use to ease calling Matlab routines at runtime (covered later in this book). Similar to the simulator form, this form will search for Matlab installations and give you the ability to choose the appropriate directory. Here, two environment variables will optionally be set, \$SPV\_MATLAB\_INC and \$SPV\_MATLAB\_LIB. As for the previous form, you may have to shut down and restart the Visual Studio and restart it so that the new environment variable will take effect.





### 2.2.7. Step 6: Finish and Build

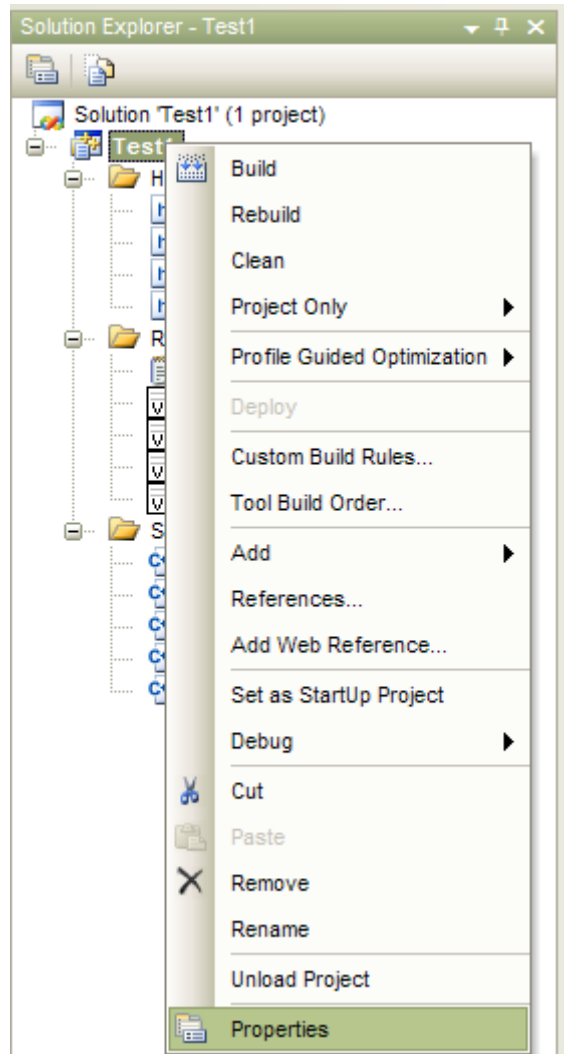
At this point the wizard will complete. It will launch the simulator, compile the sample HDL code and create a C++ project. As mentioned above, you may have to shut down and restart the Visual Studio 2005 environment and reopen the workspace – you will see a pop-up box telling you to do so, if it is indeed necessary. At this point, you may build the project with Build→Build Solution



### 2.2.8. Setting up the debugger

The wizard is not capable of setting up the debugger. You must do so manually. These are the steps.

#### 2.2.9. Step 1: Right Click on project name at left pane, and choose “Properties” for Debugger Settings Form



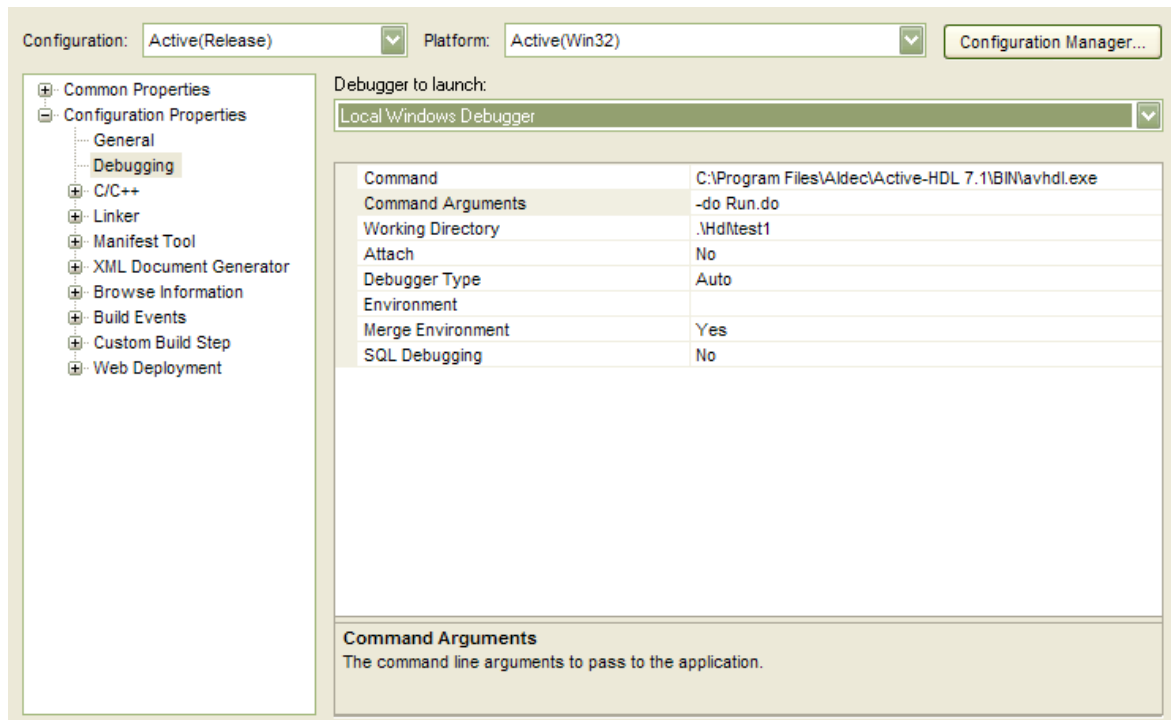
### 2.2.10. Step 1: Choose Debug tab and fill in the fields

Examples for ModelSim and ActiveHDL as shown below.

Notice that the command line argument has a `-do` switch that will cause the simulator to run the `Run.do` script. There you will find the actual commands that are passed to the simulator. They will include a design load, possibly a directory change, a wave viewer command to display the top level signals, and a run command.

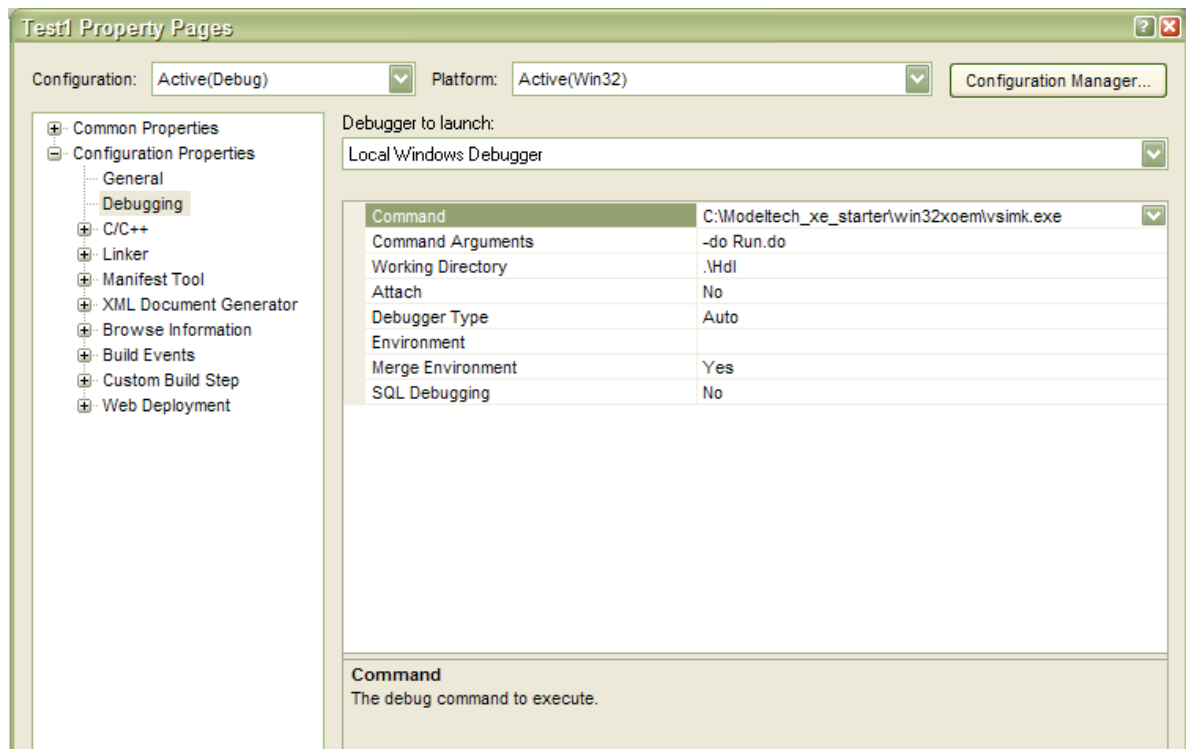
#### ○ ActiveHDL

Remember to substitute the project name you supplied to the wizard instead of "Test1" in the "Working directory:" field.



## ○ ModelSim

For ModelSim, note that the executable for **debug** is vsimk.exe and NOT vsim.exe or modelsim.exe. If modelsim.exe or vsim.exe is written, replace it by writing vsimk.exe. This is because vsim spawns a vsimk process, which actually runs the simulation, but the debugger will not be aware of this.



## 2.3. SpvCppSim (Optional)

Null simulation is used when developing verification code without a simulator. A null simulator enables a C++ project to be built without linking to a HDL simulator. This allows for verification development to progress before the hardware development has advanced to the point where it can be tested without taking up a license for the HDL simulator. Later, when the hardware is ready for testing, the project can easily be converted to work with the hardware simulation.

The steps required for setting up **SPV** with a null simulation, in general, require the user to:

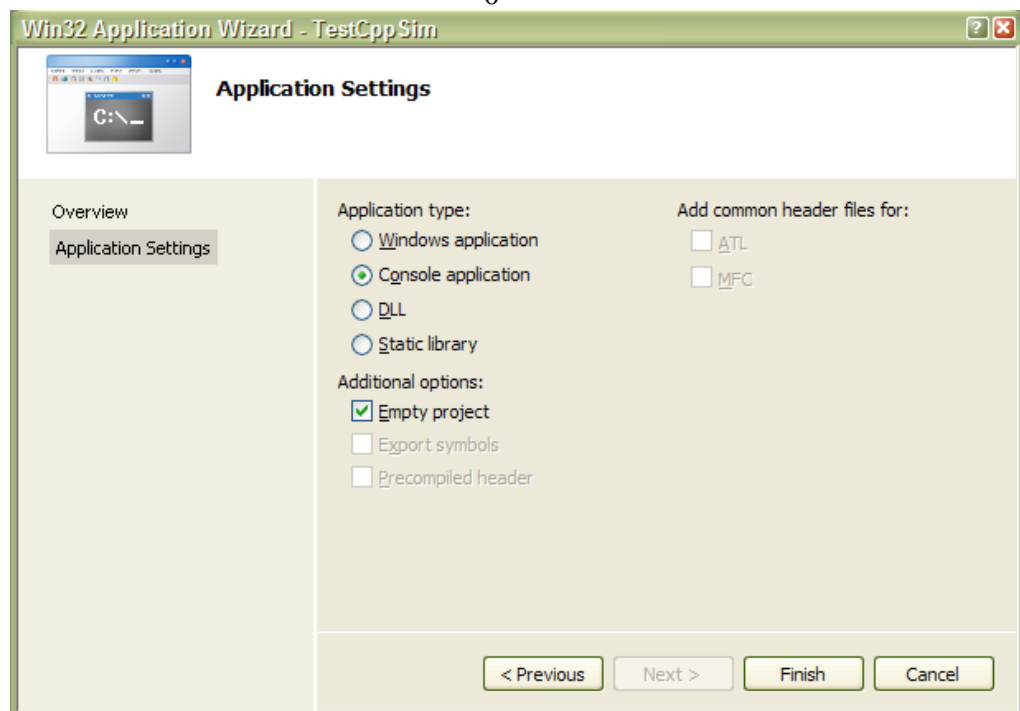
- Add the **SPV** header file directory to the development environment.
- Add the appropriate library files to the development environment.
- Create an application project, and a library project. The former will act instead of an HDL “simulator” and the latter will be the verification code. This separation makes it easier to port the verification to a true HDL simulator later on.

### 2.3.1. Visual C++ 2005

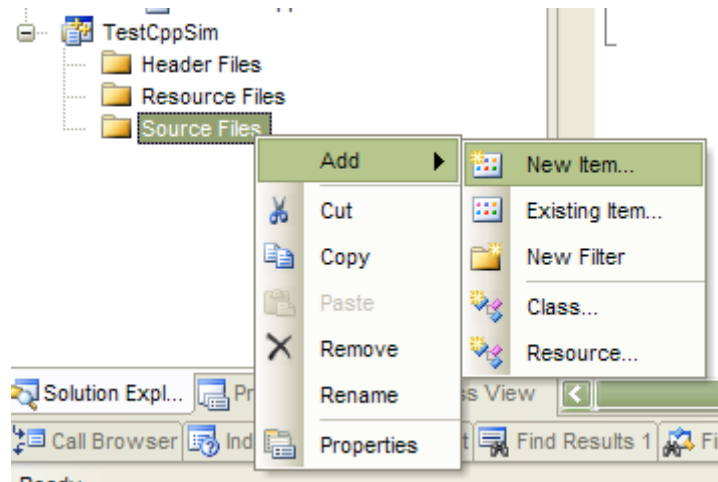
This section describes how to set up **SPV** with a Null Simulation when using Visual C++ 2005

1. Select File→New Project. In the Visual C++ tab, select "Win32" and then choose "Win32 Console Application". Give the project a name in the "Name" text box. In the Solution box, select "Add To Solution" (if you want to add this project to the current solution). Press the "OK" button.
2. Choose "Next", and then uncheck "Precompiled Header" and check "Empty project" and press "Finish". If you choose one of the other options, you will have to delete the **main** function as this function is realized within **SPV**.

0



3. Now add **SpvMain**. Select the "Source Files" and right click. Choose Add→New Item... In the files tab, select C++ File and give the file a name (SpvMain), and press "Add".

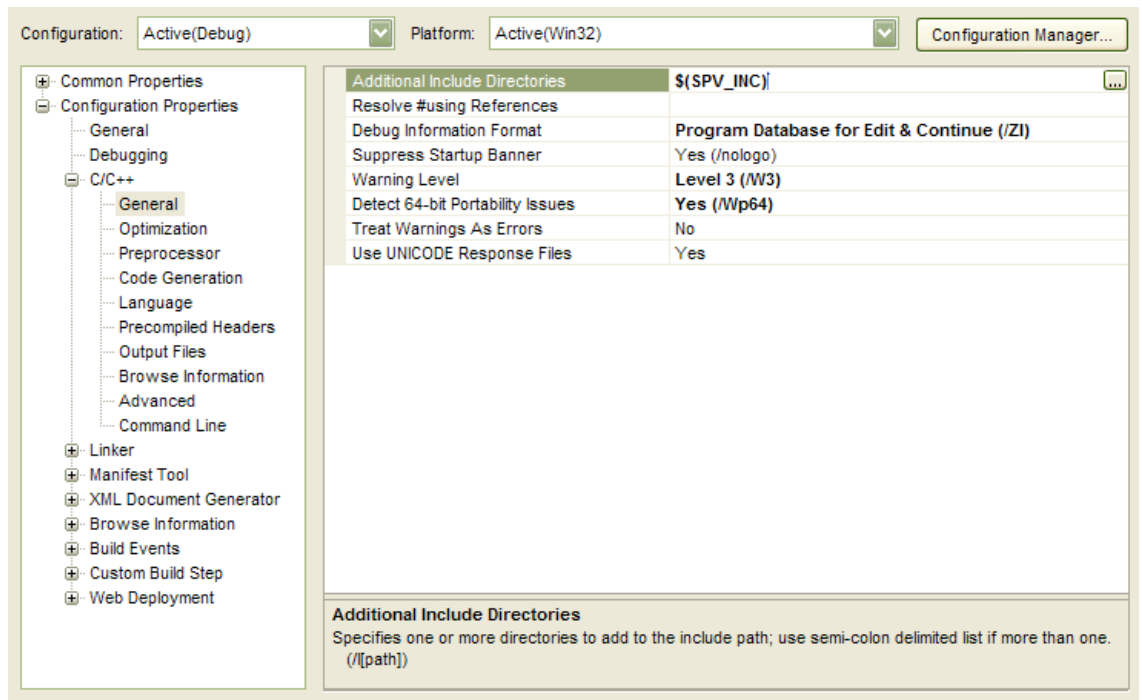


4. In the new file, include **SpvPlugin.h** and add **SpvMain()**. The file should look something like this

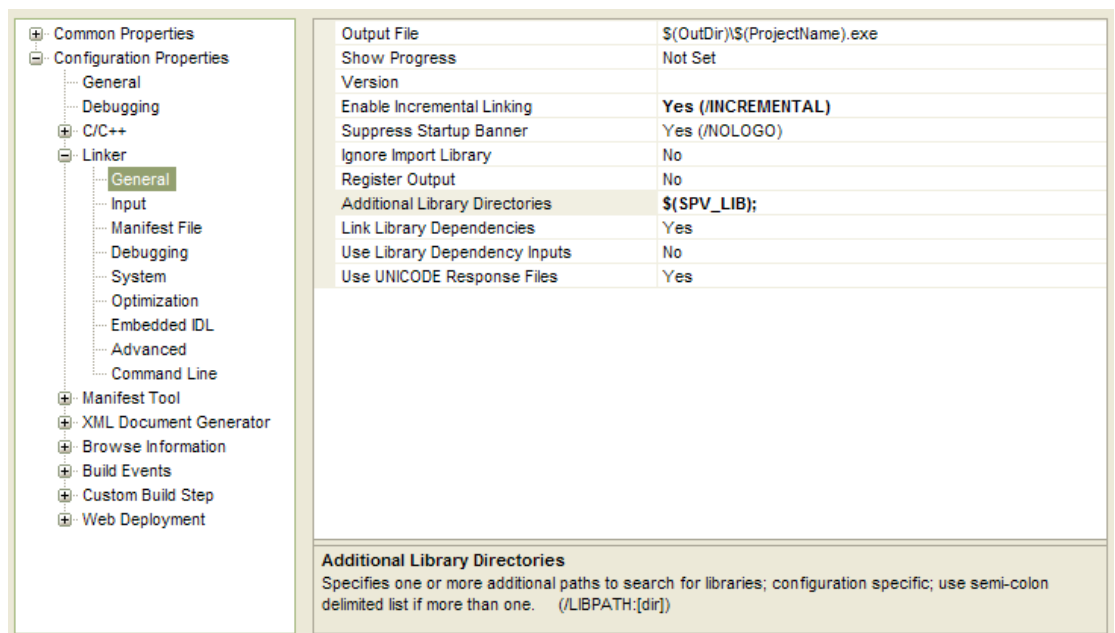
```
#include "SpvPlugin.h"

void SpvMain(int argc, char** argv)
{
}
```

