VPIphotonics Design Suite™

Developer Guide

VPIphotonics
DESIGN AUTOMATION

How to contact VPIphotonics GmbH

www.VPIphotonics.com	Web site
info@VPIphotonics.com	General information
sales@VPIphotonics.com	Sales, ordering information
support@VPIphotonics.com	Technical support, license key requests

April 2024. Document Part Number: TC0-UM03-16 24108

Contents

Preface

Optical Design and Configuration Documentation Library

The complete manual set can be found online. The following manuals are available:

- *VPIphotonics Design Suite™ User Interface Reference* – for a complete description of the options available in the Photonic Design Environment and VPIphotonicsAnalyzer user interface
- *VPIphotonics Design Suite™ Simulation Guide* – describes the simulation process in the Photonic Design Environment
- *VPIphotonics Design Suite™ Developer Guide* – describes advanced simulation techniques and automation methods, including simulation scripts, macros and cosimulation with external programming solutions and third-party tools
- *VPIphotonics Design Suite™ Photonic Modules Reference* – multivolume set with details on photonic and electronic numerical models (*Photonic Modules*) and communication/analysis models (*Signal Processing Modules*)
- *VPItransmissionMaker™Optical Systems User's Manual* – for modeling photonic systems and subsystems for access, aggregation and core network applications
- *VPIcomponentMaker™Fiber Optics User's Manual* – for modeling doped-fiber/-waveguide amplifier and laser applications, as well as multi-pump Raman amplifiers and hybrid amplification schemes
- *VPIcomponentMaker™Photonic Circuits User's Manual* – for modeling semiconductor optical amplifiers, laser-based applications, and photonic integrated circuits
- *VPIlinkConfigurator™User's Manual* – for performing cost-optimized equipment configuration of linear and branched networks allowing easier design, and faster, more competitive response to application requests and RFQs
- *VPIlinkConfigurator™Customization Reference* – for customizing VPIlinkConfigurator to individual requirements using its Application Programming Interfaces
- *VPIlinkDesigner™User's Manual* – for performing cost-optimized optical network design
- *VPIdeviceDesigner™Getting Started* – for a brief introduction into the main capabilities of VPIdeviceDesigner

- *VPIdeviceDesigner™User's Manual* – for detailed information on the product functionality of VPIdeviceDesigner and recommended design workflows
- *VPIdeviceDesigner™Library Reference* – for describing the application user interface of the VPIdeviceDesigner Python library
- *Release Notes* – for detailed information on new features and recent updates.

Conventions

This section explains the typographic conventions and other notations used to represent information in this guide. Elements of the user interface are indicated as follows:

- Buttons, menus and controls appear in a **medium sans serif font**, such as **File**.
  A sequence of menu commands is indicated as follows: **File > Save**.
  (In this case, select the **Save** command from the **File** menu.)
- Module names appear in *medium italics*.
- The names of module libraries, port names and equation variables appear in *italics*.
- The names of module parameters appear in a **`bold fixed-width`** font.
- Parameter settings, programming examples, command names, and constants are indicated with a `regular fixed-width` font.
- Variables and optional arguments are shown in an *`italic fixed-width`* font.

Chapter 1

Macros

VPIphotonics Design Suite™ allows you to combine a sequence of commands in a script that can be called from a menu or within a schematic to automate a task or series of tasks. These scripts are called "macros" and written in VPI's *PDE Scripting Language*, which offers direct access to the simulation features of the Photonic Design Environment, and also supports additional commands that cannot be performed via the graphical user interface.

Macros may be customized to work with a particular simulation setup and attached to a specific schematic, or perform more general functions that can be applied to any schematic.

One example of a macro would be a script that modifies the parameter settings of specific modules on a schematic to select between different variations of a single simulation setup.

Another macro might modify only standard global parameters, or parameters of modules of a certain class (for example, fibers), and could, therefore, be applied to many different setups.

The PDE Scripting Language is a simple language, but it is very powerful, and can be extended in many ways. Some of the features of the PDE Scripting Language are:

- Two dialects are supported: the first based on *Tcl* ([1],[2]) and the second based on *Python*. Both of them have a rich set of mathematical, file, text manipulation, control, and other commands. This gives them power to be extended in creative ways, such as programming in expertise, reading from design specification files, and opening other applications (such as audio files for online tutorials).
- Commands to mimic keyboard and mouse interactions with the PDE are provided. Thus repetitive tasks can be replaced by macros which can be used to automate editing tasks, search for and set parameters, or add grounds to unterminated ports.
- The script language includes an extension that adds a *wizard* facility, which displays a multiple-page interactive dialog. This can be used to prompt a user's interaction, such as: '*How many kilometers long is the link you wish to build?*'. It can be used for more trivial (but useful) tasks, such as automated replacement of one module with another, user-chosen type, and input value conversion or handy calculator, or to specify the range of an automated test. Because the PDE Scripting Language is

based on general-purpose scripting languages, the wizards can include mathematical calculations, in which design rules can be coded.

- In some cases, macros can be used to fully automate routine tasks, such as optimization: a script can change the schematic topology (or the number of channels) to maximize its performance.

  In other cases, macros may be written to assist in the design process, and user intervention may be necessary to run this type of macro. An example is an attempt to close a modified schematic without saving it: the PDE scripting engine would normally display a dialog window asking the user if it is necessary to save the changes.

The features and syntax of the PDE Scripting Language are quite similar to the Simulation Scripting feature described in Chapter 3, "Simulation Scripts". However, there are a few important differences:

- PDE macros are executed prior to (and, sometimes, after) the simulation run. They have access to schematic topology, initial settings, files attached to schematics, etc. However, there is no fine-grained access to the simulation workflow, final parameter values (except those saved to an output file) and so on.

  This also means that some of the macro commands may differ in semantics and results from their counterparts in simulation scripting (though the command *syntax* may be very similar). Refer to the descriptions of each command for details.

Programming in the PDE Scripting Language has many advantages typical of a high-level scripting language:

- The commands are mapped onto physical actions such as keyboard and mouse movements to place, link, edit parameters and run simulations. Thus anyone who has used VPI Design Suite manually will understand its commands set, and what is required to form a simulation.
- The effects of commands are easy to see by running the PDE simultaneously with the script editor.
- The language can be used at a very basic level, without complex structures, however complex structures and control commands can be added as the user becomes experienced, and these procedures can be easily shared.

This chapter introduces the PDE Scripting Language by providing examples of how wizards are created, simulations are synthesized, and schematics can be interrogated. A detailed description of all PDE Scripting Language commands is provided in Appendix A, "PDE Scripting Language Reference". Moreover, Appendix B, "Build-In Macros" contains a list of all the built-in macros along with a brief description of their functionality.

Conventions

Language commands are represented with a `regular fixed-width` font. Variables and optional arguments are shown in an ***`italic fixed-width`*** font.

Optional parts of a Tcl command are enclosed in question marks such as `?something?`, and for Python they are enclosed in square brackets.

Tcl Basics

This section provides a brief introduction to Tcl concepts to help you understand how the language is used in VPI Design Suite. For more detailed information, refer to the *Tcl/Tk Documentation* available from the **Help** menu, or visit the Tcl Developer Xchange website at http://www.tcl.tk/doc/.

Two excellent reference books on *Tcl* are authored by Ousterhout [1] and Welch [2], from which this information has been distilled.

Commands

A *Tcl* script is a string which contains one or more commands. In contrast to other scripting and compiled languages as Python, MATLAB or C++, Tcl has no special constructs for procedure definitions, conditionals or loop statements. Those are also represented by commands.

White space characters such as spaces or tabs are used to separate commands and their arguments. Command sequences are separated by semicolons or newlines and consist of a command name and zero or more arguments:

```
command1 arg1 arg2 arg3 ...; command2 arg1 arg2 arg3 ...
command3 arg1 arg2 arg3 ...
```

The command may be one of Tcl's built-in commands, a command provided by the Photonic Design Environment or a Tcl procedure defined in a script.

A Tcl procedure can be defined with the `proc` command:

```
proc name arglist body
```

The first argument is the procedure name, the second is a list of parameters and the third is a command body that contains one or more Tcl commands. For example:

```
proc Diag {a b} {
      set c [expr sqrt($a * $a + $b * $b)]
      return $c
}
```

Note: Because `proc` is a command and *body* is an argument, the opening brace must appear on the same line. (The brace at the end of the first line starts the third argument to `proc`, which is the command body.) This also applies to commands that are used to control program logic (like `for`, `if`, etc.).
Also, the space between the *`arglist`* and *body* braces is not optional – it serves to separate the `proc` arguments.

After the procedure is defined, it can be used like any other Tcl command:

```
set d [Diag 3 4]
```

Substitution

Before calling a command, the Tcl interpreter performs several types of substitution to calculate the values of arguments.

Variable Substitution

Variable substitution is used to access the value of a variable. The dollar sign ("`$`") is used to trigger variable substitution in one of the following forms:

- `$`*`name`*
  Where ***name*** is the name of a scalar variable
- `$`*`name`*`(`*`index`*`)`
  Where ***name*** is the name of an array variable and ***`index`*** is an element in the array.

When a statement like this is evaluated, it is replaced with the value of the variable.

The `set` command is used to assign a value to a variable:

```
set name value
```

The following program will display a dialog with the message *"Bob has 2 apples."*:

```
set a 10; set b(Name) Bob; set b(4) 5
messagedialog "$b(Name) has [expr $a/$b(4)] apples."
```

Array variables in Tcl represent associative arrays and are closer to maps or dictionaries in other languages than to regular arrays. Items in an array are accessed not by their index but by name (key).

Command Substitution

Command substitution is used to call one command to construct arguments for another. A nested command is delimited by square brackets ("`[]`"). The Tcl interpreter takes everything between the brackets and evaluates it as a command. For example:

```
set d [Diag 3 4]
```

In this case, the nested command is `Diag 3 4`. After substitution, the initial command will resolve to:

```
set d 5.0
```

Arbitrary nesting of commands is supported, and nested commands are evaluated first, so their result can be used in arguments to the outer command.

```
set val [lindex [lindex [lindex 2] 2] 3]
```

Backslash Substitution

Backslash substitution occurs if a backslash character ("\") appears within a string. In most cases, the backslash is dropped and the following character is treated as an ordinary character. This mechanism can be used to insert literal characters that would otherwise have a special meaning (like braces, square brackets or the backslash character itself).

Certain backslash character sequences are treated specially:

- `\n` - is replaced with the newline character (`0xa`)
- `\r` - is replaced with the carriage-return character (`0xd`)
- `\t` - is replaced with a tab (`0x9`)
- `\<newline>`*`whiteSpace`* - a single space character replaces backslash, newline and all whitespace characters after the new line.

For example, the following construct could be used to assign a value of "`$a[\`" to a variable:

```
set str \$a\[\\
```

Backslash-newline substitution may be used to break long lines:

```
set totalCost [expr $amount1*$cost1 + $amount2*$cost2 + \
              $amount3*$cost3]
```

Grouping: Double Quotes and Braces

In Tcl, everything is a string and it is up to each command to interpret those strings. In contrast to other languages, there is no need to wrap something in quotes to indicate that it is a string. However, in some cases it may be necessary to group several characters that include whitespace or even newlines into one argument. This can be done with double quotes (`" "`) or curly braces (`{}`). The main difference between these methods is that all types of substitution are performed on characters in double quotes and only backslash-newline substitution is performed within curly braces.

For example:

```
set s Hello
messagedialog "The length of $s is [string length $s]."
messagedialog {The length of $s is [string length $s].}
```

The first message will show *"The length of Hello is 5."*, but the next example prevents substitution, so the string will appear as-is: *"The length of $s is [string length $s]."*.

Braces are especially valuable when providing input for commands such as `if`, `for`, `proc`, etc., which also perform substitution, but later during their execution.

Comments

Like many other scripting languages, Tcl uses the number sign[1] (#) as a line comment marker to embed annotations in scripts: any text between the # and the end of the line is ignored. However, in Tcl, the # must occur at the *beginning* of the command in order to be treated as a comment marker. A # that occurs elsewhere is not treated specially.

To append an inline comment to the end of a command, precede the # with a semicolon. In this case, the # will be recognized as a comment marker.

```
# Here are some calculations
set a [expr 2 * 3]
set b [expr sqrt($a)]; # Just some note
```

Math Expressions

The Tcl interpreter itself does not evaluate math expressions, but the `expr` command is provided for this purpose. It supports the same syntax as C expressions and a broad range of math functions, such as `sin`, `cos`, `log`, `sqrt`, etc.

```
set x [expr 3.0*6.0]
set y [expr sin(2.0/4.0)]
```

Internally, `expr` handles integer, floating point and boolean values. Integer values are promoted to floating point values as needed, but it is important to keep in mind that since Tcl itself operates with strings, the values consumed and produced by `expr` are converted from strings to numbers and vice versa during command execution.

Floating point values are distinguished from integers by a decimal point or engineering notation. This requires careful handling to avoid unexpected results:

```
# The following line produces 0
set a [expr 1/2]
# The following line produces 0.5
set a [expr 1.0/2]
```

Tcl Lists

A list is an ordered tuple of values. In other languages, this is sometimes known as an “array” or “vector”. A Tcl list provides a convention on how to interpret a certain string. The elements in a list are separated with white spaces. Lists may contain other lists and support arbitrary nesting. They can be created directly or using the `list` command:

```
# The following two commands create the same list
set l {1 2 Hello {two words}}
set l [list 1 2 Hello {two words}]
```

Tcl provides an extensive set of built-in commands to work with lists, such as `llength`, `lindex`, `lappend`, `linsert`, etc.

1. Also known as a “pound character”, “hash symbol” or “octothorpe”.

Execution Control Statements

As mentioned previously, Tcl has no special syntax for execution control statements but instead provides a set of built-in commands for the same purposes.

A conditional statement is represented with the `if` command. Loops can be created with `for`, `foreach` and `while`.

```
foreach inst [getblocks] {
        set port_res ""
        foreach port [getports $inst] {
                set types [getsignaltypes $inst $port]
                if {[lsearch $types elwave] >= 0 && [isportvisible $inst $port]} {
                        lappend port_res "$port supports elwave"
                } else {
                        lappend port_res "$port doesn’t supports elwave"
                }
        }
        lappend res "$inst $port_res"
}
messagedialog $res
```

References

[1] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, Reading, MA, USA, ISBN 0-201-63337-X.

[2] B. Welch, *Practical Programming in Tcl and Tk*, 4th Edition, Prentice Hall, ISBN 0-13-038560-3, 2003.

Python Basics

This section provides a brief introduction to Python concepts to help you understand how the language is used in VPI Design Suite. For more detailed information, visit the Python 3.11 documentation website at https://docs.python.org/3.11/.

Statements and Expressions

Statements in Python are instructions that perform specific actions or control the execution flow. Python supports such statements as assignment, expression, `import`, `return`, `pass`, `raise`, `if`, `for`, and several more:

```
import vpi_tc_pde as pde # Import statement
a = 10 # Assignment statement
(a + fun(3))*5 # Expression statement

def fun(prm): # Function definition statement
      if prm < 0: # Conditional statement
            return 0 # Return statement
      else:
            return prm
```

Expression statements are used to compute a value or call a procedure. Math operators may be used in expressions directly:

```
(degrees(asin(0.5)) + exp(x)) * cos(radians(60))
```

Execution flow statements like `if`, `for`, `while` are used to define conditional and loop constructs. Such statements are ended with a colon (`:`) and followed by a block of subordinate statements:

```
for inst in pde.getblocks():
      port_res = ""
      for port in pde.getports(inst):
            types = pde.getsignaltypes(inst, port)
            if "elwave" in types and isportvisible(inst, port):
                  port_res += "{} supports elwave".format(port)
            else:
                  port_res += "{} doesn’t supports elwave".format(port)
      res += inst + port_res
pde.messagedialog(res)
```

Blocks and Indentation

There are no braces {} or other symbols to define a block of statements in Python. Instead, blocks are identified by having the same indentation:

```
if a>b:
      # This block of statements will be executed if condition is true
      res = a
      pde.writemessage("res = {}".format(res))
else:
      # And this block if false
      res = b
      pde.writemessage("res = {}".format(res))
```

Comments

A hash sign (#) that is not inside a string begins a comment. All characters after the # and up to the end of the physical line are part of the comment and are ignored.

```
# First comment
a = 10 # second comment
```

Variables and Datatypes

Variables are named values. Their names should begin with a letter or underscore and are case-sensitive. Variables do not need to be declared and their datatypes are inferred from the assignment statement. Python supports the following basic data types: numbers (integers, long integers, floating point numbers, and complex numbers), strings, and collections (tuples, lists, dictionaries). In addition, Booleans are considered as a subtype of plain integers.

```
counter = 100 # An integer assignment
length = 1000.0 # A floating point
name = "Azimuth" # A string
value = '{TimeWindow}/2' # String in single quotes
root = 5.7-1.5j # A complex value
res = [] # An empty list
d = {"title" : "Power", "value" : 15} # A dictionary
```

After assigning a variable, it is possible to refer to it and perform various operations with it:

```
res.append(counter*2 + root + d["value"])
```

Functions

Functions (sometimes called procedures or routines) are reusable pieces of code. Once a function is defined, it may be called multiple times. Functions calculate and return some value or change some state.

```
coeff = 3 # Assign global variable

def fun(value): # Define a function
      global coeff # Declare a variable as global to access it from function
      res = sin(value * coeff)/2 # Calculate some value
      return res # Return it

a = fun(55) # Call function
```

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope. This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

Classes

Python is an object-oriented language and allows to define and use such OOP idioms as classes and objects. A class is a user-defined prototype for an object that defines a set of data members and methods, and an object is a unique instance of a particular class.

```
class Rectangle(object): # Definition of a class
      def __init__(self, width, height): # Constructor
            self._width = width
            self._height = height

      @property
      def width(self): # Definition of the property
            return self._width

      @property
      def height(self):
            return self._height

      def getArea(self): # Definition of the method
            return self._width*self._height

r = Rectangle(10, 20) # Create instance of a Rectangle class
pde.writemessage(r.width) # 10
pde.writemessage(r.getArea()) # 200
```

Modules and Packages

A module is a grouping entity that may define functions, classes, and variables. Regular modules correspond to Python files. A package is a hierarchical directory structure which contains other packages (directories) and modules (Python files). A directory is considered a package if it contains the file `__init__.py`. Modules and packages provide the ability to logically organize Python code. There are a lot of standard packages and modules, and more can be found on the Internet.

```
import numpy as np # Import "numpy" module under the alias "np"
from math import * # Import everything from the "math" module

a = np.zeros(5) # Call the "zeros" function from the "numpy" module
b = sin(pi/2) # Access "pi" constant and "sin" from "math" without any prefix
```

Commands specific to the PDE Scripting Language Macro itself reside in the module named **`vpi_tc_pde`**. It needs to be imported at the beginning of the macro file to be able to access those commands. It is possible to import this module under some alias, which is preferable because it allows you to avoid name conflicts, or import everything into the current namespace which sometimes may lead to name clashes:

```
import vpi_tc_pde as pde # Import with alias "pde"
pde.writemessage(pde.curuniverse()) # Command shall be preceded with module alias

from vpi_tc_pde import * # Import everything into the current namespace
writemessage(curuniverse()) # Access the command directly
```

Note: All subsequent examples imply that `vpi_tc_pde` module is imported under the alias `pde`.

Math Expressions

Python natively supports basic math calculations with numeric datatypes in any expression. These are +, -, *, /, % (modulus) and ** (exponentiation). Precedence of operators can be controlled using parentheses. Many useful mathematical functions are available in the standard `math` module:

```
import math
val = math.sin((val+1)**2)
```

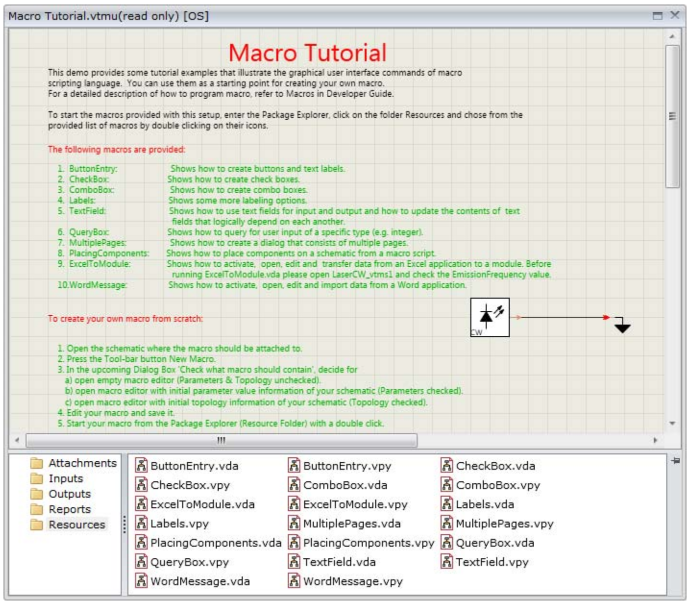
VPIdesignSuite is distributed with the very powerful libraries *NumPy* and *SciPy*. The first provides matrix calculation, and the second contains routines for optimization, linear algebra, FFT, and so on.

```
from math import *
import numpy as np
from scipy.fftpack import fft, ifft

a = np.array([20,30,40,50])
b = np.arange(4) # array([0, 1, 2, 3])
c = a-b # array([20, 29, 38, 47])
b**2 # array([0, 1, 4, 9])
10*np.sin(a) # array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])

a = np.ones(3, dtype=np.int32)
b = np.linspace(0,pi,3) # Returns evenly spaced numbers over a specified interval
c = a+b # array([ 1., 2.57079633, 4.14159265])
# array([0.54030231+0.84147098j,-0.84147098+0.54030231j,-0.54030231-0.84147098j])
d = np.exp(c*1j)

a = np.array([[1.0, 2.0], [3.0, 4.0]])
a.transpose()
np.linalg.inv(a)

x = np.array([1.0, 2.0, 1.0, -1.0, 1.5])
y = fft(x) # Fast Fourier Transform
yinv = ifft(y) # Inverse discrete Fourier transform
```

Locating, Creating and Running Macros

Macros that are associated with a specific schematic are commonly stored as attachments to the schematic's work package in the `Resources` folder. Some of the built-in VPI demonstrations include macro attachments (`.vda` and `.vpy` files). You can run the attached macros by double-clicking them, or right-clicking them and choosing **Run** from the context menu.

In particular, the examples described in this chapter can be found in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial*, which is shown in Figure 1-1.



Macro Tutorial.vtmu(read only) [OS]

Macro Tutorial

This demo provides some tutorial examples that illustrate the graphical user interface commands of macro scripting language. You can use them as a starting point for creating your own macro.
For a detailed description of how to program macro, refer to Macros in Developer Guide.

To start the macros provided with this setup, enter the Package Explorer, click on the folder Resources and chose from the provided list of macros by double clicking on their icons.

The following macros are provided:

1. ButtonEntry:	Shows how to create buttons and text labels.
2. CheckBox:	Shows how to create check boxes.
3. ComboBox:	Shows how to create combo boxes.
4. Labels:	Shows some more labeling options.
5. TextField:	Shows how to use text fields for input and output and how to update the contents of text fields that logically depend on each another.
6. QueryBox:	Shows how to query for user input of a specific type (e.g. integer).
7. MultiplePages:	Shows how to create a dialog that consists of multiple pages.
8. PlacingComponents:	Shows how to place components on a schematic from a macro script.
9. ExcelToModule:	Shows how to activate, open, edit and transfer data from an Excel application to a module. Before running ExcelToModule.vda please open LaserCW_vtms1 and check the EmissionFrequency value.
10.WordMessage:	Shows how to activate, open, edit and import data from a Word application.

To create your own macro from scratch:

1. Open the schematic where the macro should be attached to.
2. Press the Tool-bar button New Macro.
3. In the upcoming Dialog Box 'Check what macro should contain', decide for
   a) open empty macro editor (Parameters & Topology unchecked).
   b) open macro editor with initial parameter value information of your schematic (Parameters checked).
   c) open macro editor with initial topology information of your schematic (Topology checked).
4. Edit your macro and save it.
5. Start your macro from the Package Explorer (Resource Folder) with a double click.

Figure 1-1 *Macro Tutorial schematic, containing examples for this chapter*

Macros that provide more generic commands for common applications can be added to the Macros button menu on the Home tab of the ribbon or to the Favorites toolbar for easy access from any schematic.

VPIphotonics provides a variety of useful macros with the default installation. Most of them can be found in the Macros button menu on the Home tab of the ribbon. You can add your own macros to this list as described below. Some resource-specific macros can be found in the Macros submenu of a resource's context menu.

Attaching a New Macro to an Existing Design

You can write macros from scratch or base them on existing ones.

Two types of user macros are supported in VPI Design Suite: Python macros (`.vpy`) and Tcl macros (`.vda`). It is possible to select a required language dialect by selecting one of these two file extensions.

To create a new macro, open the schematic that you want to run the macro on, and click Home > New Macro. The New Macro dialog appears so you can choose one of the two supported extension types (`.vpy` or `.vda`), and where to save the macro. (The dialog shows the `Resources` subfolder of the current design package by default.) Likewise, typing a path to the future macro and/or a macro name with the `.vpy` extension in the File name field will create a new Python macro.

Then you will be prompted to select whether you want the parameters of the schematic, the topology of the schematic, or both to be automatically included in the new macro (as described below). Once you close this dialog, the new macro will be created, and opened for editing in Notepad (or the external editor specified in the application preferences).

There are a number of ways to base a macro on an existing script. You can copy and paste a macro from the `Resources` folder of one package to the `Resources` folder of another schematic. Standard copy and paste keyboard shortcuts (`CTRL+C` and `CTRL+V`) also work on text selected within the text editor, so you can simply copy the required section of an existing macro and paste it into a macro of your own. You can also right-click a macro in the `Resources` folder and select Save As.

Macro with Schematic Parameters

Snapshots of the state (values) of all parameters on a schematic can be captured at any time. The parameters are saved in a macro as an attachment to the schematic. The complete set of parameters can be restored later by running the macro.

To save the current parameter settings as a macro:

1. Click the New Macro button on the Home tab of the ribbon.
   The New Macro dialog appears.
2. Use the New Macro dialog to choose where to save the macro and the language dialect.
   Normally it should be stored in the `Resources` subfolder of the current design package.
   The Create window appears.
3. Select the Parameters checkbox.
4. Clear the Topology checkbox.
5. Click OK.
   A new macro is created containing all the commands necessary to set the value of each parameter in the schematic to its current value.
   Whenever this script is run, it restores the parameter settings to the values that were saved in the macro.

This feature is useful for demonstrating several versions of parameter settings, say to a customer, or for saving particular settings while experimenting with different parameter values, so that they can be easily restored later.

Macro with Schematic Topology

Snapshots of the schematic topology (layout of components) can be captured at any time. The topology is saved in a macro as an attachment to the schematic.

The current parameter settings can also be saved as part of the macro. If the parameters are not saved in to the macro, all module parameters will be set to their default values when the topology is recreated using the macro.

Of course, particular parameter sets can be saved separately as parameter macros, as described in the previous section.

To save the current topology as a macro:

1. Click on the New Macro button on the Home tab of the ribbon.
   The New Macro dialog appears.
2. Use the New Macro dialog to choose where to save the macro and the language dialect.
   Normally it should be stored in the `Resources` subfolder of the current design package.
   The Create window appears.
3. Select the Topology checkbox.
4. To save the current parameter settings along with the topology, select the Parameters checkbox.
5. Click OK.
   A new macro is created containing the commands necessary to generate all the elements of the current setup (including text boxes), place them in the same locations, and connect them together. If the Parameters checkbox was also selected, the commands required to set the value of each parameter in the schematic to its current value will also be included.
   Whenever this script is run (by double-clicking its icon), it recreates the current setup.

WARNING: The **default** command in the automatically-generated macro is `reset`. This deletes the contents of the schematic. The remainder of the macro will then recreate the entire schematic and set its parameters. You will lose your work if you run the macro without saving your changes elsewhere.

If you want a new schematic (universe) for the new topology, the `reset` command can be replaced with the `newuniverse` command:

In Tcl:

```
newuniverse [uniquename] SDF
```

In Python:

```
newuniverse(uniquename(), "SDF")
```

Similarly, the `domain` command can be added after the `newuniverse` command if the non-default simulation domain is necessary (such as SMATRIX domain for PIC device models).

Adding the line `run 1` (`run(1)`) at the end of the script causes it to automatically run the simulation.

Editing Macros in Designs

A simple text editor can be used to modify macros using the PDE Scripting Language. To edit an existing macro, right-click the icon of the macro in the Resources folder, and select Open from the context menu. Macros can be edited while the PDE is in use, so you can verify that the change you make to the macro actually produce the desired results.

Advanced users may prefer to edit macros in a special editor (or IDE), that provides syntax highlighting, auto-completion, and similar features.

Creating and Editing Macros for General Use

The Macros button menu on the Home tab of the ribbon provides access to the default macros shipped with VPI Design Suite. It may also contain macros for additionally installed products (toolkits) and user-defined macros.

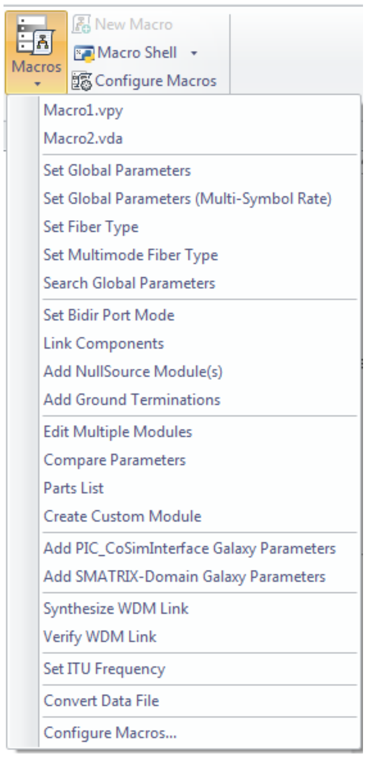
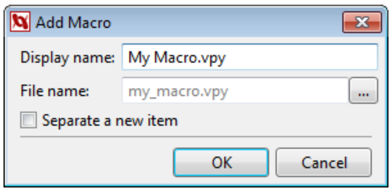


Figure 1-2 *Macros button menu*

To add a new user macro to the Macros menu, click Add in the Macros Manager. To launch the Macros Manager, press Configure Macros on the Home tab of the ribbon or click the Configure Macros menu item in the Macros menu.

The **Add Macro** dialog allows you to specify a file name for the macro file (or a path to an existing macro file) and a display name that should appear in the **Macros** menu. The language dialect can be selected by specifying a proper file extension (`.vda` for Tcl and `.vpy` for Python).



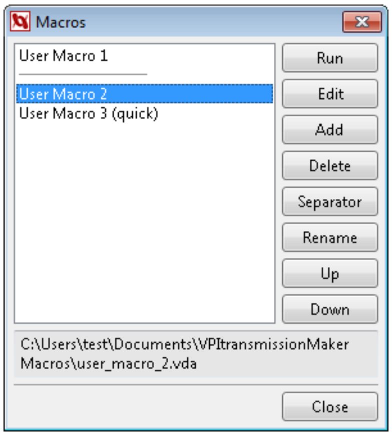
Display name: My Macro.vpy
File name: my_macro.vpy
Separate a new item

Figure 1-3 *Add Macro dialog*

The newly added macro will open for editing in Notepad or in the editor specified in the application preferences.

Customizing the Macros Button Menu

You can easily organize your macros in the **Macros** menu via the Macros Manager. To launch the Macros Manager, press **Configure Macros** on the **Home** tab of the ribbon or click **Configure Macros** in the **Macros** menu.



User Macro 1
User Macro 2
User Macro 3 (quick)

C:\Users\test\Documents\VPItransmissionMaker Macros\user_macro_2.vda

Figure 1-4 *Launching the Macros Manager*

The Macros Manager allows you to reorganize your macros by moving them up and down in the **Macros** menu and by adding separators between them for better usage experience, as well as to run, edit or rename macros, and add new or remove existing macros (see Figure 1-4).

Prefixing any character in the macro display name with a caret ('^') will cause that character to act as a menu shortcut character, allowing the macro to be run by typing `ALT+H,M,M` followed by your menu shortcut character.

To add the caret prefix to your macro display name, in the Macros Manager, click **Add** when creating a new macro or click **Rename** if the macro already exists.

All your macros will be stored in the `VPIdesignSuite Macros` subfolder of the `My Documents` directory. This directory may contain special files: `macros.xml` and `macros.txt`. These files are macros menu configuration files for different versions of VPIphotonics Design Suite™. Please don't remove them not to loose your macros in the **Macros** menu.

Customizing the Built-in Macros

You can customize any of the standard macros as follows:

1. Locate the desired macro in the installation directory (typically `C:\Program Files\VPI\VPIdesignSuite 11.5\macros`) via the `macros.xml` file.
2. Open the Macros Manager and press **Add**. To launch the Macros Manager, press the **Configure Macros** button on the **Home** tab or click the **Configure Macros** menu item in the **Macros** menu located on the same ribbon tab.
3. In the **Add Macro** dialog, click the browse button and choose the macro file located at the first step.
4. Specify a display name for the customized version of this macro.
5. In the opened text editor, press **OK** and modify the macro according to your requirements.
6. Press **Save** in the text editor when modification is completed and close the text editor alongside with the Macros Manager.

Note: Some macros can require additional files for work. For example, *Synthesize WDM Link* macro (`WDM_Section.vda`) requires `.par` files. You have to copy these required files manually from the VPI Design Suite installation directory to the `<My Documents>\VPIdesignSuite Macros` directory.

Protecting Macros

If you wish to protect your ideas implemented as macros, you can encrypt them.

VPIphotonics Design Suite™ allows you to encrypt both Tcl macros (`.vda`) and Python macros (`.vpy`). It is also possible to encrypt Python modules (`.py`) so that you can use (import) them in macros without disclosing their logic.

For more information about encrypting files and protecting your intellectual property, refer to Chapter 5 of the *VPIphotonics Design Suite™ Simulation Guide*.

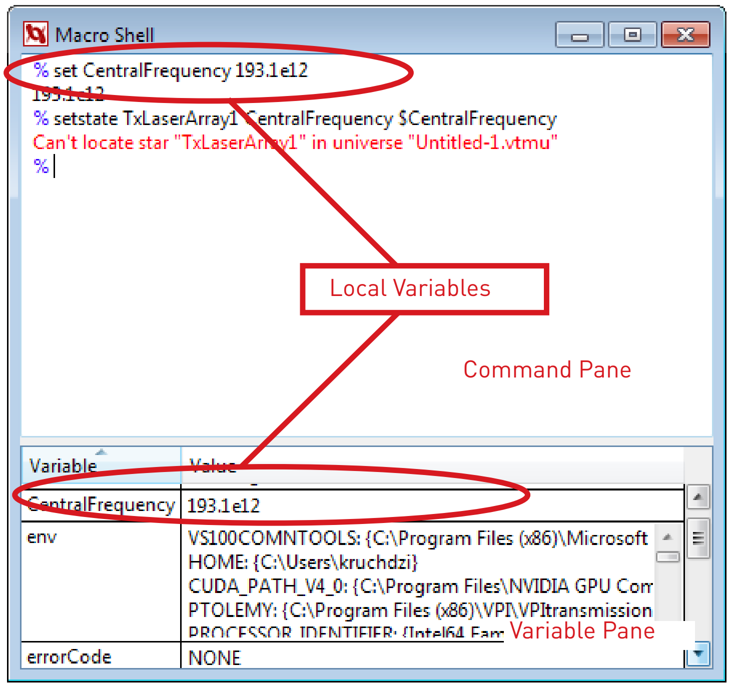
Macro Shell

VPI Design Suite contains two integrated macro shells (one for each PDE Scripting Language dialect) that allow you to run commands in interactive mode so the results appear immediately. The primary purpose of the shells is to simplify the development and debugging of macros that control the work of the schematic.

Tcl Macro Shell

The Tcl macro shell window consists of two parts:

- The *command pane* (top portion) and
- The *variable pane* (bottom portion), as shown in Figure 1-5.



```
% set CentralFrequency 193.1e12
193.1e12
% setstate TxLaserArray1 CentralFrequency $CentralFrequency
Can't locate star "TxLaserArray1" in universe "Untitled-1.vtmu"
%
```

Variable	Value
CentralFrequency	193.1e12
env	VS100COMNTOOLS: {C:\Program Files (x86)\Microsoft HOME: {C:\Users\kruchdzi} CUDA_PATH_V4_0: {C:\Program Files\NVIDIA GPU Com PTOLEMY: {C:\Program Files (x86)\VPI\VPItransmission PROCESSOR_IDENTIFIER: {Intel64 Fam
errorCode	NONE

Figure 1-5 *Tcl macro shell window*

To process a command, type it in the command pane and press `ENTER`. If the command is complete according to the syntax of the Tcl language, it is executed by the internal Tcl interpreter immediately. The results are shown immediately in the command pane.

If the specified command contains errors, an error message is issued in red. (Figure 1-5 shows an example of an attempt to set the value of a nonexistent module parameter.) If the command performs actions with the schematic, for example, modifies schematic parameters, this schematic changes immediately. For example, in order to set the **EmissionFrequency** parameter of the *Tx_OOK* transmitter Tx_OOK_vtmg1, the following command must be run:

```
setstate Tx_OOK_vtmg1 EmissionFrequency 198.5e12
```

The commands can be either typed manually or copied using Windows standard keyboard shortcuts such as CTRL+C and CTRL+V. The shell window "remembers" the command history so you can access previously entered commands using the up arrow and down arrow on the keyboard.

The variable pane contains a table with the variables stored in memory. In order to copy their values (or part of a value), select the necessary value and press CTRL+C. The variables are sorted in alphabetical order (by default) and can be sorted in reverse by clicking the column header.

The Macro Shell window displays the content of *Tcl* arrays, while each array element (a key-value pair) is displayed as

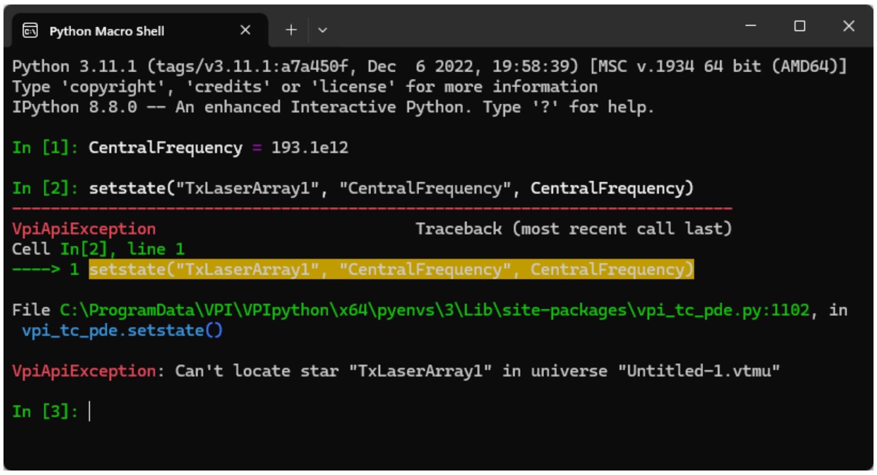
```
key: {value}
```

The text for macros and simulation scripts can be edited in other code editors such as Visual Studio or Notepad++.

Scripts can be run using the `source` command. For example, the command `source {C:\script1.tcl}` runs the script stored in the file `C:\script1.tcl`. The variables used in this script will be visualized in the variable pane. This feature can be very helpful for debugging.

Python Macro Shell

The macro shell for Python is a console window that works in text mode as shown in Figure 1-6.



```
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec  6 2022, 19:58:39) [MSC v.1934 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: CentralFrequency = 193.1e12

In [2]: setstate("TxLaserArray1", "CentralFrequency", CentralFrequency)
---------------------------------------------------------------------------
VpiApiException                           Traceback (most recent call last)
Cell In[2], line 1
----> 1 setstate("TxLaserArray1", "CentralFrequency", CentralFrequency)

File C:\ProgramData\VPI\VPIpython\x64\pyenvs\3\Lib\site-packages\vpi_tc_pde.py:1102, in
 vpi_tc_pde.setstate()

VpiApiException: Can't locate star "TxLaserArray1" in universe "Untitled-1.vtmu"

In [3]: |
```

Figure 1-6 *Python macro shell*

Commands can be entered after the command prompt (“`In [xxx]:`”) and executed by pressing `ENTER`. Once a command is executed, its result will be printed after the “`Out [xxx]:`” string.

The shell window “remembers” the command history so you can access previously entered commands using the up and down arrows on the keyboard.

To get extended information about the shell, execute the “`?`” or “`%quickref`” commands.

Python Environments

Besides the official Python distribution, there are several other distributions, such as Intel® Distribution for Python or WinPython. Most of them are based on the official distribution but provide an extensive verified set of libraries.

VPI Design Suite is shipped with its own Python distribution: VPIpython. This distribution is isolated from the rest of the Python installations on your machine (the system Python or any third-party distribution). At the same time VPI Design Suite is designed to use either VPIpython or any third-party distributions.

VPIpython consists of the Python core and isolated working environments (further called Python environments). The Python core is immutable and contains standard libraries and suits as a template for Python environments.

A Python environment may be understood as a copy of the core which allows you to install additional libraries. Upon installation, one Python environment is created automatically and the libraries necessary for the functioning of VPI Design Suite are installed into it.

The notion of a Python environments becomes very useful when it is necessary to create several so-called sandboxes (for example, for experimenting with a third-party library). When experiments are over, unnecessary environments may be removed without breaking the whole system.

Multiple Python environments may be added through the application Preferences but only one may be made active. An active environment is used both for macro execution and Python cosimulation.

Adding Python Environments

It is possible to create a standard Python environment or add some third-party distribution as a working Python environment. In both cases Photonic Design Environment attempts to install necessary additional libraries into the new environment.

Creating Python Environments

A Python environment is created as a slightly modified copy of the core. The core libraries are not copied into the new environment but used directly from the core. This saves disk space.

To create a new Python environment:

1. On the File menu, click Preferences.
   The Preferences dialog appears.
2. Select the Python Environments tab.
3. Click the Create button.
   The Create dialog appears.
4. Enter the name of the new environment.
   Selecting a meaningful name helps you to later remember the purpose of the environment.
5. Choose a version of the Python interpreter.
6. If necessary, enter the path to the Python engine DLL (`python311.dll`). It should match the version of the Python interpreter.
   Usually this may be left unaltered to let it be found automatically.
7. If necessary, enter the path to additional libraries which should reside in some common location.
   Multiple locations may be separated with a semicolon.
8. Click OK.
9. Select those products whose packages you want to install.
10. Click OK.
    A new environment is being created which may take some time. During that process, a minimum necessary set of libraries is being installed into the newly created environment.
11. Once the environment is created, it may be made active by clicking in the Active column.

Adding Third-Party Python Distributions

It may happen that a configured Python installation already exists on your system. In this case, it may be added to the list of environments to be used for running macros or cosimulation.

To add an existing Python environment:

1. On the File menu, click Preferences.
   The Preferences dialog appears.
2. Select the Python Environments tab.
3. Click the Add Existing button.
   The Browse For Folder dialog appears.
4. Select the location of the environment to be added and click OK.
   The Add Existing dialog appears.
5. Enter the name of the new environment, the path to the Python engine DLL, and a list of locations with additional libraries if necessary as described above.
6. Click OK.

7. Select those products whose packages you want to install.
8. Click OK.
   The environment is being added. During that process, a selected set of libraries is being installed into the environment.
9. Once the environment is created, it may be made active by clicking in the Active column.

Installing Third-Party Libraries

The true power of Python lies in its tremendous libraries — both standard and third-party. The VPIpython comes with a minimum set of libraries necessary for the application to function (like *NumPy* and *SciPy*), and it is easy to install more.

Via Package File

If a third-party library is available as a source distribution (`.tar.gz` or `.zip`) or a `.whl` package, it is possible to install it through the application Preferences.

To install the library from the package file:

1. On the File menu, click Preferences.
   The Preferences dialog appears.
2. Select the Python Environments tab.
3. Select the target environment by clicking its name.
4. Click the Install Package button.
   The Open dialog appears.
5. Select the desired package file and click Open.
   The package is being installed which may take some time.

Note: Packages may have other packages as dependencies. If you have access to the Internet, these will be downloaded automatically. Otherwise it is necessary to download and install dependencies first.

Using Command Prompt

Many libraries are distributed through the *Python Package Index* available at https://pypi.python.org/pypi. They can be installed using the `pip` utility.

To list installed libraries:

1. On the File menu, click Preferences.
   The Preferences dialog appears.
2. Select the Python Environments tab.
3. Select the target environment by clicking its name.

4. Click the Run Console button.
   The Windows command interpreter window appears.
5. Type `pip list` in the command prompt and hit `ENTER`.
   The list of installed libraries (packages) will be printed.
6. Execute the `pip show <package_name>` command.
   The detailed information about the package is printed.

To search for and install a library:

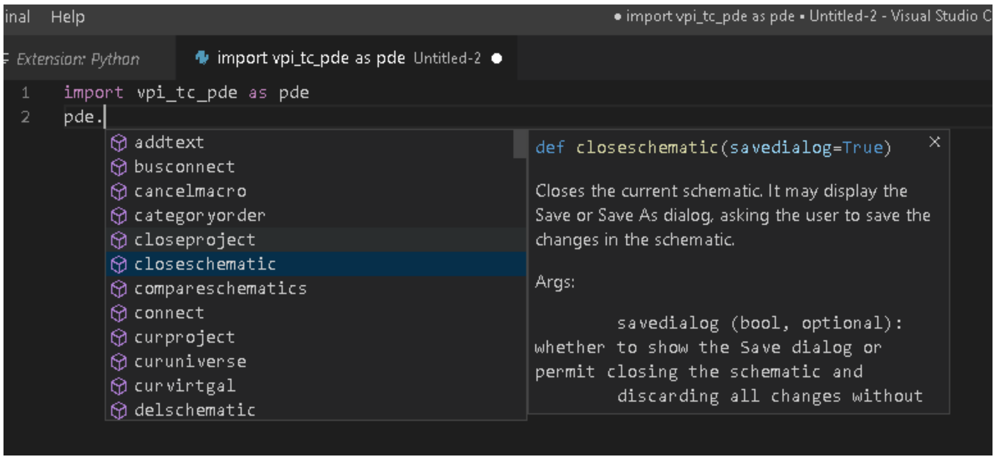
1. Follow steps 1-4 described above.
2. Execute the `pip search <query>` command.
   The list of packages whose name or summary contains `<query>` will be printed.
3. Execute the `pip install <package_name>` command.
   The package and its dependencies are being downloaded and installed.

To uninstall a library:

1. Run the console as described above.
2. Execute the `pip uninstall <package_name>` command.
   The package is being uninstalled. Note that the dependencies installed during the package installation are not uninstalled automatically.

Accessing vpi_tc_pde and vpi_tc_ptcl Documentation Stubs

Documentation stubs allow autocomplete features provided by many Python development environments (PyCharm, Visual Studio, Visual Studio Code, etc.) when using `vpi_tc_ptcl` and `vpi_tc_pde` modules. The stubs are automatically deployed into the Python environments created with the Photonics Design Environment (see Figure 1-7).


inal
Help
● import vpi_tc_pde as pde ● Untitled-2 - Visual Studio C
Extension: Python
import vpi_tc_pde as pde Untitled-2 ●

```
import vpi_tc_pde as pde
pde.
```

```
addtext
busconnect
cancelmacro
categoryorder
closeproject
closeschematic
compareschematics
connect
curproject
curuniverse
curvirtgal
delschematic
```

```
def closeschematic(savedialog=True)

Closes the current schematic. It may display the
Save or Save As dialog, asking the user to save the
changes in the schematic.

Args:

        savedialog (bool, optional):
whether to show the Save dialog or
permit closing the schematic and
        discarding all changes without
```

Figure 1-7 *Displaying documentation in Visual Studio Code*

To enable this feature, `python.exe` from the active Python environment must be selected as Python interpreter in your IDE.

1. On the **File** menu, click **Preferences**.
   The **Preferences** dialog appears.
2. Select the **Python Environments** tab.
3. In the **Environments** list, select the line marked as **Active**.
4. Observe the **Root Path** field in the **Details** section. **python.exe** is located in the **Scripts** subfolder of this path.

Debugging Python Code

When developing Python macros, it is frequently necessary to debug Python code to spot and fix errors of the algorithm. Several approaches that prove helpful when analyzing and debugging Python code are described in the *Debugging of Python Scripts* white paper. This white paper shows how to debug Python scripts inside VPI Design Suite. The paper summarizes some available techniques for spotting algorithmic errors in custom Python code, be it cosimulation, a function used in parameter expressions or simulation scripts, or macros. Please contact support@VPIphotonics.com to get the white paper.

Web Access to Macro API

Besides Tcl and Python macros, the PDE scripting is also available from another computer, using a special API based on Web technology (HTTP protocol), according to a principle presented in Figure 1-8. The PDE scripting interface can be invoked from any modern programming language, the only requirement being that the calling program is able to send GET and POST requests [3] via HTTP.

Applications of the Web access include exporting and running simulations from software that is installed on non-Windows systems.

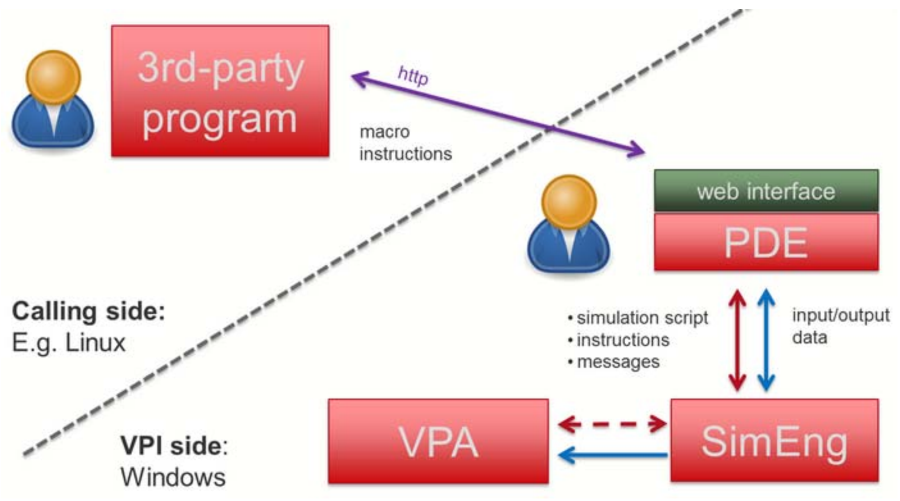


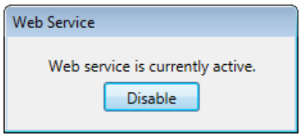
Figure 1-8 *Principle of the Web access to PDE API*

According to the terminology used to describe Web applications, the computer and/or software that uses HTTP requests to access PDE functionality is called the *client*; PDE and the computer it is installed into is a *server*, and the PDE component responsible for handling the requests is referred to as a *service*.

Controlling the Web Service

To start the Web access (service) in the Photonic Design Environment, click **Enable Web Service** on the **Tools** > **Server** ribbon.

Enabling the Web service temporarily blocks the user interface of PDE, as shown in Figure 1-9.


Web Service

Web service is currently active.

Figure 1-9 *Blocking of user actions when using Web service*

The aim of this blocking is to prevent accidental data loss or errors if the local user changes the PDE state, for example, closes or modifies the current schematic, disrupting the logic of the calling side.

Clicking **Disable** in the above dialog stops the remote access and unblocks the PDE user interface.

The Web service can also be controlled programmatically, including switching it off using an HTTP request (see "enablewebservice" on page 350).

For correct work of the Web service, it is necessary that the computer running the Photonic Design Environment should be visible to other computers in the network. In many cases, for example, if the computer is part of the Windows domain, the network discovery is already switched on. Otherwise, it is necessary to change the Windows settings (administrative privileges are required):

- Windows 10: Settings > Network & Internet > Ethernet.
- Windows 8.1 and earlier: Control Panel > Network and Sharing Center > Advanced sharing settings.

Caution: Besides the simple blocking described above, the Web service feature currently provides no client authentication or other security options.

Web access is expected to be used in a secure environment, such as a corporate network behind a suitable firewall.

To enable the Web service, the Photonic Design Environment needs an additional license to be present on the same VPIlicenseServer that is used by the whole product.

Principles of the Web Service

The functionality of the Web access to PDE is based on the following approach:

- The client computer calls the server using an HTTP *request*, with a special encoding of the API parameters: schematic path, parameter values and so on, as described in "Request Format" below.
- The result is returned from the server to the client using an HTTP response, again using a predefined encoding of return values and data structures, as described in "Response Format".
- In addition to the Macro API, the Web service supports exchange of files with the remote computer, as further discussed in "Transfer of Files to and from the Remote Party".

Request Format

According to the internet standards (see, for example, [4]), a simple HTTP request looks like an address, or a Web page, or a search request. For instance, to set a parameter value of a module in a current schematic, the following request can be used:

```
http://vpi.example.com:8475/pde/v1/setstate?inst=laser1&value=3.7&par
      am=AveragePower
```

This request consists of the following parts:

- `http://vpi.example.com` is the protocol (or 'scheme') and the server name. The server name is completely dependent on local settings and can be either
    - a full name (like in this example) of the computer, containing its name and the domain,

  + a (short) computer name, like `vpi_host` or `localhost` (the latter can be used to have both client and server on the same physical computer), or
  + an IP address, like `192.164.0.123`.
- `:8475` is the number of the port on which the PDE is listening for requests. The default port number `8475` is registered in Windows Firewall settings upon VPIphotonics Design Suite™ installation. If this port should be used by some other program, please contact VPIphotonics support for further details.
- `/pde/v1` is the (virtual) path on the server that is used to separate different services and to provide a way for future extensions.
- `setstate` is technically part of the path, and is used to specify a particular API call.
- The (optional) part after the question mark, `?`, `inst=laser1&value=3.7&param=AveragePower`, specifies the parameters of the request. It consists of one or several `parameter=value` pairs, delimited by the ampersand symbol, `&`.

In the above example, all values sent to the API call can be used 'as is'. However, many symbols (such as spaces, colons, or slashes) are not allowed in Web requests or have special meaning in the HTTP request syntax. For such symbols, a special 'percent encoding' should be used [4].

The principle of percent-encoding is that a disallowed symbol is replaced by the percent symbol, `%`, followed by the original symbol code.

For example, a request to place a module onto a schematic should contain the module type identifier, or URN (see "Identifying Packages using a URN" on page 31). To illustrate, for the *LaserCW* module with the following URN

```
URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:
```

the request will look like

```
http://vpi.example.com:8475/pde/v1/star?inst=laser1&module=URN%3AVPI_
    LIB%3A%3ATC+Modules%5COptical+Sources%5CLaserCW.vtms%3A
```

In some API calls, such as `selectstars`, or `waitrun`, the parameter values should be comma-delimited, with commas encoded as `%2C` in the actual request, like

```
http://vpi.example.com:8475/pde/v1/selectstars?args=laser1%2Cground2%
    2CFork2
```

to select modules with IDs `laser1`, `ground2`, and `Fork2`.

Note: Unlike the Tcl and Python macros, Web access supports only named parameters in requests, and the order of parameters is completely arbitrary.

Errors and Status Codes

Depending on if the request was successful or not, the PDE Web service can return one of the following error or status codes [3]:

- `200`: The 'success' code, meaning the response will contain useful data (if the API call returns it).
- `400`: Denotes a 'bad request', or an expected error due to incorrect parameters or context of the API call. A more detailed error message is returned together with the status code.
- `404`: Means the command is not found, usually returned if the Web API command contains a misprint.
- `429`: Returned if Web service cannot process a request right now. The `Retry-After` header of the response contains time (in seconds) that the client should pause before resubmitting the request.
- `503`: The 'service unavailable' status. Usually returned if the Web service is not started or after switching it off using the `enablewebservice?enable=false` call.

Response Format

The response from the PDE is returned in the JSON format [5], [6] (stands for JavaScript Object Notation). Nowadays, this format is widely used beyond the JavaScript language and is a standard de facto in many modern Web applications.

In short, the JSON format may transfer the following data:

Objects	Hash-table, represented as `{key1: value1, key2: value2, ...}`, where `keyN` is a string name of a structure part and a `valueN` is any element of the JSON format, including another object.
Arrays	Square-bracket enclosed sequence of any JSON values, like `[value1, value2, ...]`.
Strings	Sequence of arbitrary symbols.
Numbers	Numbers in string representation using base-10 representation. Floating-point values (including scientific or exponential notation) and integer numbers are supported.
Special values	JSON supports three special values or 'literal tokens': `true`, `false`, and `null`.

Finally, it is worth mentioning that manual processing of both requests and responses or writing a specialized function to handle these is rather impractical. There exist specialized libraries for almost any modern programming language enabling to handle such requests in a transparent manner. With such a library, only PDE-specific logic should be implemented.

Transfer of Files to and from the Remote Party

As in the Web access VPIphotonics Design Suite™ works on a different computer than the client code, the problem of transferring input and output files appears.

On the one hand, file transfer can be based on using shared drives on a computer with Web access. However, this approach is not always convenient and requires additional administrative efforts.

As a more straightforward alternative, Web access implements three API calls for processing input and output files:

- `download`: transfers the specified file from server to client.
- `upload`: transfers the specified file from server to client; unlike all other Web access commands, POST request is used instead of GET.
- `delete`: moves to Recycle Bin the specified file or directory on the server.

The files are transmitted using chunked transfer encoding, as further described in the Internet standards, for example, [7].

These API calls use a virtual path different from the main API, for example:

```
http://vpi.example.com:8475/pde/v1/file/download
```

For security, the file transfer API calls are limited to only files in standard folders of the current schematic (`Resources`, `Inputs`, `Outputs`, `Attachments`, or `Reports`).

Currently, files up 200 Megabytes in size can be transmitted using Web service.

References

[3] R. Fielding, and J. Reschke, Eds., "*Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*", *RFC 7231*, June 2014, https://tools.ietf.org/html/rfc7231.

[4] T. Berners-Lee, R. Fielding, and L. Masinter, "*Uniform Resource Identifier (URI): Generic Syntax*", *STD 66, RFC 3986*, January 2005, https://tools.ietf.org/html/rfc3986.

[5] Ecma International, "*The JSON Data Interchange Format*", *Standard ECMA-404*, October 2013, http://www.ecma-international.org/publications/standards/Ecma-404.htm.

[6] T. Bray, Ed., "*The JavaScript Object Notation (JSON) Data Interchange Format*", *RFC 7159*, March 2014, https://tools.ietf.org/html/rfc7159.

[7] R. Fielding, and J. Reschke, Eds., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014, https://tools.ietf.org/html/rfc7230.

Introduction to Macro Commands by Function

The PDE Scripting Language commands can be grouped by function:

- opening, saving, exporting, sizing, redrawing, clearing, closing, deleting or comparing a schematic (Universe or Galaxy)

  ```
  closeschematic, delschematic, exportschematic, newuniverse,
     openschematic, redraw, reset, saveschematic, setview, zoomall,
     compareschematics.
  ```

- running a schematic (Universe)

`usesimscript, run, stopsimulation, waitrun.`

- selecting, placing, sizing, rotating and deleting modules (Star or Galaxy), virtual galaxies and text

  `addtext, delstar, deselectall, deselectstars, mirrorhoriz, mirrorvert, portplace, rotate, selectstars, setbidirmode, setbitmap, setbounds, setpos, star, starplace, virtgal, setvirtgal.`

- linking modules with buses or wires, controlling visual link path

  `busconnect, connect, disconnect, linkroute.`

- creating and setting parameters

  `newstate, setstate.`

- parameter and module sweep commands

  `newmodulesweep, sweep, sweeprule.`

- interrogating a schematic (parameters, topology)

  `evaluateparams, getbidirmode, getbidirportnames, getblocks, getbounds, getconnectedblocks, getconnection, getcontenttype, getmaster, getportdir, getports, getsignaltypes, getstates, getterminals, getvirtgal, getxpos, getypos, isconnected, isportvisible, isschematicmodified, isvirtgal, packagefolder, parseschematic, portplace, selectedstars, statevalue.`

- communicating with the user

  `messagedialog, modify, querydialog, selectfiledialog, wizard, enablewaitdlg, writemessage.`

- an automation controller for external applications that support COM Automation (such as Microsoft Office)

  `createobject, releaseobject, putproperty, getproperty, invoke.`

- miscellaneous utility and configuration commands

  `categoryorder, curuniverse, curvirtgal, domain, prodset, getpythonexepath, uniquename.`

In addition, the standard Tcl or Python language functions are supported. These enable program control, mathematical manipulation, error handling, and file input/output.