5. Select the Project on the "Solution Explorer" and right click. Choose "Properties"→. In the "C++" tab choose "General" from the "Configuration Properties" list. Under "Additional Include Directories" enter either *SpvDirectory\SpvVersion\spv\_Product\include* OR *\$(SPV\_INC)* and make sure SPV\_INC in the environment variables is pointing to the include directory of SPV.
6. For Matlab support add *\$(SPV\_INC)\dpf*

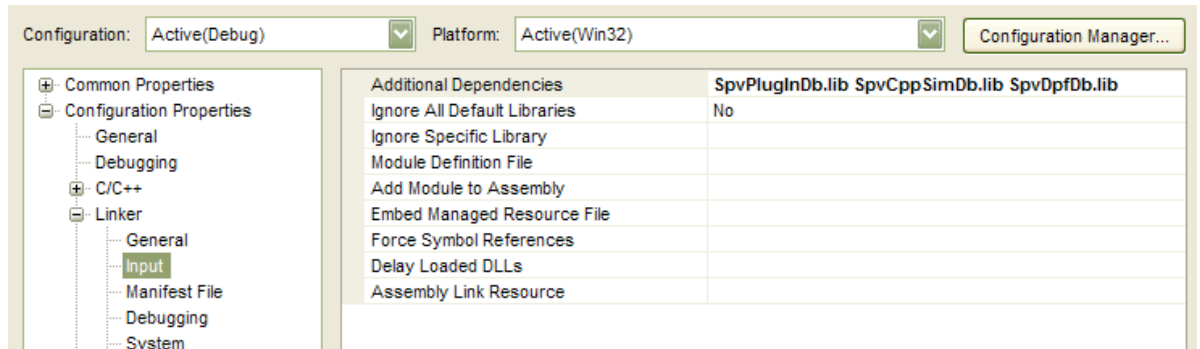


7. In the "Linker" tab, choose "General". Under "Additional Library Directories" add a *SpvDirectory\SpvVersion\lib\win32\VC80* or write `$(SPV_LIB)` and make sure your environment variable `SPV_INC` is pointing to the lib directory of SPV (as above).



8. In the "Link" tab, choose "Input" from the "Category" list. Under "Object/Library modules:" add at the beginning of the line the following:  
`Spv<Simulator><VerilogOrVHDL><Rl(Release)Db(Debug>.lib`  
 So, if you are using `SpvSimulator`, then you will need to provide `SpvCppSimDb.lib` for debug, `SpvCppSimRl.lib` for release version.

To conclude, we use SpvCppSimDb.lib in the example.  
 Also, please add SpvPlugInDb/Rl.lib for **supporting external** parameters file.  
 Add SpvDpfDb.lib to add Matlab support. (Make sure to add in the include \$(SPV\_INC)\dpf



A sample SpvSim application, that is similar to the projects produced by the SpvAppWizard for Verilog and VHDL, is included with document, in the SpvSimExample directory. The code snippets in this book can be tried there without the need for an HDL simulator.

## 2.4. HDL Simulation

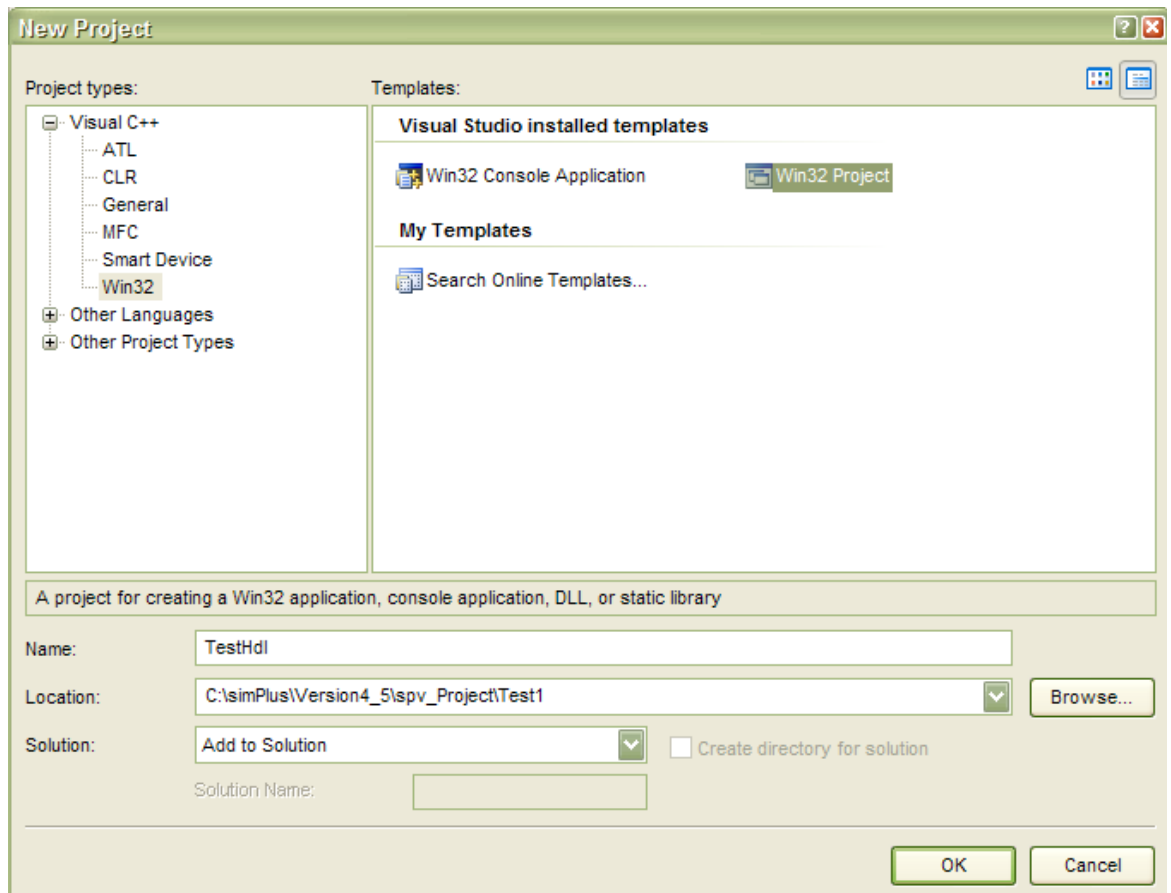
Verification with an HDL simulation means that the verification code is integrated with an HDL simulator running Verilog or VHDL code. Various simulators work in varying ways. Generally speaking, the verification code must be linked either statically or dynamically to the simulator.

### 2.4.1.1. Visual C++ 2005

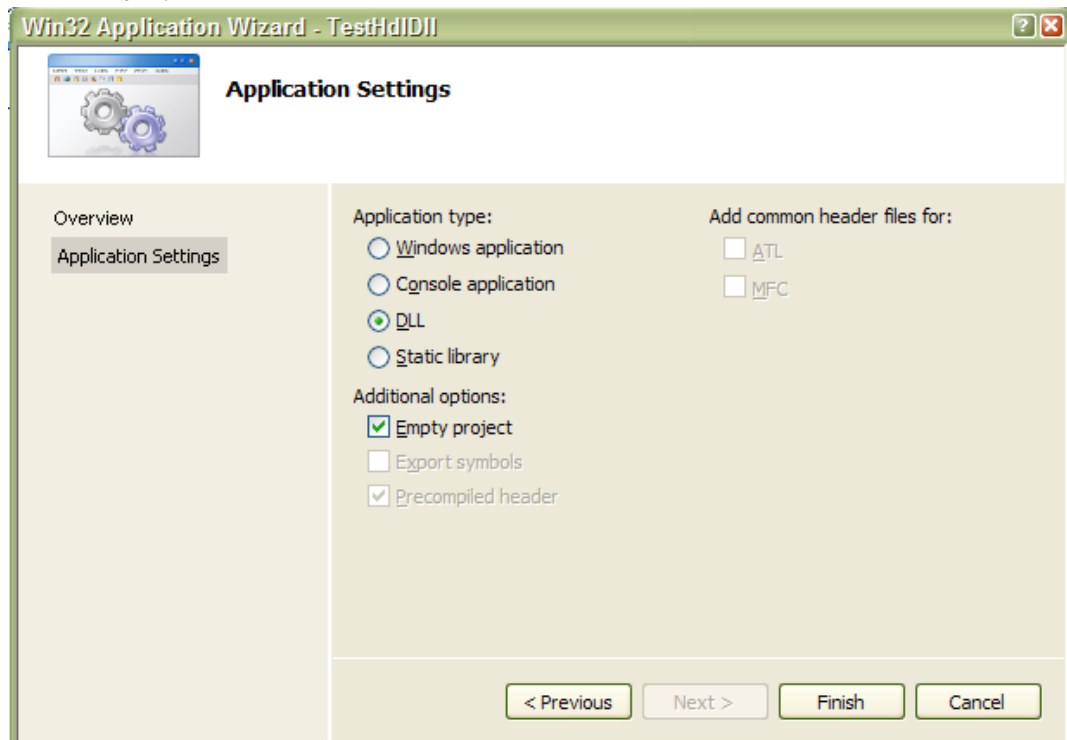
With an HDL simulator, the verification code must be compiled to a dynamic library.

1. Select File→New->Project. In the projects tab, select " Win32 Project". Give the project a name in the "Name" text box. Select "Add to Solution" if you want to add to the current project. Make sure the location is inside your project. Press the "OK" button.

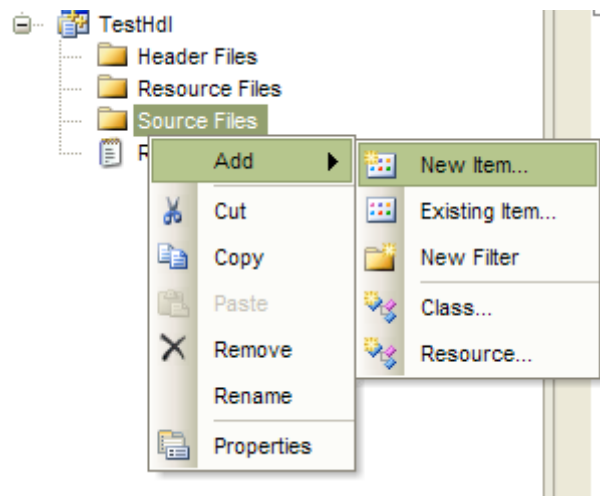




2. Press next on the first screen ("Welcome to the Win32 Application Wizard").
3. Choose "DLL" in the Application Type. Check "Empty Project" and press "Finish".



4. Now add SpvMain. Select the "Source Files" and right click. Choose Add→New Item... In the files tab, select C++ File and give the file a name (SpvMain), and press "Add".

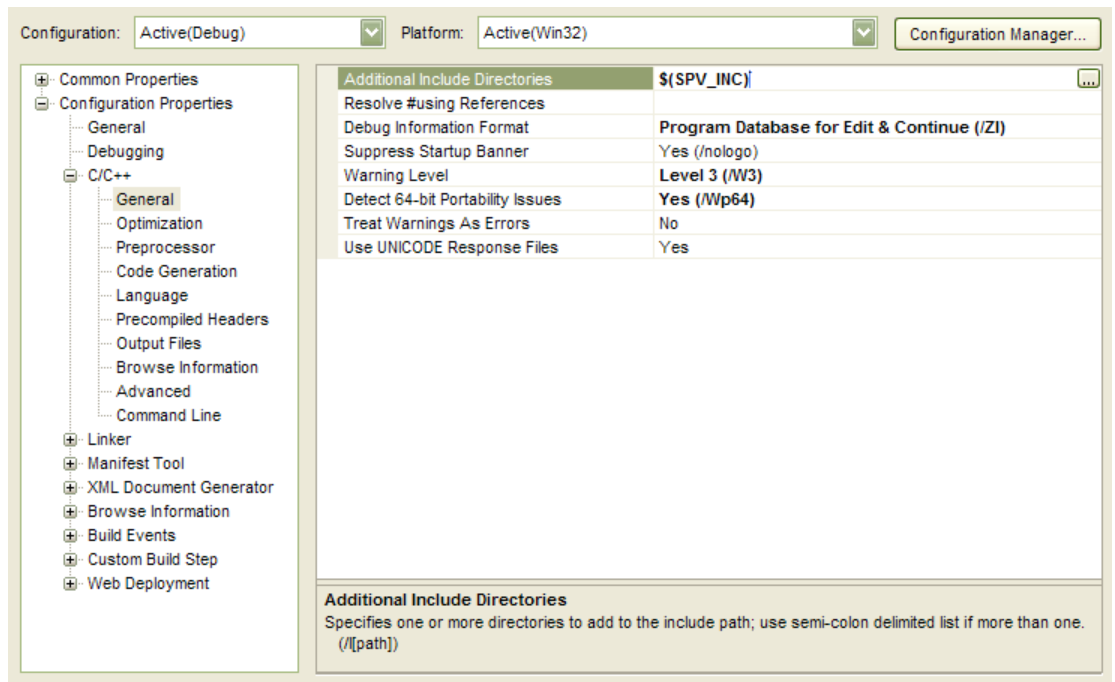


5. In the new file, include **SpvPlugin.h** and add **SpvMain()**. The file should look something like this

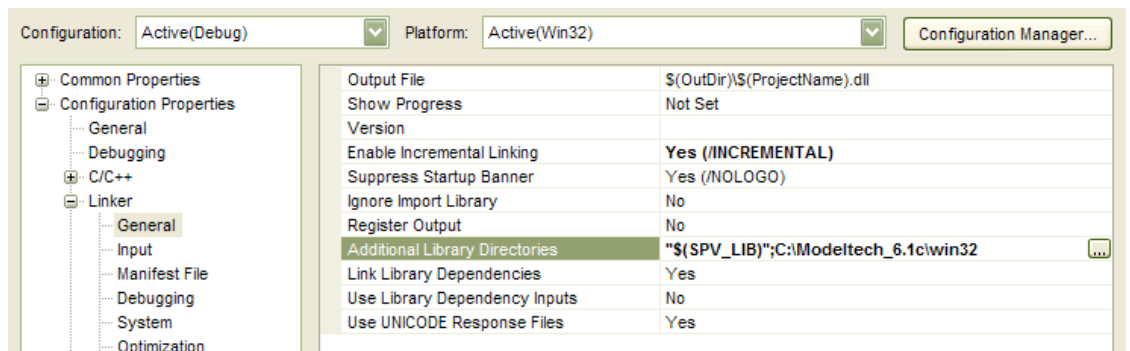
```
#include "SpvPlugin.h"

void SpvMain(int argc, char** argv)
{
}
```

6. Select the Project on the "Solution Explorer" and right click. Choose "Properties" →. In the "C++" tab choose "General" from the "Configuration Properties" list. Under "Additional Include Directories" enter either *SpvDirectory\SpvVersion\spv\_Product\include* OR *\$(SPV\_INC)* and make sure SPV\_INC in the environment variables is pointing to the include directory of SPV.
7. For Matlab support add *\$(SPV\_INC)\dpf*



8. In the "Linker" tab, choose "General". Under "Additional Library Directories" add a *SpvDirectory\SpvVersion\lib\win32\VC80* or write *\$(SPV\_LIB)* and make sure your environment variable *SPV\_INC* is pointing to the lib directory of SPV (as above).



9. Add the directory where the simulator's interface library is kept. (Not relevant to null sim projects) Add a comma and:  
For ModelSim: C:\Modeltech\_6.1c\win32 (if your installation directory for ModelSim is not C:\Modeltech\_6.1c, then substitute yours).  
For Active-HDL ActiveHDL: C:\Program Files\Aldec\Active-HDL 7.1\pli\lib (if your installation directory for Active-HDL ModelSim is not C:\Program Files\Aldec\Active-HDL 7.1, then substitute yours).

Alternately, you can use the *\$(SPV\_SIM\_LIB)*, which is what the SPV wizard uses by default. However, the wizard searches for simulators and gives you the opportunity to set this variable. So, if you have never used the wizard before, then you will have to set *SPV\_SIM\_LIB* yourself to the directory described above.

10. In the "Link" tab, choose "Input" from the "Category" list. Under "Object/Library modules:" add at the beginning of the line the following:  
Spv<Simulator><VerilogOrVHDL><Rl(Release)Db(Debug>.lib.

So, if you are using ModelSim, then you will need to provide SpvMtiDb.lib for debug, SpvMtiRl.lib for release version.

To conclude, we use SpvMtiDb.lib in the example.

Also, please add **SpvPlugInDb/Rl.lib** for **supporting external** parameters file.

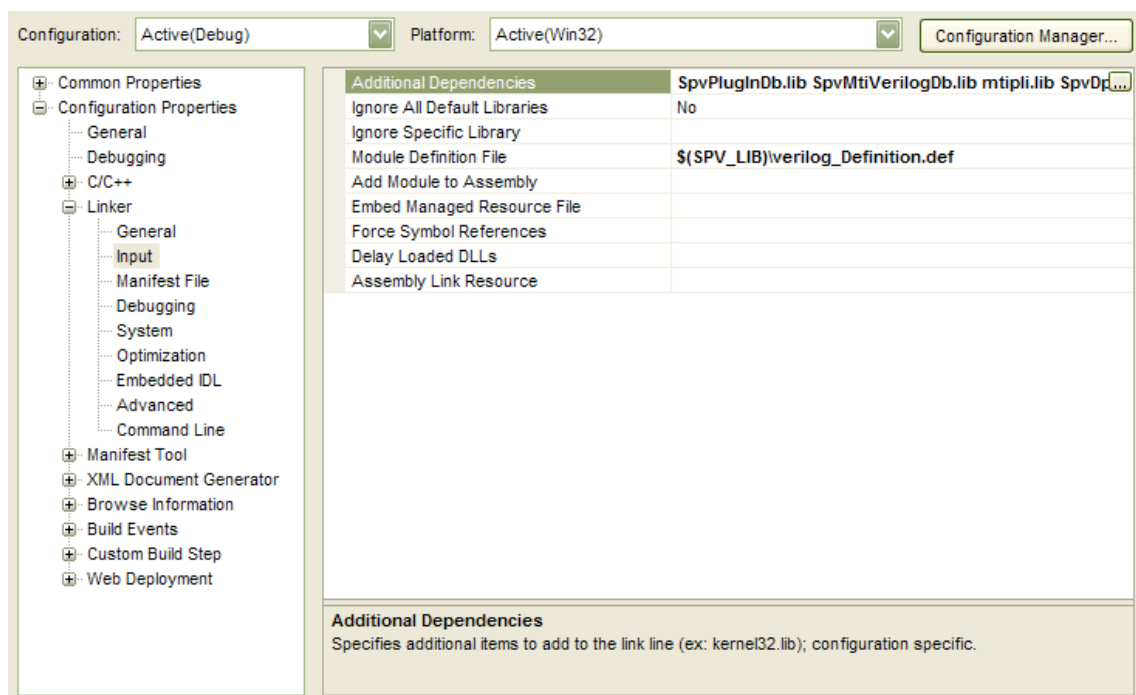
Add SpvDpfDb.lib to add Matlab support. (Make sure to add in the include \$(SPV\_INC)\dpf

Add mtipli.lib to add the modelsim library.

Add aldecpli.lib for Active-HDL ActiveHDL.

If you are using Modelsim then in the "Module Definition File" add either `SpvDirectory\SpvVersion\lib\win32\VC80\verilog\vhdl_Definition.def` or `$(SPV_LIB)\verilog\vhdl_Definition.def`.

If you are using ActiveHDL then in the "Module Definition File" add either `SpvDirectory\SpvVersion\lib\win32\VC80\vhpi_Definition.def` or `$(SPV_LIB)\vhpi_Definition.def`.



11. In the "Debugging" tab, under "Command:", enter `C:\Modeltech_6.3\win32\vsim.exe` (if your installation directory for ModelSim is not `C:\Modeltech_6.3`, then substitute yours).
12. Also in the "Debugging" tab, under "Working directory:", enter the directory of the Verilog or VHDL project.
13. Also in the "Debugging" tab, under "Program Arguments:", enter all of the top level modules that are to be used in the simulation, separated by spaces.

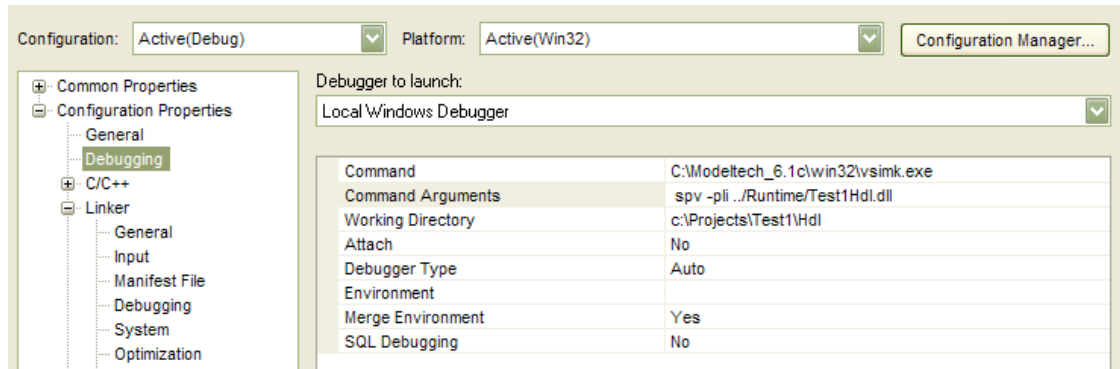
For Verilog projects only, add a space and "spv". After the modules, add `-pli PathToDLL`, where *PathToDLL* is the path to the DLL, either absolute or relative to the working directory.

Alternatively, for ModelSim, *PathToDLL* can be specified in the

ModelSim.ini file as the **veriuser** attribute.

Note that when working outside of Visual Studio with the former alternative, the modules and `-pli` switch must appear at the command line.

For VHDL projects, the DLL name is specified in the stub file. See the section on stub files, below.



## 2.5. Stub Files

Linking the verification code to the HDL simulation requires some additions to the HDL code. Stub files supply the necessary additions to the HDL project that supply the missing link.

### 2.5.1. Verilog

For most Verilog projects, the stub file is static. In any case, it must be built with the HDL project and specified as an additional top level module at simulation execution. It must, at minimum, be composed of:

```
module spv;
parameter from_File_Name = "";
parameter date = 14092005;
parameter dec_Sum = 0;

initial $spv;

endmodule
```

The link to the verification DLL is supplied at the command line, by specifying `spv` as an additional top level module and with the `-pli` switch: `-pli dll_Name.dll`. Replace `dll_Name` with the name of your verification DLL. Note that if the directory where `dll_Name` is located is not on the OS path, then you will have to specify a full path name to the DLL, either absolute or relative to the simulation work directory.

### 2.5.2. VHDL

VHDL projects require a stub file be built with them. It must, at minimum, be composed of:

```
ENTITY Spv is
END Spv;

ARCHITECTURE Arc of Spv is
    attribute foreign : string;
    -- ACTIVE-HDL only:
        attribute foreign of Arc : architecture is "VHPI dll_Name.dll";
connect_Spv_Vhpi";
    -- MODELSIM only:
        attribute foreign of Arc : architecture is "vhdl_Init_simPlus dll_Name.dll";
BEGIN
END;
```

Replace *dll\_Name* with the name of your verification DLL. Note that if the directory where *dll\_Name* is located is not on the OS path, then you will have to specify a full path name to the DLL, either absolute or relative to the simulation work directory.

Additionally, in the top level module, under the architecture line, add

```
    component simPlus
    end component;
```

Now, at the end of the instances declared in the begin: section, add

```
    sp : simPlus;
```

This line creates an instance of the module defined in the stub file. It is critical that this instance be the last instance declaration in the begin: section - any signal appearing after it will not be recognized by the verification code!

The final top level architecture code should look something like this:

```

LIBRARY ieee;
use ieee.std_logic_1164.all;

ENTITY Tb is
END Tb;

ARCHITECTURE TbArc of Tb is

---- Component declarations ----
COMPONENT Comp
END COMPONENT Comp;

--SPV stub module component
COMPONENT Spv
END COMPONENT;

BEGIN

---- Component instantiations ----

CompInst : Comp

--VERY IMPORTANT - Instantiate SPV stub module here. Must
--be at the END of the top level module
sp : Spv;

end TbArc;

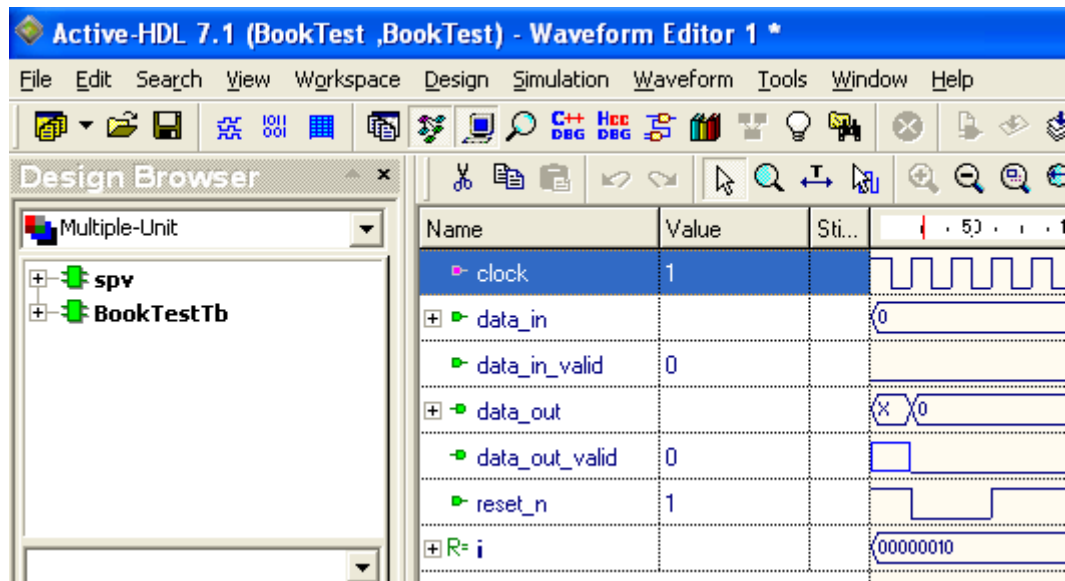
```

## 3. Signals and Processes

### 3.1. *First Steps*

#### 3.1.1. Driving signals

Use the wizard to create a “Simple” project. (Here, we’ll assume that the project name is “BookTest”.) Run the simulation as described above. You should see something like this:



Open the BookTestDriver.cpp file, there will be a ThreadFunc() function, with the following code:

```

SpvEvent pClock("BookTestTb.clock", AtPos);

SpvSig resetN("BookTestTb.reset_n");
SpvSig dataIn("BookTestTb.data_in");
SpvSig dataEn("BookTestTb.data_in_valid");
//TODO: Instantiate other signals here

// Waiting 2 Clocks, Resetting for 2 clocks and continue
Wait(pClock,2);
resetN = 0;
Wait(pClock,2);
resetN = 1;

//TODO: Set initial reset values for all signals here

//TODO: Drive data here

Wait(pClock);

//TODO: Set initial reset values for all signals here

//TODO: Drive data here

Wait(pClock);

```



Taking a quick look at the code, we can see that the first line defines a simulator event called pClock which is the positive edge of the BookTestTb.clock signal. The next three lines declare C++ representatives of simulator signals; reset\_n, data\_in, and data\_in\_valid, respectively. The next line uses pClock to block the process for 2 clock cycles after which it drives reset low, blocks for another 2 clocks, and drives reset high. To complete the reset sequence, data\_in and data\_in\_valid should be driven to their reset values, whatever those may be. To do so, we'll simply add two more lines.

```
dataEn = 0;  
dataIn = 0;
```

In other words, “driving” a register value is the same as assigning its **SpvSig** representative. If there were other inputs to our DUT, we would declare more **SpvSig** instances and assign those as well.

The ThreadFunc() function is the definition of an **SPV** thread process. An **SPV** thread process starts execution after the **StartTProc()** function is called, which we do in BookTestDriver's Start() function.

```
void BookTestDriver::Start()  
{  
    StartTProc(this, (SPVPM)&BookTestDriver::ThreadFunc);  
}
```

The first parameter of the **StartTProc()** function will almost always be *this*. The second is the class member function which will be the process code. The third, optional parameter (not shown here), is a string name which will be used as an identifier for the process (mostly in debug output). Note that the string name does not have to be the same as the function name. Now, let's have our driver process actually do something beyond resetting the DUT. After the reset initializations, we'll add:

```

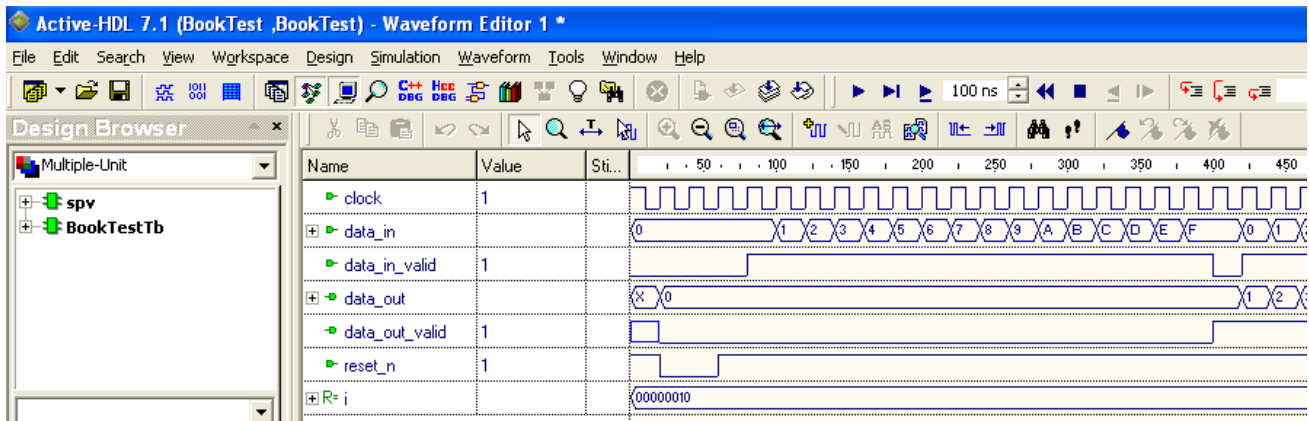
while(1)
{
    Wait(pClock);
    dataEn = 1;
    for(unsigned i = 0; i < 0x10; i++)
    {
        dataIn = i;
        Wait(pClock);
    }
    dataEn = 0;
}

```

Here, we've put the process into an endless loop, where at the beginning of each loop we raise the `data_in_valid` signal, drive 16 clocks of `data_in` (counting from 0 to 15), and lower `data_in_valid`. Note that the calls to **Wait()** have no second parameter. This is because the second parameter, "repeat", is 1 by default.

Running the simulation now yields:

In the next sections, we'll discuss the elements we've introduced here more in



depth.