A complete description of the scripting language commands is provided in Appendix A, “PDE Scripting Language Reference”. The remainder of this section provides examples of the use of the language.

Building a Schematic

Macros can be used to build a simulation from scratch, or to add components and connections to an existing schematic. The first action is to open a new or previous schematic onto which modules are placed and connected.

Identifying Packages using a URN

A number of macro commands used to build schematics require you to specify the schematic or module that you wish to work with. For example, to work with an existing design, you must specify the schematic that you wish to open. To place a module on a schematic, you must specify which module you want to add.

All packages are identified using a uniform resource name (URN). A URN is a string with five pieces of data. Each piece of data is separated by a colon. The five pieces of data are:

`"URN"`:*`NID`*:*`NID_VERSION`*:*`NSP`*:*`NSP_VERSION`*

- URN — prefix
- *`NID`* — namespace identifier (either `VPI_LIB`, `VPI_DEMO`, `TKIT_LIB`, `TKIT_DEMO`, `USER_LIB`, `CHILD` or `LOCAL`). The first four represent a path, which depends on the installation location. The `USER_LIB` represents all user libraries independently of their real location. The `CHILD` NID represents the path to the parent package of the resource in question. The `LOCAL` is used to represent the full path when neither of other NIDs suite.
- *`NID_VERSION`* — version of the namespace identifier *(usually empty)*
- *`NSP`* — namespace path. The path to a package, relative to the NID.
- *`NSP_VERSION`* — version of the namespace path *(usually empty)*.

VPI Design Suite resolves a URN to a file system path by concatenating the paths from the *`NID`* and the *`NSP`*.

Note: Virtual galaxies don't have URNs.

For example:

The `BER Estimator Comparison I` package has the following URN and resource:

“`URN:VPI_DEMO::OS\Simulation Techniques\BER Estimation\BER Estimator Comparison I.vtmu:`” and “`URN:CHILD::TxExtModLaserRZ.vtmg:`”.

If `VPI_DEMO` is stored as `C:\Program Files\VPI\VPIdesignSuite 11.5 \tooldata\demos`, the resulting path for the package and resource would be:

```
C:\Program Files\VPI\VPIdesignSuite 11.5\tooldata\demos\OS\
   Simulation Techniques\BER Estimation\BER Estimator
   Comparison I.vtmu
```

and

```
C:\Program Files\VPI\VPIdesignSuite 11.5\tooldata\demos\OS
   \Simulation Techniques\BER Estimation\BER Estimator
   Comparison I.vtmu_pack\Resources\TxExtModLaserRZ.vtmg
```

Creating a New Schematic

For a **new** universe, the `newuniverse` command creates a new universe with the given name, domain, and product set (optional). If the command runs successfully, it returns the URN of the created schematic.

A **unique name** can be generated using the `uniquename` command, which returns a name in the form of “UNTITLED-1,2,3 etc”, which is unique to the current session.

The following command is equivalent to the default behavior of the `newuniverse` command with no parameters.

Tcl:

```
newuniverse [uniquename] Auto -prodset “OS,FO”
```

Python:

```
newuniverse(uniquename(), "Auto", “OS,FO”)
```

Parameters can be added to a schematic (galaxy, universe) or virtual galaxy using the `newstate` command. The Simulation Domain can be changed using the `domain` command. To change the product set of the schematic you can use the **prodset** command.

Working with an Existing Schematic

To work on an **existing schematic**, the `openschematic` command opens the schematic with the given path or URN.

If the schematic you wish to work on is one of a number of schematics that are already open, you can **select** the specific open universe that you want commands to apply to by using the `curuniverse` command.

If called without an argument, the `curuniverse` command **returns** the URN of the current schematic. With the latter syntax, the command is quite frequently combined with the `packagefolder` command to get access to the files attached to a schematic.

Tcl:

```
set path "[packagefolder [curuniverse]]\\Attachments\\Message.doc"
```

Python:

```
path = packagefolder(curuniverse()) + "\\Attachments\\Message.doc"
```

Note: It is better to use `file join` in Tcl and `os.path.join()` in Python for constructing paths from separate pieces.

To clear an open universe of all modules, the `reset` command erases all contents of the current schematic.

To close the current schematic (universe or galaxy), use the command `closeschematic`. It may ask for a Save As if it has not been previously saved, or if it is read-only. The command has an optional flag that is used to suppress the dialog, if it is not necessary to save the changes.

To delete an existing schematic (or galaxy), use the `delschematic` command. You will be prompted to confirm the deletion. The command additionally closes the opened schematic before deletion.

Note: When a hardcoded string is used to open, select or delete a schematic (i.e., not the content of some *Tcl* variable), enclose the name or URN in braces, for example: `openschematic {c:\My Projects\design.vtmu}`.

To compare two open schematics (or galaxies), use the `compareschematics` command. The command returns `1` (`True`) when schematics are identical and `0` (`False`) otherwise. Different schematic components (profiles, global parameters, modules, module parameters, and connections) can be selected for comparison by specifying the corresponding options. The comparison result can be saved to a file.

Selecting and Placing Modules

Modules, Types and Instances

- A *Module* is either a galaxy module, a star module or a module sweep. It is a prototype for module instance.
  + A Module Name is the basic generic name of a module as it appears in the library, for example *LaserSM_RE*.
  + A Module Path uniquely identifies a module stored in VPI Design Suite. It is either a URN or a unique (absolute) path.
- A *Module Instance* is an instance of a module on a schematic. For example, a *LaserSM_RE* module may appear several times and each instance of it may have different parameters.
- A *Virtual Galaxy* is an entity which groups module instances on a schematic and has parameters specific to galaxies. In contrast to regular galaxies, virtual galaxies don’t exist on their own and cannot be reused (there is only one instance of each virtual galaxy).

- A *Type* is the type of a schematic or module: universe, galaxy, virtual galaxy or star.
- An *Instance ID* is a unique name given to a module or virtual galaxy every time it appears on a schematic. Each time it requires a different instance ID, because it will be representing a different physical laser. When placing modules using the PDE, the instance ID is generated from the module name by default (for example, `LaserSM_RE_vtms1`, `LaserSM_RE_vtms2`), but it can be changed using the parameter editor. When placing modules using the PDE Scripting Language, the instance ID **must** be specified when the module instance is created.

Selecting The Target Context

Many functions that work with instances, parameters, or text blocks depend on the current context (*schematic* or *virtual galaxy*). For example, the `star` command adds a module instance to the current virtual galaxy of the current schematic. If no virtual galaxy is currently designated as the target context, the module is added to the main topology. The same approach is used by the `addtext` command.

To designate a schematic and its main topology for use as the current context, use the `curuniverse` command and specify the target as an argument. To designate a virtual galaxy as the current context in the current schematic, use the `curvirtgal` command and specify the target as an argument. To reset the current context back to the main topology, use the `curvirtgal` command with the `-reset` option.

Specifying a Position

Positions are specified from the top left corner of the schematic (or virtual galaxy if it is now made current). Each ruled square of the schematic grid is 2 units by 2 units. Positions are quantized to the nearest unit. An icon is usually 4 units by 4 units. The reference point of the instance icon is usually in the center.

Adding, Placing and Deleting a Module Instance

A module can be selected from the resources and placed on the schematic and deleted using the following commands:

- `star` creates a new instance from a generic module in the current schematic or virtual galaxy. It also allows placing an instance in a desired position during creation.
- `starplace` places an instance described by its instance ID in a desired position with a desired rotation and mirror.
- `setpos` places an instance described by its instance ID at position (*xpos*, *ypos*).
- `setvirtgal` assigns an instance to a virtual galaxy or the main schematic topology.
- `delstar` deletes a module instance from the current schematic.

Note: If you know the desired location for a new instance before creating it, the `star` command with the placement specified is a preferred way to proceed.
Before creating a new instance, you can perform some analysis of its visual size and ports via the `getbounds`, `getportdir`, `getports`, and `getterminals` commands.

Adding, Placing and Deleting a Virtual Galaxy

A virtual galaxy can be added, moved and deleted using the following commands:

- `virtgal` creates a new virtual galaxy in the current schematic.
- `setpos` places the virtual galaxy at the specified position
  (the reference point of the virtual galaxy is the left top corner).
- `starplace` places the virtual galaxy at the specified position
  (the reference point of the virtual galaxy is the left top corner).
- `setbounds` changes the bounds of the virtual galaxy
  (can be used to move and/or change size of the virtual galaxy).
- `delstar` deletes a virtual galaxy from the current schematic.

Note: If you know the desired location and size of the virtual galaxy before creating it, use this information as the arguments of the `virtgal` command.

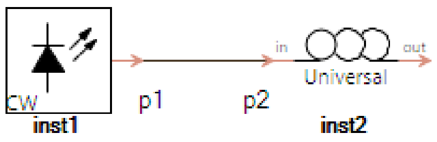
Working with Parameters

- `setstate` is used to set the values and expression type of instance or virtual galaxy parameters.
- `newstate` creates a new schematic (global) parameter of the current schematic (universe or galaxy) or parameter of a virtual galaxy, or modifies an existing parameter if the name already exists.
- `statevalue` finds the values (and other attributes) of parameters of an instance on the current schematic, or of global parameters of the current schematic or virtual galaxy parameters if the instance specification is omitted.
- `evaluateparams` calculates the actual values of parameters by evaluating expressions specified for the parameters.
- `domain` sets the simulation domain (`SDF`, `BDF`, ...) of a schematic (universe or galaxy) or a virtual galaxy. If the name is omitted, the command returns the current domain.

Connecting Modules

Module instances can be interconnected by `wire`, `bus` or `labeled` type of links. There are separate commands for these:

- `connect` creates a wire or labeled link between two instances.



Some of the modules, such as (de)multiplexers *BusCreate* and *BusSplit*, have multiport input and/or output ports. To ensure proper connection order, the module's input or output can be specified in the `connect` command using the `"port_name#terminal_index"` notation. The list of available terminals can be found using the `getterminals` command.

- `busconnect` creates a bus connection between two multiports of instances. A bus is a collection of number of parallel independent connections between the port terminals.
- The existence of a connection can be checked by using the command `isconnected`, which can also check if the given port (or terminal) is currently connected. It returns `1` (`True`) if the port is connected and `0` (`False`) otherwise.
- `disconnect` disconnects one end of a wire from the specified port of an instance.
- A desired visual link path can be specified as an option of the `connect` and `busconnect` commands. It can be retrieved or altered via the `linkroute` command.

Some of the modules, such as *WgStraight* or *BusMergeBi*, have a special type of ports: bidirectional. A bidirectional port is actually a group of ports some of which may be visible or hidden depending on the current port representation (bidirectional, only input, only output or output and input). Special commands exist to switch between modes or query of the port configuration:

- `setbidirmode` sets the mode of the bidirectional port group.
- `getbidirmode` queries current configuration of the bidirectional port group.
- `getbidirportnames` returns the names of all ports which belong to one bidirectional port group.
- `isportvisible` queries the state of some port in the bidirectional port group.

Creating a Universe, Placing Modules, Linking Instances—Example

Note: This example is the *PlacingComponents* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial*.

`PlacingComponents` macro creates the following schematic. Note that a delay is used in the program to slow down the drawing process so that you can see the components being placed. Without the delay, the created schematic would appear almost instantaneously. The final schematic is presented in Figure 1-10.

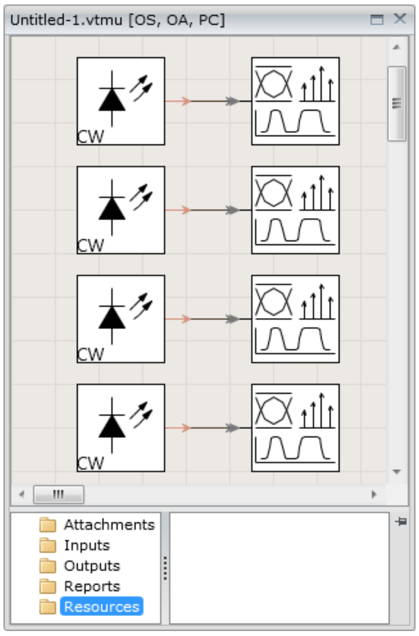


Figure 1-10 *Automatic placement of the components on a schematic*

In Tcl:

```
# Description: Example of how to place components
#
# Procedure to place and connect components
proc placeComponents {startIndex count} {
	global distX distY posX posY minBoundX minBoundY maxBoundX maxBoundY
	# reset bounds
	set minBoundX 1000
	set minBoundY 1000
	set maxBoundX 0
	set maxBoundY 0

	# loop to create instances
	for { set i $startIndex } { $i < [expr $startIndex + $count] } { incr i } {
		# create instance of given LaserCW and place it
		star Laser$i {URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:}
		setpos Laser$i $posX $posY

		# get the icon's bounds as list of four elements { x1 y1 x2 y2 }
		set laserBounds [getbounds Laser$i]
		# calculate height
		set laserHeight [expr int([lindex $laserBounds 3] - [lindex $laserBounds
1] + 0.5)]

		# create instance of SignalAnalyzer and place it
		star SignalAnalyzer$i {URN:VPI_LIB::TC
Modules\Analyzers\SignalAnalyzer.vtms:}
		set visBounds [getbounds SignalAnalyzer$i]
```

```
             set visWidth  [expr int([lindex $visBounds 2] - [lindex $visBounds 0] +
0.5)]
             setpos SignalAnalyzer$i [expr $posX + $distX + $visWidth]  $posY

             # connect laser and visualizer
             connect Laser$i output SignalAnalyzer$i input

             # requery to reflect new position
             set laserBounds [getbounds Laser$i]
             set visBounds [getbounds SignalAnalyzer$i]
             # calculate min bounds
             set minBoundX [expr min([lindex $laserBounds 0], $minBoundX)]
             set minBoundY [expr min([lindex $laserBounds 1], $minBoundY)]
             # calculate max bounds
             set maxBoundX [expr max([lindex $visBounds 2], $maxBoundX)]
             set maxBoundY [expr max([lindex $visBounds 3], $maxBoundY)]

             # calculate next y-position
             set posY [expr $posY + $laserHeight + $distY]

             # wait for a half second betwen placements (animation)
             after 500
             redraw
      }
      return "$i"
}

# create new universe
newuniverse

# initialize position of modules and distance between them
set posX     5
set posY     5
set distX    2
set distY    1

# number of components to place
set number 4

# first place components to the main topology
set nextIndex [placeComponents 1 $number]
# calculate virtual galaxy bounds from results of recent placement
set vgX [expr $posX - 2 - 1]
set vgY [expr $posY - 2]
set vgWidth [expr round($maxBoundX - $minBoundX + 2)]
set vgHeight [expr round($maxBoundY - $minBoundY + 2)]

# create virtual galaxy
virtgal VirtualGalaxy1 $vgX $vgY $vgWidth $vgHeight

# initialize position of modules in virtual galaxy
set posX     3
set posY     3

# place some items into virtual galaxy
# (designated as current context after creation)
```

```
placeComponents $nextIndex $number

# make all modules visible
zoomall
```

In Python:

```
# Description:  Example of how to place components

import time
import vpi_tc_pde as pde

# Procedure to place and connect components
def place_components(startIndex, count, posX, distX, distY):
      global minBoundX, minBoundY, maxBoundX, maxBoundY, posY

      minBoundX = 1000
      minBoundY = 1000
      maxBoundX = 0
      maxBoundY = 0

      index = startIndex
      # loop to create instances
      while (index < startIndex+count):
            # create instance of given LaserCW and place it
            laserCW = "Laser{}".format(index)
            pde.star(laserCW, "URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:")
            pde.setpos(laserCW, posX, posY)

            # get the icon's bounds as list of four elements { x1 y1 x2 y2 }
            laserBounds = pde.getbounds(laserCW)

            # calculate height
            laserHeight = int(laserBounds[3] - laserBounds[1] + 0.5)

            # create instance of SignalAnalyzer and place it
            signalAnalyzer = "SignalAnalyzer{}".format(index)
            pde.star(signalAnalyzer, "URN:VPI_LIB::TC
Modules\Analyzers\SignalAnalyzer.vtms:")
            visBounds = pde.getbounds(signalAnalyzer)
            visWidth = int(visBounds[2] - visBounds[0] + 0.5)
            pde.setpos(signalAnalyzer, posX + distX + visWidth, posY)

            # connect laser and visualizer
            pde.connect(laserCW, "output", signalAnalyzer, "input")

            # requery to reflect new position
            laserBounds = pde.getbounds(laserCW)
            visBounds = pde.getbounds(signalAnalyzer)

            # calculate min bounds
            minBoundX = min(laserBounds[0], minBoundX)
            minBoundY = min(laserBounds[1], minBoundY)

            # calculate max bounds
            maxBoundX = max(visBounds[2], maxBoundX)
            maxBoundY = max(visBounds[3], maxBoundY)

            # calculate next y-position
            posY = posY + laserHeight + distY

            # wait for a half second betwen placements (animation)
            time.sleep(0.5)
            pde.redraw()
            index += 1
      return index

# create new universe
pde.newuniverse()

# initialize position of modules and distance between them
posX =  5
posY =  5
distX = 2
distY = 1

# number of components to place
number = 4
# first place components to the main topology
nextIndex = place_components(1, number, posX, distX, distY)

# calculate virtual galaxy bounds from results of recent placement
vgX = posX - 2 - 1
vgY = posY - 2
vgWidth = int(round(maxBoundX - minBoundX + 2))
vgHeight = int(round(maxBoundY - minBoundY + 2))

# create virtual galaxy
pde.virtgal("VirtualGalaxy1", vgX, vgY, vgWidth, vgHeight)

# initialize position of modules in virtual galaxy
posX = 3
posY = 3

# place some items into virtual galaxy
# just after creation it becomes current
# and all components will be added to it
place_components(nextIndex, number, posX, distX, distY)

# make all modules visible
pde.zoomall()
```

Adding Text to a Schematic

- `addtext` adds text to a schematic at the specified position. Optionally, the font, font size, color, and alignment of the text; and size, background and border color of the text box can be set in the command.

Selecting Modules on a Schematic

These commands select, deselect, and identify chosen instances, just as clicking on them with a mouse would do. Selected instances are usually grayed out.

- `selectstars` selects all instances on the schematic given as arguments.
- `deselectall` deselects all selected components in the current schematic.
- `deselectstars` deselects only the instances that are passed as parameters to it.
- `selectedstars` returns a list with the instance IDs of all currently selected stars.

Resizing and Redrawing a Schematic

- `setview` sets the size and position of the schematic visible area (viewport) to the given coordinates.
- `zoomall` sizes the zoom of the viewport so that all blocks are visible.
- `redraw` repaints the schematic, in case there are unwanted glitches in the graphics.

Repositioning an Instance

- `mirrorhoriz` horizontally inverts an instance (with respect to the **vertical** axis, similar to the Mirror Y button in the Photonic Design Environment).
- `mirrorvert` vertically inverts an instance (with respect to the **horizontal** axis, similar to the Mirror X button in the Photonic Design Environment).
- `rotate` rotates the icon of a given instance clockwise by 90 (default), 180, or 270 degrees.
- `setpos` command places an instance (or a virtual galaxy) in a desired position.
- `starplace` allows placing an instance in a desired position with a desired rotation and mirror.

Note: If you want to change a position and rotation and/or mirror of an instance, `starplace` is preferred.

Setting a Bitmap for a Module

- `setbitmap` changes the icon of the current schematic to the bitmap image in the supplied file. The schematic cannot be read-only.

Saving a Schematic

- `saveschematic` saves the current schematic. If a name (or URN) is given the current schematic is saved under that name.

Exporting a Schematic

- `exportschematic` exports the current schematic using specified format.

Running a Schematic

- `usesimscript` enables or disables use of a user-defined simulation script for the subsequent runs.
- `run` runs the simulation. The `run` command returns the unique ID of a simulation. It can be used later in the `waitrun` command.
- `stopsimulation` terminates the specified simulation.
- `waitrun` waits until the specified simulation finishes.

Note: Although superficially similar, the `run` command in the PDE Scripting Language used for macros is different from the `run` command in the language used for simulation scripts. The simulation script `run` command directly executes the simulation, and does not return until it has completed. The macro `run` command submits the job for simulation, exactly as pressing the **Run** button. It returns once the job has been submitted, and does not wait for the simulation itself to complete.

Interrogating a Schematic and Finding its Topology

The following commands obtain information about the instances and topology of an existing schematic. They are very useful as part of an analysis process, or for implementing macros to modify a schematic, for example, by substituting components. They can also be used to create parts lists, or automatic checking.

- `getbidirmode` queries the current configuration of the bidirectional port group.
- `getbidirportnames` returns the names of all ports which belong to one bidirectional port group.
- `getblocks` gets the instance IDs of all instances in the top level of the current universe or, if the optional parameter is given, all instance IDs of instances corresponding to the module specified by the parameter. May also return virtual galaxies.
- `getbounds` gets the bounds of an instance.
- `getconnectedblocks` gets a list of all instances connected to the specified port of the specified instance.
- `getconnection` gets the other ends of a connection, which has one end specified in the command.

- `getcontenttype` returns a string representing the type (star, galaxy, or universe) of the specified package. If the path is omitted, the type of the current schematic is returned.
- `getmaster` gets the generic master module type of the given instance (that is, the module from which the instance was created).
- `getportdir` gets the direction of a given port.
- `getports` gets a list with ports of a given block. The list will contain either all ports, if the optional parameter is omitted, or only the input ports or output ports depending on the optional parameter.
- `getsignaltypes` gets a list of signal types (electrical, optical, anytype, integer, float, etc.) for a given port.
- `getstates` gets a list with names of all parameters of a given instance or a list of all global parameters of the current schematic if the instance ID is omitted.
- `getterminals` gets a list of all terminals of a given port (for multiport modules such as the *BusCreate* and *BusSplit* modules). Individual terminals of multiport can be then accessed using the `"port_name#terminal_index"` notation.
- `getvirtgal` gets the instance ID of the virtual galaxy to which the module instance belongs or returns an empty string if the module instance resides on the main schematic topology.
- `getxpos` gets the x position of the instance.
- `getypos` gets the y position of the instance.
- `isportvisible` queries the state of some port in the bidirectional port group.
- `isvirtgal` is used to determine whether the specified instance ID corresponds to a virtual galaxy.
- `portplace` gets the position and rotation of a given port. This command can be used to find the orientation of an instance in the schematic.

The topology along a linear path can be found automatically. This is useful for logging the length of a light path, for example, by noting the lengths of the fibers within it. It can also be used as a precursor to setting up instrumentation along the path.

- `parseschematic` traces a path along a linear string of interconnected instances.

User Interaction

As the preceding examples demonstrate, basic macros that save and restore parameter settings or schematic topologies and automate many other complex or repetitive tasks can run without user interaction. However, by using special macro commands, you can create interactive macros which may operate depending on user input or selection.

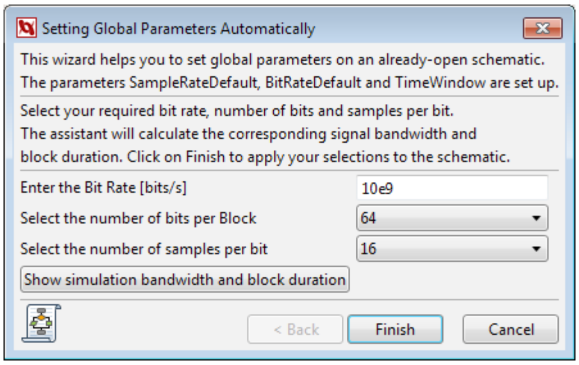
It is possible, for example, to create “self-designing” modules. A self-designing module consists of a schematic whose topology and/or parameters are modified by an interactive macro, so that the user can easily reconfigure the module.

An example of a self-designing module is provided in the demonstration *Optical Systems Demos > Simulation Techniques > General > Efficient WDM System Design*. The *Mean Field Tx Array* galaxy in the demonstration is a self-designing transmitter module. By opening the galaxy, and running the attached *Mean Field Tx Setup* macro, you can easily configure the transmitter arrays to generate a combination of parameterized signals, single frequency band, and multiple frequency band signals.

The interactive interface created by the macro is known as a "wizard". *Wizards* for user input can be combined with mathematical functions and control logic written in the PDE Scripting Language to implement sophisticated new features. For example, an optical or electrical filter response can be designed using analytical expressions within a macro, with a wizard interface that prompts the user to enter the specifications for the filter.

Generating a Wizard Dialog Page

A *wizard dialog page* allows a user to interact with the macro, for example, to enter design specifications, or to set test specifications. A typical example (the dialog created by the `Set Global Parameters` macro) is shown in Figure 1-11.


Setting Global Parameters Automatically

This wizard helps you to set global parameters on an already-open schematic.
The parameters SampleRateDefault, BitRateDefault and TimeWindow are set up.

Select your required bit rate, number of bits and samples per bit.
The assistant will calculate the corresponding signal bandwidth and block duration. Click on Finish to apply your selections to the schematic.

Enter the Bit Rate [bits/s] 10e9
Select the number of bits per Block 64
Select the number of samples per bit 16

Figure 1-11 *An example of a wizard dialog page*

The `wizard` command is used to create wizard dialogs. It has many options and variations that allow different types of fields and controls to be placed in the dialog (see Appendix A, "PDE Scripting Language Reference"). The following examples demonstrate a few of the most common and useful operations with wizards.

If a wizard dialog with a single page is required, the following command structure should be used. Note that a page must be given a name so that texts and interactive dialog elements can be referenced to the page. In this example, we shall call the page page.

The following script shows how this is done. The commands are explained after the code.

In Tcl:

```
# description WizardDialog.vda: Shows how a wizard dialog is created

# create a wizard page and give it a title
```

```
wizard addpage page
wizard title "Wizard of prescribed size"
wizard size 500 100

# open wizard window (must be last)
wizard open {}
```

In Python:

```
# description WizardDialog.vpy: Shows how a wizard dialog is created

# create a wizard page and give it a title
pde.wizardaddpage("page")
pde.wizardtitle("Wizard of prescribed size")
pde.wizardsize(500, 100)

# open wizard window (must be last)
pde.wizardopen(lambda: None)
```

- `wizard addpage` ***`pagename`***: adds an empty page named ***pagename*** to the wizard.
- `wizard title` ***`title`***: sets the title of the wizard window.
- `wizard size` ***`width height`***: sets the width and height (in pixels) of the wizard page area to ***`width`*** × ***`height`***.
- `wizard open` ***`command`***: displays the wizard window.
  The ***command*** argument must contain a Tcl script (one command or several commands in braces). The command(s) will be executed when the **Finish** button is pressed. For Python, the ***command*** argument must be a function that is called when **Finish** is pressed (in the example above a lambda function returning `None` is used).

As can be seen from the examples above, the wizard commands have a slightly different syntax for Python. Whenever Tcl has a command `wizard something`, in Python it is `wizardsomething` (written as one solid word), and in case of ***`pagename`*** `something` it is `wizardsomething(`***`pagename`***`)`.

Note that this box has no contents as yet. It will look like the example in Figure 1-12.

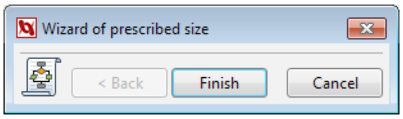


Figure 1-12 *Wizard dialog page (100 high, 500 wide) generated with the script above*

The page automatically displays the **Finish** and **Cancel** buttons. The **Finish** button will run any commands contained in the brackets after `wizard open` (or call a function in Python). For example, this could be a set of commands to construct a schematic.

Generating Wizard Dialogs with Multiple Pages

Note: The example script code in this section is taken from the *MultiplePages* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial.*

Wizards with multiple pages are created by using multiple `wizard addpage` commands. Multiple wizard commands can be combined on a single line, so long as the syntax is unambiguous, therefore to create a wizard with four pages, the following line can be used:

```
wizard addpage page1 addpage page2 addpage page3 addpage page4
```

In Python, it also can be written in one line although the syntax is different (commands are separated with semicolon):

```
pde.wizardaddpage("page1"); pde.wizardaddpage("page2");
    pde.wizardaddpage("page3"); pde.wizardaddpage("page4")
```

The above command(s) creates four pages. By default the order in which they are displayed is determined by the sequence of calling the `addpage` command, but it may be altered later. The wizard command `nextpage`, which is prefixed by the page name, is used to define the page that is displayed when the **Next** button is clicked.

In Tcl:

```
# define page
page1 text "This is page one"
page1 nextpage page2

# define page
page2 text "This is page two"
page2 nextpage page3

# define page
page3 text "This is page three"
page3 nextpage page4

# define page
page4 text "This is page four"
```

In Python:

```
# define page
pde.wizardtext("page1", "This is page one")
pde.wizardnextpage("page1", "page2")

# define page
pde.wizardtext("page2", "This is page two")
pde.wizardnextpage("page2", "page3")
```

```
# define page
pde.wizardtext("page3", "This is page three")
pde.wizardnextpage("page3", "page4")

# define page
pde.wizardtext("page4", "This is page four")
```

Similarly, the command *`pagename`* `prevpage` *`pagename`* defines the page that is opened when the **Back** button is clicked. By default it is the previous page determined by reversing the `nextpage` commands.

The above example also demonstrates the use of the *`pagename`* `text` command to place a text label in a wizard dialog. Any number of graphical components and controls can be added to the wizard page using commands of the form *`pagename command options`*. In addition to `text`, `nextpage`, and `prevpage`, the following commands can be used:

- `label` adds a label (similar to `text`, but doesn't force newline).
- `newline` moves subsequent components onto a new line.
- `separator` adds a separator (horizontal rule, running across the wizard dialog).
- `checkbox` adds a checkbox.
- `combo` adds a combo box (a drop-down choice list).
- `entry` adds a text entry field.
- `button` adds a new button to the page.