## 3.2. Signals

### 3.2.1. What is a signal?

#### 3.2.1.1. Definition

Signal is a generic term for any hardware component that has a value, for example, a register. Supported components include:

Verilog:

- `reg`

- wire
- event

#### VHDL

- std\_logic
- integer
- bool
- enum
- bit
- arrays of std\_logic, bool, bit

### 3.2.1.2. Linking C++ to a signal

The **SpvSig** class enables reading and writing values to and from signals in the **SPV** framework, according to the rules described above. Connection to an HDL simulation signal is via the full path name of the signal in **SpvSig**'s constructor or in its **Init()** function.

The following example assumes a Verilog "top" module with a submodule instance, "subtop", which contains a wire, "bus", and a register "clk".

```
SpvSig clk("top.subtop.clk");

//wire is read-only
SpvSig bus("top.subtop.bus");
```

For VHDL under ModelSim:

```
SpvSig clk("/subtop/clk");

//wire is read/write
SpvSig bus("/subtop/bus");
```

Note that there is no need to set the bit lengths of the signals, as these are defined in the HDL code. Note also that in the VHDL form, the top level module is implicit.

### 3.2.1.3. Differences between Verilog and VHDL simulations

SPV provides a level of abstraction from the particular HDL language being used. However, there are some differences.

- For ModelSim only: Verilog signal paths are separated by “.” while Vhdl is separated by “/”. For ActiveHDL projects, the format is the same for both Verilog and VHDL.
- For ModelSim only: Verilog signal paths start at the top level module while in VHDL, the top level is implicit.
- Wires in Verilog cannot be driven directly. A register must be declared in the Verilog and assigned to the wire. This register can then be driven from SPV.

### 3.2.1.4. Reading Signal Values

The binary value of a signal represented by can be extracted with the **Uint()** or **Uint64()** functions, which return the first 32 and 64 bits of the signal respectively. (Later, we’ll see another option, using the SPV bit vector class, **SpvBitVec**, for wider signals) If the signal is narrower than 32 bits (for **Uint()**) or 64 bits (for **Uint64()**), the higher bits will be zeroed.

(In many cases, the **Uint()** function may not even be necessary. The rule is, if the compiler doesn’t complain, its OK.)

Continueing in the ThreadFunc() function....

```
//Get value of data
unsigned sigVal = dataIn.Uint();

//Get 64 LSB bits of sig val.
//SPV_UINT64 is a macro that expands to the compiler's 64 bit unsigned integer type.
SPV_UINT64 sigVal64 = dataIn.Uint64();

//Product with Uint()
unsigned prodVal1 = dataIn.Uint() * 3;
```

### 3.2.1.5. Signal Assignment and Simulator Synchronization

**SpvSig** can be assigned binary values with the C++ assignment operator or with the **Assign()** function. However, the values will not be updated in the simulation until **SPV** returns control to the simulator, or in other words, until the next **Wait()**. This means that reading a value immediately after assignment will not reflect the new value assigned.

### 3.2.1.6. Signal Slices

It is also possible to assign sections, or slices, of a signal and read them as well. This is done through the round parenthesis and square brackets operators, as below:

```
dataIn = 0;

Wait(1);

//sigVal1 will be 0
unsigned sigVal1 = dataIn.Uint();

dataIn = 4;

//sigVal1 will be 0 because assignment hasn't yet been synchronized with the simulator.
unsigned sigVal2 = dataIn.Uint();

Wait(1);
```

If the slice's indices are out of range of the signals width, a runtime error

```
dataIn = 0;

Wait(1);

//sigVal3 will be 0
unsigned sigVal3 = dataIn.Uint();

//Set 2 MSB bits of data to one - don't touch other bits
dataIn(2, 3) = 0x3;

Wait(1);

//sigVal4 will be 0xC
unsigned sigVal4 = dataIn.Uint();

//Set bit 3 to zero - don't touch other bits
dataIn[3] = 0;

Wait(1);
```

will result with a message posted to the log. Note that there is no meaning to the order of the slice indices – they will be sorted from greater to lower.

### 3.2.1.7. Value Domains

Signals in **SPV** are similar, conceptually and in use, to bit vectors. However, signals have states beyond simple 0 and 1. Signal bits can be set to X or Z, where X represents "undefined" and Z means "passive". When a signal bit in **SPV** is set to X or Z, it is said to be in the X domain or Z domain, respectively. When a signal is neither X nor Z, it is in the binary domain. A signal cannot be in the X and Z domains simultaneously.

**SPV** will report each domain separately. That is, by default, all signal operations will be in the binary domain but the X or Z domains may be specifically requested. In such a case, the signal value reported is a bit mask where all 1 bits are in the specified domain and all zero bits are outside of that domain. If the user attempts to read the binary domain of signal bits in the X domain, the X domain bits will be reported as 0's. For bits in the Z domain, the binary domain value will be 1's.

When setting a bit to the X domain when it has been in the Z domain, the bit will be taken out of the Z domain automatically. The reverse is true as well. Setting the binary domain value of a bit will take it out of any other domain.

Examples:

### 3.2.2. Logical Expressions

```
//Set dataIn to binary 0
dataIn = 0x0;

Wait(1);

//Set 2 LSB bits to X
dataIn(XVal) = 0x3;

//Set 2 MSB bits to Z
dataIn(ZVal) = 0xC;

Wait(1);

//dataVal will be 0xC0 because the Z's are seen as 1's and the X's as 0's
unsigned dataVal = dataIn.Uint();

//Set dataIn to 0x5 BINARY.
dataIn = 0x5;

Wait(1);

//X Val will be 0
unsigned dataXVal = dataIn(XVal).Uint();
```

Sometime it is necessary to create a logical combination of signals or we desire to use C++ variables instead of signals. For example, say we want

to record `data_in` on the positive edge of the clock, but only when the `data_in_valid` signals is high. (We will need such an event later, in the collector). One solution is to use the `pClock` event as we have until now and check the status of `data_in_valid` in the loop, as in:

```
unsigned vals[10];
unsigned i = 0;

//Collect 10 samples, but only when valid sig is
high
while(i < 10)
{
    Wait(pClock);
    if(dataEn == 1)
    {
        vals[i] = dataIn.Uint();
        i++;
    }
}
```

In general, all of the logical comparison operators can be used with **SpvSig**.

While we're at it, we'll introduce another (yet similar) way to accomplish wait-until-condition-is-met with the `WAIT_UNTIL` macro, like this:

```
unsigned vals[10];
unsigned i = 0;

//Collect 10 samples, but only when valid sig is
high
while(i < 10)
{
    //Block here until condition is met
    WAIT_UNTIL(pClock, dataEn == 1);
    vals[i] = dataIn.Uint();
    i++;
}
```

### 3.3. Processes

#### 3.3.1. What is a process?

A process is a unit of parallel execution. In **SPV**, processes are built upon C++ functions.

#### 3.3.2. Process types

In **SPV**, processes come in several flavors; *Atomic*, *Error*, and *Thread*.

An **error process** is the most limited of the three. It is signal sensitive but is intended to detect error conditions only. All error processes write debug output to the screen and log file (spv.log). Beyond that, an error process is limited to three predefined behaviors – stop, finish, and continue.

- Stop – temporarily stops the simulation. The simulation may be continued from the same point in the simulator via the simulator's user interface.
- Finish – ends the simulation. The simulator application will be closed.
- Continue – Log only. The simulation is not interrupted.

**Atomic processes** require the user to define a function that contains the signal sensitive code. This function is the user's opportunity to react to the simulation. The process is atomic in that the function must execute in its entirety immediately (in terms of simulation time) – there is no way to suspend execution mid-function while the simulator resumes execution. However, the process is cyclical. That is, the process function will be called whenever its trigger event occurs.

**Thread processes** also require a user defined function, but here is it possible to suspend execution (block) mid-function. The suspension will usually be defined in terms of a signal event or a quantity of simulation time. Thread processes make it possible to define complex logic that would require multiple atomic processes or a complex state machine in a single atomic process. However, thread processes tend to be more expensive than atomic or error processes from a performance perspective. All of the processes we will use in the coming chapters will be thread processes.

### 3.3.3. Executing processes

#### 3.3.3.1. SpvEvent

The **SpvEvent** class listens for changes in a signal. When such a change occurs, the event object will check if the change is of a type defined when the event object was initialized. Five **transition types** are supported:

- *Positive edge* (**AtPos**) – the signal value has risen.
- *Negative edge* (**AtNeg**) – the signal value has fallen.
- *Change* (**AtChange**) – the signal value has changed. (effectively, Positive or Negative edge)
- *Equal* (**Equ**) – the signal value transited to some specified value.
- *Unequal* (**NEqu**) – the signal value has transited to any value except some specified value.

The event object is used to block a thread process by calling the **Wait()** function. If the signal change satisfies the event's trigger type, then the process is unblocks and execution continues, either to the end of the thread function or until the next **Wait()**. The **Wait()** function has an alternate form



where you can specify an **SpvSig** and the transition type instead of an event. This is convenient when you want to wait on a signal that you are also reading or writing. (i.e. that you have already instantiated as an **SpvSig**) However, what happens behind the scenes is that **SPV** will create a temporary **SpvEvent**. Therefore, in a loop it is worthwhile to define an **SpvEvent** and reuse it, rather than using this alternate **Wait()** form. As an example, we could write a toggle function as:

```
SpvSignal someSig("BookTestTb.some_sig");

//Wait for any change in some_sig
Wait(someSig, AtChange);

//Wait 10 simulation cycles
Wait(10);

//Toggle some_sig
someSig = ~someSig;
```

Here, we've also introduced yet a third form of **Wait()** where we specify a length of time to block, in simulation time units. (Simulation time scale units depend on the settings of the HDL simulation. The **SpvSig::GetTimeUnit()** function returns the simulation time unit as an exponent relative to secs; e.g. femto-secs is -15.)

## 4. Using Bit Vectors

### 4.1. Overview

#### 4.1.1. What is a bit vector?

A bit vector is an array of bits treated as an unsigned integer. Such an array could represent a binary number, packet data, a random bit pattern, or any other digitally encoded data.

The **SpvBitVec** class encapsulates such a list of bits. It provides many services in the form of its public functions and operator overloads, including integer (unsigned) arithmetic, bitwise operations stream output.

**SpvBitVec** is often used together with **SpvSig** to record and change HDL component values.

The generation family of classes can be used to generate bit patterns (random or otherwise) that can be stored in an instance of **SpvBitVec**.

### 4.1.2. Other Bit Vector Classes

Other classes that are part of the bit vector family include:

- **SpvBitVecSlice** – references to a section of a bit vector.
- **SpvBitVecBool** – references a single bit of a bit vector.
- **SpvBitVecArr** – a collection class of bit vectors.
- **SpvBitVecCollect** – aggregates bit vector chunks as one large bit vector.
- **SpvBitVecIter** – the inverse of **SpvBitVecCollect**, it does out bit vector chunks from one large bit vector.

These classes will be explained in the coming sections. Missing above is **SpvBitVecKind**, which will be explained in the section on stream output.

## 4.2. *Bit vector data*

A **SpvBitVec** can receive its data from one of four sources; One of the **SpvBitVec** initialization functions, an unsigned integer, a signal, or bit pattern generation.

### 4.2.1. Construction and Initialization

The constructor functions create bit vectors and the initialization functions provide a quick way of filling a bit vector with data as well as controlling the size of the bit vector. These two types of functions are linked, as some of the constructor overloads initialize as well.

When constructing a bit vector, the size and initial value can be optionally specified as well. The different forms of constructor will create an uninitialized bit vector, duplicate an existing bit vector, copy the binary domain of a signal, or copy the bit pattern of an unsigned integer.

```

//Make sure to: #include <SpvHfile.h>

//Create uninitialized bit vector of 32 bits
SpvBitVec vecUnitialized;

//Create bit vector with bit pattern of unsigned integer value 67
//and size of 32 bits (default)
SpvBitVec vecUnsigned(67);

//Create bit vector with bit pattern of unsigned integer value 68
//and size of 8 bits
SpvBitVec vecSmallUnsigned(68, 8);

//Create a second bit vector pattern with bit pattern of
//unsigned integer value 68 and size of 8 bits
SpvBitVec vecCopy(vecSmallUnsigned);

//Create a signal linked to the hardware component top.clk
//We assume such a component exists, otherwise this will cause
//a runtime error.
//create a bit vector with a copy of the signal's s size and bit pattern
SpvSig sig("top.clk");
SpvBitVec vecSignalCopy(sig);

```

The assignment operator and **Copy()** function have similar effects to the above. The difference between these two is that the assignment operator will not change the bit size of the left hand bit vector, while **Copy()** will cause the calling object to be a duplicate of the source bit vector (the parameter) in every way. Note that duplicating a source vector via the new vector's constructor, as we did above, is the same as calling **Copy()** – that is, both size and data are taken from the source. The general rule is, whenever assignment operations between bit vectors of differing sizes are executed, any bits "missing" from the right operand are considered zero, while any "extra" bits are ignored.

A bit vector's sized can be changed with **Resize()**. Its size can be locked with **MakeSizeConst()**.

Continuing from the code above...

```

        //Copy 68 to vecUnitialized. The size of vecUnitialized
        //is unchanged at 32 bits.
vecUnitialized = vecSmallUnsigned;

        //Copy 68 to vecUnitialized. The size of vecUnitialized
        //is changed to 8 bits.
vecUnitialized.Copy(vecSmallUnsigned);

        //Attempt to copy 256 to vecUnitialized.
        //Because vecUnitialized has a bit size of 8 bits, the 9th
        //bit of the integer is cut off and effectively,
        //zero has been copied to vecUnitialized.
vecUnitialized = 256;

        //Change bit size of vecUnitialized to 9 - the new bit is zero
        //or in other words, the value of the bit vector remains the same
vecUnitialized.Resize(9);

        //Copy 256 to vecUnitialized
vecUnitialized = 256;

        //Copy the first 9 bits of sig.
vecUnitialized = sig;

        //Lock size of vecUnitialized
vecUnitialized.MakeSizeConst();

        //value of vecUnitialized is 68, but its size remains 9 bits
//instead of 32. A warning message is output to the log
vecUnitialized.Copy(vecSmallUnsigned);

```

The last three initialization functions, **One()**, **Zero()**, and **Gen()** set all the bits in the bit vector to one, zero, and random values, respectively.

```

        //Set all 9 bits to one
vecUnitialized.One();

        //Set all 9 bits to zero
vecUnitialized.Zero();

        //Generate random bit pattern for 9 bits
vecUnitialized.Gen();

```

Once again continuing from the above code...

#### 4.2.2. Bit vectors, integers, and operators

**SpvBitVec** is designed to allow a bit vector to be manipulated, for most intents and purposes, as an unsigned integer of arbitrary bit length. Furthermore, copying data to and from a bit vector is easy. The **UInt()** function returns the first 32 bits of the bit vector and assigning an unsigned integer to a bit vector requires no special additions. The entire bit vector can be read as an array of 32 bit sections with **UIntInd()**. As seen above, an integer can be assigned to a bit vector thanks to overloading of the assignment operator. Also, all of the arithmetic and bitwise operations can be performed on a mix of integers and bit vectors.

The **UInt64()** function returns the first 64 bits of the bit vector and assigning a 64 bit unsigned integer to a bit vector is accomplished with **SetUInt64()**. The SPV macro, **SPV\_UINT64**, defines a 64 bit unsigned integer in a compiler-portable manner and should be used instead of the compiler's type if portability is an issue.

```

//Create vec1 initialized to 5 and vec2 initialized to 0 with
//a bit size of 8 bits.
SpvBitVec vec1(5), vec2(0, 8);

//i = 1 + 5 == 6
unsigned i = 1 + vec1.Uint();

//vec2 = 5 * 6 == 30
vec2 = i * vec1;

//vec2 = 5 ^ 6 == 3    (bitwise XOR)
vec2 = i ^ vec1;

//1's complement of 8 bits, vec2 = ~3 == 252 == 0xfc
vec2 = ~vec2;

//Set new size of bit vector to 64 bits
vec2.Resize(64);

//shift bits 32 bits to the left, with assignment
vec2 <<= 32;

//i = the second set of 32 bits in the vector == 252 == 0xfc
i = vec2.UintInd(1);

//As for Uint(), but with 64 bit unsigned
//j will be 0x0000000fc0000000
SPV_UINT64 j = vec2.Uint64();

//shift 24 bits (0xfc00000000000000)
j <<= 24;

//and assign to vector
vec2.SetUint64(j);

```

### 4.3. *Displaying bit vectors*

Bit vectors can be output to any output stream (any class derived from **ostream**, such as the C++ standard *cout*) by using the overloaded insertion (<<) operator.