Finally, the command `wizard gotopage` *`pagename`* immediately opens the page given in pagename. Some of these commands are illustrated in the following examples, and all are described in detail in Appendix A, "PDE Scripting Language Reference".

Adding Messages to the Message Log

The command `writemssage` can be used to add informational, warning or progress messages to the Message Log.

In Tcl:

```
# add infomational message
writemessage "Informational message"
# add warning message
writemessage "Warning message" -type warning
# add progress message
writemessage "Progress message" -type progress
```

In Python:

```
# add infomational message
pde.writemessage("Informational message")
# add warning message
pde.writemessage("Warning message", "warning")
```

```
# add progress message
pde.writemessage("Progress message", "progress")
```

Displaying Alerts and Simple Messages

The command **messagedialog** can be used to display a message in a pop-up dialog.

In Tcl:

```
# open wizard window
wizard open {
      # message when Finish is pressed
      messagedialog "Thank you for using the 'nextpage' command"
}
```

In Python:

```
# define funcion which will be called when Finish is pressed
def on_wizard_finish():
      # message when Finish is pressed
      pde.messagedialog("Thank you for using the 'nextpage' command")

# open wizard window
pde.wizardopen(on_wizard_finish)
```

In addition to displaying messages to the user once a macro has run, the **messagedialog** command is extremely useful when developing scripts for displaying debugging messages. For example, a message dialog can be used to display the intermediate results in a complex calculation so that you can check that all steps are being performed correctly.

Requesting Input with Query Boxes

Note: The example script code below is taken from the *QueryBox* macro in *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial*.

A query box is a dialog that requires the user to enter a single value of a specified type before continuing (see Figure 1-13). Query boxes are opened using the **querydialog** command.

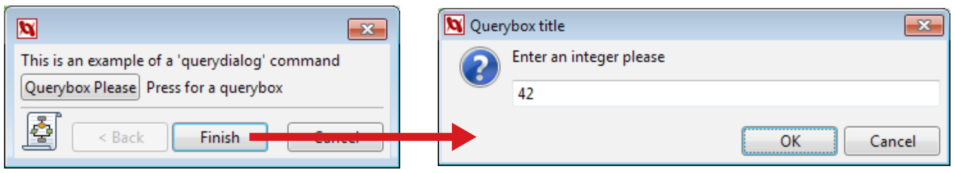


Figure 1-13 *Example showing the use of a query box, and dialog that appears when Finish is pressed*

`QueryBox` macro produces a query dialog that expects an integer when a button is pushed.

In Tcl:

```
# Description:  Shows how a querybox is generated

# init variables
set variable -1
set bButtonPressed  FALSE

# create a wizard page
wizard addpage page
# define page
page text "This is an example of a 'querydialog' command"


page button "Querybox Please" -command {
set variable [querydialog integer "Enter an integer please" "Querybox title" true 0]
set bButtonPressed TRUE
}

page label "Press for a querybox"

# open wizard window
wizard open {
      #message when Finish is pressed
      if  {$bButtonPressed == "TRUE"} {
            messagedialog "You entered: $variable"
      } else {
            messagedialog "You entered nothing."
      }

}
```

In Python:

```
# Description:  Shows how a querybox is generated

import vpi_tc_pde as pde

# init variables
variable = -1
bButtonPressed = False
```

```
# create a wizard page
pde.wizardaddpage("page")
# define page
pde.wizardtext("page", "This is an example of a 'querydialog' command.")

def on_button_click():
      global bButtonPressed, variable
      variable = pde.querydialog("integer", "Enter an integer please", "Querybox
title", True, "0")
      bButtonPressed = True

pde.wizardbutton("page", "Querybox Please", command=on_button_click)

pde.wizardlabel("page", "Press for a querybox")

def on_wizard_finish():
      # message when Finish is pressed
      if bButtonPressed:
            pde.messagedialog("You entered: {}".format(variable))
      else:
            pde.messagedialog("You entered nothing.")

# open wizard window
pde.wizardopen(on_wizard_finish)
```

Placing Labels on a Page

The command *pagename* `label` adds the text `string` to the page *pagename*, with alignment.

In Tcl:

pagename `label` *label* ?`-tooltip` *tooltip*?

In Python:

`wizardlabel(`*pagename*`,` *label*`[,` *tooltip*`])`

With the `-tooltip` option a tooltip containing *tooltip* will pop up when the cursor is placed over the label.

Separating Text

To ensure that each entry has a line of its own (otherwise the entries will appear on the same line), a `newline` command can be added after pagename. This is similar to a newline command for printers.

In Tcl:

pagename `newline`

In Python:

```
wizardnewline(pagename)
```

Only the first newline will be applied. Multiple newlines are ignored. For extra space use the text command with blank text.

Alternatively, a thin visible line can be used to separate lines of text.

In Tcl:

```
pagename separator
```

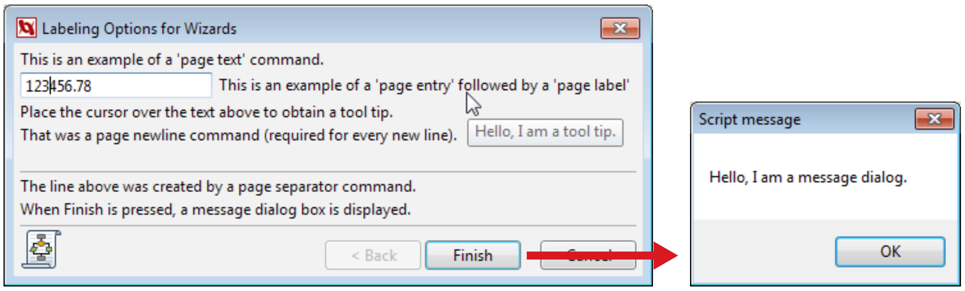
In Python:

```
wizardseparator(pagename)
```

Labeling a Wizard Dialog Page—Example

Note: The example script code in this section is taken from the *Labels* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial*.

Figure 1-14 shows examples of a title, label, text, entry, label, tool-tip, newline, separator, and a message dialog within a wizard dialog.


Labeling Options for Wizards

This is an example of a 'page text' command.

123456.78 This is an example of a 'page entry' followed by a 'page label'

Place the cursor over the text above to obtain a tool tip.

That was a page newline command (required for every new line). Hello, I am a tool tip.

The line above was created by a page separator command.

When Finish is pressed, a message dialog box is displayed.

Script message

Hello, I am a message dialog.

Figure 1-14 *Examples of labels and entries within a wizard dialog, and dialog that appears when Finish is pressed*

The program that generated it (`Labels` macro) is shown below.

In Tcl:

```
# Description: Shows some wizard page labeling commands
#
# variable for display in the 'entry' box
set X 123456.78

# create a wizard page
wizard addpage page
wizard title "Labeling Options for Wizards"

# define page
page text "This is an example of a 'page text' command."
page entry X
page label "This is an example of a 'page entry' followed by a 'page label'" -tooltip
"Hello, I am a tool tip."
page newline
page text "Place the cursor over the text above to obtain a tool tip."
page newline
page text "That was a page newline command (required for every new line)."
page newline
page text " "
# multiple page newlines only give a single newline so use page text " "

page separator
page text "The line above was created by a page separator command."
page text "When Finish is pressed, a message dialog box is displayed."

# open wizard window
wizard open {
      messagedialog "Hello, I am a message dialog."
}
```

In Python:

```
# Description: Shows some wizard page labeling commands
#

import vpi_tc_pde as pde

# create a wizard page
pde.wizardaddpage("page")
pde.wizardtitle("Labeling Options for Wizards")

# define page
pde.wizardtext("page", "This is an example of a 'page text' command.")
pde.wizardentry("page", "X")
# variable for display in the 'entry' box
pde.modify("X", 123456.78)

pde.wizardlabel("page", "This is an example of a 'page entry' followed by a 'page
label'", "Hello, I am a tool tip.")
pde.wizardnewline("page")
pde.wizardtext("page", "Place the cursor over the text above to obtain a tool tip.")
pde.wizardnewline("page")
pde.wizardtext("page", "That was a page newline command (required for every new
line).")
pde.wizardnewline("page")
pde.wizardtext("page", " ")
# multiple page newlines only give a single newline so use page text " "

pde.wizardseparator("page")
pde.wizardtext("page", "The line above was created by a page separator command.")
pde.wizardtext("page", "When Finish is pressed, a message dialog box is displayed.")
```

```
def on_wizard_finish():
        # message when Finish is pressed
        pde.messagedialog("Hello, I am a message dialog.")
# open wizard window
pde.wizardopen(on_wizard_finish)
```

Data Entry

There are four methods of entering data, depending on the type of data:

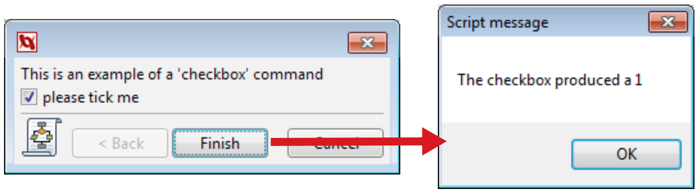
- checkboxes represent Boolean data (yes/no or on/off)
- combo boxes represent lists of choices
- entry boxes allow arbitrary text or data to be entered
- buttons can be used to cause functions to execute, or to initiate other operations.

Each of these methods is illustrated in the following examples. See Appendix A, "PDE Scripting Language Reference" for details on the command syntax and available options.

Check Box Command—Example

Note: This example is the *CheckBox* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial.*

The CheckBox macro shows how a checkbox is displayed, and how it creates a `variable=1` if checked (see Figure 1-15).



This is an example of a 'checkbox' command

☑ please tick me

The checkbox produced a 1

Figure 1-15 *Creating a checkbox variable and dialog that appears when Finish is pressed*

In Tcl:

```
#  Description:  Shows how a checkbox is used.
# create a wizard page
wizard addpage page
# define page
page text "This is an example of a 'checkbox' command"
page checkbox "please tick me" variable
# open wizard window
wizard open {
        #message when Finish is pressed
        messagedialog "The checkbox produced a $variable"
}
```

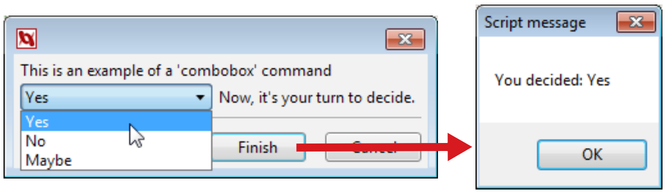
In Python:

```
#  Description:  Shows how a checkbox is used.
import vpi_tc_pde as pde

# create a wizard page
pde.wizardaddpage("page")
# define page
pde.wizardtext("page", "This is an example of a 'checkbox' command.")
pde.wizardcheckbox("page", "please tick me", "variable")

def on_wizard_finish():
        # message when Finish is pressed
        variable = pde.getvariable("variable")
        pde.messagedialog("The checkbox produced a {}".format(variable))

# open wizard window
pde.wizardopen(on_wizard_finish)
```

Using a Combo Box—Example

Note: This example is the *ComboBox* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial*.

The ComboBox macro displays a wizard with a three-entry combo box. Once **Finish** is clicked, the result of the choice is displayed as in Figure 1-16.


This is an example of a 'combobox' command

Now, it's your turn to decide.

You decided: Yes

Figure 1-16 *Combo box wizard and dialog that appears when Finish is pressed*

In Tcl:

```
#  Description: Shows how a combobox is used.

# create a wizard page
wizard addpage page
```

```
# define page
page text "This is an example of a 'combobox' command"

page combo variable Yes|No|Maybe
page label "Now, it's your turn to decide."

# open wizard window
wizard open {
      #message when Finish is pressed
      messagedialog "You decided: $variable"
}
```

In Python:

```
# Description: Shows how a combobox is used.
import vpi_tc_pde as pde

# create a wizard page
pde.wizardaddpage("page")
# define page
pde.wizardtext("page", "This is an example of a 'combobox' command.")

pde.wizardcombo("page", "variable", ["Yes","No","Maybe"])
pde.wizardlabel("page", "Now, it's your turn to decide.")

def on_wizard_finish():
      # message when Finish is pressed
      variable = pde.getvariable("variable")
      pde.messagedialog("You decided: {}".format(variable))

# open wizard window
wpde.wizardopen(on_wizard_finish)
```

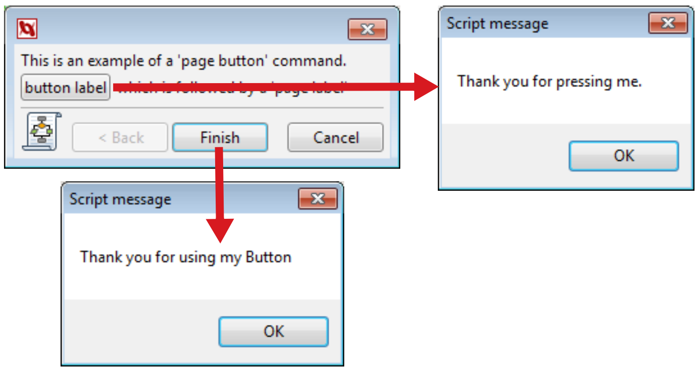
Entry Command—Example

The *pagename* `entry` command is illustrated by the *TextField* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial*. This example is described in detail in "Controlling Wizard Fields" on page 57.

A Wizard with a Button—Example

Note: This example is the *ButtonEntry* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial.*

The `ButtonEntry` macro shows how a button is displayed, and how it calls a procedure (in this case a message dialog) when clicked (see Figure 1-17).


This is an example of a 'page button' command.

button label which is followed by a 'page label'

Script message

Thank you for pressing me.

Script message

Thank you for using my Button

Figure 1-17 *ButtonEntry dialog and dialogs that appear when buttons are pressed*

In Tcl:

```
# Description: Shows how a button calls a procedure

# create a wizard page
wizard addpage page
# define page
page text "This is an example of a 'page button' command."

page button "button label" -command {messagedialog "Thank you for pressing me."}

page label "which is followed by a 'page label'"

# open wizard window
wizard open {
	#message when Finish is pressed
	messagedialog "Thank you for using my Button"
}
```

In Python:

```
# Description: Shows how a button calls a procedure
import vpi_tc_pde as pde

# create a wizard page
pde.wizardaddpage("page")
# define page
pde.wizardtext("page", "This is an example of a 'page button' command.")

def on_button_click():
	pde.messagedialog("Thank you for pressing me.")

pde.wizardbutton("page", "button label", command=on_button_click)

pde.wizardlabel("page", "which is followed by a 'page label'")
```

```
def on_wizard_finish():
       #message when Finish is pressed
       pde.messagedialog("Thank you for using my Button")

# open wizard window
pde.wizardopen(on_wizard_finish)
```

Controlling Wizard Fields

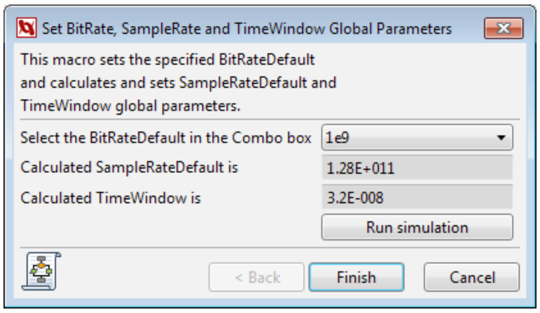
Note: The example script code in this section is taken from the *Set Global Parameters* macro in the demonstration *Optical Systems Demos > Subsystems > Transmitters > RE Model - Direct Modulation* and from the *TextField* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial.*

The command `modify` changes a variable that is bound to a text field, combo box, or checkbox of the wizard. This command works in the same way as the variable assignment (for example, the `set` command in Tcl), but it also modifies the graphical component in the wizard bound to the variable. That is, the user sees the variable change immediately.

Note that in Python this command doesn't use the actual variable and the value should be accessed using the `getvariable` command.

Simulation with Adjustable BitRate—Example

The `Set Global Parameters` macro provides a tool to adjust the values of the main global parameters (**`TimeWindow`** and **`SampleRateDefault`**) according to the value of the other parameter (**`BitRateDefault`**). It uses the input in an entry box as output. The dialog is shown in Figure 1-18.



This macro sets the specified BitRateDefault and calculates and sets SampleRateDefault and TimeWindow global parameters.

Select the BitRateDefault in the Combo box: 1e9
Calculated SampleRateDefault is: 1.28E+011
Calculated TimeWindow is: 3.2E-008

Figure 1-18 *Calculation of global parameters*

In Tcl:

```
set tcl_precision 17
```

```
# global variables
set SampleRateDefault [format "%G" "[expr 128*1E9]" ]
set TimeWindow [format "%G" "[expr 32/1e9 ]" ]

# create a wizard page
wizard addpage page1

wizard title "Set BitRate, SampleRate and TimeWindow Global Parameters"
wizard width 340
wizard height 190

page1 text "This macro sets the specified BitRateDefault"
page1 text "and calculates and sets SampleRateDefault and"
page1 text "TimeWindow global parameters."
page1 separator

page1 newline
page1 label "Select the BitRateDefault in the Combo box "
page1 combo BitRateDefault 1e9|2e9|4e9|8e9|16e9 -tooltip "Select the BitRateDefault"
-command {
modify SampleRateDefault [format "%G" "[expr 128*$BitRateDefault]"]
modify TimeWindow [format "%G" "[expr 32/$BitRateDefault]"]
}

page1 newline
page1 label "Calculated SampleRateDefault is"
page1 entry SampleRateDefault -editable false -tooltip "SampleRateDefault =
2^n*BitRateDefault, n = 7"

page1 newline
page1 label "Calculated TimeWindow is"
page1 entry TimeWindow -editable false -tooltip "SampleRateDefault =
2^m/BitRateDefault, m = 5"

page1 newline
page1 label " "
page1 button "Run simulation"   -tooltip "Press this button to run the simulation with
the selected parameters" -command {
newstate BitRateDefault float $BitRateDefault Global
newstate SampleRateDefault float $SampleRateDefault Global
newstate TimeWindow float $TimeWindow Global
run 4}

# open wizard window
wizard open {
#do nothing
}
```

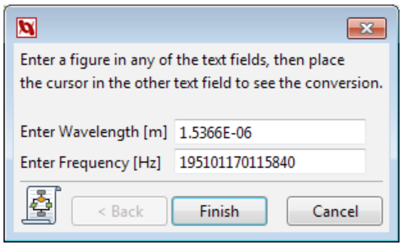
In Python:

```
import vpi_tc_pde as pde

# create a wizard page
pde.wizardaddpage("page")
pde.wizardtitle("Set BitRate, SampleRate and TimeWindow Global Parameters")
pde.wizardwidth(340)
pde.wizardheight(190)

# define page
pde.wizardtext("page", "This macro sets the specified BitRateDefault")
pde.wizardtext("page", "and calculates and sets SampleRateDefault and")
pde.wizardtext("page", "TimeWindow global parameters.")
pde.wizardseparator("page")

pde.wizardnewline("page")

def on_combo_command():
	bitRateDefault = pde.getvariable("BitRateDefault")
	sampleRateDefault = 128*float(bitRateDefault)
	timeWindow = 32/float(bitRateDefault)
	pde.modify("SampleRateDefault", "{:g}".format(sampleRateDefault))
	pde.modify("TimeWindow", "{:g}".format(timeWindow))

pde.wizardlabel("page", "Select the BitRateDefault in the Combo box")
pde.wizardcombo("page", "BitRateDefault", ["1e9","2e9","4e9","8e9","16e9"], tooltip =
"Select the BitRateDefault", command=on_combo_command)
pde.modify("BitRateDefault", 1e9)

pde.wizardnewline("page")
pde.wizardlabel("page", "Calculated SampleRateDefault is")
pde.wizardentry("page", "SampleRateDefault", editable=False,
tooltip="SampleRateDefault = 2^n*BitRateDefault, n = 7")
pde.modify("SampleRateDefault", "{:g}".format(128*1e9))

pde.wizardnewline("page")
pde.wizardlabel("page", "Calculated TimeWindow is")
pde.wizardentry("page", "TimeWindow", editable=False, tooltip="TimeWindow =
2^m/BitRateDefault, m = 5")
pde.modify("TimeWindow", "{:g}".format(32/1e9))

def on_button_command():
	pde.newstate("BitRateDefault", "float", pde.getvariable("BitRateDefault"),
"Global")
	pde.newstate("SampleRateDefault", "float", pde.getvariable("SampleRateDefault"),
"Global")
	pde.newstate("TimeWindow", "float", pde.getvariable("TimeWindow"), "Global")
	bitRateDefault = float(pde.getvariable("BitRateDefault"))
	pde.run(4)

pde.wizardnewline("page")
pde.wizardlabel("page", " ")
pde.wizardbutton("page", "Run simulation", tooltip = "Press this button to run the
simulation with the selected parameters", command = on_button_command)

# open wizard window
pde.wizardopen(lambda: None)
```

Wavelength to Frequency Converter—Example

An alternative way to update a macro variable with a text entry is to use the `-variable` and `-focuslost` options on the *pagename* `entry` command. The `-variable` option specifies the name of the global *Tcl* variable that will be changed when the user moves the input focus (cursor) out of the text entry field. This variable will then have a value returned by the procedure specified via the `-focuslost` option and the wizard elements that use the variable will be updated (similar to the `modify` command). And similarly to the `modify` command, a value can be accessed in Python using the `getvariable` command.

The `TextField` macro is a handy converter between wavelength and optical frequency, or vice versa. It uses the ability of an entry box to be an input or an output with the `-variable` option. Its dialog is shown in Figure 1-19.


Enter a figure in any of the text fields, then place
the cursor in the other text field to see the conversion.

Enter Wavelength [m] 1.5366E-06

Enter Frequency [Hz] 195101170115840

Figure 1-19 *Wavelength to frequency converter*

In Tcl:

```
# description Shows how fields can be used as inputs/outputs
# speed of light
set C 2.99792458e8

# procedure to convert wavelength to frequency
proc wavelengthToFrequency {w} {
      global C
      return [expr $C / $w]
}

# create a wizard page
wizard addpage page

# initialize variable to hold wavelength
set wave 1536.6e-9
# initialize variable to hold frequency
set freq [expr $C / $wave]
page text "Enter a figure in any of the text fields, then place"
page text "the cursor in the other text field to see the conversion."
page text " "

# create a label to display title for textfield to enter wavelength
page label "Enter Wavelength \[m\]"

# create a textfield to enter wavelength
# if focus has been lost the frequency of the currently
# entered wavelength is calculated and assigned to "freq"
page entry wave -variable freq -focuslost {wavelengthToFrequency $wave}

# add newline
page newline

# create a label to display title for textfield to enter frequency
page label "Enter Frequency \[Hz\]"

# create a textfield to enter frequency
# if focus has been lost the wavelength of the currently
# entered frequency is calculated and assigned to "wave"
page entry freq -variable wave -focuslost {wavelengthToFrequency $freq}

# open wizard window
wizard open {
      #
      # the following commands will be executed when "Finish" has been pressed
      #
      # display window with wavelength and frequency
      messagedialog "Wavelength: $wave\nFrequency: $freq"
}
```

In Python:

```
# Description:  Shows how fields can be used as inputs/outputs
import vpi_tc_pde as pde

# speed of light
C = 2.99792458e8

# initialize variable to hold wavelength
wave = 1536.6e-9

# procedure to convert wavelength to frequency
def wavelengthToFrequency():
      return C/float(pde.getvariable("wave"))

# procedure to convert frequency to wavelength
def frequencyToWavelength():
      return C/float(pde.getvariable("freq"))
# create a wizard page
pde.wizardaddpage("page")

# define page
pde.wizardtext("page", "Enter a figure in any of the text fields, then place")
pde.wizardtext("page", "the cursor in the other text field to see the conversion.")
pde.wizardtext("page", " ")

# create a label to display title for textfiled to enter wavelength
pde.wizardlabel("page", "Enter Wavelength [m]")

# create a textfield to enter wavelength
# if focus has been lost the frequency of the currently
# entered wavelength is calculated and assigned to "freq"
pde.wizardentry("page", "wave", variable="freq", focuslost=wavelengthToFrequency)
pde.modify("wave", wave)

# add newline
pde.wizardnewline("page")

# create a label to display title for textfield to enter frequency
pde.wizardlabel("page", "Enter Frequency [Hz]")

# create a textfield to enter frequency
# if focus has been lost the wavelength of the currently
# entered frequency is calculated and assigned to "wave"
pde.wizardentry("page", "freq", variable="wave", focuslost=frequencyToWavelength)
pde.modify("freq", C/wave)

def on_wizard_finish():
      # message when Finish is pressed
      pde.messagedialog("Wavelength: {} m\nFrequency: {} Hz".format(
pde.getvariable("wave"), pde.getvariable("freq")))

# open wizard window
pde.wizardopen(on_wizard_finish)
```

External Program Automation

The COM object command allows communication with an external server using the late binding approach. Macros can serve as automation controllers for external applications that support COM Automation (such as Microsoft Office).

You can automate work with external applications that support the COM interface using the following commands:

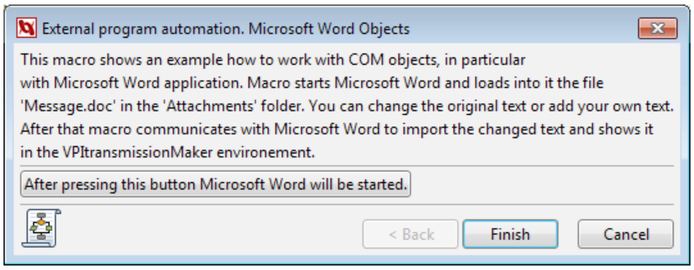
- `createobject` creates an external ActiveX object instance based on ProgID;
- `releaseobject` releases an ActiveX object instance previously created with `createobject`;
- `putproperty` modifies property of an ActiveX object;
- `getproperty` returns property value of an ActiveX object;
- `invoke` invokes a method on an object instance with a set of arguments.

The above commands are not available for Python. The same functionality may be reproduced using `pywin32`, a third-party library distributed together with VPI Design Suite.

Open Word Document, Edit, Display Contents, Close Word

Note: This example is the *WordMessage* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial.*

The `WordMessage` macro starts Microsoft Word and opens the `Message.doc` file stored in the `Attachments` folder. You can change the original text or add your own. After that, the macro communicates with Microsoft Word to import the modified text and show it in the VPI Design Suite environment.


External program automation. Microsoft Word Objects

This macro shows an example how to work with COM objects, in particular
with Microsoft Word application. Macro starts Microsoft Word and loads into it the file
'Message.doc' in the 'Attachments' folder. You can change the original text or add your own text.
After that macro communicates with Microsoft Word to import the changed text and shows it
in the VPItransmissionMaker environement.

After pressing this button Microsoft Word will be started.

Figure 1-20 *Microsoft Word Application Automation*

In Tcl:

```
#set path to the Message.doc
set path [packagefolder [curuniverse]]\\Attachments\\Message.doc

#create external object instance and make it close without alerts
set Wrd [createobject "Word.Application" ]

#make Word-object visible when open
putproperty $Wrd "Visible" True

#return the property of object
set wbooks [getproperty $Wrd "Documents"]

#open Message.doc from the specified path
set wbook [invoke $wbooks "Open" $path]

#message
messagedialog "Please switch to Microsoft Word and edit the text. Press OK when you are
done."

#activate document
set sheet [getproperty $Wrd "ActiveDocument"]
putproperty $sheet "Saved" True

#define a data range
set range [getproperty $sheet "Paragraphs" ]
```

```
#get the value of a line
set line [invoke $range "Item" 1]

#set a range of the line
set line_range [getproperty $line "Range"]


#message from Message.doc
messagedialog "The text from Microsoft Word: ' [getproperty $line_range "Text"] '"

#close active document without modifications
invoke $sheet "Close"

#close Wrd
invoke $Wrd "Quit"

#relesase objects range, line, line_range, sheet, wbook, wbooks, Wrd
releaseobject $range
releaseobject $line
releaseobject $line_range
releaseobject $wbook
releaseobject $wbooks
releaseobject $Wrd
releaseobject $sheet
```

In Python:

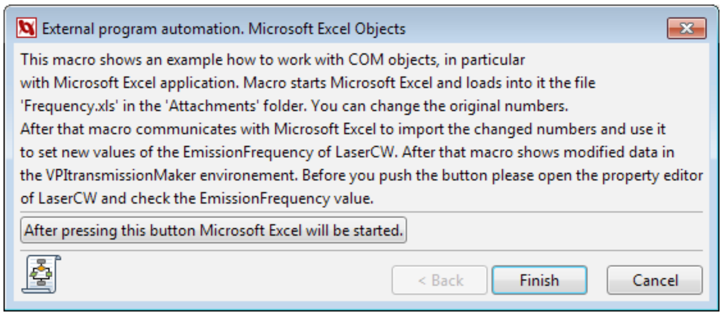
```
# set path to the Message.doc
path = os.path.join(pde.packagefolder(pde.curuniverse()), "Attachments", "Message.doc")

# create external object instance
word = win32com.client.Dispatch('Word.Application')

# make Word-object visible when open
word.Visible = True

# open Message.doc from the specified path
doc = word.Documents.Open(path)

# message
pde.messagedialog("Please switch to Microsoft Word and edit the text. Press OK when you
are done.")

# activate document
word.ActiveDocument.Saved = True

# get the value of a line
line_text = word.ActiveDocument.Paragraphs.Item(1).Range.Text

# message from Message.doc
pde.messagedialog(u"The text from Microsoft Word: '{}'".format(line_text))

# close active document without modifications
word.Documents.Close()
```

```
# close Word
word.Quit()
```

Open Excel Document, Edit, Set Instance Parameter, Close Excel

Note: This example is the *ExcelToModule* macro in the demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial.*

The `ExcelToModule` macro starts Microsoft Excel and opens the `Frequency.xls` file in the `Attachments` folder. You can change the original numbers in the Excel worksheet. After that, the macro communicates with Microsoft Excel to import the changed numbers and uses this data to set new values for the **`EmissionFrequency`** parameter of the *LaserCW* module. After that, the macro shows the modified data in the VPI Design Suite environment.


External program automation. Microsoft Excel Objects

This macro shows an example how to work with COM objects, in particular with Microsoft Excel application. Macro starts Microsoft Excel and loads into it the file 'Frequency.xls' in the 'Attachments' folder. You can change the original numbers. After that macro communicates with Microsoft Excel to import the changed numbers and use it to set new values of the EmissionFrequency of LaserCW. After that macro shows modified data in the VPItransmissionMaker environement. Before you push the button please open the property editor of LaserCW and check the EmissionFrequency value.