Controlling the output format is accomplished by using the overloaded parenthesis operator with one of the values of the **OutputKind** enumeration – either **SpvDec**, **SpvBin**, or **SpvHex**. These create a special object of type **SpvBitVecKind**, which is not a bit vector in and of itself, but rather a reference to the source bit vector with the addition of an output mode. Using **SpvDec** has the same effect as printing the bit vector with no output mode.

Continuing from the code above...

```
//remember to include #include <ostream>

//prints 18158513697557839872
cout<<vec2<<endl;

    //same as previous
    cout<<vec2(SpvDec)<<endl;

    //prints fc000000_00000000
    cout<<vec2(SpvHex)<<endl;

    //prints
11111100_00000000_00000000_00000000_00000000_00000000_00000000_000
00000
    cout<<vec2(SpvBin)<<endl;
```

Similar code could be used to output to a file, like so...

```
ofstream outFile;

outFile.open("output.txt", ios_base::out | ios_base::trunc);

    //prints fc000000_00000000to the file output.txt
    outFile<<vec2(SpvHex)<<endl;

    outFile.close();
```

#### 4.4. *Debugging Support*

Debuggers are built to show built in types and to open up classes and structures to their internal elements. Classes like **SpvBitVec** are not directly supported by the debugger and even if their internal structure were open to the debugger, you probably wouldn't understand much of what you'd see there.

Most debuggers have some mechanism for executing functions from the watch window. **SPV** takes advantage of this to provide debugging support. The **Str** function returns a string containing a text representation of the contents of the vector. **Str** takes a single parameter of type **SpvBitVecKind**. (either SpvDec, SpvHex, SpvBin, as above).

Try placing a breakpoint on one of the output lines of vec2 above. In the watch window (or command line) of your debugger, try adding vec2.Str(SpvHex). You should see the value of vec2 in the debugger. Try the same thing with vec2.Uint(), vec2.Uint64(), vec2.UintInd(0), vec2.UintInd(1), and vec2.Size().

## 4.5. *About header files*

In the above examples, we used the `SpvHfile.h` header which includes all of the **SPV** headers. While it is possible to include only the required headers for each file, we recommend that you take the easy way out and include `SpvHfile.h`.

## 4.6. *Reference classes*

Several classes in the bit vector family are not themselves bit vectors, but references to an existing **SpvBitVec**. The performance of some of these classes is optimized by reusing a pool of existing objects. Accordingly, the user should not save references or pointers to instances of these classes, lest the referenced instance be reused in the interim. Initialized instances of these class are not obtained directly, but rather are returned by one of the **SpvBitVec** operators. If you want to save a reference to a vector, copy the reference to a new instance, as demonstrated below.

The relevant classes are, **SpvBitVecKind** (discussed above), **SpvBitVecSlice**, and **SpvBitVecBool** (both discussed below), which all inherit from **SpvBitVecRef**. They all implement **GetVal()** which returns a reference to the bit vector that they refer to.

### 4.6.1. *Slice*

**SpvBitVecSlice** represents a section (slice) of an existing bit vector. It can be obtained by using the dual parameter parenthesis operator. This operator accepts as parameters the starting (from) and ending (to) bit indices of the section.

**SpvBitVecSlice** is designed to interact seamlessly with bit vectors, other slices, or unsigned integers – as if it was a bit vector unto itself. However, with its referential nature, any changes made to the slice are immediately reflected in the source bit vector and vice versa. If desired, the slice's value can be obtained as a bit vector with the **SpvBitVec** casting operator and/or copied with the assignment operator to another bit vector.



```

//Create two bit vectors initialized to 0x55 and 0xF1
SpvBitVec src1(0x55), src2(0xF1);

//Assign 4 bits (4 - 7, inclusive) of src2 to
//the first 4 bits of src1. src1 is now 0x5F
src1(0,3) = src2(4, 7);

//Save a copy of the slice via initialization of an SpvBitVectorSlice
SpvBitVecSlice refCopy = src1(0,3);

//Same as: src1(0,3) = src2(4, 7);
//Note that assignment of a slice is different than initialization
//as shown above, even though both use the "equals" sign.
//Assignment changes the value of the referenced bit vector
//whereas initialization creates a new reference to the bit vector.
refCopy = src2(4, 7);

//Assign the value 2 to bit 4 - 5. src1 is now 0x6F
src1(4,5) = 2;

//Print the first 6 bits of src1 in hexadecimal.
//Output is 2f
cout<<src1(0,5)(SpvHex);

//This has the same effect as
//src1(0,3) = 0xF
refCopy = 0xF;

//Create independent copy of slice
SpvBitVec contentCopy = src1(0, 3);

//This has no effect on src1
contentCopy = 0;

```

#### 4.6.1.1. Debugging Slices

The **SpvBitVec** class contains a debugging function for slices called **SliceStr**, which takes as parameters the starting and ending indices of the slice and a third parameter of type **SpvBitVecKind**. Set a breakpoint in the code above and try setting a watch expression of : src1.SliceStr(4, 5, SpvHex).

### 4.6.2. Bool

**SpvBitVecBool** represents a single bit in a bit vector. It can be obtained by using the square parenthesis (index) operator. This operator accepts as a parameter the index of the desired bit.

Similar to **SpvBitVecSlice**, any change in the boolean value referenced is immediately reflected in the source vector and vice versa. An independent copy can be obtained with the casting operator of type *bool*.

```
spv_Bit_Vector src(0x36);

    //src == 0x37
src[0] = 1;

    //prints The second bit is 1
if(src[1])
    cout<<"The second bit is 1";
else
    cout<<"The second bit is 0";

    //Copy the 8th bit to the first bit. src == 0x36
src[0] = src[7];

    //Independent copy of bit
bool bitCopy = src[0];

    //Has no effect on src
bitCopy = 1;
```

## 4.7. Bit vector arrays

**SpvBitVecArr** encapsulates an array of bit vectors. However, it is more than simply an array. It provides additional functionality such as wholesale initialization and generation of all the vectors in the array, and concatenation of the array bit vectors to a single vector.

A bit vector array can be thought of as a matrix of bits, with the bit vectors making up the rows. **SpvBitVecArr** overloads the square bracket (index) operator to provide intuitive access to the bits in the matrix. A function, **GetColumnVec()**, provides the ability to retrieve a column of bits as a bit vector (though you can't change the bit vector that it returns).

```

SpvBitVecArr vecArray;

    //Initialize array to 4 vectors of 8 bits each
    //with default generator limited to values
    //between 16 and 32
vecArray.Init(4, 8, 16, 32);

    //Generate random values between 16 and 32
    //for all array vectors
vecArray.Gen();

    //Print vector array
    //Printed what is below (but your output may be different)
    //30
    //30
    //20
    //29
cout<<vecArray<<endl;

    //Fill vector array with 4, 8, 16, 32
    //Vector matrix will be (in binary digits, columns numbered from
RIGHT (lsb) to LEFT (msb)):
    //00000100
    //00001000
    //00010000
    //00100000
for(int i = 0; i < vecArray.Size(); i++)
    vecArray[i] = 4 << i;

    //Print array vector as one long array
    //Output is 20100804
cout<<vecArray.Pack()(SpvHex)<<endl;

    //Print each row of the bit matrix (in hex digits)
    //Output is 04 08 10 20
for(i = 0; i < vecArray.Size(); i++)
    cout<<vecArray[i](SpvHex)<<" ";
cout<<endl;

    //Print each column of bits
    //Output is 0 0 1 2 4 8 0 0 (starting with column 0 at left)
for(i = 0; i < vecArray[0].Size(); i++)
    cout<<vecArray.GetColumnVec(i)<<" ";
cout<<endl;

```

#### 4.7.1. Debugging Bit Vector Arrays

The **SpvBitVecArr** class contains two debugging functions, **Str** and **SliceStr** which work similarly to the functions of the same name of **SpvBitVec** with one extra parameter – the index of the vector in the array is passed as the first parameter.

### 4.8. *Aggregating and Dispensing bits*

**SpvBitVecCollect** collects a number of bits, one chunk at a time. It is often used to collect the state of a register or bus over time. The total number of bits and the chunk size are set at initialization. By default, bits are collected starting at the LSB, but this can be changed at initialization.

To use **SpvBitVecCollect**, first initialize the collector with the maximum number of total bits for collection and the default chunk size, either in the constructor or with the **Init()** function. Then call **SetNext()** to place a chunk into the collection. Repeat calls to **SetNext()** as needed. Note that a second, optional parameter to **SetNext()** can be a chunk size that is different than the default set initialization. The **IsLast()** function returns true after the call to **SetNext()** which fills the collector to the maximum number of bits set at initialization. Once the collector is full (**IsLast()** returns true), **SetNext()** may not be called again until the **Init()** function is once again called. **Init()** not only initializes the collector, it also clears any current data.

**SpvBitVecCollect** also supplies the following functions:

- **BitsCollected()** – returns the number of bits already inside the collector
- **BitsRemaining()** – returns the number of vacant bits remaining
- **IsEmpty()** – returns true when **BitsCollected()** is zero
- **LastCollection()** – returns whatever the last call the **SetNext()** received.
- **Collected()** – returns the current contents of the collector. A cast to **SpvBitVec** will have the same effect.

```

//Initialize array to 4 vectors of 8 bits each
//with default generator limited to values
//between 16 and 32
SpvBitVecArr vecArray(4, 8, 16, 32);

//Fill vector array with 4, 8, 16, 32
for(int i = 0; i < vecArray.Size(); i++)
    vecArray[i] = 4 << i;

//Create a collector
SpvBitVecCollect collector;

//Initialize its size to 32 and its chunk size to 8
collector.Init(32, 8);

//Collect bits - note the termination condition
//Here a bit vector from a bit vector array is collected,
//but a signal or unsigned could be collected as well.
for(int j = 0; !collector.IsLast(); j++)
    collector.SetNext(vecArray[j]);

//Retreive the aggregate bit vector with the
//help of the SpvBitVec casting operator
//(here the cast is implicit in the assignment)
SpvBitVec aggregate = collector;

//Print the collected bits. Output is 20100804
cout<<aggregate(SpvHex)<<endl;

```

**SpvBitVecIter** performs the inverse operation of **SpvBitVecCollect**. Whereas **SpvBitVecCollect** collects chunks of data, **SpvBitVecIter** takes a complete bit vector and serves it up in chunks. One common use is to feed data over time to a bus.

**SpvBitVecIter** is initialized in its constructor or the **Init()** function with a source bit vector and a default chunk size. Chunks are retrieved with a call to **Next()**, where the optional parameter is a chunk size which may be different from the default specified at initialization. Similarly to **SpvBitVecCollect**, **IsLast()** function returns true after the call to **Next()** which empties the iterator.

**SpvBitVecCollect** also supplies the following functions:

- **BitsIterated()** – returns the number of bits already dispensed by the iterator.
- **BitsRemaining()** – returns the number of bits remaining inside the iterator.

- **IsEmpty()** – returns true when **BitsRemaining()** is zero.
- **LastIteration()** – returns whatever the last call to **Next()** returned.

Continueing from the code above...

```
//Create an iterator on aggregate with a chunk size of 2
SpvBitVecIter iter;
iter.Init(aggregate, 2);

//Print out aggregate 2 bits at a time
//Output is 0 1 0 0 0 2 0 0 0 0 1 0 0 0 2 0
while(!iter.IsLast())
    cout<<iter.Next()(SpvHex)<<" ";

cout<<endl;
```

## 5. Generation

### 5.1. *What is generation?*

Stimulus generation is a critical topic in hardware verification. In order to properly verify the HDL code of a hardware project, as many combinations of test input as possible must be tested. Manually creating such large masses of test cases is unfeasible, thus the need for a more automated solution. When we refer to **Generation** in this chapter, we mean any method of creating a bit pattern according to a given set of rules. The classes that implement generation are called **generators**.

**SPV** supplies a ready made set of generators that cover many common generation needs. They are separated into two types, non-random and random. Also supplied is an extensible class for user defined generation classes.

### 5.2. *Generators*

A generator in **SPV** is a class the derives from the **SpvGen** base class and implements its interface, particularly, the **Gen()** function. **SpvGen** defines some basic functionality for all generators. Amongst them are the functions:

- **SetName()** – Set a string name for an instance of a generator.
- **Name()** – Retrieves the name set with **SetName()**.
- **Reset()** – Only relevant to non-random generators, this function returns the generator to its initial state.
- **Size()** – Returns the bit width of the generator as set at initialization. While there are uses for the bit size, generally this should be set to 32 (the maximum) at initialization.

- **Gen()** – This function, which is the only one in **SpvGen** with no default implementation, must return the generated value, according to the type of the generation class. The **SPV** built-in generator classes implement this function. User defined generation classes must implement it.

### 5.2.1. Non-Random (deterministic)

#### 5.2.1.1. Constant

The constant generator, **GenConst**, generates the same bit pattern always. This is the most trivial of generators and alone it is not very useful. Its usefulness comes from its role as a building block for more complex generators (See Composite Generators and file based generation below).

```
//generates 18
SpvGenConst genConst(32, 18);

//Set name to generator
genConst.SetName("Constant_Gen");

//Generate constant value and store
unsigned val = genConst.Gen();

//Print name and generated value
//Displays: Constant_Gen generated 18
cout<<genConst.Name()<<" generated "<<val<<endl;
```

#### 5.2.1.2. Sequential

Sequential generators generate predetermined values, one after another.

##### 5.2.1.2.1. *SpvGenNextStep*

**SpvGenNextStep** is initialized with: a range of values, the step value, the initial value, and the order (ascending or descending) at initialization. The first call to **Gen()** returns the initial value specified. With every addition call to **Gen()** it returns the last value incremented (or decremented, according to the order – default is ascending) by step. When the upper bound is reached, the value wraps around.

```

        //Generate values from 0 to 10, in ascending
//order with a step of 2, starting at 4
    SpvGenNextStep genStep1(32, 2, 10, 0, 4);

        //Same as genStep1, but start at 5
    SpvGenNextStep genStep2(32, 2, 10, 0, 5);

        //Generate 10 values - Note what happens when the
//generators wrap around.
//Output is...
//4 5
//6 7
//8 9
//10 0
//1 2
//3 4
//5 6
//7 8
//9 10
//0 1
    for(int p = 0; p < 10; p++)
    {
        cout<<genStep1.Gen()<<" ";
        cout<<genStep2.Gen()<<endl;
    }

    cout<<endl;

```

#### ***5.2.1.2.2. SpvGenInRangeListOrder***

**SpvGenInRangeListOrder** is initialized with a list of value ranges and returns one value at each call to **Gen()**, in the order specified – ascending (first to last) or descending (last to first).



```

SpvGenInRangeListOrder genSequence(32, "8 12, 15, 18 20");

    //Prints 8 9 10 11 12 15 18 19 20 8 9 10 11 12 15 18 19 20 8 9
for(int k = 0; k < 20; k++)
    cout<<genSequence.Gen()<<" ";
cout<<endl;

    //Reinitialize the generator with the Init() function
    //with the same parameters, except reverse the sequence order
genSequence.Init(32, "8 12, 15, 18 20", false);

    //Prints 20 19 18 15 12 11 10 9 8 20 19 18 15 12 11 10 9 8 20 19
for(int l = 0; l < 20; l++)
    cout<<genSequence.Gen()<<" ";
cout<<endl;

```

### 5.2.2. Random

Generators that do not generate deterministically are called random generators. The "randomness" of the generators can be controlled with initialization constraints limiting the domain of possible values.

The "random" generation is actually pseudo-random, depending on the a seed value for **SPV**'s randomization internals. That is, if the seed value is the same, the generation should return the same. At every programming session, a seed for the random generators may be manually supplied, or taken from the system time clock. If neither of these options is chosen, a constant default is used. In any event, the seed value may be retrieved by calling **SpvConfig::GetSeed()**.

By setting the seed manually, it is possible to recreate the generation of interesting test cases. The **SpvConfig::UseSeed()** function sets the seed type, and if necessary, the seed itself. Its first parameter is the seed type, **DefaultSeed**, **RandomSeed**, or **UserSeed**. In practice, not calling **SpvConfig::UseSeed()** at all will seed the generators with **DefaultSeed**. **RandomSeed** uses the system time clock to create a seed which will change at every execution. **UserSeed** takes the second parameter to **SpvConfig::UseSeed()** as the seed. Use **RandomSeed** to ensure constantly changing random generation, **SpvConfig::GetSeed()** to record the random seeds, and **UserSeed** to replay test runs of particular interest. Note that the seed functions are *static* functions and are called via **SpvConfig** with a double colon, as written.

**IMPORTANT:** Recreating the generation of a test run with **UserSeed** cannot be guaranteed if either the verification code or the HDL change between the original test run and the replay. Generally speaking, as long as the generators' **Gen()** functions are called in the same order, the values returned will be the same.