After pressing this button Microsoft Excel will be started.

Figure 1-21 *Microsoft Excel Application Automation*

In Tcl:

```
#set path to the Frequency.xls
set path [packagefolder [curuniverse]]\\Attachments\\Frequency.xls

#create external object instance
set Excel [createobject "Excel.Application"]

#make it visible when open
putproperty $Excel "Visible" True

#cancel warnings for Excel application during running script
putproperty $Excel "DisplayAlerts" False

#return the property of object
set wbooks [getproperty $Excel "Workbooks"]

#open Frequency.xls from the specified path
set wbook [invoke $wbooks "Open" $path]
```

```
#message
messagedialog "Please switch to Microsoft Excel and edit the text. Press OK when you are
done. Look through the EmissionFrequency of LaserCW" "Please edit numbers!" "warning"

#activate worksheet
set sheet [getproperty $wbook "ActiveSheet"]

#define a range of data column#D and 3, 4,5 rows
set range [getproperty $sheet "Range" "D3:D5"]

#cycle for the specified range
for {set i 1} { $i < 4} {incr i 1} {
   #get the value of i-th cell
   set cell [getproperty $range "Cells" $i]

   #set the value of EmissionFrequency of LaserCW_vtms1 from Frequency.xls
   setstate LaserCW_vtms1 EmissionFrequency [getproperty $cell "Value"]

   #message before the value
   messagedialog "The value of the EmissionFrequency of LaserCW = [getproperty $cell
"Value"] Hz" "Frequency values from the file" "plain"

   #release object cell
   releaseobject $cell
}

#close Excel object
invoke $Excel "Quit"

#restore warnings for Excel applications after run of script
putproperty $Excel "DisplayAlerts" True

#release objects range, sheet, wbook, wbooks, Excel
releaseobject $range
releaseobject $sheet
releaseobject $wbook
releaseobject $wbooks
releaseobject $Excel
```

In Python:

```
# set path to the Frequency.xls
path = os.path.join(pde.packagefolder(pde.curuniverse()), "Attachments",
"Frequency.xls")

# create external object instance
excel = win32com.client.Dispatch('Excel.Application')
# make Excel visible when open
excel.Visible = True

# cancel warnings for Excel application during running script
excel.DisplayAlerts = False

# open Frequency.xls from the specified path
```

```
wbook = excel.Workbooks.Open(path)

# message
pde.messagedialog("Please switch to Microsoft Excel and edit the text. Press OK when
you are done. Look through the EmissionFrequency of LaserCW.", "Please edit numbers!",
"information")

# define a range of data column#D and 3, 4, 5 rows
cell_range = wbook.ActiveSheet.Range("D3:D5")

# cycle for the specified range
for i in range(1, 4):
      # get the value of i-th cell
      cell = cell_range.Cells(i)

      # set the value of EmissionFrequency of LaserCW_vtms1 from Frequency.xls
      pde.setstate("LaserCW_vtms1", "EmissionFrequency", cell.Value)

      # message before the value
      pde.messagedialog("The value of the EmissionFrequency of LaserCW = {}
Hz".format(cell.Value), "Frequency values from the file", "plain")

# close Excel
excel.Quit()

# restore warnings for Excel applications after run of script
excel.DisplayAlerts = True
```

Working with Sweeps

The **sweep** command allows the interactive simulation function to be driven from scripts. A sweep has many parameters to define and assign, as described in Chapter 7 of the *VPIphotonics Design Suite™ Simulation Guide*.

Opening a Sweep

To load an interactive simulation (sweep) from an existing `.vsw` file, use the **sweep** command with the **-load** option in Tcl or the **sweepload** command in Python. To open an interactive simulation window (editor), use the **-open** option in Tcl or the **sweepopen** command in Python.

In Tcl:

```
sweep sweepname -load path
sweepname -open
```

In Python:

```
sweepload(sweepname, path)
sweepopen(sweepname)
```

Note: You can use the packagefolder and curuniverse macro commands to obtain the path to the Resources folder of the current schematic where your sweeps are stored.

An example in Tcl:

```
set name {Length & Dispersion.vsw}
set sweeppath [file join [packagefolder [curuniverse]] Resources $name]
sweep sweep1 -load $sweeppath
sweep1 -open
```

An example in Python:

```
import os
name = r"Length & Dispersion.vsw"
sweeppath = os.path.join(packagefolder(curuniverse()), "Resources", name)
sweepload("sweep1", sweeppath)
sweepopen("sweep1")
```

Creating a Sweep

You can create interactive simulations (sweeps) in macros. However, it's impossible to save an interactive simulation from macros. You may work around it by opening the interactive simulation window (editor) and saving it manually via the **Save** button in the bottom right corner of the interactive simulation window.

In Tcl, the **sweep** command is used to create a new named sweep or load an existing sweep from a file. The ***sweepname*** command is used to modify an existing one (created by a previous call of the **sweep** command).

```
sweep sweep1 -setoption interactive off -setoption parallel on
sweep1 -open
```

In Python, a set of commands, such as sweep, sweepsetoption, sweepaddrule, and sweepassign, should be used.

```
sweep("sweep1")
sweepsetoption("sweep1", "interactive", False)
sweepsetoption("sweep1", "parallel", True)
sweepopen("sweep1")
```

Defining Sweep Rules

The sweeprule command is used to define a new sweep rule (sweep control) that can be used with the **sweep** command.

Note: Sweep rules with a different number of iterations cannot have the same depth.

Running a Sweep

You can run interactive simulations (sweeps) from macros using the `run` macro command. Note that you have to load or create a sweep first: it's impossible to run a sweep by specifying the path to the `.vsw` file in the `run` command arguments.

An example in Tcl:

```
set name {Length & Dispersion.vsw}
set sweeppath [file join [packagefolder [curuniverse]] Resources $name]
sweep sweep1 -load $sweeppath
run 1 -sweep sweep1
```

An example in Python:

```
import os
name = r"Length & Dispersion.vsw"
sweeppath = os.path.join(packagefolder(curuniverse()), "Resources", name)
sweepload("sweep1", sweeppath)
run(1, sweep="sweep1")
```

Module Sweeps

The `newmodulesweep` command creates a new module sweep. A module sweep is similar to a galaxy, but has no links, galaxy ports, and junction nodes. It is used to compare the performance of several devices in a system by substituting them into a main schematic. For adding and removing instances, changing parameter settings, and saving the schematic, use the same commands that are used for universes and galaxies.

Miscellaneous Commands

You may find useful the following Tcl/Python and macro commands:

- `categoryorder` returns a number between 1 and 5 representing the position of a given `categoryname` in the list of known categories (**`Global`**, **`Physical`**, **`Numerical`**, **`General`**, **`Enhanced`**) or 6, if it is unknown.
- `exec` executes the external program
  ```
  set python [getpythonexepath]
  exec $python {C:\Scripts\schematic analysis.py}
  ```
  The `-sync` option makes this command synchronous (`exec` will wait for the process to finish before returning): `exec ?-sync?` *`command_line`*
- `exit` closes the VPI Design Suite PDE when invoked from a macro. It is recommended to use the `return` command to exit the macro. For Python, it only exits the macro because it is run in the separate process.

Useful Tips

Creating Dangling Ends to Wires

A *Junction* module may be used as a point of connection of several wires, so this point can be connected to other instances manually. For this, create a connection from port `p1` (input or output) of instance `b1` to the node `Junction` (optionally with a given `delay`).

To create a new junction (with instance ID `Junction1`), use

```
URN:VPI_LIB::TC Modules\Wiring Tools\junction.vtmy:
```

The name of the junction's "port" is `junction`, so to connect the output port *`outport`* of a module instance *`inst`* to the junction, use

In Tcl:

```
connect inst outport Junction1 junction ?delay?
```

In Python:

```
pde.connect(inst, outport, "Junction1", "junction"[, delay])
```

Running a New Process from a Python Macro

If you wish to launch a new process from within a Python macro via the `subprocess` module, please use the *`close_fds`*`=True` flag for the `Popen` constructor as described in the Python subprocess module documentation at https://docs.python.org/3.11/library/subprocess.html#popen-constructor. Otherwise a new process may break the Python scripting support in the VPI Design Suite PDE under certain conditions (e.g., a child process started by a macro still running while the PDE was restarted).

Chapter 2

Cosimulation

This chapter covers the cosimulation capabilities provided by VPIphotonics Design Suite™.

Cosimulation is a technique in which some part of the simulation is handled by an application other than the VPI Design Suite simulator. Cosimulation enables you to implement new simulation modules, in addition to those provided by VPIphotonics.

In many cases, new modules can be constructed as galaxies, based on existing modules. However, there may be times when you will want to implement a new module that cannot be built this way. For example, you may wish to simulate a new type of component that is completely different from anything already on the market. Alternatively, a galaxy-based model may be too complex, or too inefficient to be practical. Another circumstance in which cosimulation is useful is when you have an existing proprietary simulation model developed using one of the supported cosimulation environments, that you wish to integrate into a larger simulated system within VPI Design Suite.

The application environment in which cosimulation takes place is referred to as the *target environment* within this chapter. Four target environments are currently supported. These are:

- MATLAB (version 7.9 or higher).
  Your cosimulation module is implemented as one or more MATLAB functions, in one or more source files. The cosimulation script has access to all the power of MATLAB, including any Toolboxes you have installed. MATLAB cosimulation is described in detail in "Cosimulation with Matlab" on page 93.
- Python (version 3.11.1 with the NumPy 1.23.5 and other packages, which are installed by default during the VPI Design Suite installation under both Windows and Linux). Your cosimulation module is implemented as one or more Python functions, in one or more source files. Python is a freely available, interpreted, interactive, object-oriented programming language. Combined with the numerical and scientific extension packages, Python provides a powerful environment for cosimulation. Python cosimulation is described in detail in "Cosimulation with Python" on page 114.
- Compiled C/C++ code in a dynamic-link library (a Windows DLL, or a Unix shared library). The use of native code provides the greatest flexibility, and potentially the highest computational efficiency, of all cosimulation methods, although in some cases a greater development effort may be required. VPI Design Suite includes the

MIT FFTW package to support faster development of numerical processing code. C/C++ library cosimulation is described in detail under "Cosimulation with Dynamic-Link Libraries" on page 150.

- COM component (only available on Windows). Using Microsoft Component Object Model (COM) technology allows you to implement a cosimulation module in almost any modern language targeting the Microsoft Windows platform. This environment also provides a more object-oriented approach than other environments. COM cosimulation is described in detail under "Cosimulation with COM Components" on page 214.

Other cosimulation capabilities include:

- *Rx_mQAM_DSP_BER* and *BER_mQAM_DSP* (from *Module Library* > *Receivers*),
- *LinkAnalyzer* (from *Module Library* > *Analyzers*),
- *PIC_CoSimInterface* from *Module Library* > *Cosimulation* provides a possibility to create custom PIC Elements (linear photonic devices whose functionality is completely described by a user-defined S-matrix),
- cosimulation with ADVANCED DESIGN SYSTEMS (Keysight Technologies) target environment.

The *Rx_mQAM_DSP_BER*, *BER_mQAM_DSP* and *LinkAnalyzer* modules provide access to MATLAB, Python and C library target environments. *PIC_CoSimInterface* supports Python and C library target environments. Further details can be found in *Photonic Modules Reference* for corresponding modules.

The *ADS_Interface* module from *Module Library* > *Cosimulation* serves as an interface to the ADVANCED DESIGN SYSTEMS cosimulation environment. Further details can be found in the *Photonic Modules Reference* and the white papers located in the `Attachments` folder of the demos in *Optical Systems Demos* > *Simulation Techniques* > *Cosimulation* > *ADS*.

The remainder of this chapter is organized as follows: the first section gives an overview of the general cosimulation interface to VPI Design Suite; this is followed by separate sections covering each of the four cosimulation environments.

The examples for this chapter can be found in the demonstrations folder *Optical Systems Demos* > *Simulation Techniques* > *Cosimulation*. Some of these examples are also available in demonstrations for other products, VPIcomponentMaker Fiber Optics and VPIcomponentMaker Photonic Circuits.

The Cosimulation Interface

To use the cosimulation interface in VPI Design Suite you first create a galaxy, containing the required input and output ports, a cosimulation data interface module for each port, and a single cosimulation module that drives simulation in the target environment. By creating new user parameters in the galaxy, you can define a custom parameter set for your cosimulation module. When your cosimulation galaxy is placed in another schematic, these parameters can be edited just like any other module.

Note: A virtual galaxy can be used for fast creation or prototyping of cosimulation interfaces.

For each simulation run, the input and parameter data are stored in the target environment and a specified function is executed within that environment; afterwards, the expected output data is collected and fed back into the simulation.

When you develop your cosimulation code in any of the supported environments, you can provide up to three functions, which will be called by the cosimulation interface at different times during the simulation run. These functions are:

- An *initialization* function that is executed upon start-up of the simulation. At the time of execution of this function, the initial parameter settings are available; however, there is not yet any input data since the simulation run has not yet commenced.
- A *run* function that is executed each time simulation data is available at the inputs of your cosimulation module. At execution time of this function, current parameter settings *and* valid input data are available to your cosimulation code.
- A *wrapup* function that is executed after completion of the simulation. At execution time of this function, the parameter settings are available; however, there is no input data since the simulation run has completed.

Creating the Cosimulation Interface

Modules used to build cosimulation galaxies are found in the *Cosimulation* folder of the module library. The main interface to the target environment is provided by the *CoSimInterface* module. This module performs the following functions:

- copies the cosimulation galaxy and global parameter settings into the target environment.
- copies simulation input data received via the connected cosimulation input modules (*CoSimInputOpt*, *CoSimInputEl*, etc.) into the target environment.
- executes specific functions or specified commands within the target environment.
- copies output data out of the target environment and sends it to the output ports of the connected cosimulation output modules (*CoSimOutputOpt*, *CoSimOutputEl*, etc.).

The *CoSimInterface* module has a number of parameters, which enable you to choose the target environment, specify the path to a directory containing cosimulation code, specify the name of the script or library files that might be required, specify whether logical information is available in cosimulation and name the functions to execute at start-up, run and wrap-up of the simulation. These parameters are used slightly differently for each target environment; they are described in detail in the corresponding sections below.

Parameters

All parameters of the cosimulation galaxy are copied into the target environment by the *CoSimInterface* module. However, how those are accessed in the cosimulation code is heavily dependent on the target environment type.

In addition to the parameters of the cosimulation galaxy, the following global (schematic) parameters are copied to the target environment:

- **`TimeWindow`**
- **`SampleModeBandwidth`**
- **`SampleModeCenterFrequency`**
- **`GreatestPrimeFactorLimit`**
- **`BoundaryConditions`**

To avoid ambiguity in MATLAB, Python and Library cosimulation environments, where all such names appear in one scope, schematic parameters are prefixed by 'topology_', for example, the parameter **`TimeWindow`** becomes **`topology_TimeWindow`** in the cosimulation environment. The COM cosimulation interface uses a different approach to provide access to those parameters.

Access to these parameters simplifies cosimulation code, when the input signal is not available (for example, for signal creation using cosimulation or when the *NullSource* module is connected to cosimulation interface).

Apart from the inputs and parameters which are defined individually for each application, the cosimulation interface always provides one additional variable to the target environment:

- `ifc_name` holds a string (the instance ID of the *CoSimInterface* module) that uniquely identifies the cosimulation instance within the simulation. This can be used to distinguish between a number of instances of the same cosimulation module. The same information is accessible via the cosimulation context object in COM cosimulation.
- `ifc_CRCnew` is initialized by the cosimulation interface to hold a checksum calculated over the interface structures (that is, the inputs, outputs and parameters). It can be checked at run-time against a CRC calculated when the galaxy was first created. A special utility function `XXX_isValidIfc()` is automatically generated for Library cosimulation in the header file to do that. This function can be used in the `XXX_init()` function to verify that the galaxy has not been modified since the library was compiled. The value of `ifc_CRCnew` should be sent as an argument to this function.

Ports

In addition to the main *CoSimInterface* module, a number of dedicated data interface modules are provided. A data interface module is required between each input or output port of the cosimulation galaxy, and the *CoSimInterface* module. The function of the data interface modules is to define the specific *type* (for example, *optical signal*, *electrical signal*, *float*, etc.) of the port. The following data interface modules are provided:

- Floating-point numbers and matrices (*CoSimInputFlt*, *CoSimOutputFlt*, *CoSimInputMxFlt*, *CoSimOutputMxFlt*)
- Complex numbers and matrices (*CoSimInputCx*, *CoSimOutputCx*, *CoSimInputMxCx*, *CoSimOutputMxCx*)
- Electrical samples and block signals (*CoSimInputEl* and *CoSimOutputEl*)
- Optical samples and block signals (*CoSimInputOpt*,*CoSimInputOpt_N*, *CoSimOutputOpt* and *CoSimOutputOpt_N*)
- Bit sequence (*CoSimInputBitSeq*).

Each of these modules has a parameter **`Name`**, which you use to define the name of the corresponding input or output variable in the target environment.

Note: These names must be unique across all inputs, outputs and parameters of that cosimulation interface! This means that no two input, output or parameter variables may have the same name.

For dynamic languages, MATLAB and Python, names of ports and parameters should not coincide with reserved keywords.

The electrical and optical input data interface modules have an additional parameter **`InputDataType`**, which may be set to `Samples`, `Blocks`, or `Any`. This parameter specifies whether the corresponding cosimulation input is able to accept signals in the form of individual samples, blocks, or either type.

Figure 2-1 shows an example cosimulation galaxy. The cosimulation interface has two inputs: one optical (top left) and one scalar floating-point (bottom left). It also has one optical output.

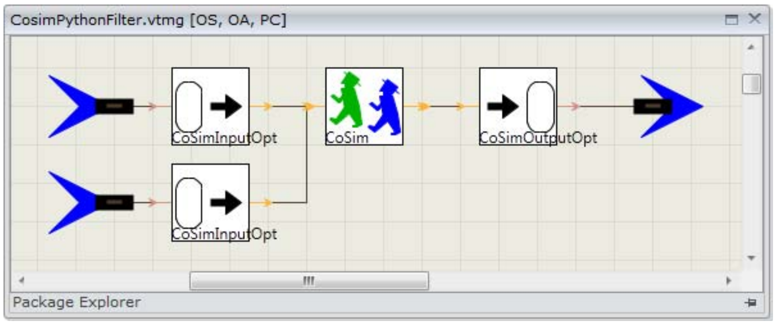


Figure 2-1 *Example cosimulation interface galaxy with one optical and one float input and one optical output*

Mapping Signals to Data Structures in the Target Environment

While the representation of scalar numbers and matrices is usually straightforward in the target environment, VPI optical and electrical signals need to be mapped to nested structures to preserve the internal representation consisting of sampled bands, noise bins, and parameterized signals (for block mode signals), or sampled data (for sample mode signals). The VPI signal representations are managed by a component of the VPI Design Suite software known as the Optical Network Simulation Layer (ONSL). They are therefore referred to as 'ONSL signals'.

This section describes the general structure of these mappings for each ONSL signal type. The structures described here are generic, covering the features common to all target environments, and may not represent exactly the variables passed into a given target environment. For example MATLAB, Python and Library cosimulations use structures and names very similar to those presented in this section, whereas COM cosimulation uses COM objects to store data. Additionally, property names may significantly differ from those described here. For these details, you should also read the corresponding section for each target environment.

Logical information is also available in the cosimulation environment. It can be accessed, processed and output from the cosimulation program. For this purpose, the global parameter **`LogicalInformation`** should be set to `On` (or `CurrentRun`) and the **`LogicalInformation`** parameter of the *CoSimInterface* module should be set to `Yes`.

Quantized Values

It is important to understand that all time and frequency values that define the layout of signals are handled as quantized values:

- In the time domain, the basic quantization grid spacing is defined by a floating-point variable `dt`, which specifies a fundamental timestep in seconds. All other time quantities are defined as *integer multiples* of this unit. For example, the time window is defined by an integer variable `T`, such that the actual duration of the time window (in seconds) is computed as **`TimeWindow`** = `T` × `dt`.
- In the frequency domain, the basic quantization grid spacing is defined by a floating-point variable `df`, which specifies a fundamental frequency step, in Hz. This frequency step is, by definition, the resolution of a discrete Fourier transform of a full block of sampled data, and is therefore related to the time window according to `df` = 1/**`TimeWindow`**.

Note: Though all variables in units of time and frequency step have an integer value, the actual type of these variables is double in Matlab, float in Python and 64-bit integer in dynamic-link libraries, respectively. Integer type for these variables was used in cosimulation with the dynamic-link libraries in previous versions of VPI Design Suite. Cosimulation with existing libraries is currently supported, but it is recommended to recompile the libraries.

- Sampled bands are represented simply as arrays, typically without any specific timing information other than the global time data described above. Each sampled band in an MFB optical signal may consist of a different number of samples, depending upon the sample rate of the signal. By definition, each sampled band fills the complete time window, and so the sample period in seconds can be determined by dividing the **`TimeWindow`** by the array length. Alternatively, the sample period in units of the quantization unit `dt` can be obtained by dividing `T` by the array length.

Optical Signals

Optical signals are represented using nested *structures* as shown in Table 2-2 to 2-5. The top level structure shown in Table 2-2 contains some general signal information starting with the signal boundary conditions (`boundaries`), the time and frequency grid spacing already mentioned (`dt` and `df`), the timestamp (start time) and duration of the signal (`t0` and `T`) and the upper and lower frequency limits (`f1` and `f0`). Note, that these time and frequency values are given as integer multiples of the respective grid spacing. The timestamp `t0` deserves further explanation. For signals with *periodic* boundary conditions, the timestamp has no well-defined meaning since the block of data represents one period of a waveform that extends in principle over all time. In this case, the time stamp will typically be zero. However, in the case of signals with *aperiodic* boundary conditions, each block of data generated by a transmitter follows on in time from the preceding block. In this case, the timestamp will increase by one time window for each block generated.

In addition there are three *arrays* which contain the:

- sampled frequency bands
- parameterized signals
- noise bins.

Each of these signal representations is stored in a separate structure located in the respective array. The contents of these structures are also displayed in Table 2-2 to 2-5.

An optical *sampled band* is characterized by the frequency limits (`f0` and `f1`), and the sample array of the electrical field. The storage of the optical field depends on the state of polarization of the signal.

- For an *arbitrary polarization state* the arrays `Ex` and `Ey` are used for the *x*- and *y*-polarization components of the electrical field. Such signals may have time-dependent polarization, since for each sample time the *x* and *y* components of the polarization vector are stored.
- If the signal has *constant polarization* (degree of polarization is 100%) a single array `E` is used. The polarization information is represented by the state of polarization on the polarization ellipse by giving the azimuth (`azi`) and ellipticity (`ell`) in degrees.

Parameterized signals (PS) and *noise bins* (NB) are characterized by the frequency limits (`f0` and `f1`) and the four components of the Stokes vector (`S`). Note that the power of parameterized signals and noise bins is stored in the first component of the Stokes vector.

Parameterized signals also contain the following statistical characteristics of the signal: *AveragePulsePosition*, *ExtinctionRatio*, and *PulsePositionVariance*.

Parameterized signals also contain the following tracking characteristics of the signal (if the global schematic parameter **`TrackingMode`** is set to either `Detailed` or `FinalValues`): *Event, Entry, PhysicalPath, TopologicalPosition, TransitTime, Path, AccumulatedGVD, Power, S1, S2, S3, NonlinearCoefficient, AccumulatedDGD, SPM, Bo, Be, NoisePSD, NoisePower, DistCohTotalPower,* and *DistIncohTotalPower*.

Distortions (D) are a special kind of parameterized signals that represent parasitic signals leading to degradation of signal quality. Similarly to parameterized signals, distortions are characterized by the frequency limits (`f0` and `f1`) and the four components of the Stokes vector (S). Contrary to parameterized signals, distortions do not contain tracking data. Please refer to the *VPItransmissionMaker™Optical Systems User's Manual* for details on distortions.

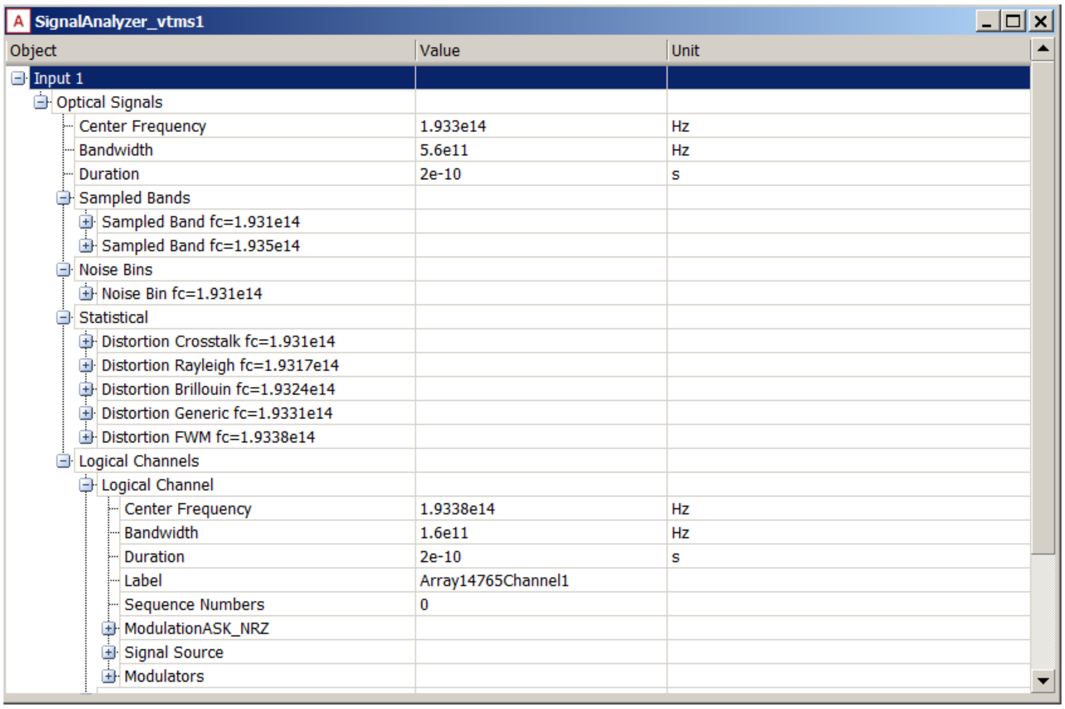
Another special kind of signals are null signals, which can be produced by the *NullSource* module or any other deactivated module (e.g., *LaserCW* with parameter **`Active`** set to `No`). In the cosimulation environment, the structure of such signals is identical to that of ordinary signals, with the following exceptions:

- the `type` field contains the keywords '`osignal_null`', '`osample_null`', '`esignal_null`', '`esample_null`' for optical and electrical bands and samples,
- all other fields are empty.

Note: If an input signal is delayed, which is represented by a null signal, and the parameter **`InputDataType`** of the *CoSimInputOpt* or *CoSimInputOpt_N* module is set to `Any`, the signal type at delayed iterations (when the signal has not yet reached the module) is always assumed to be '`osignal_null`', although the actual signal representation could be in Sample mode.

A special mode of VPIphotonicsAnalyzer (described under "Signal Structure" in Chapter 10, "Analyzing Simulation Results" of *VPIphotonics Design Suite™ Simulation Guide*) allows you to actually look into the ONSL signal representation of any signal at its input. This mode can be useful when debugging cosimulation code in VPI Design Suite, as it enables you to look at the details of the signal structures being passed into and out of your cosimulation module. Figure 2-2 shows an example of the signal viewed in the Signal Structure mode.

Note: VPIphotonicsAnalyzer does not show individual samples (these are packed into blocks before visualization), nor does it support 'special' signals, like null signals.



SignalAnalyzer_vtms1

Object	Value	Unit
Input 1		
Optical Signals		
Center Frequency	1.933e14	Hz
Bandwidth	5.6e11	Hz
Duration	2e-10	s
Sampled Bands		
Sampled Band fc=1.931e14		
Sampled Band fc=1.935e14		
Noise Bins		
Noise Bin fc=1.931e14		
Statistical		
Distortion Crosstalk fc=1.931e14		
Distortion Rayleigh fc=1.9317e14		
Distortion Brillouin fc=1.9324e14		
Distortion Generic fc=1.9331e14		
Distortion FWM fc=1.9338e14		
Logical Channels		
Logical Channel		
Center Frequency	1.9338e14	Hz
Bandwidth	1.6e11	Hz
Duration	2e-10	s
Label	Array14765Channel1	
Sequence Numbers	0	
ModulationASK_NRZ		
Signal Source		
Modulators		

Figure 2-2 *An optical signal shown in the Signal Structure mode of VPIphotonicsAnalyzer*

Similar functionality is provided by the obsolete module *DebugInfoOutputTk* located in the *Analyzers* folder.

Optical signals in Sample mode are represented as shown in Table 2-1.

Table 2-1 *Representation of optical signals (Sample mode)*

Struct Entry	Type	Description
`type`: 'osample'; 'osample_null'	string	signal type: optical sample; signal produced by inactive module or module *NullSource*
`dt`	float	time grid spacing (1/**`SampleRate`**)
`df`	float	frequency grid spacing (1/**`TimeWindow`**)
`t0`	int	timestamp of signal (grid points)
`fc`	int	center frequency (grid points)
`fs`	int	sample rate, bandwidth (grid points)
`Ex`	complex	x-polarization of optical field
`Ey`	complex	y-polarization of optical field

Distortions are contained in the '`channels`' array of the optical signal block representation. Their structure is identical to the structure of parameterized signals, as described in Table 2-4. The differences are that the `type` field contains the distortion type (either `DistortionRayleigh`, `DistortionBrillouin`, `DistortionGeneric`, `DistortionFWM`, or `DistortionCrosstalk`).

Table 2-2 *Representation of optical signals (Block mode)*

Struct Entry	Type	Description
`type`: 'osignal'; 'osignal_null'	string	signal type: optical signal; signal produced by inactive module or module *NullSource*
`boundaries`	string	boundary conditions of the signal
`dt`	float	time grid spacing (seconds)
`df`	float	frequency grid spacing (Hz). `df`=1/TimeWindow
`t0`	int	time stamp (start time) of signal (grid points)
`T`	int	duration of signal (grid points)
`f0`	int	lower frequency limit (grid points)
`f1`	int	upper frequency limit (grid points)
`noise`	array	contains noise bins if present (see Table 2-5)
`channels`	array	contains parameterized signals if present (see Table 2-4)
`bands`	array	contains sampled bands if present (see Table 2-3)

Additionally, for distortions, the fields *AveragePulsePosition*, *ExtinctionRatio*, and *PulsePositionVariance* are set to zero and all the tracking parameters are empty arrays.

Table 2-3 *Sampled bands (Block mode)*

Struct Entry	Type	Description
`type`: 'oband'	string	signal type: sampled band
`f0`	int	lower frequency limit (grid points)
`f1`	int	upper frequency limit (grid points)
`azi`	float	azimuth of polarization ellipse in degrees
`ell`	float	ellipticity of polarization ellipse in degrees
`E`	complex array	optical field in case of fully polarized signal
`Ex`	complex array	x-polarization of optical field

Table 2-3 *Sampled bands (Block mode)*

Struct Entry	Type	Description
`Ey`	complex array	y-polarization of optical field
`labelset`	array of 'Label' structures	label(s) of the logical channel(s) attached to the optical band

Table 2-4 *Parameterized channels (Block mode)*

Struct Entry	Type	Description
`type`: 'ochannel'	string	signal type: parameterized channel
`f0`	int	lower frequency limit (grid points)
`f1`	int	upper frequency limit (grid points)
`S`	float array	Stokes vector, not normalized
`AveragePulsePosition`	float	Average position of a pulse
`ExtinctionRatio`	float	Extinction ratio of the signal
`PulsePositionVariance`	float	The variance of a pulse position
`labelset`	array of 'Label' structures	label(s) of the logical channel(s) attached to the signal
`Event`	int array	The ordinal number of a module on the signal path
`Entry`	int array	Number of entry
`PhysicalPath`	float array	Total length of fiber the signal has passed
`TopologicalPosition`	float array	Topological position of the signal
`TransitTime`	float array	Time of signal propagation through a link
`Path`	string array	Array with IDs of the modules the signal has passed
`AccumulatedGVD`	float array	Accumulated dispersion of the signal
`Power`	float array	Power of the optical signal
`S1, S2, S3`	float arrays	Components of not normalized Stokes vector
`NonlinearCoefficient`	float array	Coefficient of fiber nonlinearity
`AccumulatedDGD`	float array	Accumulated differential group delay of the signal
`SPM`	float array	Phase changes of the optical signal due to self-phase modulation
`Bo`	float array	Optical bandwidths
`Be`	float array	Electrical bandwidths

Table 2-4 *Parameterized channels (Block mode)(Continued)*

Struct Entry	Type	Description
`NoisePSD`	float array	Power spectral density of the optical noise at the signal wavelength
`NoisePower`	float array	Optical noise power in the optical bandwidth Bo
`DistCohTotalPower`	float array	Total power of the coherent distortions located in the electrical bandwidth Be
`DistIncohTotalPower`	float array	Total power of the incoherent distortions in the whole spectral range outside Be

Table 2-5 *Noise bins (Block mode)*

Struct Entry	Type	Description
`type`: 'onoise'	string	signal type: Noise Bin
`f0`	int	lower frequency limit (grid points)
`f1`	int	upper frequency limit (grid points)
`S`	float array	Stokes vector, not normalized

The signal structure described above applies when *single* port input/output modules are used to pass optical signals to and return them from the *CosimInterface* module. The single port input/output modules are *CoSimInputOpt* and *CoSimOutputOpt*, respectively. Their *multiple* port counterparts are *CoSimInputOpt_N* and *CoSimOutputOpt_N*, which support multiple optical signals arriving from several sources or a bus with a width greater than 1. In this case, the data passed to or received from cosimulation target is a cell array (MATLAB), list (Python), or special collection (COM). Each element of this container is the data corresponding to a single port signal as described above. Elements are numbered from `1` to `N` for MATLAB and from 0 to `N-1` for Python cosimulation, where `N` is the number of inputs or outputs.

Note: The *CoSimInputOpt_N* and *CoSimOutputOpt_N* modules are not currently supported by dynamic-link library cosimulation.

When using multiple port input/output modules, special variables are created in the cosimulation environment to store the numbers of inputs and outputs. "*Mapping VPI Signals to Matlab Data Types*" and "*Mapping VPI Signals to Python Data Types*" describe how to access these variables in MATLAB and Python cosimulations.

Electrical Signals

Electrical signals are represented in a very similar manner to optical signals, but are restricted to contain only one sampled band in a nested structure band (see Table 2-6).

Table 2-6 *Representation of electrical signals (Block mode)*

Struct Entry	Type	Description
`type`: 'esignal'; 'esignal_null'	string	signal type: electrical signal; signal produced by inactive module or module *NullSource*
`boundaries`	string	boundary conditions of the signal
`dt`	float	time grid spacing
`df`	float	frequency grid spacing (1/**`TimeWindow`**)
`t0`	int	start of time axis (grid points)
`T`	int	duration of signal (grid points)
`fs`	int	total signal bandwidth (grid points)
`band`	structure	sampled band (see Table 2-7)

Table 2-7 *Sampled bands (Block mode)*

Struct Entry	Type	Description
`type`: 'eband'	string	signal type: electrical sampled band
`fs`	int	sample rate (grid points)
`E`	float array	electrical signal as real valued baseband signal
`labelset`	array of 'Label' structures	label(s) of the logical channel(s) attached to the signal

Electrical signals in Sample mode are represented as shown in Table 2-8.

Table 2-8 *Representation of electrical signals (Sample mode)*

Struct Entry	Type	Description
`type`: 'esample'; 'esample_null'	string	signal type: electrical sample; signal produced by inactive module or module *NullSource*
`dt`	float	time grid spacing (1/**`SampleRate`**)

Table 2-8 *Representation of electrical signals (Sample mode)*

Struct Entry	Type	Description
`df`	float	frequency grid spacing (1/`TimeWindow`)
`t0`	int	timestamp of sample (grid points)
`fs`	int	sample rate, bandwidth (grid points)
`E`	float	real valued sample of signal

Logical Channels

The structure of the logical information that can be accessed from within the cosimulation environment is similar to the internal representation of the logical information in the simulation kernel (it can be visualized by VPIphotonicsAnalyzer in the Signal Structure mode or via the obsolete *DebugInfoOutputTk* module). The logical information is associated with logical channels.

The logical channels represent a nested structure as shown in Table 2-9. There is one global array of logical channels. It can be accessed by the variable `lchannels` in the MATLAB or Python environment. In library cosimulation, the structure *ONSL_LogicalInformation* holds the list of all logical channels. In COM cosimulation, logical information is accessible via the cosimulation context object.

Table 2-9 *Logical channel structure*

Struct Entry	Type	Description
`type`: '`lchannel`'	string	
`label`	string	label string of logical channel
`seqnumber`	int	sequence number, is used to discriminate between logical signal blocks belonging to different time instants of one channel
`modulateelabel`	struct of type 'Label'	label specifying logical channel representing the signal that was input as modulated signal to modulation of this logical channel (if present). 'info' field = "Modulatee"
`modulatorlabel`	struct of type 'Label'	label specifying logical channel representing the signal that was input as modulating signal to modulation of this logical channel (if present). 'info' field = "Modulator"
`signalsource`	struct of type 'SignalSource'	describes source of the signal (if present).
`modulation`	struct of type 'Modulation'	describes modulation type (if present)

Table 2-9 *Logical channel structure (Continued)*

Struct Entry	Type	Description
`pulseshape`	struct of type 'PulseShape'	describes shape of the pulse (if present)
`bitstream`	int array (has zero size if absent)	describes bitstream (if present)
`linecoding`	struct of type 'Line-coding'	describes linecoding employed (if present)

The following tables describe the data that can be accessed in cosimulation through the structures listed in Table 2-9.

Table 2-10 *Structure containing information about SignalSource*

Struct Entry	Type	Description
`type`: 'source'	string	
`name`	string	source name (ID of source)
`dt`	float	time grid spacing (1/SampleRate)
`df`	float	frequency grid spacing (Hz). `df`=1/TimeWindow
`t0`	int	time stamp (start time) of signal (grid points)
`T`	int	duration of signal (grid points)
`f0`	int	lower frequency limit (grid points)
`f1`	int	upper frequency limit (grid points)

Table 2-11 *Structure containing information about modulation*

Struct Entry	Type	Description
`type`: 'modulation'	string	
`subtype`	string	Type of modulation. Can be ask, fsk, psk, plain, user
`bitrate`	int	bit rate of the modulated signal (in grid points)
`df`	float	frequency grid spacing (Hz). `df`=1/**TimeWindow**
`modulationindex`	float	modulation index of the modulated signal (if type == '**ask**')

Table 2-11 *Structure containing information about modulation (Continued)*

Struct Entry	Type	Description
`frequencydeviation`	float	frequency deviation of the modulated signal as frequency offset between spaces and marks in Hz (if type == '**fsk**')
`phasedeviation`	float	phase deviation of the modulated signal as phase offset between spaces and marks in degrees (if type == '**psk**')
`usersubtype`	string	Name of user-specified modulation
`numberofuser-parameters`	int	Number of parameters for user-specified modulation type
`usersparameters`	array of structs of type '`Userparameter`' that include struct entries '`userparameter`' containing the value of a user parameter, and 'type' containing the type of user parameter 'arrayofstrings'.	contains parameters for user-specified modulation type

Table 2-12 *Structure containing information about pulse shape*

Struct Entry	Type	Description
`type`: 'pulseshape'	string	
`subtype`	string	Type of pulse shape. Can be Rectangular, Stimulus, Gauss, Sech, RaisedCosine, RootRaisedCosine, Arbitrary
`bias`	float	bias of the pulse
`dutycycle`	float	duty cycle of the pulse (if type == 'Rectangular')
`shift`	float	shift of the pulse's high (if type == 'Rectangular')
`amplitude`	float	amplitude of the pulse (if type == 'Rectangular', 'Stimulus', 'Gauss', 'Sech', 'RaisedCosine', 'Root-RaisedCosine')
`centerpos`	float	center position of the pulse in seconds (if type == 'Stimulus', 'Gauss', 'Sech', 'RaisedCosine', 'Root-RaisedCosine')
`t_fwhm`	float	full width at half maximum of the pulse in seconds (if type == 'Gauss', 'Sech')
`extrabits`	float	maximum number of neighboring bits from each side which are taken into account for pulse overlap (if type == 'Gauss', 'Sech', 'RaisedCosine', 'RootRaisedCosine')

Table 2-12 *Structure containing information about pulse shape (Continued)*

Struct Entry	Type	Description
`order`	float	order of the gauss pulse (if type == 'Gauss')
`chirp`	float	chirp of the pulse (if type == 'Gauss')
`pulsewidth`	float	pulse width (if type == 'RaisedCosine', 'RootRaised-Cosine')
`alpha`	float	excess bandwidth of the pulse (if type == 'RaisedCo-sine', 'RootRaisedCosine')
`pulsename`	string	pulse name supplied by the user (if type == 'Arbi-trary')
`pulse`	complex array	pulse supplied by the user (if type == 'Arbitrary')
`duration`	int	duration of the pulse held in the array. This is inter-preted as bit duration (if type == 'Arbitrary')
`dt`	float	time grid spacing (1/TimeWindow) (if type == 'Arbi-trary')

Table 2-13 *Structure containing information about LineCoding applied to the signal*

Struct Entry	Type	Description
`type`: 'coding'	string	
`subtype`	string	Type of line coding. Can be Manchester, ami, Unipolar, Bipolar
`lastbit`	int	delay element's contents for differential Manchester coding (if type == 'Manchester', 'ami')
`isdifferential`	int	indicates if differential Manchester coding is to be employed (if type == 'Manchester').1 – yes, 0 - no

In addition to logical channels, the signal labels are stored in structures shown in Table 2-2. For parameterized signals, optical and electrical bands, additional fields are available:

Table 2-14 *Structure containing information about logical channel labels*

Struct Entry	Type	Description
`type`: 'label'	string	
`label`	string	Label of the optical channel
`seqno`	array	The sequence number (index) of the logical channel

Table 2-14 *Structure containing information about logical channel labels (Continued)*

Struct Entry	Type	Description
`info`	string	"modulatee" if the label is of modulatee channel, "modulator" if the label is of modulator channel, "" - other

Iteration (Sweep) Information

VPI Design Suite simulations support several levels of iterations, as described under "Hierarchy of Iterations, Runs and Sweeps" in Chapter 9, "Advanced Simulation Features" of the *VPIphotonics Design Suite™ Simulation Guide*. Information on all current iteration levels is available for modules as a special structure called iteration (or sweep) information, or `SweepInfo`. This information is used by modules like *Ramp* to control their functionality, like resetting the output value on a specific sweep level. Iteration information is also used in VPIphotonicsAnalyzer to handle signals using the iteration selector.

`SweepInfo` is available in cosimulation environments as an array of structures describing each iteration level:

Table 2-15 *Different iteration levels in* `SweepInfo`

Iteration level	Description
0	Module iterations
1	Runs
2	Sweep depth 0
3	Sweep depth 1
...	...
n	Sweep depth n-2

On each level, the `SweepInfo` contains a pair of integer values for the current iteration, run or sweep step number and for the total number, respectively:

Table 2-16 *Structure containing information about iteration level*

Struct Entry	Type	Description
`current`	int	Number of current iteration on some level
`total`	int	(Indicative) maximal number of iterations on some level

The `SweepInfo` structure follows several agreements on its values:

- The current and total values start from 1.
- The total number of *iterations* coincide with the current value (it is very hard to estimate the total number of iterations for a given module, especially in complex simulation setups; moreover, in some cases the total number of iterations is not known beforehand).
- The total value for *sweeps* can be less than the current value, especially for sweeps implemented using simulation scripts
(in fact, it is only the total number of runs that it is fixed in advance).
- The minimal number of levels in sweep information is 3 (iterations, runs and sweep with depth 0 are always included).

Function Parameters Evaluation

Function parameters are treated a little bit differently to regular (non-function) parameters. While regular parameters always have one representation (an expression), function parameters are numeric parameters whose value can be specified using one of three equivalent forms: an expression that evaluates up to a constant, a symbolic Python expression that contains variables like `$x`, and a path to a file from which value can be read. In cases of a Python expression and a path to a file, the parameters are interpreted by the cosimulation module in a specific way. For more details about function parameters, please refer to "Function Parameters" in Chapter 4 of the *VPIphotonics Design Suite™ Simulation Guide* and "Built-in Analytical Models of PIC Elements" in Chapter 5 of the *VPIcomponentMaker™Photonic Circuits User's Manual*.

If Python expression may be calculated (for example, it does not contain variables like `$x`) and its result is numerical, then the expression type of this parameter is transformed to the type `Standard` and the calculated value is provided to the cosimulation environment. Otherwise, the Python expression is passed as a string and it should be evaluated by the cosimulation code. To simplify this evaluation all cosimulation environments (except MATLAB) provide a special function.

One significant peculiarity which should be kept in mind is that Python expressions specifying function parameters can refer to functions and variables defined in the Python initialization script (please refer to "Initialization Scripts" in Chapter 4 of the *VPIphotonics Design Suite™ Simulation Guide* for more details).

Expressions are evaluated in the context of the init script of the parent resource of the instance of the cosimulation galaxy which the parameter belongs to. For example, in Figure 2-3 the cosimulation galaxy ***G1_vtmg1*** has two function parameters: **A** and **B**. Parameter **A** is specified by a Python expression which refers to the function `f_U1($F)`, defined in the init script of the universe `U1`. It is possible to evaluate this parameter in the cosimulation code of the *CoSimInterface* module ***C_vtms2***, while an attempt to evaluate the parameter **B** will produce an error because the function `f_G1($F)` referred to in the expression for **B** is not defined in the init script of the universe `U1`.

If the cosimulation interface is not wrapped into a galaxy and lies on the schematic, it uses the global parameters and they are evaluated in the scope of the init script of the universe (for example, see the *CoSimInterface* module ***C_vtms1*** in Figure 2-3).

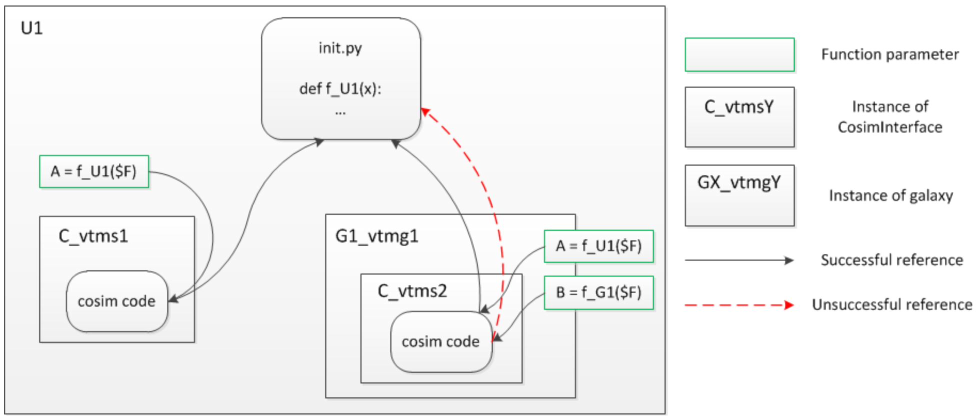


Figure 2-3 *Context of function parameters evaluation in a cosimulation environment*

All the same, it is frequently required to create deeply nested setups where expression from the upper level should be used in the parameter of the nested instance (for example, a cosimulation galaxy nested in another galaxy). To make this possible the expression shall be referred to using a *direct reference*. In this case it will be evaluated in the proper context using the expression type of the original parameter. For example, as shown in Figure 2-4, expressions for parameters **A** and **B** of the cosimulation galaxy *G3_vtmg1* will be evaluated in the cosimulation code in the context of the init script of the schematic *U1* because they are passed by direct references. An attempt to evaluate the parameter **C**, however, will produce an error because the universe parameter **C** is not referred to using direct reference but in the expression '{C}*2', and the function `f_U1($F)` referred to in the original expression is not defined in the init script of the galaxy `G2_vtmg1`.

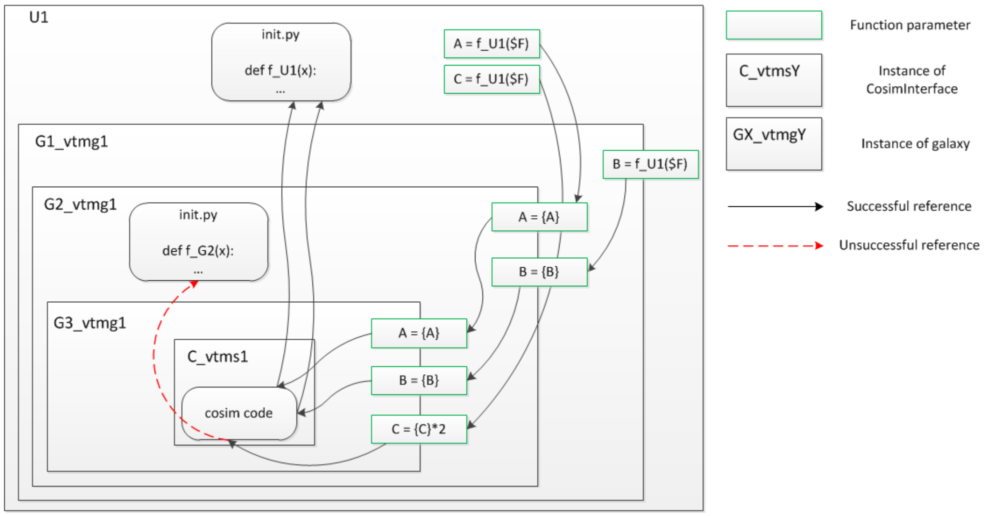


Figure 2-4 *Context of function parameters evaluation in cosimulations in case of direct reference*

Additional General Cosimulation Issues

Memory Management

The VPI Design Suite simulation system is responsible for releasing any resources utilized by the cosimulation interface. In particular, this includes the memory used to hold parameters, input data and output data in the target environment. Variables and structures holding input and output data are deleted after executing the interface's run function. Variables holding parameter values are deleted at shut-down of the interface. You should not rely on any variable provided by the cosimulation interface persisting between two calls to the target environment. You must exercise care when creating new structures to be passed out of the target environment, as described in the documentation below for each specific target environment.

Managing Variable Names

As has already been mentioned, *all* variable names must be unique, that is, the **`Name`** parameters of the input and output data interfaces, and the galaxy parameters must not contain any duplicate names.

However, it is only names that exist within the scope of the cosimulation interface that must be unique. Therefore, by encapsulating the cosimulation module within a galaxy, as has been recommended, you can ensure that the names you use will be unique regardless of the names used globally or by other modules in any simulation setup.

In effect, the galaxy enclosing the cosimulation interface creates an isolated *namespace* that prevents any clashes of names with other modules or cosimulation interfaces.

Cosimulation with MATLAB

This section describes the use of the cosimulation interface with MATLAB as the target environment.

System Requirements

To use the MATLAB cosimulation interface, you will need to have a properly licensed version of MATLAB installed on your system. In order to allow to execute the MATLAB engine, two special libraries provided by the MATLAB installation, known as *libeng* and *libmx*, must be accessible.

Linux Platforms

On Linux platforms, MATLAB version 7.9 or higher is required.

The path to the necessary MATLAB installation can be specified either via the `MATLAB` environment variable or on the **Cosimulation** tab of the Server Administration Tool.

The environment variable can be set by including the path to the MATLAB installation directory in the `<server_install_dir>/.profile` file:

```
MATLAB=/usr/local/matlab
export MATLAB
```

Under Linux, the MATLAB libraries `libeng.so` and `libmx.so` are located in the `bin/glnxa64` subdirectory. Thus for a typical setup, the path specified in the Server Administration Tool would be `/usr/local/matlab/bin/glnxa64/libeng.so`.

Windows Platform

On the Windows platform, MATLAB version 7.9 or higher is required. It provides the libraries `libeng.dll` and `libmx.dll`, which reside in a subfolder of the MATLAB binary directory, typically `C:\Program Files\MATLAB\R2019b\bin\win64`.

You may add this directory *(or the corresponding MATLAB path for your system)* to the `PATH` environment variable, which can be defined via **Control Panel > System and Security > System > Advanced system settings > Environment Variables**.

Alternatively, the path to `libeng.dll` may be specified in the **Cosimulation** settings on the **Simulation** tab of the application preferences.

Note: In order to use MATLAB cosimulations, the 64-bit version of MATLAB must be installed.

Setting Up the Cosimulation Interface

To choose MATLAB as the external program, set parameter **`InterfaceType`** of the *CoSimInterface* module to `Matlab`. In this mode, the interpretation of the various parameters of the cosimulation module is as follows:

- **`Path`**: the path provided here is added to the MATLAB search path. You should set this parameter to point to the directory containing the `.m` files implementing your MATLAB cosimulation code.
- **`InitCommand`**: set this parameter to the MATLAB command that should execute upon initialization of the interface. If left blank, no initialization command will be run.
- **`RunCommand`**: set this parameter to the MATLAB command to execute each time the module is fired during the simulation run.
- **`WrapupCommand`**: set this parameter to the MATLAB command to execute prior to shut-down of the interface.
- **`ShareInterface`**: if this parameter is set to `ON`, all instances of cosimulation interfaces in one setup will share the same MATLAB process. Thus only one MATLAB license is required, but care must be taken that these different interfaces do not

interfere. If set to `OFF`, each cosimulation interface will use its own MATLAB engine, which will require as many licenses as cosimulation interfaces.

Note: It is possible to attach the MATLAB script files to the cosimulation galaxy package and run them from the `Inputs` package folder. However, it is important to ensure that when there are multiple source scripts, all scripts that are not part of the standard installation, or installed on the default search path, are included in the `Inputs` folder. To use the attached scripts, leave the **`Path`** parameter blank.

- **`LogicalInformation`**: set this parameter to `Yes` to access the logical channels during the cosimulation. Note that the global parameter **`LogicalInformation`** should also be `On` or `CurrentRun`.

Simulation Job Variables

Several global variables are available to access the simulation job information like current iteration, sweep details, or whether the simulation is running at the Remote Simulation Server or locally (see "Iteration (Sweep) Information" on page 90).

To access the iteration information, use the following command:

```
global sweepinfo
```

In MATLAB, the sweep information is the cell array of the named structures. For example, the total number of runs can be found as

```
runs_total = sweepinfo{2}.total;
```

Note: In MATLAB, all arrays are unity-based.

As another example, the current sweep step number in zero depth is

```
sw0 = sweepinfo{3}.current;
```

To access the current run number, the following should be declared:

```
global runnumber
```

Note: This variable is supported but deprecated in favor of `sweepinfo`.

To understand whether a simulation is running at the Remote Simulation Server, you can use the `isremotesimulation` global variable:

```
global isremotesimulation
```

Handling Parameters

Parameters are passed to the MATLAB cosimulation environment as variables. Regular (non-function) parameters are mapped to native MATLAB types like `double`. Function parameters are represented as a structure with the following fields:

```
{type, expr_type, value}
```

where `type` is a parameter type (for example, `'floatarray'`), `expr_type` is an expression type of the function parameter (`'standard'`, `'python'`, or `'file'`), and `value` is the value of the parameter.

An expression type and value follow the rules provided under "Function Parameters Evaluation" on page 91. The MATLAB cosimulation environment has no means to evaluate Python expressions.

There are schematic parameters — **`TimeWindow`**, **`SampleModeBandwidth`**, **`SampleModeCenterFrequency`**, **`GreatestPrimeFactorLimit`**, and **`BoundaryConditions`** — available inside the MATLAB cosimulation environment after declaring them as global in the script, for example,

```
global topology_TimeWindow
```

The parameter **`ifc_name`** is available after declaring it as a function parameter, for instance,

```
function output=MyFunction(input,ifc_name).
```

Mapping VPI Signals to MATLAB Data Types

The MATLAB cosimulation interface uses the MATLAB *structure* data type to map ONSL internal signals. The arrays of sampled bands, parameterized signals, and noise bins in the structure representing an optical block signal are implemented as *cell arrays*.

All elements of all structures must be created. Even if arrays are not required, empty arrays must be declared and assigned. For example, the second component of the Stokes vector of the 42nd noise bin in the optical signal `sig` can be referenced using the expression

```
sig.noise{42}.S[2]
```

When using multiple port input/output modules to pass optical signals to and from the cosimulation environment (as described on page 84), the corresponding structure type is *struct array* in MATLAB notation and the same component of noise bin in the *third input* can be referenced as

```
sig(3).noise{42}.S[2]
```

The numbers of input and output signals can be accessed via the `input_buswidth` and `output_buswidth` variables. For example, the lines

```
NumberOfInputs=input_buswidth.input1
NumberOfOutputs=output_buswidth.output3
```

give the number of inputs and number of outputs to the modules *CoSimInputOpt_N* and *CoSimOutputOpt_N*, respectively with the **Name** parameter set to `input1` and `output3`. To access these variables, they must be declared as global in *both* the **`InitCommand`** of the *CoSimInterface* module and in the MATLAB script:

```
global input_buswidth output_buswidth;
```

To access the logical channels, the global variable should be declared:

```
global lchannels
```

After this declaration, this variable will refer to the global list. And, for example, it is possible to get the label of the modulatee channel:

```
lchannels{1}.modulateelabel.label
```

Debugging MATLAB Cosimulation Code

When developing cosimulation scripts, it is frequently necessary to debug MATLAB code to spot and fix errors of the algorithm. Currently, MATLAB does not support debugging in 'embedded mode', that is, when MATLAB runs from other software, such as VPIphotonics Design Suite™.

However, there are several approaches that prove helpful when analyzing and debugging MATLAB cosimulation code, as described below.

Printing Messages

The simplest approach is to print the necessary variables, using one of the MATLAB functions `disp()` or `fprintf()`, or just omitting a semicolon at the end of the expression:

```
y = x*z
```

When the MATLAB cosimulation (single iteration of the cosimulation module) finishes, all the text from the MATLAB console is copied to the **Message Log** of the Photonic Design Environment.

In many cases, examination of such logging is enough to spot an error in cosimulation code.

Plotting Data

On the other hand, cosimulation normally processes large arrays of data, for example, optical or electrical signals in sampled representation. The examination of individual values of such arrays is uninformative.

As a workaround, such data can be displayed graphically — for example,

```
plot(ElecSignalIn.band.E);
```

will plot the electrical signal contained in variable `ElecSignalIn`, as shown in Figure 2-5.

Note, until the MATLAB figure window is closed, the simulation status remains 'running', and another simulation usually can not be started (depending on available licenses and simulation settings).

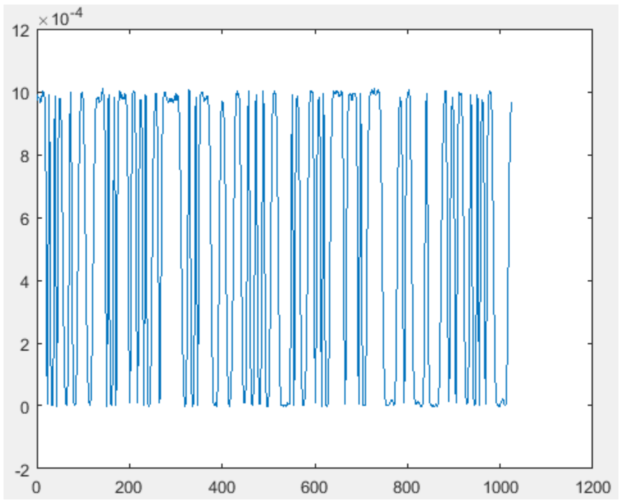


Figure 2-5 *Example of electrical signal plotted using MATLAB*

Please check the following demos for further examples:

Optical Systems Demos > Long Haul > Collision-Induced Jitter (Matlab)

Optical Systems Demos > Long Haul > Jitter versus Distance (Matlab)

Optical Systems Demos > Test & Measurement > Importing Data > Importing Signal (Matlab)

Offline Debugging

Finally, when the cosimulation code is very complex, or uses complex data structures, it is possible to save the MATLAB variables (workspace) within a function and then debug the code in the MATLAB Editor/Debugger environment.[1]

For example, for the following simple MATLAB (cosimulation) function:

```
function y = fun(x)
   y = 2*x;
```

1. The approach below is adapted from the documentation at http://www.mathworks.com/help/matlab/matlab_external/debugging-matlab-functions-used-in-engine-applications.html.

```
end
```

To debug fun, first it is necessary to modify the function to save the MATLAB workspace to the file debug_fun.mat.

```
function y = fun(x)
   save c:\MyProjects\debug_fun.mat
   y = 2*x;
end
```

Note: As VPIphotonics Design Suite™ uses a special temporary folder for simulation, it is recommended to specify the absolute path to the target .mat file.