### 5.2.2.1. GenInRange

**GenInRange** generates random bit patterns between a specified lower and upper bound (inclusive).

Here we will use the **SpvConfig::UseSeed()** with **UserSeed** to force a consistent reproduceable generation. For values that will vary for each run, use **RandomSeed** instead. Note that for all the random generator examples, your output may differ from that shown here.

```
//Force seed to value to constant value (1),
//which will make the results reproduceable
//between runs
SpvConfig::UseSeed(UserSeed, 1);

//Uncomment this line for varying generation at each run
//SpvConfig::UseSeed(RandomSeed);

//Constrain generator to 7 bits between 10 and 100
SpvGenInRange rangeGen(32, 100, 10);

//Will print 51 27 92 34
for(int l = 0; l < 4; l++)
    cout<< rangeGen.Gen()<<" ";
cout<<endl;
```

### 5.2.2.2. SpvGenInRangeList

**SpvGenInRangeList** is similar to **SpvGenInRangeListOrder** with the important difference that it **SpvGenInRangeList** will generate values from its initialization randomly. Otherwise, usage is the same.

**SpvGenInRange** can be seen as a degenerate case of **SpvGenInRangeList** where only one set of ranges can be specified.

```

//Generate values from 8 to 12, 15, and 18 to 20
SpvGenInRangeList rangeListGen(32, "8 12,15,18 20");

//My output is 19 18 10 10 19 11 18
//18 8 10 8 12 12 10 10 19 8 18 12 10
for(int i = 0; i < 20; i++)
    cout<<rangeListGen.Gen()<<" ";
cout<<endl;

```

### 5.2.2.3. SpvGenNotInRangeList

**SpvGenNotInRangeList** is inverse of **SpvGenInRangeList**. It generates values not in the initialization list (AKA an exclusion list). Otherwise, usage is the same - almost. There is one important nuance. Here, the bit width parameter is very important because it defines the value to exclude from.

```

//Generate values excluded from 8 to 12, 15, and 18 to 20
//Note that the range to exclude from is 0 to 32 because of
//the generation size of 5 bits
SpvGenNotInRangeList rangeNotListGen(5, "8 12,15,18 20");

//My output is 16 4 29 23 25 21 7
//1 21 2 17 31 21 23 13 13 2 4 17 26
for(int j = 0; j < 20; j++)
    cout<<rangeNotListGen.Gen()<<" ";
cout<<endl;

```

### 5.2.3. User Defined

User defined generators are the most flexible of all. The generation is coded by the user in the form of a pure virtual function, **Gen()**. Within the **Gen()** function, the user is responsible to return a unsigned integer value which is the generation result.

A class derived from **SpvGen** can generate based on anything the user is capable of . However, often such a class will include within it other generators from which it will choose, or combine, to form its own generation.

The example below (Windows only) generates 0 in the morning and 1 in the afternoon!. This is not a very practical example (and even dangerous, because the generation seed cannot guarantee the reproducibility of the values generated) but it does serve to demonstrate that the user can do almost anything...

```

#include <time.h>
class AmPmGen : public SpvGen
{
public:
    //Generates 0 in the afternoon (PM), non-zero in the morning (AM)
    virtual unsigned Gen()
    {
        time_t currentTime;
        tm *localTime;

        //Get current system time
        time(&currentTime);

        //Convert system time to local day time
        localTime = localtime(&currentTime);

        //In the morning generate 0, in the evening generate 1
        return localTime->tm_hour < 13;
    }
};

void UserDefinedGeneratorFunc()
{
    //Instantiate AM/PM generator
    AmPmGen genPM;

    //Output will either be, AM or PM, depending on when you run the
code!
    const char* amPmStr = genPM.Gen() ? "AM" : "PM";
    cout<<amPmStr<<endl;
}

```

#### 5.2.4. Composite Generation

Composite generation is a powerful concept. It allows assembly of new generators based on those already existing. In this way, more complex generators can be modularly built.

##### 5.2.4.1. Weighted

Weighted generators enable creation of a generator made up of other generators, where each child generator is assigned a statistical weight. Generation is internally a two step process; Random generation for deciding which child generator to use and then execution of the chosen generator's **Gen()** function.

**SpvGenWeighted** realizes a weighted generator based on percentage weights. The size of the generated bit vector is assigned at initialization. The **AddGenElem()** function attaches an existing generator to the composite. Its first parameter is a reference to the child generator and its second parameter is the percentage. All of the percentages together should add up to %100. If the percentages add up to less than %100, the percentages are normalized to %100. If they add up to more or less than 100, the weights are normalized.

```
//Build weighted generator with 10 percent assigned to
//to rangeListGen and 90 percent to rangeNotListGen
SpvGenWeighted genWeighted(32);
genWeighted.AddGenElem(rangeListGen, 10);
genWeighted.AddGenElem(rangeNotListGen, 90);

//My output is 26 4 1 21 30 17 27 1
//23 24 2 28 24 13 24 16 21 3 21 11
for(int k = 0; k < 20; k++)
    cout<<genWeighted.Gen()<<" ";
cout<<endl;
```

#### 5.2.4.2. Repetitive

**SpvGenRepeat** specifies a sequence of generator choices. Notice that this is not the same as **SpvGenInRangeList**, where the repetition is of values. Here it is a *generator* that is being repeatedly chosen for generation and not necessarily a specific value. This way it is possible to specify, say, 2 generations between some range and then 10 generations of a different range.

Similar to **SpvGenWeighted**, an **AddGenElem()** is defined, with the first parameter being the generator for repetition, but the second parameter is the repetition count instead of a statistical weight. Continuing from the previous examples...

```

//Build repetitive choice generator with 2 repetitions per generator
//Third generator is const generator generating zero
SpvGenRepeat genRepeat(32);
genRepeat.AddGenElem(rangeListGen, 2);
genRepeat.AddGenElem(rangeNotListGen, 2);
genRepeat.AddGenElem(SpvGenConst(32, 0), 2);

//My output is 8 9 31 28 0 0 12 11
//22 6 0 0 18 20 13 16 0 0 8 15
for(unsigned m = 0; m < 20; m++)
    cout<<genRepeat.Gen()<<" ";
cout<<endl;

```

Pay attention to the third generator in the code above. We use the **SpvGenConst** class to cause the repetition generator to generate a constant value (zero) every 5th and 6th call to **Gen()**. This demonstrates why we need a “Constant” generator – the repetition generator’s **AddGenElem()** takes only generators, that is, objects derived from **SpvGen**, so to cause the generation of a constant value in our composite repetition generator we need a generator that will return a constant value.

The composite generators themselves can be used as parameters to **AddGenElem()**. In the example below, the repetition generator from the code above will be used 50 percent of the time, while the value 99 will be generated the other 50 percent. This example will also demonstrate the **ClearGenerators()**, which will reset the the generator list for the weighted generator (it can also be used with a repetitive generator). It further demonstrates using the the **Reset()** function for restarting the generation sequence of the repetition generator. Continuing from the previous examples...

```

//Restart repetition generator
genRepeat.Reset();

//Clear generator list from weighted generator and rebuild
genWeighted.ClearGenerators();
genWeighted.AddGenElem(genRepeat, 50);
genWeighted.AddGenElem(SpvGenConst(32, 99), 50);

//My output is 99 19 99 99 99 99 99
//99 8 22 99 7 99 0 99 0 11 20 13 30
for(unsigned n = 0; n < 20; n++)
    cout<<genWeighted.Gen()<<" ";
cout<<endl;

```

### 5.2.5. File Defined Generators

File defined generators are generators that are defined in an external text file. File defined generators enable parameterization of generation for the end user without requiring C++ code changes and recompilation.

The **SD** (System Directory) class reads a text file with named generator definitions, as well as other configuration information. These generators and configuration setting can then be retrieved by the user via their names.

#### 5.2.5.1. Text file format

The text file format is as follows:

All parsed lines start with a line type followed by a colon. Line types start at the first character of a line and are all upper case letters.

Numerical values in the file may be decimal or hexadecimal (marked by a 0x prefix). Strings are case sensitive. A double slash (//) marks a comment from that point till the end of the line.

Parsing will start only from the first line with a START line type and will end at the at first line with a STOP line type.

Most lines have a name parameter. If another line is defined with the same name, then the last definition is conclusive, i.e. the last definition overrides the previous ones.

STRING line types define a string alias for another string value. A STRING line follows the format:

STR: {*Name*} {*String*}

NUMBER line types define a string alias for a signed integer value of 32 bits. A NUMBER line follows the format:

NUMBER: { *Name*} {*Value*}

BIT\_VEC line types define a string alias for a unsigned integral value of arbitrary bit size. A BIT\_VEC line follows the format:

BIT\_VEC: { *Name*} {*BitSize*} {*Value*}

Similarly, BOOL line types define a string alias for a true/false value. A BOOL line follows the format:

BOOL: { *Name*} {*true/false*}

The following example is the text file generated by the SPV AppWizard. It contains a series of configuration definitions intended for use in initializing the seed value for generation.

```

START:

// Seed Values
STR:          USE_DEF          0
STR:          USE_RAND         1
STR:          USE_FIX          2
NUMBER:       RAND_SEED        USE_FIX
NUMBER:       SEED              123456789          // For case of
RAND_SEED == USE_FIX use this value

STOP:

```

The file is parsed with the **SD::ReadGenFile()** function. After parsing, the “Get” functions can be used to retrieve values from the online database. The following “Get” functions are defined:

- **SD::GetInt()** – Returns NUMBER definitions
- **SD::GetBool()** – Returns BOOL definitions
- **SD::GetStr()** – Returns STR definitions
- **SD::GetBitVec()** – Returns BIT\_VEC definitions

The “Get” functions come in two forms. The first takes a reference to the variable that will receive the value from the file as its first parameter and returns true/false which indicates whether the Name (second parameter) actually existed in the file. The second returns the appropriate type (int, bool, etc.) instead of the reference parameter but will cause an exception if there is no matching line for the Name parameter. There is also a **SD::IsExists()** function which returns true if a line of any type exists for the name indicated in its parameter.

The C++ code that uses the definitions from the last example is below. The feature implemented by the code is to allow the end user to set the seed type and seed value from outside the C++ code. It just uses the file definitions to determine how (and if) to call the **SpvConfig::UseSeed()** function.



```

SD::ReadGenFile("../././Runtime/GenParameters.txt");

// Seed Config
int seedType, seedVal;

if(SD::GetInt(seedType,"RAND_SEED") == true)
    if(SD::GetInt(seedVal,"SEED") == true)
        SpvConfig::UseSeed((SpvSeedType)seedType,seedVal);

```

Generator line types are defined with the following general format:

**{GenType}: {Name} {BitSize} {GenParameters}**

where

*GenType* - Type of generator to create. There are a number of types, each roughly corresponding to one of the **SPV** generation classes. Below are the mappings from *GenType* to class and the *GenParameter* for each.

- GC – Corresponds to **SpvGenConst** – { *Value* }
- GR – Corresponds to **SpvGenInRange** – { *From* } { *To* }
- GS– Corresponds to **SpvGenNextStep** – { *Step* } { *From* } { *To* }  
{ *StartVal* } { *IsAscending:=true/false* }
- GRANGE0 – Corresponds to **SpvGenInRangeListOrder** –  
{ *RangeList* } { *IsUpwards:=true/false* }
- GRANGE – Corresponds to **SpvGenInRangeList** and **SpvGenNotInRangeList**, depending on the last parameter –  
{ *RangeList* } { *IsInRange:=true/false* }
- GPERCENT – Corresponds to **SpvGenWeighted.** –  
{ *GenList* } { *WeightList* }
- GQUANTITY – Corresponds to **SpvGenRepeat.** – { *GenList* }  
{ *RepeatList* }

All “List” parameters must be enclosed in quotes. *RangeList* is a comma separated list of ranges or single values. For example: “1-4,6,8-10” which translates to 1 through 4, 6, and 8 through 10. Note that there are no spaces allowed and each range has a hyphen.

The *GenList* parameters are comma separated lists of generator names previously defined in the file. The *WeightList* and *RepeatList* parameters are comma separated values, each corresponding (by order) to a generator in the *GenList*.

*Name* - The string literal that will identify the generator. If Name is the same as a previously defined generator with the addition of a period followed by a number, it will define a specific case of that previously defined generator. This is known as a special generator, or point generator, which will be explained later on.

*BitSize* - The size of the bit pattern to be generated.

Examples:

```
//Const (here, always 3)
GC: ConstExample 32 3

//Range (here, 0 to 3)
GR: RangeExample 32 0 3

//Step up/down (by last param). Step size is second param.
//Starting value is penultimate param. (here, 0, 3, 6, 9, 12, 15, 2, 5, etc)
GS: StepExample 32 3 0 15 0 true

//Range list/not in list (by last param) (here, 0 to 4, and 15)
GRANGE: RangeListExample 32 "0-4,15" true

//Range list order up/down (by last param)
GRANGE: RangeListOrderExample 32 "0-4,15" true

//Weighted generation: 10% for DataGen1, 90% for DataGen2
GC: DataGen1 32 3
GC: DataGen2 32 7
GPERCENT: WeightedExample 32 "DataGen1,DataGen2" "10,90"

//Repetitive generation: 10 times for DataGen1, 90 times for DataGen2
GQUANTITY: RepeatExample 32 "DataGen1,DataGen2" "10,90"
```

#### 5.2.5.2. Using file generators in your code

Below is an example of the code that could be used to take advantage of file defined generators. Note that the type used by **SD::GetGen()** is a pointer to **SpvGen**. This is significant because it means that the code is agnostic to the specific generator type defined. The example below looks for a generator named “ConstExample”, but “ConstExample” could be defined as a generator of any type without changing the C++ code.

```

SpvGen* fileGen = NULL;
if(SD::GetGen(fileGen, "ConstExample"))
{
    //Will print: File generator returned: 3
    cout<<"File generator returned: "<<fileGen->Gen()<<endl;
}
else
    cout<<"Could not find generator!"<<endl;

```

### 5.2.5.3. INCLUDE

Often there will be many test configurations in a project for the same DUT. Each configuration may have different generator definitions, for example, one test is very random while another is more directed at a particular case. We could define completely separate text file for each configuration, but then we would probably find that many of the definitions are being duplicated – an clear maintenance problem. The solution is to define a file with the default configuration for all the lines that we reference in the code. We then use the INCLUDE directive to import these definitions. Now recall from above that if a line name is redefined, the last definition is conclusive, so we can redefine the definitions we want to change. In this way, each test configuration only represents the difference between it and the configuration defaults instead of wholesale duplication.

In the example below, we assume that the defintions from the previous example have been placed in a file called “GenDefaults.txt” and that we want to override the “ConstExample” generator. Note that INCLUDE: must appear before the START: directive. Further note that there is no need to change the C++ code.

```

INCLUDE: “../../Runtime/GenDefaults.txt”

START:

//Redefine here as range of 5 to 10, instead of constant 3
GR: ConstExample 32 5 10

STOP:

```

There is one important nuance in the redefinition of generators. If inside we had overridden the “DataGen1” generator, this would have had no effect on the “WeightedExample” and “RepeatExample” generators defined in GenDefaults.txt even though both are defined with DataGen1. The reason for

this is because at the time that these were parsed, DataGen1 had a particular definition – the override only affects DataGen1 after “WeightedExample” and “RepeatExample” have already been parsed and created.

#### 5.2.5.4. Point (Special) Generators

There is one more method for generator override. Say that you have multiple instances of a module in your DUT and you want to define consistent generation across all of them, you’d simply use the same generator for all the instances. But what if the end user sometimes needs to override the generation for a particular instance or instances?

The file format solves this use-case by specifying that if a generator name ends with a dot followed by a number, then that generator will be considered to be a special case of a definition of the same name without the dot. This special case will be identified by the number following the dot. The special case can be recalled from the C++ code by specifying an additional parameter to the **SD::GetGen()** functions. This final parameter is the numerical identification of the special generator. Now here’s the important part; If the special generator ID specified exists, that special definition will be the one returned. But if not, the default (without the point) definition will be returned.

For example:

```
INCLUDE: "../Runtime/GenDefaults.txt"

START:

//Define a special case with ID 2 as a range from 5 to 10
GR: ConstExample.2 32 5 10

STOP:
```

And the C++ code:

```

SpvGen* fileGen = NULL;
    //Will print:
    //File generator returned: 3
    //File generator returned: 3
    //File generator returned: 7
for(unsigned i = 0; i < 3; i++)
{
    if(SD::GetGen(fileGen, "ConstExample", i))
        cout<<"File generator returned: "<<fileGen->Gen()<<endl;
    else
        cout<<"Could not find generator!"<<endl;
}
,

```

One last thing. Even though we've been talking about generators exclusively in the previous paragraphs, the same apparatus also works with other types STR, NUMBER, BOOL, etc, with their respective "Get" functions.

See the SPV\_User\_External\_Control documentation for more details.

## 6. Coverage

### 6.1. *What is Coverage?*

#### 6.1.1. Definition

As mentioned in the Generation Chapter, verification of HDL code requires as many combinations of test input as possible. However, we need a way to check that, in fact, we have covered all the possible states that the HDL code could be expected to be in. In other words, we need a way to check the comprehensiveness of the test suite. **Coverage** is keeping track of the occurrence of a specified circumstance or set of circumstances.

#### 6.1.2. Types of Coverage

##### 6.1.2.1. Simple Coverage

Coverage that relates to the value of one free variable is called simple coverage. An example of simple coverage is tracking the occurrences of values of a single signal. When an occurrence of a particular value has happened, that value is said to be "hit".

##### 6.1.2.2. Cross Coverage

Cross coverage is tracking the occurrence of all the possible combinations (the cross product) of several free variables, for example, tracking the hits of all eight combinations of three single-bit signals. In **SPV**, simple coverage is treated as a degenerate case of cross coverage.

#### 6.1.3. Coverage Classes

SPV provides two related classes for coverage; **SpvCrossCover** and **SpvCoverage**. **SpvCoverage** inherits from **SpvCrossCover**, and so includes its functionality, but provides a simplified interface to coverage initialization for the most common cases.

##### 6.1.3.1. SpvCoverage

**SpvCoverage** is designed for the most straight-forward use of coverage – recording signals states at some event. Usage is simple; Just call **AddItem()** for each signal in the coverage and **Start()** to supply the sampling event (an **SpvEvent** instance) and the coverage name, which will identify the coverage object. The example below demonstrates using **SpvCoverage** for a cross coverage of two columns, each recording a signal, and sampling on the positive clock edge.

```

SpvEvent pClock("BookTestTb.clock");

SpvCoverage* cov = new SpvCoverage;

//Add BookTestTb.data_in for first coverage column
cov->AddItem("BookTestTb.data_in", "Data In");

//Add BookTestTb.data_in_valid for second coverage column
cov->AddItem("BookTestTb.data_in_valid", "Data In Valid");

//Name coverage object "sig1Xsig2" and sample on postive
//Clock edges
cov->Start("sig1Xsig2", pClock);

```

When using **SpvCoverage**, the coverage database will be automatically recorded to a file and will aggregate over multiple runs of the simulation. Optionally, you may specify that a file be created to save the coverage results from a simulation and the level of detail saved in the file. If you opted for a high detail level, then you may also opt to have the coverage object read the file and continue its hit tracking from the status saved in the file. In this way, it is possible to keep coverage statistics across multiple simulations, which is particularly important when executing an unattended simulation series.

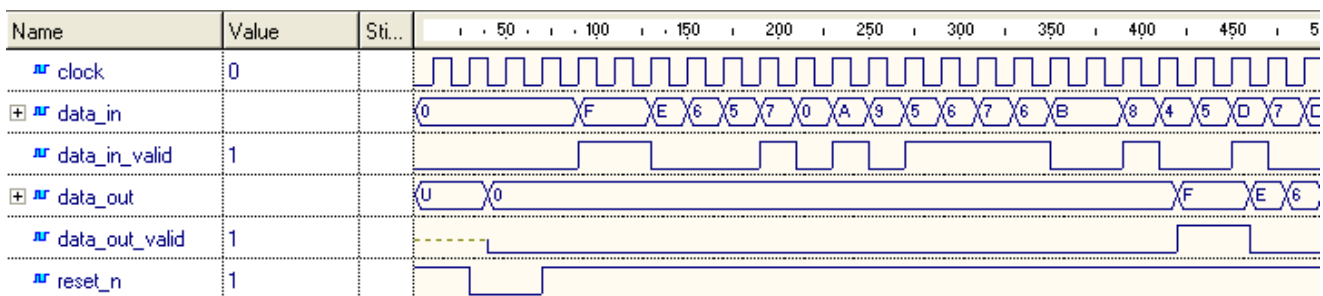
For our example, let's drive random data and a random valid signal and collect coverage on both of these.

```

for(unsigned i = 0; i < 100; i++)
{
    dataEn = GenUnsigned(0, 1);
    dataIn = GenUnsigned(0, 0xF);
    Wait(pClock);
}

```

The waveform from this drive looks like:



The coverage results can be viewed with the SPV Coverage Viewer. Under Windows this is installed in the Start menu in the Simplus group. When activating the viewer, choose Open and locate the SpvCovDir subdirectory created in the simulation directory.

The result for our example is:

SPV Coverage

File Tools

Choose Test: Test No 0:-> Date: (Sun Feb 18 14:59:43:437 2007)

All

(59.0%)

GM Coverage (21.1%)

sig1Xsig2 (96.9%)

Coverage Control

Select

Next Next Next

Fix Coverage Grid

Coverage Info

Cov. Percent	Efficiency	Combinations	No. Hits	Unique Hits	Hits to Full
96.88%	20.53%	32	151	31	1

Hits	Data In	Data In Valid
16	0	0
5	0	1
2	1	0
4	1	1
4	2	0
4	2	1
0	3	0
5	3	1
44	4	0
2	4	1
1	5	0
3	5	1
2	6	0
1	6	1
3	7	0
3	7	1
5	8	0
3	8	1
1	9	0
3	9	1
2	10	0
3	10	1
3	11	0
3	11	1

The leftmost column is the list of coverage objects for our simulation. Here we can see sig1Xsig2, which is what we called our coverage object in the call to **Start()**. The spinner arrows at the bottom allow the user to switch between the runs of the simulation.

In the main window, the white column shows the entry values for the first element in the row to its right.

The number of hits for each entry is displayed in its box. Entries with at least one hit are green while those with no hits are red.

The horizontal row of boxes above the entry ID box show statistics regarding the coverage. Starting from the left, is the percentage covered,



defined as (entries hit at least once) / (total entries). Followed by efficiency, which is the (total hits) / (entries hit at least once). The next box is the randomization seed that will cause a replay of the generation. The last box is simply the number of entries in the coverage object.

The top combo box marked “Coverage Info” shows information on the individual items of the coverage, but only after selecting a field by clicking the combo box and selecting a field.

The “44” is the number of hits for the marked entry, which is the entry for dataEn == 0 and dataIn == 4. The reason why there are so many hits on this entry, 0,4 is because after the 100 drives in the for statement, the simulation was allowed to continue running a bit. As it happened, the last values driven, were 0 for dataEn and 4 for dataIn. These values were sampled at every clock from the last drive until the simulation terminated.

### 6.1.3.2. SpvVirCover and Coverage

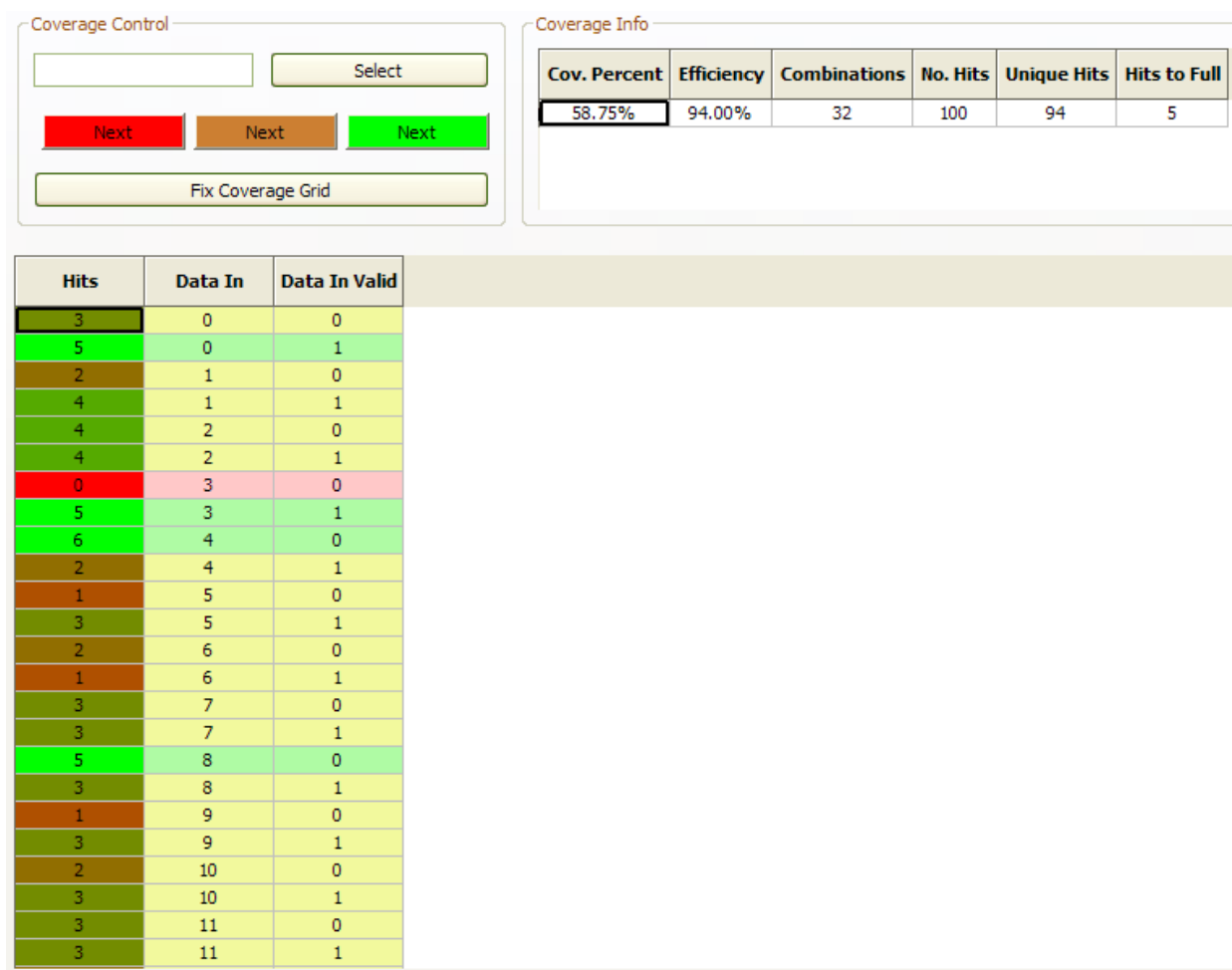
Now let’s say we aren’t interested in sampling values at each clock, but asynchronously on changes to dataEn and dataIn. We need to create an event on the change of either of these signals. To accomplish this we can use **SpvVirCover** to create a “virtual” signal who’s value will rise each time you would like. SpvVirCover is an extension of the SPV language, and it is located in BookTest library.

Now at the coverage definition:

```
#include "../VirCover/SpvVirCover.h"
...
m_VirCover = new SpvVirCover();
m_VirCover->SetFullQuantity(5);
m_VirCover->Init("sig1Xsig2");
m_VirCover->AddItem("BookTestTb.data_in", "Data In");
m_VirCover->AddItem("BookTestTb.data_in_valid", "Data In Valid");
...
m_VirCover->Start();
for(unsigned i = 0; i < 100; i++)
{
    dataEn = GenUnsigned(0, 1);
    dataIn = GenUnsigned(0, 0xF);
    m_VirCover->Trigger();
    Wait(pClock);
}
```

We use Trigger to trigger the coverage whenever we would like. Also, we define SetFullQuantity 5, this defines that 100% coverage is only when we have generated 5 elements of each item.

Erasing the contents of SpvCovDir, so as not to show previous runs, running the simulation, and then running the SPV Coverage Viewer, we get:



Notice that there are fewer hits overall (100). This is because we won't sample situations where the combination of dataEn and DataIn didn't change. Also, we can see that combination 4,0 has 6 hits which is green because it is above 5. Combination 8,1 has only 3 hits which is not red (0 hits) and not green (above or equal to 5 hits).

#### 6.1.4. Adaptive Generation

As we saw in the last section, random generation may often leave some values not covered. This is because each random generation is independent of all the generations before it. If we want to raise the probability of covering all the values within the same amount of simulation time, we need to make the generation smart enough to take earlier generations into account. In other words, we want generation that can change itself with feedback from the coverage, or **adaptive generation**.

During runtime, the coverage objects may be queried for the number of hits on a value (**IsCovered()**), the percentage of values with at least one hit (**CoverPercent()**), the efficiency of the hits (**CoverEfficiency()**). (As an aside, the hit count for a value can be manually incremented using **AdvanceEntryCount()**.)

The **GenerateUnCoverElement()** function can be used to generate a random value that has not been covered (no hits). **GenerateMinCoverElement()** is similar in that it randomly generates a value from within those entries with the least number of hits. The **GetFirstValueThat()** function returns the first value that has been covered/not covered (specified by parameter) from a given reference point. The search order (ascending or descending) can be specified by parameter as well. The **GetFirstRangeThat()**, and **GetRangeListThat()** have a similar functionality, but they return a range or list of covered/not covered values respectively.

Now, let's change the signal drive to use the feedback from the coverage for generation. For the sake of demonstration of the coverage feedback, we'll enter an endless loop of generation by feedback.

```
m_VirCover->Start();
while(m_VirCover->CoverPercent() < 100)
{
    const SpvFastList<unsigned> &currGenList = m_VirCover-
>GenerateMinCoverElement();
    dataEn = currGenList[1];
    dataIn = currGenList[0];
    m_VirCover->Trigger();
    Wait(pClock);
}
```

Taking a look at the code above, we can see that the interior of the generation loop has changed slightly. Instead of generating randomly, we ask the coverage for one of the least covered entries. It returns an array (of type **SpvFastList<unsigned>**) which, for our purposes, is simply an array of values where each value corresponds to one item of the coverage entry. The order of the values is the same order that we called **AddItem()** when we initialized the coverage. So to retrieve the value for dataEn, we take the second value in the list and for dataIn we take the first value in the list.

Erasing the contents of SpvCovDir, so as not to show previous runs, running the simulation, and then running the SPV Coverage Viewer, we get:

**Coverage Control**

Next
Next
Next

**Coverage Info**

Cov. Percent	Efficiency	Combinations	No. Hits	Unique Hits	Hits to Full
100.00%	96.39%	32	160	160	5

Hits	Data In	Data In Valid
5	0	0
5	0	1
5	1	0
5	1	1
5	2	0
5	2	1
5	3	0
5	3	1
5	4	0
5	4	1
5	5	0
5	5	1
5	6	0
5	6	1
5	7	0
5	7	1
5	8	0
5	8	1
5	9	0
5	9	1
5	10	0
5	10	1
5	11	0
5	11	1

## 7. Collecting and Comparing

Now that we can generate and check the effectiveness of the generation, we should proceed to the other side of the verification project – the collection of data and comparison of the actual data versus expected results.

We'll use the same classes for signals and processes to create two new processes, this time in the BookTestCol.cpp file, where the wizard has been kind enough to create the function shells for us. Each of these processes will have the responsibility to collect data from the DUT's interface. One will collect from the input and the other from the output. The input collection process will store the data it monitors at the input in a FIFO structure. The output process will draw an element from the FIFO for each value is monitors at the output and will compare the two after running the input data through a transfer function. The transfer function's job is to simulate the DUT's effects on the input data as it progresses to the output. This is known as the reference model, or often, the "golden" model.

### 7.1.1. Collection Processes

The code below is the input process, implemented in the `BookTestCol::PacketInThread()` function. It simply waits on the positive clock edge and pushes data into the FIFO whenever the valid signal is high. Notice here that some of the signals (reset) and the clock event are not defined in the function. They are defined in the header file as members of the class, as shown by the `m_` convention (member). The reason for this is because these objects will be needed in more than one process, so instead of duplicating them, we define one at the class level and initialize them somewhere else – in this case, in the constructor.