If the cosimulation uses global variables, such as lchannels or runnumber, the save command should be placed *after* the declaration of global variables.

Run the schematic with cosimulation, then start MATLAB and load debug_fun.mat:

```
load debug_fun.mat
```

(or just drag the .mat file onto MATLAB Editor/Debugger).

Variable x should now be available in the Workspace pane of MATLAB Editor/Debugger and contain the value from the VPIphotonics Design Suite™ Simulation Engine.

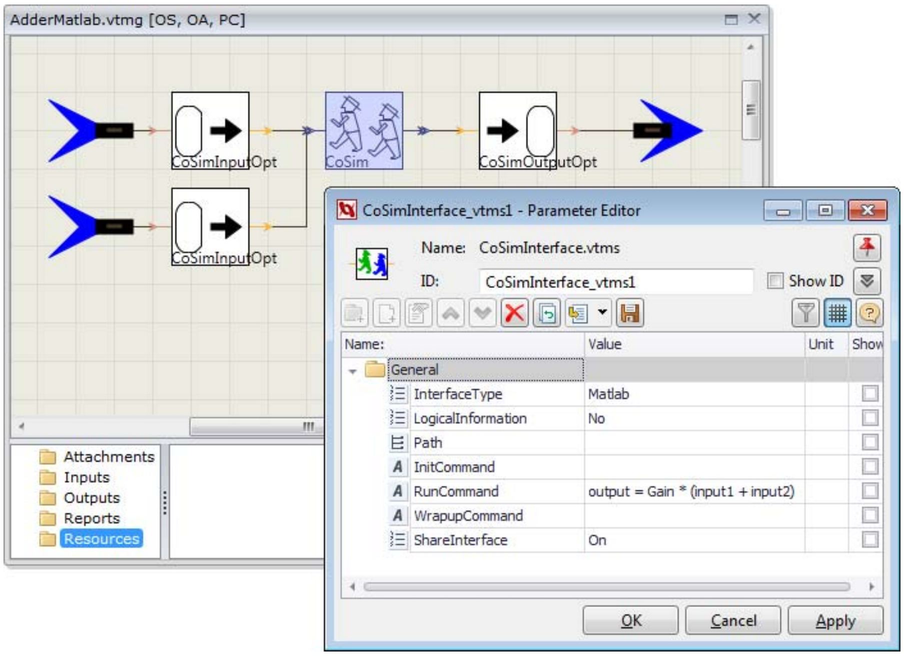
To debug fun(), open the function in the MATLAB Editor/Debugger, and then call the function from the MATLAB command line:

```
myfun(x)
```

Using the MATLAB Cosimulation Interface

A Simple Example: Adding Two Floating-Point Values

A very simple example is a cosimulation module that takes two floating-point value inputs, adds them and scales the result by a gain that is provided as a parameter. The output is then passed out of the cosimulation interface. Figure 2-6 shows the setup of the module. The parameter **Name** of the upper input declaration module has been set to input1, and that of the lower module to input2. The **Name** of the output module is output. The galaxy has a parameter **Gain**, which is set by default to 1.0.



Name: CoSimInterface.vtms
ID: CoSimInterface_vtms1

Name:	Value	Unit	Show
General			
InterfaceType	Matlab		☐
LogicalInformation	No		☐
Path			☐
InitCommand			☐
RunCommand	output = Gain * (input1 + input2)		☐
WrapupCommand			☐
ShareInterface	On		☐

Figure 2-6 *Cosimulation galaxy setup for the MATLAB adder module*

The parameter **RunCommand** of the *CoSimInterface* module is set to

```
output = Gain * (input1 + input2)
```

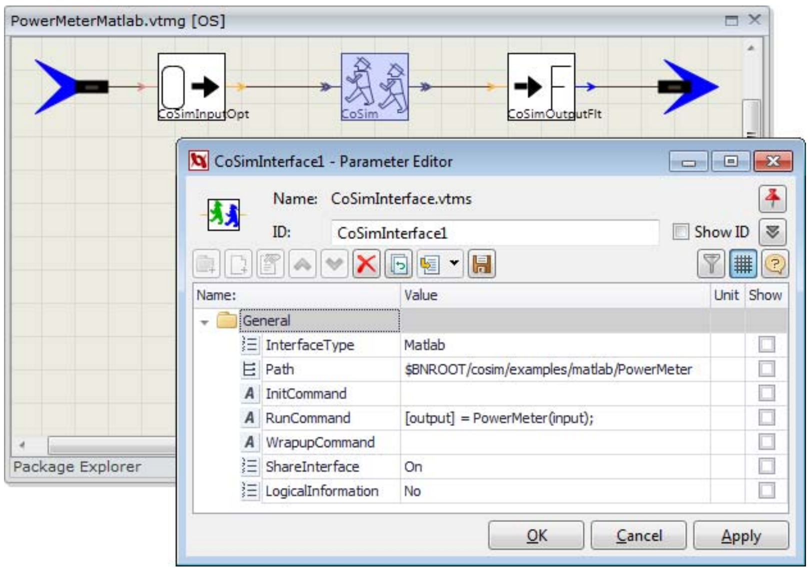
Using this galaxy, any two input sequences are added and scaled accordingly. Note that since this simple example can be implemented using a single line of MATLAB code, it is not necessary to create any `.m` files, or to set the value of the **Path** parameter.

Example 1 — Calculating the Total Optical Power of an Input Signal using MATLAB

Another example of using MATLAB functions in VPI Design Suite is demonstrated in the setup shown in Figure 2-7. The figure shows the cosimulation galaxy used for calculating the total power of an optical input. The galaxy has one optical input and returns a float value output. Therefore the data interface modules *CoSimInputOpt* and *CoSimOutputFlt* have been used to convert the VPI Design Suite signal representation into the appropriate MATLAB form and vice versa. The RunCommand of the *CoSimInterface* is set to

```
[output] = PowerMeter(input);
```

Hence, the MATLAB function `PowerMeter.m` located in the directory defined by the parameter *CoSimInterface* > Path will be executed with the parameter **input**. The variable *input* represents the optical signal passed in to the *CoSimInterface* module by the module *CoSimInputOpt.* The output of the MATLAB function is written into the variable *output*. The square brackets around [*output*] imply that the data handed over might be an array (although in this case it is not). The *output* is then passed on to the module *CoSimOutputFlt* where it is converted into VPI Design Suite float values.



Name: CoSimInterface.vtms

ID: CoSimInterface1

Name:	Value	Unit	Show
General			
InterfaceType	Matlab		☐
Path	$BNROOT/cosim/examples/matlab/PowerMeter		☐
InitCommand			☐
RunCommand	[output] = PowerMeter(input);		☐
WrapupCommand			☐
ShareInterface	On		☐
LogicalInformation	No		☐

Figure 2-7 *Example of the cosimulation interface galaxy for implementing a power meter in MATLAB. The parameter settings of the CoSimInterface are shown.*

The MATLAB function `PowerMeter.m` that actually performs the calculation is shown in Figure 2-8. This code can be found in the subfolder `simeng\cosim\examples\matlab\PowerMeter` of the VPI Design Suite installation directory. The function sums the power of all different signal representations used in VPI Design Suite, that is, optical sampled bands, parameterized signals and noise bins, and returns the calculated total power.

The optical power of the noise bins and parameterized channels is located in the first element of the Stokes vector, which is accessed in the MATLAB function by `input.noise{n}.S(1)` and `input.channels{channel}.S(1)`.

The power of each sampled band is calculated by averaging the square of the magnitude of the E-field vector over the total number of samples in the band. Note that the program has to distinguish between constant polarization signals (stored in the array `input.bands{band}.E`) and arbitrary polarization signals (stored in the arrays `input.bands{band}.Ex` and `input.bands{band}.Ey`). For constant polarization the `E` array contains the complex values of the E-field for all samples of the band and the `Ex` and `Ey` arrays will be empty.

On the other hand, when the band is arbitrarily polarized the information about the E-field will be stored in the `Ex` and `Ey` arrays and the `E` array will be empty. The test that distinguishes between these two cases is at line 16 of `PowerMeter.m`. Note that the MATLAB expression

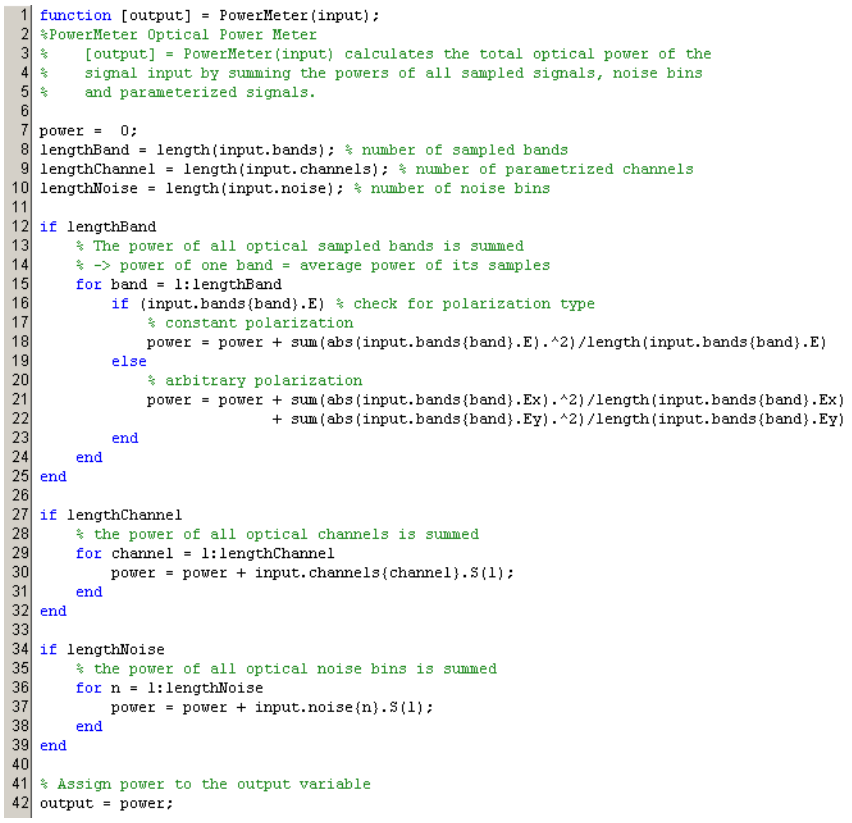
```
    if (input.bands{band}.E)
```

is equivalent to

```
    if length(input.bands{band}.E) > 0
```

and you will see both forms used in VPI examples. That is, the array E always exists, but will have zero length if it is unused. Indeed, you may always assume that if a structure exists, all fields within that structure will exist, although in the case of arrays and cell arrays, they may have zero size.

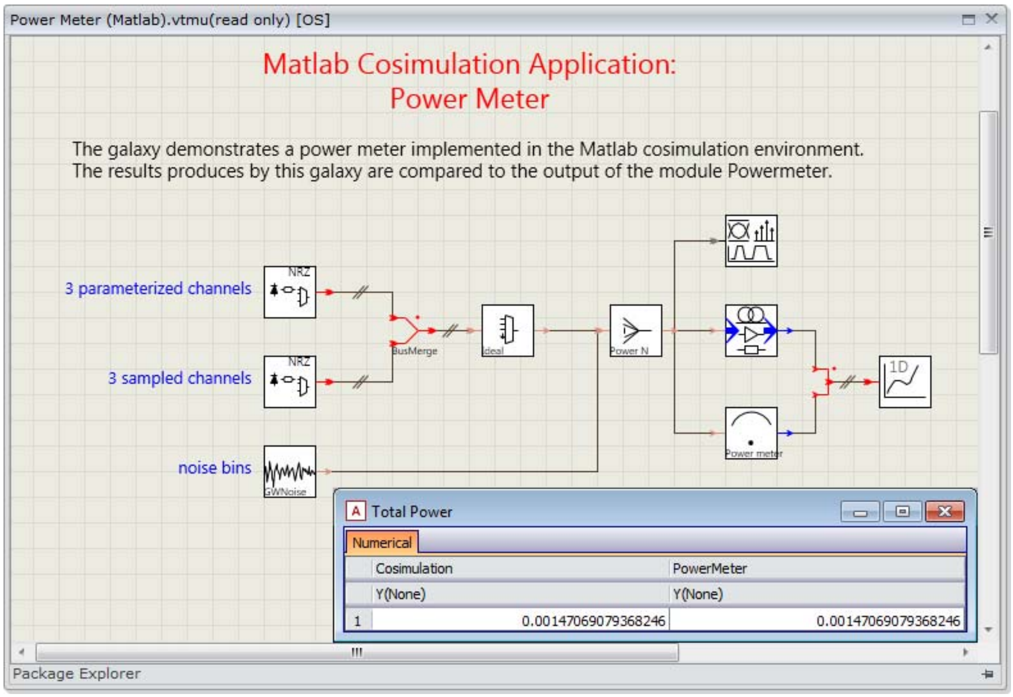
The power in the E array is calculated in line 18 if it was nonempty. Otherwise the power in the two arrays Ex and Ey is calculated in lines 21 and 22.



```
 1 function [output] = PowerMeter(input);
 2 %PowerMeter Optical Power Meter
 3 %    [output] = PowerMeter(input) calculates the total optical power of the
 4 %    signal input by summing the powers of all sampled signals, noise bins
 5 %    and parameterized signals.
 6
 7 power =  0;
 8 lengthBand = length(input.bands); % number of sampled bands
 9 lengthChannel = length(input.channels); % number of parametrized channels
10 lengthNoise = length(input.noise); % number of noise bins
11
12 if lengthBand
13     % The power of all optical sampled bands is summed
14     % -> power of one band = average power of its samples
15     for band = 1:lengthBand
16         if (input.bands{band}.E) % check for polarization type
17             % constant polarization
18             power = power + sum(abs(input.bands{band}.E).^2)/length(input.bands{band}.E)
19         else
20             % arbitrary polarization
21             power = power + sum(abs(input.bands{band}.Ex).^2)/length(input.bands{band}.Ex)
22                           + sum(abs(input.bands{band}.Ey).^2)/length(input.bands{band}.Ey)
23         end
24     end
25 end
26
27 if lengthChannel
28     % the power of all optical channels is summed
29     for channel = 1:lengthChannel
30         power = power + input.channels{channel}.S(1);
31     end
32 end
33
34 if lengthNoise
35     % the power of all optical noise bins is summed
36     for n = 1:lengthNoise
37         power = power + input.noise{n}.S(1);
38     end
39 end
40
41 % Assign power to the output variable
42 output = power;
```

Figure 2-8 *MATLAB function PowerMeter.m. It calculates the total power of all used optical bands, channels and noise.*

The demonstration *Power Meter (Matlab)* in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Matlab* illustrates this example in operation. In the demonstration, an optical signal is created that consists of a combination of parameterized signals, sampled bands, and noise bins. The total power in this optical signal is calculated using the MATLAB code, and using the standard *Powermeter* module. Figure 2-9 shows the result of running this demonstration, illustrating that the MATLAB script arrives at the same result as the standard power meter.


Power Meter (Matlab).vtmu(read only) [OS]

Matlab Cosimulation Application:
Power Meter

The galaxy demonstrates a power meter implemented in the Matlab cosimulation environment.
The results produces by this galaxy are compared to the output of the module Powermeter.

Total Power

Numerical

	Cosimulation	PowerMeter
	Y(None)	Y(None)
1	0.00147069079368246	0.00147069079368246

Package Explorer

Figure 2-9 *Results of running the Power Meter (Matlab) demonstration*

Example 2 — Implementing an Optical Bandpass Filter Using MATLAB

The demonstration *Filter (Matlab)* in the demonstrations *folder Optical Systems Demos > Simulation Techniques > Cosimulation > Matlab* is a more elaborate example that illustrates processing of an optical input signal to produce an optical output signal, and using multiple input ports of different types. This demonstration implements a Gaussian bandpass filter for optical signals using a MATLAB script.

The setup of the MATLAB optical filter demonstration is shown in Figure 2-10. As in the power meter demonstration, an optical signal consisting of a combination of parameterized signals, sampled bands, and noise bins is generated. The two inputs to the cosimulation module are the optical signal, and a floating-point value representing the center frequency of the filter. The cosimulation galaxy is shown in Figure 2-11, along with its parameters. The bandwidth of the filter is set using a parameter. Of course, the center frequency could also be set using a parameter, and in most cases this would be the most logical approach. However, this example is designed to illustrate the different options available, and to demonstrate the use of multiple inputs of differing types.

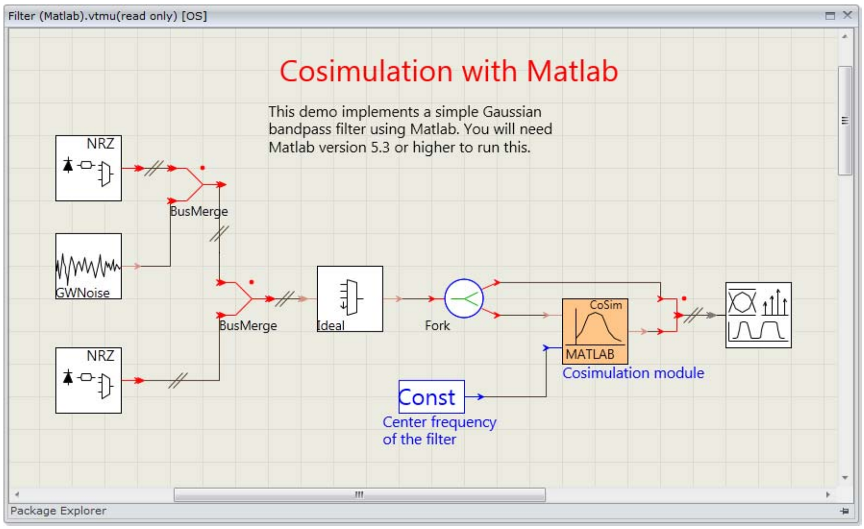
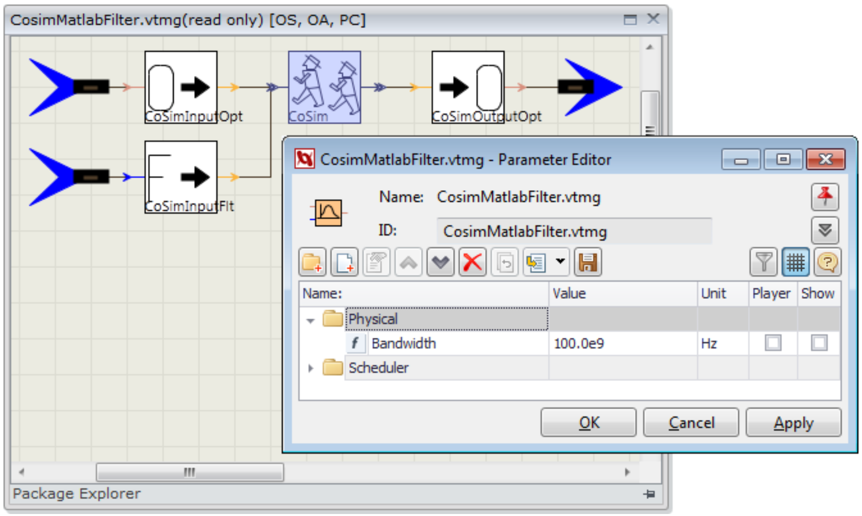

Filter (Matlab).vtmu(read only) [OS]
Cosimulation with Matlab
This demo implements a simple Gaussian bandpass filter using Matlab. You will need Matlab version 5.3 or higher to run this.
NRZ
GWNoise
NRZ
BusMerge
BusMerge
Ideal
Fork
CoSim
MATLAB
Cosimulation module
Const
Center frequency of the filter
Package Explorer

Figure 2-10 *Setup for the MATLAB optical filter demonstration*


CosimMatlabFilter.vtmg(read only) [OS, OA, PC]
CoSimInputOpt
CoSim
CoSimOutputOpt
CoSimInputFlt
Package Explorer
CosimMatlabFilter.vtmg - Parameter Editor
OK
Cancel
Apply

Name: CosimMatlabFilter.vtmg
ID: CosimMatlabFilter.vtmg

Name:	Value	Unit	Player	Show
Physical				
Bandwidth	100.0e9	Hz	☐	☐
Scheduler				

Figure 2-11 *Cosimulation galaxy for the MATLAB optical filter*

The MATLAB files `filterSignalGaussBP.m` and `gaussBP.m` that implement the optical filter can be found in the subfolder `simeng\cosim\examples\matlab\GaussBPFilter` of the VPI Design Suite installation directory – under Windows, typically

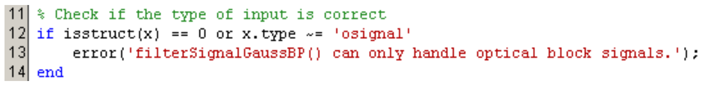
```
C:\Program Files\VPI\VPIdesignSuite 11.5
```

The file `gaussBP.m` is a simple function implementing the Gaussian filter function itself. It is called as required by the code in `filterSignalGaussBP.m`, which is the main MATLAB program executed by the *CoSimInterface* module using the **RunCommand**

```
output = filterSignalGaussBP(input,centerFrequency,Bandwidth);
```

Note that in this function call, `input` is the MATLAB structure representing the optical signal defined by the *CoSimInputOpt* module, `centerFrequency` is the MATLAB variable defined by the *CoSimInputFlt* module, while `Bandwidth` is the MATLAB variable that is created from the galaxy parameter **`Bandwidth`**. In MATLAB cosimulation, there is no difference from the point of view of the MATLAB code, between variables created from inputs, and variables created from galaxy parameters.

Although not always necessary when creating cosimulation module purely for your own use, it is good practice to check on input to a cosimulation function, that the types of the inputs being passed are the expected types. Figure 2-12 illustrates how this is done in MATLAB code for the optical block mode signal expected as input to the filter. Two consecutive checks are done at line 12. The first test confirms that the input variable `x` is, indeed, a structure. The second test checks that the `type` field of the structure is equal to the string '`osignal`' which identifies it as an optical signal (see Table 2-2). If either of these tests fail, an error is generated.

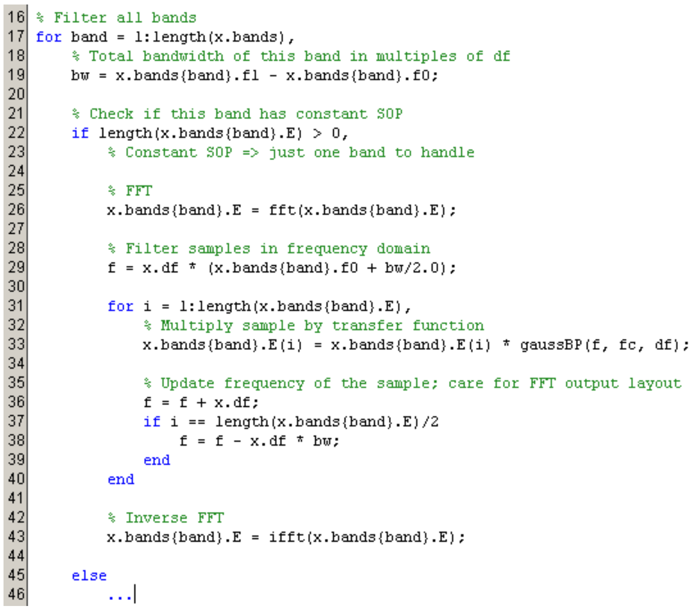


```
11 % Check if the type of input is correct
12 if isstruct(x) == 0 or x.type ~= 'osignal'
13     error('filterSignalGaussBP() can only handle optical block signals.');
14 end
```

Figure 2-12 *Optical block signal type checking code in MATLAB*

Note: The cosimulation interface sends all error messages to the VPI Design Suite PDE and they are displayed there like ordinary simulation errors. Moreover, when MATLAB code produces any textual output (like those produced by the MATLAB `disp` function or a statement without a terminating semicolon), the cosimulation interface displays all output in the VPI Design Suite **Message Log**. However, the output is sent to the message window only when the MATLAB code has finished running. More advanced messaging functions are available in Python cosimulation, as described under "Errors and Warning Messages in Python Code" on page 116.

The fragment of MATLAB code that implements the filtering of sampled bands with constant polarization is shown in Figure 2-13.



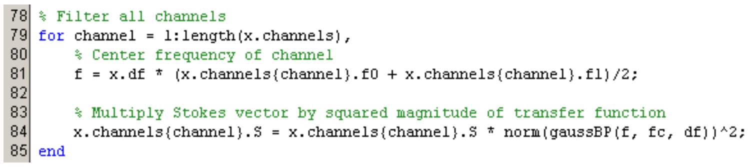
```
16 % Filter all bands
17 for band = 1:length(x.bands),
18     % Total bandwidth of this band in multiples of df
19     bw = x.bands{band}.f1 - x.bands{band}.f0;
20
21     % Check if this band has constant SOP
22     if length(x.bands{band}.E) > 0,
23         % Constant SOP => just one band to handle
24
25         % FFT
26         x.bands{band}.E = fft(x.bands{band}.E);
27
28         % Filter samples in frequency domain
29         f = x.df * (x.bands{band}.f0 + bw/2.0);
30
31         for i = 1:length(x.bands{band}.E),
32             % Multiply sample by transfer function
33             x.bands{band}.E(i) = x.bands{band}.E(i) * gaussBP(f, fc, df);
34
35             % Update frequency of the sample; care for FFT output layout
36             f = f + x.df;
37             if i == length(x.bands{band}.E)/2
38                 f = f - x.df * bw;
39             end
40         end
41
42         % Inverse FFT
43         x.bands{band}.E = ifft(x.bands{band}.E);
44
45     else
46         ...
```

Figure 2-13 *MATLAB code fragment that filters fully polarized optical sampled bands*

Recall that a constant polarization optical band comprises a single array of complex samples — the code that filters an arbitrarily polarized band is similar, except that both arrays are filtered. This code fragment illustrates a number of important principles. Lines 19, 29, 36 and 38 demonstrate one approach to the use of quantized frequencies. The total bandwidth of the signal is stored as an integer `bw` (line 19) representing the number of frequency grid steps. The physical center frequency of the band (in Hz) is computed (line 29) as the initial value of the optical frequency `f`. Filtering is performed in the frequency domain by multiplying each component of the discrete Fourier transform of the sampled band (computed using the FFT at line 26) by the filter amplitude (line 33). At each iteration, the current frequency is incremented by the basic frequency quantization grid spacing `df` (line 36). The standard layout of the array containing the FFT result consists of the samples corresponding to all frequencies from the center frequency up to the maximum frequency first, followed by the samples corresponding to the minimum frequency up to the center frequency. Thus, at the half-way point, the physical frequency reaches the maximum frequency, and must be reduced to the minimum frequency by subtracting the bandwidth. This is done using the quantized bandwidth `bw` multiplied by the grid spacing `df` (line 38).

Note that the filtering process is iterated over all sampled bands present using the loop variable `band`. Thus, for example, the expression `x.bands{band}.E` refers to the `E` array (fully polarized sample band), of sampled band number `band`, where the sampled bands are all stored in the cell array `bands` of the optical input signal `x`.

The MATLAB code fragment that filters parameterized signals is shown in Figure 2-14. Again, it consists of a loop over all parameterized signals present in the input. For each channel, the center frequency is computed (line 81), taking into account the quantization of all frequencies. The filtering consists of multiplying all components of the Stokes vector by the squared-magnitude of the filter transfer function at the center frequency. Note that the code at line 84 multiplies all four elements of the array `x.channels{channel}.S` by the scalar value `norm(gaussBP(f, fc, df))^2`. The process for filtering noise bins is identical, because noise bins are described using exactly the same parameters as parameterized signals.



```
78 % Filter all channels
79 for channel = 1:length(x.channels),
80     % Center frequency of channel
81     f = x.df * (x.channels{channel}.f0 + x.channels{channel}.f1)/2;
82
83     % Multiply Stokes vector by squared magnitude of transfer function
84     x.channels{channel}.S = x.channels{channel}.S * norm(gaussBP(f, fc, df))^2;
85 end
```

Figure 2-14 *MATLAB code fragment that filters parameterized signals*

The filtering is performed in-place on the input optical signal `x`, that is, the input signal is actually modified by the filter transfer function. In the final line of the MATLAB function, the modified input signal is simply copied to the output variable (`y = x`, at line 99). This way, it is not necessary to construct a new optical signal from scratch.

This technique is suitable whenever a cosimulation module performs linear operation(s) on an input signal to produce an output signal, which is a fairly common requirement.

Since MATLAB assignments have copy semantics, it is simple to create additional copies of the input (or any other) signal, simply by an assignment of the form

```
signalCopy = signal
```

This is useful when multiple outputs must be generated from a single input, or when an unmodified copy of the input is required during the computations, as well as a modified copy. However, care should be taken to avoid unnecessary assignments: the consequence of a simple statement such as the above may be the allocation and copying of many parameterized signals, noise bins, and sampled bands, and each sampled band may consist of a large number of samples. This can have a significant impact on the efficiency of your cosimulation code.

Example 3 — Creating New ONSL Signals Using MATLAB

It is possible to create ONSL signal structures from scratch in MATLAB, so long as some simple rules are followed.