The FIFO object, `m_DataFifo`, is an instance of the STL deque class (deque == double ended queue), which is also defined in the header, so that both processes may access it. The `push_back()` function pushes a value onto the rear of the FIFO. As you may imagine there is also `push_front()` function. Removing elements is via the `pop_back()` and `pop_front()` functions. Accessing elements is either through the `front()` and `back()` functions, if the access is at the edges, or through the index operator (`[]`) for random access. Finally, the `size()` function returns the number of elements in the queue.

```
BookTestCol::BookTestCol()
{
    m_PClock.Init("BookTestTb.BookTest.clock", AtPos);
    m_ResetN.Init("BookTestTb.BookTest.reset_n");
}

void BookTestCol::PacketInThread()
{
    SpvSig data("BookTestTb.BookTest.data_in");
    SpvSig dataEn("BookTestTb.BookTest.data_in_valid");

    Wait(m_ResetN, AtPos);

    while(1)
    {
        Wait(m_PClock)
        if(dataEn == 1)
            m_DataFifo.push_back(data.Uint());
    }
}
```

The output process is somewhat more involved because it will not only record, but it will also compare and declare errors. (A fuller, more modular verification design, would separate the functions of comparator, reference model, and collector into separate classes and objects, but here we'll keep things simple.)

```

void BookTestCol::PacketOutThread()
{
    SpvSig data("BookTestTb.BookTest.data_out");
    SpvSig dataEn("BookTestTb.BookTest.data_out_valid");

    Wait(m_ResetN, AtPos);

    while(1)
    {
        Wait(m_PClock;
        if(dataEn == 1)
        {
            unsigned received = data.Uint();

            if(m_DataFifo.empty())        //Check fifo status
            {
                SPV_OUT(<<"["<<SimTime()<<"] Unexpected: Received,
"<<received<<endl;
                SpvConfig::StopSim(true ;
                return;
            }

            unsigned input    = m_DataFifo.front();
            unsigned expected = TransferFunc(input);
            if(!Compare(received, expected))
            {
                SPV_OUT(<<"["<<SimTime()<<"] Mismatch: Received, "<<received
                <<"", Expected, "<<expected<<endl);
                SpvConfig::StopSim(true);
            }
            else
            {
                SPV_OUT(<<"["<<SimTime()<<"] Compare OK: Received,
"<<received<<endl);
            }
            m_DataFifo.pop_front();
        }
    }
}

```

Here we also wait on the positive clock edge and only do something when the valid signal is high. First we read the output data. Then we check the FIFO status – perhaps it is empty, which would indicate unexpected data. Then we read the head of the FIFO and pass it to a transfer function (see below) and the result of the transfer function, “expected”, is fed to the comparator function which reaches a decision on whether the data is good or not. If not, we print an error message and terminate the simulation. Finally, we pop the head value from the FIFO and start over.

### 7.1.2. Compare

The compare function below is quite simple. Trivial, actually. But, it won't always be so. What if the reference model was not exact? For example, what if the reference model is a floating point model while the DUT is fixed point? There could be round off error involved. Now, ideally, the reference model would be exact, but the world is not an ideal place. So, to avoid false positives (i.e. false flagging of errors), we could design a certain tolerance into the comparison and this would make the comparison a bit less trivial. Conceivably, the tolerance could event be dependent on runtime factors and so on, and end up quite non-trivial.

```
bool BookTestCol::Compare(unsigned received, unsigned expected)
{
    if(received == expected)
        return true;
    else
        return false;
}
```

### 7.1.3. Transfer Function

The transfer function below here is also quite simple, but only because our DUT is trivial. Much of the effort involved in verification in general and DSP projects in particular involves getting the reference model right. In the next chapter we'll see how to call Matlab routines for the refernce model.

```
unsigned BookTestCol::TransferFunc(unsigned input)
{
    return input;
}
```

Try changing the transfer function to do some sort of manipulation on the input data. You should see the collector/comparator catch the DUT's lack of conformance. Now modify the DUT until the test doesn't report any errors and VOILA, you've verified the DUT.

One last word on the comparison and transfer functions. Here, our check is bit accurate, but not clock accurate. That is, no check is made to see if the data comes out when it is supposed to. Sometimes this is necessary as well. In that case, we would not just be checking the data when the output valid signal is high, but we'd be checking the output's valid signal itself, in addition to the data, at every clock. That would mean that our Transfer function would have to be aware of the timing of the input valid and take that into account when predicting both the output valid and the output data.

There is also a completely different kind of check – a rule check. Here, there is no reference model, but there are rules for proper behavior. For example, maximum latency from input to output may not be higher than 100 ns. Here, we would also monitor the input and output, but we'd be looking for rule violations as opposed to comparison to a reference model.

## 8. Calling Matlab routines from SPV

Matlab is commonly found in hardware applications, particularly DSP projects. **SPV**'s companion library, **DPF**, contains interface classes that greatly ease calling Matlab routines for either signal generation and/or as a reference model for the DUT. This chapter explains how to use the interface class to call Matlab routines from within an **SPV** verification application.

### 8.1. Setup

If you used the wizard to create your project with linking to DPF (the default) and Matlab was already installed, you should be already be set up. If not you will have to do the following to your project settings:

- Add the Matlab include directory, usually `<MatlabRoot>\extern\include`, to the “Additional include directories” field of the C/C++→preprocessor settings. For example:  
`C:\Program Files\MATLAB704\extern\include`
- Add the Matlab library directory, usually `<MatlabRoot>\extern\lib\<Platform>\<CompilerVendor>\<CompilerVersion>`, to the “Additional library path” field of the link→input settings. For example,  
`C:\Program Files\MATLAB704\extern\lib\win32\microsoft\msvc60`
- To the link→General settings, in the “Object/library modules” field, add:
  - `libmx.lib`
  - `libeng.lib` (only really needed if you use the Matlab engine, see below)
  - `DpfMtd.lib` or `DpfMt.lib` (for debug and release projects, respectively)

### 8.2. Two ways to call

There are two ways **SPV** supports calls Matlab routines on-line:

- Invoke Matlab engine
- Call Matlab compiled function

The Matlab engine actually starts up an instance of Matlab, which means that it uses a Matlab license. Invoking this instance and communicating with it is accomplished via the functions in the `engine.h` Matlab header file. In short, an initialization function returns a handle to a Matlab instance. Most other functions require this handle for communication



with the Matlab engine instance. Variables can be placed in and retrieved from the engine and functions executed as an evaluation of a string command. Before exiting the simulation, you must call the termination function.

The Matlab compiler (mcc) turns .m file functions into a form that can be executed without a Matlab license (though mcc itself is licensed separately). In this case, mcc will create a binary library with an accompanying header file. The header file will include any functions that were defined as callable and two functions for initialization and termination. The initialization function must be called before any calls to the functions in the compiled library and the termination function should be called before exiting the simulation. The compiled functions themselves come in two forms; mlf and mlx. We will use the mlf-prefixed versions here, because they are easier to use and look more like regular C functions.

	<b>Engine</b>	<b>MCC library</b>
<b>Initialization</b>	Engine* handle = engOpen(NULL); engEvalString(handle, "path_to_m_file")	lib<LIBNAME>Initialize()
<b>Routine Call</b>	engPutVariable(handle, "in", in); engEvalString(handle, "matlab func call string"); out = engGetVariable(handle, "out");	mlfMyfunc(out, in)
<b>Termination</b>	engClose(handle)	lib<LIBNAME>Terminate()

More (and authoritative) information can be found in the Matlab and mcc documentation. The rest of this chapter explains how to call a Matlab function, either through the engine or as a compiled function. The actual compilation using mcc is beyond the scope of this document.

In either case, the **DPF** interface classes are the same.

### **8.3. *DpfMI interface classes***

The interface class directly supports the following Matlab types:

- One and Two dimensional (matrix) numerical arrays
  - Double – **MIDoubleArray/ MIDoubleMatrix**
  - Float – **MISingleArray/ MISingleMatrix**
  - Logical (boolean) – **MILogicalArray/ MILogicalMatrix**
  - Unsigned and Signed integers
    - 8 bit – **MIUint8Array/ MIInt8Matrix**
    - 16 bit – **MIUint16Array/ MIInt16Matrix**
    - 32 bit – **MIUint32Array/ MIInt32Matrix**
    - 64 bit (if supported by your Matlab version) – **MIUint64Array/ MIInt64Matrix**
  - All types, except logical, may also be real or complex
- Structure – **MIStructArray**
- String - **MIString**

Matlab Cells are not supported at this time.

Some important things to note:

- It is important to realize that in Matlab, single values are treated as a degenerate array of size 1, so all types, except strings are represented by array classes.
- The **DpfMI** classes wrap a the Matlab external library's `mxArray*` type. In most cases where a Matlab API function requires an `mxArray*` or `mxArray**` (e.g. output parameters to mlf-prefixed functions), a **DpfMI** class can be used transparently.
- The **DpfMI** classes are intended to ease the interface between Matlab and C++, that is, to translate between the two worlds while hiding the difficulty of the Matlab C API. Keeping this in mind will prevent problems and poor performance.
- Since **DpfMI** classes are used in the C++ realm, the indexing convention follows the C++ language. In other words, array indices start at zero and not one, as in Matlab.
- All of the Array classes support the **Resize()** function which will reallocate the array. Beware, this operation requires copying the old array to the new array.
- All of the **DpfMI** classes support the **ToString()** function which returns a textual representation of the contents of the object, up to a certain maximum number of elements. Beyond this maximum, only the dimensions of the array are printed. The maximum can be changed/retrived with the

**MIEntityBase::SetPrintLimit()/MIEntityBase::GetPrintLimit()** functions.

The DpfMI.h header file includes all of the classes described here.

## **8.4.      *Calling the routines***

Now let's tie it all together in an example:

```
#include <DpfMI.h>
#include <engine.h>
Engine* eng = engOpen(NULL);

//Assert success
_ASSERT(eng);

//If .m file is not on the default Matlab search path,
//you will need something like this.
//engEvalString(eng, "addpath('C:\\m_file_dir');");

.....

//Declare Matlab array
MIStructArray in;

//Implicitly create structure array of size 1,
//and implicitly create field, DoubleData, of size 1 and type double.
in["Data"] = 1;

//Place structure in the scope of the Matlab instance
engPutVariable(eng, "in", in);

//Call function
engEvalString(eng, "out = MyFunc(in);");

//Retrieve structure from Matlab instance. If "out" is not a structure
//this will give an exception at runtime.
MIStructArray out = engGetVariable(eng, "out");

int retVal = out["DataOut"];

.....

engClose(eng);
```

The code above opens an instance of the Matlab engine and adds a directory to the engine's search path. At this point we save the handle on the

side, because we (usually) don't want to open an instance of the engine for every function call as the overhead of the `engOpen()` is significant. The we declare an empty structure, in. We then fill a field, `Data`, with an integer. Its important to understand that by doing this assignment, several things have happened implicitly. First of all, the empty structure is no longer empty. That is, it will now have a size of one. (We could use **Resize()** to give it a larger size. Also, most of the **DpfMI** classes have the option of specifying the array size in the constructor or **Create()** functions). Furthermore, since there was no "Data" field, one is now created. Finally, since "Data" previously had no type, the type is implicitly set in the assignment to whatever is being assigned – in this case, a 32 bit signed integer. That's a lot of implicit stuff going on, but it is key to the usability of the **DpfMI** objects.

The next line, `engPutVariable()`, places the structure into the Matlab engine's environment. Once it is there we can use it in the next line, which is the call to the Matlab routine. Notice how the input parameter to the `MuFunc()` function is the same "in" that we specified in the previous line. Here we also tell Matlab to store the results in a variable called "out". But this variable only exists in the engine, so we call `engGetVariable()` to retrieve it. Notice how we use the return value as the initialization of another **MIStructArray**. This will cause the out variable to wrap the `mxAarray*` returned by `engGetVariable()`.

Finally, we use the output by assigning it to a variable. Note that if there is no field name "DataOut", it will be created. But if there was no field, then there is also no data, and that will cause an exception when we attempt the assignment.

The build-input, `engPutVariable()`, `engEvalString()`, `engGetVariable()`, use-ouput, sequence will often repeat many times in a single simulation. Ultimately, we will finish the simulation, where we call `engClose()` to close the Matlab engine and free all of its resources.

And as we're talking about freeing resources, it's a good time to mention that the **DpfMI** classes keep track of the Matlab objects they wrap. Under most conditions, they will release the associated Matlab object when it is no longer in use. However, should you want to release an object early, say because the object is very large, the **Destroy()** function will do that for you. Beware, however, and make sure that no one else is still holding on to that data – **DpfMI** has no way of knowing when non-DpfMI code is still using an object. On the flip side, it may be desirable to prevent the release of a Matlab object, say, if non-DpfMI code is still holding a reference to it when the **DpfMI** object goes out of scope. The **SetNoDestroy()** function fills this need, but *someone* will have to call Matlab's `mxDestroy()` function to release the resource yourself or risk a memory leak.

Now, let's see the same code built for the Matlab compiler:

```

#include <DpfMl.h>
#include "libMylib.h"

libMylibInitialize();
.....

//Declare Matlab array
MlStructArray in;

//Implicitly create structure array of size 1,
//and implicitly create field, DoubleData, of size 1 and type double.
in["Data"] = 1;

MlStructArray out;
mlfMyFunc(out, in);
int retVal = out["DataOut"];

.....

libMylibTerminate();

```

This comes out a bit cleaner. The initialization and destruction of the Matlab runtime is through the Initialize/Terminate functions that are created by `mcc` for each library. The calls are reference counted, so while you may call the Initialize function more than once, each call must eventually have a matching call to the Terminate function, or the library won't ever really terminate. Note that there is no need to set a search path at runtime, but there is a need to specify the library to the linker, as well as the header file to the compiler, at compile time.

The assignment of the Data field in the "in" structure is the same. The big difference here is that there is no set/get of variables and that the function is called in a C-ish fashion. (The format for the `mlf` function, in which the out parameter as the first function argument, is the format for the `mcc` that was current to Matlab 7. An earlier version of `mcc` would return the first output parameter as the return value of the `mlf` function. In this case, out could be assigned to the function's return value, similar to way we called `engGetVariable()` in the previous example.)

One more note that is relevant to both methods. Calling Matlab like this, that is, for single values, is very inefficient. Doing so for each sample in the simulation would bring performance down significantly. Matlab is designed to work on vector (arrays) of data and is optimized to that end. So you are usually better off collecting a block of data (say, with the help of the

deque class) and then calling the Matlab routine on the entire block, the trade-off being greater code complexity.

## 8.5. *More usage*

Now that we've gone through how to call a function, let's delve a bit more into the nuances of how to use the **DpfMI** objects.

One of the limitation of the **MIStructArray** is that you cannot assign into the middle of a field that is a vector. That is, the code below will not compile.

```
//Declare Matlab array
MIStructArray in;

in["Data"][6] = 1;
```

To accomplish this task, we first create a numeric array and then assign it to the structure.

```
//Declare Matlab array
MIStructArray in;

MIInt32Array a;
a.Resize(10);
a[6] = 1;
in["Data"] = a;
```

Don't worry overly much about the array copy. Matlab uses a copy-on-write scheme which only truly duplicates copies when one or both of their contents change following the copy. This is true as long as both the source array and the destination field are the same type – here, a 32 bit integer (an uninitialized field, as above, will automatically be set to the same type as the source array). If the types are different, there will be a copy and conversion of each and every element.

In the other direction, that is, when you want to access array data that already exists a field of the array, you use an array reference to access the elements, as shown below. Here, there is no copying, but the type is sensitive

– the field must actually be the same type as the cast. Notice that you can resize and array field this way, so this could be used as an alternative to the copy in the last example.

Continueing from above...

```
//Declare Matlab array
MIStructArray in;

MIInt32Array& ref = in["Data"];
cout<<ref[6]<<endl;
ref.Resize(20);
ref[15] = 100;
```

The **DpfMI** classes do self checking for most function calls or operator usage (which is really the same thing), so when doing something in a loop, you will get better performance if you access the underlying data directly. See below.

```
// Create array of type double and resize it to 10 elements
MIDoubleArray d;
d.Resize(10);

// Fill d with values using index ([]) operator
unsigned i;
for(i = 0; i < d.Size(); i++)
    d[i] = i * 1.1;

// The same, but with better performance
double* dPtr = d;
for(i = 0; i < d.Size(); i++)
    dPtr[i] = i * 1.1;

in["ArrayData"] = d;
```

The first loop uses **MIDoubleArray**'s index operator to access the elements of the array. The second loop uses the underlying memory pointer, accessed implicitly here just by assigning it to a double\*. (There is also a **GetRealPtr()** function for explicit access.) This will be more efficient than the first loop. One word of caution, however, is that when extracting the

underlying pointer, the pointer type must match the array type, or a runtime exception will occur. This is because once you directly access the array memory, there is no way to keep the user from corrupting it. One sure way to corrupt the memory is to project and use a pointer of the wrong type to the array's memory. If you are sure you know what you are doing, you can use the **GetRealVoidPtr()** to return a type-neutral pointer to the array's memory. And, of course, once you have the pointer, you can cast the type to whatever you want – C++ will let you shoot yourself in the foot if you insist....

Matlab arrays aren't very good for collecting data, if you don't know in advance how many samples will be collected. When you need this kind of queue, you can either allocate the maximum buffer size, resize as you go along (an expensive operation), or you can use STL's deque and copy the FIFO before you call a Matlab routine. For the other direction, there is no assignment from a field to a deque, but there is the **CopyTo()** function instead. By the way, versions of these also exist for STL vector, in addition to deque. In all cases, the destination array/vector/deque will be resized to the source's size.

```
deque< unsigned > fifo;  
  
....  
  
in["FifoData"] = fifo;  
in["FifoData"].CopyTo(fifo);
```