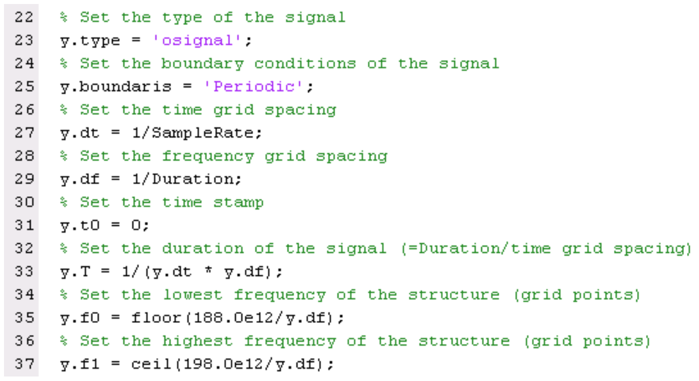
- All elements of all structures must be created. Even if arrays are not required, empty arrays must be declared and assigned. For example, a particular optical signal may contain no noise bins, and therefore may seem not to require a noise array. However, the correct operation of the cosimulation interface requires that a cell array must be created, even if it is empty. This can be done by creating and assigning an empty (that is, zero dimensioned) cell array. Similarly, Stokes vectors and sample arrays must be created, even if they are empty, for example, using a statement such as `y.bands{i}.Ex = []`.
- Scalar elements of the structures may be created either as integer or floating-point values. If integer quantities (such as the quantization multiples `t0`, `T`, `f0`, `f1` etc.) are

returned as floating-point numbers, they will be rounded to the nearest integer. Thus you need not worry about small rounding errors, or conversion to integer values, when working with these variables in MATLAB.

The demonstrations *Optical Signal Generation (Matlab)* and Electrical Signal Generation (Matlab) in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Matlab* are examples of how to create optical and electrical signals from scratch in MATLAB. The MATLAB source files for these examples can be found in the subfolders `simeng\cosim\examples\matlab\CreateOpticalSignal` and `simeng\cosim\examples\matlab\CreateElectricalSignal` of the VPI Design Suite installation directory.

Figure 2-15 illustrates the process of creating the main parameters of a new optical signal in MATLAB. Note that, according to the normal MATLAB rules, the structure `y` is not declared before use.

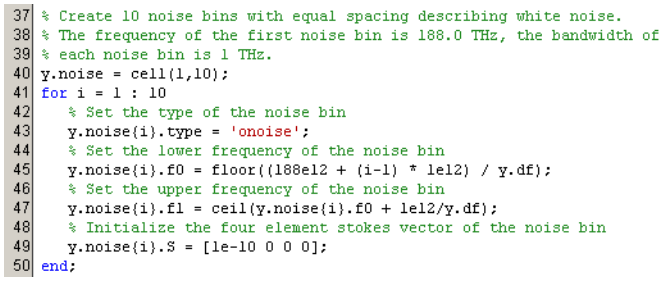
The parameter `T` is initialized using a floating-point expression — due to the constraints on `df` and `dt` it should be an integer; however, the value returned may not be an exact integer due to possible rounding errors. This will not cause any problems, according to the second rule above. The rounding functions `floor` and `ceil` are used when defining the frequency limits `f0` and `f1` in order to quantize the corresponding physical frequencies (188.0 THz and 198.0 THz) onto the global frequency grid defined by `df`.



```
% Set the type of the signal
y.type = 'osignal';
% Set the boundary conditions of the signal
y.boundaris = 'Periodic';
% Set the time grid spacing
y.dt = 1/SampleRate;
% Set the frequency grid spacing
y.df = 1/Duration;
% Set the time stamp
y.t0 = 0;
% Set the duration of the signal (=Duration/time grid spacing)
y.T = 1/(y.dt * y.df);
% Set the lowest frequency of the structure (grid points)
y.f0 = floor(188.0e12/y.df);
% Set the highest frequency of the structure (grid points)
y.f1 = ceil(198.0e12/y.df);
```

Figure 2-15 *Example of creating the parameters of an optical signal structure in MATLAB*

Following the declaration of the main parameters, noise bins, parameterized signals, and sampled bands must be created. Figure 2-16 illustrates the creation of ten noise bins. First, a cell array is created to hold the noise bin structures. Note that if no noise bins were to be created, it is necessary to create an empty cell array. This would be done using the statement `y.noise = cell(0,0)`. Then each noise bin is created in a loop. Note the creation of the Stokes vector as a MATLAB array.



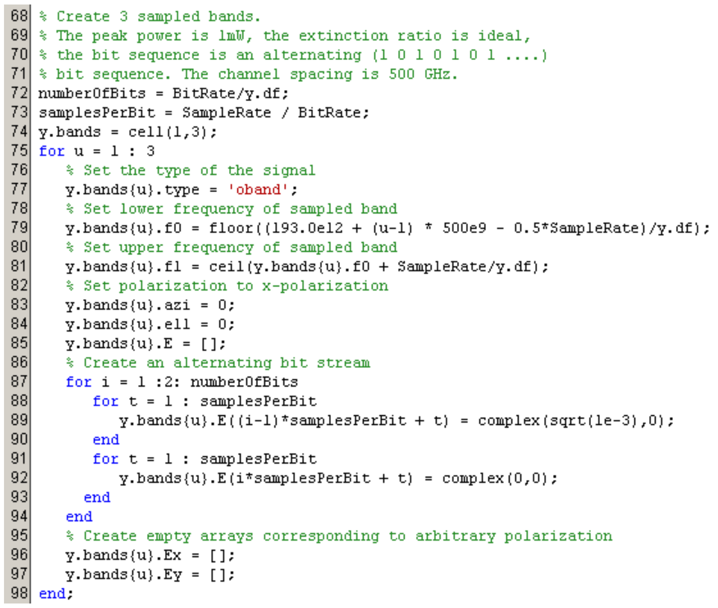
```
% Create 10 noise bins with equal spacing describing white noise.
% The frequency of the first noise bin is 188.0 THz, the bandwidth of
% each noise bin is 1 THz.
y.noise = cell(1,10);
for i = 1 : 10
   % Set the type of the noise bin
   y.noise{i}.type = 'onoise';
   % Set the lower frequency of the noise bin
   y.noise{i}.f0 = floor((188e12 + (i-1) * 1e12) / y.df);
   % Set the upper frequency of the noise bin
   y.noise{i}.f1 = ceil(y.noise{i}.f0 + 1e12/y.df);
   % Initialize the four element stokes vector of the noise bin
   y.noise{i}.S = [1e-10 0 0 0];
end;
```

Figure 2-16 *Example of creating noise bins in MATLAB*

After creating noise bins, parameterized signals must be created. The process is almost identical to that for creating noise bins — see the example code in `simeng\cosim\examples\matlab\CreateOpticalSignal` for details.

Finally, sampled bands must be created. This is illustrated in Figure 2-17. Again, a suitable cell array must be created first, and then all elements of the band structures must be created. Note that even though the arbitrary polarization sampled band arrays `Ex` and `Ey` are not used, corresponding empty arrays must be created.

For a new signal source, the logical information shown in Table 2-9 can also be added if it is required. Logical channels should at least contain a label string, sequence number (which is usually equal to the current run number), SignalSource structure (`f0`, `f1`, `t0`, `T` should be equal to the appropriate data of the physical signal) and Label (label string and sequence number of the logical channel should also be added to the label set of the new physical signal).



```
% Create 3 sampled bands.
% The peak power is 1mW, the extinction ratio is ideal,
% the bit sequence is an alternating (1 0 1 0 1 0 1 ....)
% bit sequence. The channel spacing is 500 GHz.
numberOfBits = BitRate/y.df;
samplesPerBit = SampleRate / BitRate;
y.bands = cell(1,3);
for u = 1 : 3
   % Set the type of the signal
   y.bands{u}.type = 'oband';
   % Set lower frequency of sampled band
   y.bands{u}.f0 = floor((193.0e12 + (u-1) * 500e9 - 0.5*SampleRate)/y.df);
   % Set upper frequency of sampled band
   y.bands{u}.f1 = ceil(y.bands{u}.f0 + SampleRate/y.df);
   % Set polarization to x-polarization
   y.bands{u}.azi = 0;
   y.bands{u}.ell = 0;
   y.bands{u}.E = [];
   % Create an alternating bit stream
   for i = 1 :2: numberOfBits
      for t = 1 : samplesPerBit
         y.bands{u}.E((i-1)*samplesPerBit + t) = complex(sqrt(1e-3),0);
      end
      for t = 1 : samplesPerBit
         y.bands{u}.E(i*samplesPerBit + t) = complex(0,0);
     end
   end
   % Create empty arrays corresponding to arbitrary polarization
   y.bands{u}.Ex = [];
   y.bands{u}.Ey = [];
end;
```

Figure 2-17 *Example of creating sampled bands in MATLAB*

Example 4 — Handling the Logical Information using MATLAB

Another example of using MATLAB functions in VPI Design Suite is demonstrated in the setup shown in Figure 2-18. The figure shows the cosimulation galaxy used for transferring a bit stream stored in a logical channel. The galaxy has one optical input and returns a float matrix output. Therefore the data interface modules *CoSimInputOpt* and *CoSimOutputMxFlt* have been used to convert the VPI Design Suite signal representation into the appropriate MATLAB form and vice versa. The **RunCommand** of the *CoSimInterface* is set to

```
output=LI_matlab(input);
```

Hence, the MATLAB function `LI_matlab.m` located in the directory `Inputs` of the Logical Info (Matlab) demo defined by the parameter *CoSimInterface* > **Path** will be executed with the parameter **input**. The variable *input* represents the optical signal passed in to the *CoSimInterface* module by the module *CoSimInputOpt.* The output of the MATLAB function is written into the variable *output*. The *output* is then passed on to the module *CoSimOutputMxFlt*, where it is converted into VPI Design Suite float values.

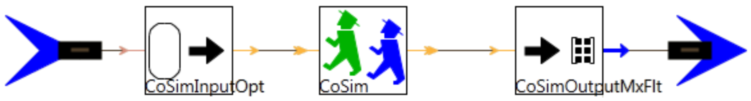
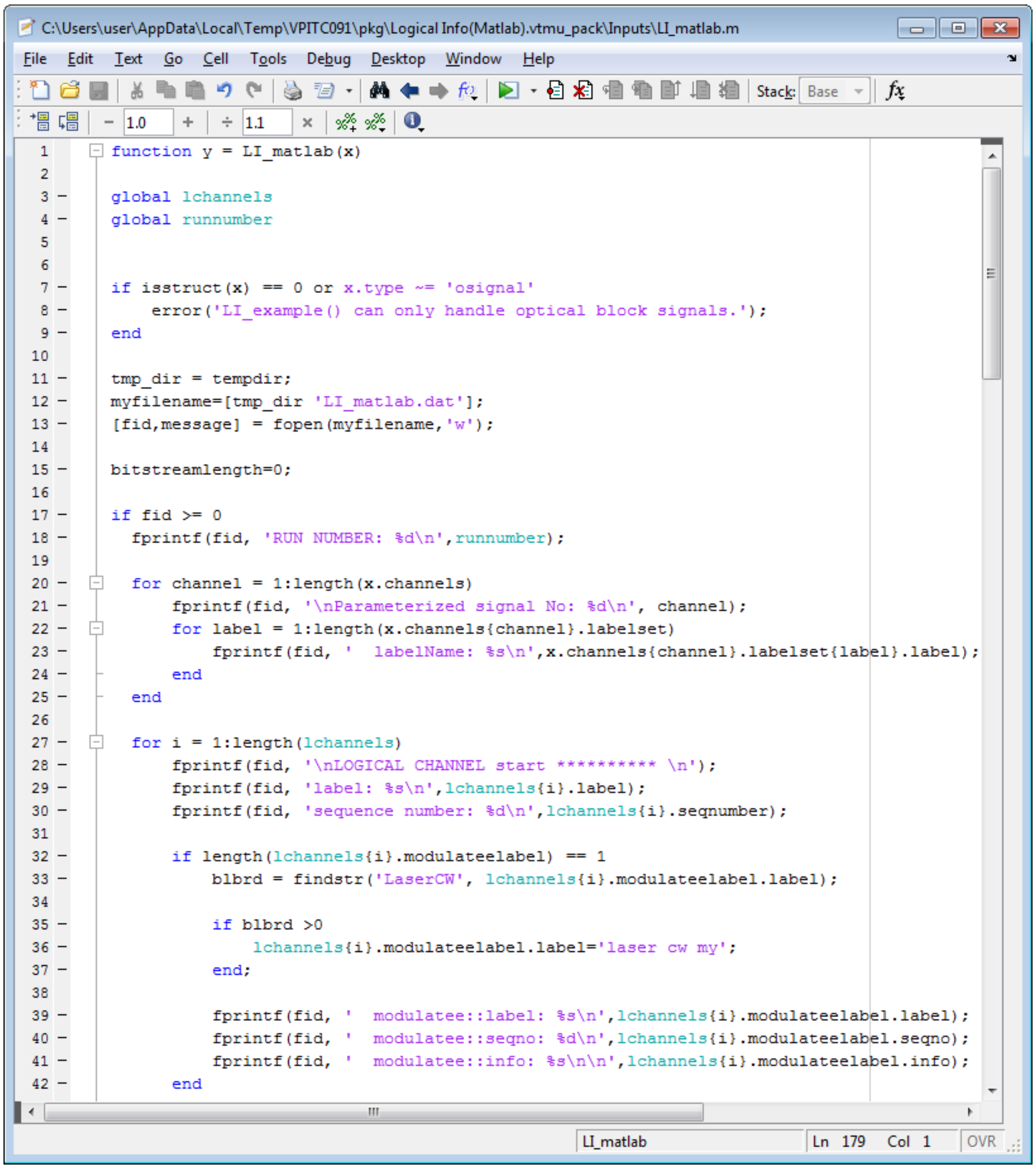


Figure 2-18 *Cosimulation galaxy for the logical info*

The MATLAB function `LI_matlab.m` writes the logical information stored in the global variable `lchannels` to a file in the temporary directory (see rows 11-13 in Figure 2-19):

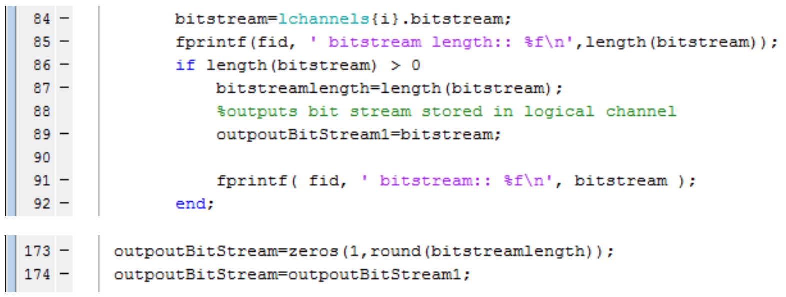

C:\Users\user\AppData\Local\Temp\VPITC091\pkg\Logical Info(Matlab).vtmu_pack\Inputs\LI_matlab.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Stack: Base
1.0
1.1
LI_matlab
Ln 179 Col 1
OVR

```
 1    function y = LI_matlab(x)
 2
 3 -  global lchannels
 4 -  global runnumber
 5
 6
 7 -  if isstruct(x) == 0 or x.type ~= 'osignal'
 8 -      error('LI_example() can only handle optical block signals.');
 9 -  end
10
11 -  tmp_dir = tempdir;
12 -  myfilename=[tmp_dir 'LI_matlab.dat'];
13 -  [fid,message] = fopen(myfilename,'w');
14
15 -  bitstreamlength=0;
16
17 -  if fid >= 0
18 -    fprintf(fid, 'RUN NUMBER: %d\n',runnumber);
19
20 -    for channel = 1:length(x.channels)
21 -        fprintf(fid, '\nParameterized signal No: %d\n', channel);
22 -        for label = 1:length(x.channels{channel}.labelset)
23 -            fprintf(fid, '  labelName: %s\n',x.channels{channel}.labelset{label}.label);
24 -        end
25 -    end
26
27 -    for i = 1:length(lchannels)
28 -        fprintf(fid, '\nLOGICAL CHANNEL start ********** \n');
29 -        fprintf(fid, 'label: %s\n',lchannels{i}.label);
30 -        fprintf(fid, 'sequence number: %d\n',lchannels{i}.seqnumber);
31
32 -        if length(lchannels{i}.modulateelabel) == 1
33 -            blbrd = findstr('LaserCW', lchannels{i}.modulateelabel.label);
34
35 -            if blbrd >0
36 -                lchannels{i}.modulateelabel.label='laser cw my';
37 -            end;
38
39 -            fprintf(fid, '  modulatee::label: %s\n',lchannels{i}.modulateelabel.label);
40 -            fprintf(fid, '  modulatee::seqno: %d\n',lchannels{i}.modulateelabel.seqno);
41 -            fprintf(fid, '  modulatee::info: %s\n\n',lchannels{i}.modulateelabel.info);
42 -        end
```

Figure 2-19 *Example opening file for writing and accessing modulatee info*

Rows 20-25 in Figure 2-19 get the channel labels which are accessible through the optical channels (see page 89). Logical information is accessed through the variable *lchannels*, for example, modulatee information is written to file in rows 32-42 shown in Figure 2-19.

To get the bit stream from the logical channel, the following code is written (see Figure 2-20):



```
84 -        bitstream=lchannels{i}.bitstream;
85 -        fprintf(fid, ' bitstream length:: %f\n',length(bitstream));
86 -        if length(bitstream) > 0
87 -            bitstreamlength=length(bitstream);
88              %outputs bit stream stored in logical channel
89 -            outpoutBitStream1=bitstream;
90
91 -            fprintf( fid, ' bitstream:: %f\n', bitstream );
92 -        end;

173 -  outpoutBitStream=zeros(1,round(bitstreamlength));
174 -  outpoutBitStream=outpoutBitStream1;
```

Figure 2-20 *Example of getting bit stream (MATLAB)*

In rows 173-174, the matrix of size 1xBitStreamLength is created and filled in by the bitstream derived from the logical channel (row 89). The last string in the code transfers this matrix to the output.

The Fast MATLAB (Matlab_Fast) Interface

The standard MATLAB interface described above uses a representation for ONSL optical signals that closely parallels the internal representation. Each noise bin and parameterized signal is contained within a separate structure, and the lists of all noise bins and parameterized signals are held in separate arrays. While this structure is logical, it is not necessarily the most efficient representation for use with MATLAB. Each structure, and each component within a structure, must be created separately by calls to the MATLAB engine interface. Furthermore, the resulting representation limits the extent to which MATLAB's powerful array-processing features can be applied to noise bins and parameterized signals. It is for these reasons that the 'fast MATLAB' (`Matlab_Fast`) cosimulation interface type has been provided. When the fast MATLAB interface is selected, noise bins and parameterized signals are represented using more efficient and array-oriented structures.

In the fast MATLAB interface, instead of representing noise bins and parameterized signals as an *array of structures*, they are represented as a *structure of arrays.* The `noise` member of the optical signal structure is not an array, but a single structure with the contents shown in Table 2-17. The lower and upper frequency limits of each noise bin are contained in a corresponding element of the `f0` and `f1` arrays respectively, while the components of the Stokes vector are stored in a row of the four-column matrix `S`.

Table 2-17 *Fast MATLAB interface optical noise bins representation*

Struct Entry	Type	Description
type: 'onoise'	string	signal type: Noise Bin
`f0`	int array	Lower frequency limit (grid points). One array entry for each noise bin.
`f1`	int array	Lower frequency limit (grid points). One array entry for each noise bin.

Table 2-17 *Fast MATLAB interface optical noise bins representation*

Struct Entry	Type	Description
S	float matrix	Stokes vectors. The matrix has four columns (one for each Stokes vector component) and one row for each noise bin.

A similar structure is used to represent parameterized signals, as shown in Table 2-18.

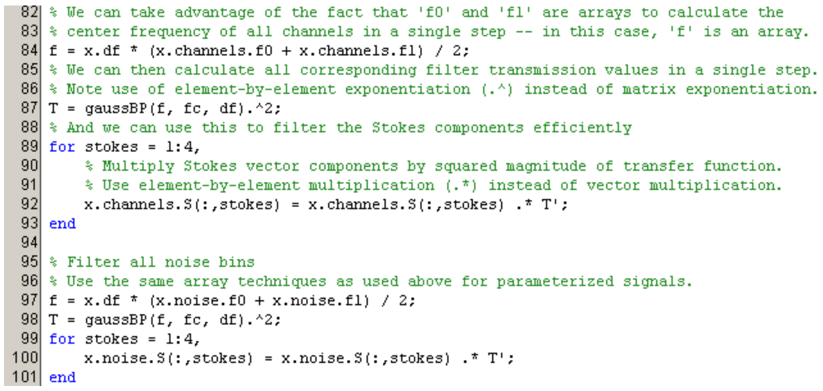
Table 2-18 *Fast MATLAB interface optical parameterized signals representation*

Struct Entry	Type	Description
type: 'ochannel'	string	signal type: Parameterized Channel
`f0`	int array	Lower frequency limit (grid points). One array entry for each noise bin.
`f1`	int array	Lower frequency limit (grid points). One array entry for each noise bin.
S	float matrix	Stokes vectors. The matrix has four columns (one for each Stokes vector component) and one row for each noise bin.

Using the fast MATLAB interface, the number of MATLAB data structures that must be written into the target environment, and read back out of it, may be greatly reduced, especially if the number of noise bins and parameterized signals is large. Instead of creating one structure for each noise bin and parameterized signal, with one string, two integers and one array in each structure, only two structures are created, each of which contains only one string, two integer arrays and one matrix. Whether or not this results in a noticeable performance improvement will depend upon the number of noise bins and parameterized signals present in the input and output signals.

Note: When the global parameter `TrackingMode` is set to `Detailed`, the fast MATLAB interface cannot be used because parameterized signal tracking implies the creation of arrays of variable length which cannot be implemented in the fast MATLAB interface.

However, an additional potential benefit of the fast MATLAB interface is that it may be possible to perform certain kinds of calculations much more efficiently in MATLAB using the fast signal representations. This efficiency may simply be in economy of expression in MATLAB code, but in many cases this will also result in faster calculations, because MATLAB is optimized for operating on arrays of values. This is illustrated by the code fragment shown in Figure 2-21, which is the relevant section of the fast MATLAB version of the Gaussian bandpass filter example. Instead of looping through each individual parameterized signal and noise bin, to calculate the center frequency and filter transmission, array operations are used to calculate an array `f` of center frequencies in a single step, followed by a corresponding array `T` of transmission values. A loop through the four components of the Stokes vector is used to apply these transmission values to every parameterized signal or noise bin, using element-by-element array multiplication. This code is simpler, more consistent with the general philosophy of MATLAB, and in cases in which the number of noise bins and parameterized signals is large, it will be more efficient.



```
% We can take advantage of the fact that 'f0' and 'f1' are arrays to calculate the
% center frequency of all channels in a single step -- in this case, 'f' is an array.
f = x.df * (x.channels.f0 + x.channels.f1) / 2;
% We can then calculate all corresponding filter transmission values in a single step.
% Note use of element-by-element exponentiation (.^) instead of matrix exponentiation.
T = gaussBP(f, fc, df).^2;
% And we can use this to filter the Stokes components efficiently
for stokes = 1:4,
    % Multiply Stokes vector components by squared magnitude of transfer function.
    % Use element-by-element multiplication (.*) instead of vector multiplication.
    x.channels.S(:,stokes) = x.channels.S(:,stokes) .* T';
end

% Filter all noise bins
% Use the same array techniques as used above for parameterized signals.
f = x.df * (x.noise.f0 + x.noise.f1) / 2;
T = gaussBP(f, fc, df).^2;
for stokes = 1:4,
    x.noise.S(:,stokes) = x.noise.S(:,stokes) .* T';
end
```

Figure 2-21 *MATLAB code to filter parameterized signals and noise bins in the fast interface format*

The fast MATLAB version of the optical bandpass filter example can be found in the subfolder `simeng\cosim\examples\matlab\GaussBPFilter` of the VPI Design Suite installation directory. The function `filterSignalGaussBP_fast` (in the file `filterSignalGaussBP_fast.m`) replaces the standard MATLAB interface function `filterSignalGaussBP`. You can use the demonstration *Filter (Matlab)* in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Matlab* to run this code if you wish to try it. Open the demonstration, and right-click on the MATLAB filter module. Select **Look Inside** from the context menu to open the cosimulation galaxy. Open the parameter editor for the cosimulation interface, and change the **`InterfaceType`** to `Matlab_Fast`, and the name of the function in the **RunCommand** to `filterSignalGaussBP_fast`. (You should not attempt to save the cosimulation galaxy — it is read-only.) Then run the demonstration. The results should be the same as those obtained with the standard MATLAB implementation.

Cosimulation with Python

System Requirements

Customized version of Python (VPIpython 3.11.1.0, based on Python 3.11.1) and all Python packages required to run demonstrations and examples are installed by the VPI Design Suite installer. The main packages installed are:

- NumPy 1.23.5
- SciPy 1.9.3
- Python for Windows Extensions 305

The NumPy package is required to use the Python interface in conjunction with matrix data or VPI ONSL signal types. The SciPy package is used to demonstrate scientific calculations in Python. Other Python packages are also recommended for typical cosimulation applications.

All of this software and documentation can be obtained freely from www.python.org or by following links from that site. For a general introduction to using Python for numerical analysis and simulation, see [1].

Any additional Python packages can also be installed; however, only for the packages listed above, VPIphotonics guarantees the correct functioning of the cosimulation interface with the supported versions.

Note: In order to use Python cosimulation, the 64-bit version of Python must be installed.

For more detailed information on how to install additional packages and configure Python environments, refer to "Python Environments" on page 20.

Setting Up the Cosimulation Interface

To choose Python as the external program, set parameter **`InterfaceType`** of the *CoSimInterface* module to Python. In this mode, the interpretation of the various parameters of the cosimulation module is as follows:

- **`Path`**: the path to a directory containing the Python script file.
- **`IfcFile`**: set this parameter to the name of the main Python script file. This file is passed to the Python interpreter for parsing during initialization of the cosimulation interface. Typically it will contain the definitions of the initialization, run and wrap-up functions.
- **`InitCommand`**: set this parameter to the Python command that should execute upon initialization of the interface. If left blank, no initialization command will be run.
- **`RunCommand`**: set this parameter to the Python command to execute each time the module is fired during the simulation run.
- **`WrapupCommand`**: set this parameter to the Python command to execute prior to shut-down of the interface.

Note: As with MATLAB scripts, it is possible to attach Python script files to the cosimulation galaxy package and run them from the Inputs package folder. However, it is important to ensure that when there are multiple source scripts, all scripts that are not part of the standard installation, or installed on the search path specified in Python environment preferences, are included in the Inputs folder. To use the attached scripts, leave the **Path** parameter blank.

- **LogicalInformation**: set this parameter to Yes to access the logical channels during the cosimulation. Note that the global parameter **LogicalInformation** should also be On or CurrentRun.

Simulation Job Functions

Several functions are available to access the simulation job information like current iteration, sweep details, or whether the simulation is running at the Remote Simulation Server or locally.

To access the current run number, the following should be used:

```
import vpi_tc_ptcl as tc

sweep_info = tc.get_sweepinfo()
sweep_level = sweep_info[1]
runNumber = sweep_level.current
```

To understand whether a simulation is running at the Remote Simulation Server, you can use the `isremotesimulation()` function of the `vpi_tc_ptcl` package:

```
import vpi_tc_ptcl as tc

if tc.isremotesimulation():
   # Do something at Remote Simulation Server
```

Errors and Warning Messages in Python Code

Run-time error messages produced in Python cosimulation, like in other types of cosimulation, are displayed in the VPI Design Suite Message Log. This applies to both ‘true’ errors and exceptions generated by the `raise` statement.

A special kind of exception can be produced by the `raise VPI_Error()` statement. In contrast to `raise RuntimeError()` or similar statements, the error message in this case will not contain traceback information, even if the extended error reporting is switched on. This permits the inclusion of error messages that do not contain extra information.

In addition to errors, the Python code can sometimes produce warnings — messages that do not interrupt the script execution but call attention to the possibility of incorrect behavior or obsolete module usage. Python warnings are displayed in the VPI Design Suite Message Log when the extended error reporting is enabled in the Submit Simulation Job dialog.

For details on extended error reporting and information shown in the log, see the **Message Log** and **Submit Simulation Job** dialog descriptions in the *VPIphotonics Design Suite™ User Interface Reference*.

In addition to the error messages and 'internal' warnings, Python cosimulation can pass user-specified messages to the VPI Design Suite **Message Log**. For this purpose, the cosimulation interface defines the special package `vpi_tc_ptcl`, which is available when the Python script is run in the cosimulation environment (see "Python Cosimulation Module vpi_tc_ptcl" on page 118).

Python Cosimulation Module vpi_tc_ptcl

The special module `vpi_tc_ptcl` is available when the Python script is run in the cosimulation environment and allows interaction between a cosimulation script and the simulation engine.

Note: This module is also available in the initialization scripts. For details on using them, see "Initialization Scripts" in Chapter 4, "Setting Simulation Parameters" of the *VPIphotonics Design Suite™ Simulation Guide*.

The module defines the following functions:

```
info(val)
info(format, val1[, val2, ...][, key1=kval1[, key2=kval2, ...]])
```

Purpose:

Displays an info message in the Message Log. Info messages will be visible when Info (or All) is checked in the Messages option of the Message Log.

Note: The function `message()` is a deprecated alias for the `info()` function.

Formatting rules:

There are two ways of formatting the message, as shown below with the help of examples:

1. In case there is only one argument, it will be formatted according to the following rules:
   + integer - as it is:
     ```
     info(1234) -> "1234"
     ```
   + float - float value (up to 15 digits are displayed):
     ```
     info(5.678) -> "5.678"
     ```
   + string, Unicode string - as it is:
     ```
     info("Hello, World!") -> "Hello, World!"
     info(u"\u00df“) -> "ß"
     ```
   + complex numbers - as complex numbers in brackets without 'j' and every component as a float value (up to 15 digits are displayed):
     ```
     info(complex(12, 54)) -> "(12,54)"
     info(complex(12.345, 54.321)) -> "(12.345,54.321)"
     ```
   + a NumPy array of integers - as integers with a space before every item (including the first entry):
     ```
     info(np.array([1, 2, 3, 4, 5])) -> " 1 2 3 4 5"
     ```
   + a NumPy array of floats - as float values (up to 15 digits are displayed) with a space before every item (including the first entry):

`info(np.array([1.1, 2.2, 3.3, 4.4, 5.5])) -> " 1.1 2.2 3.3 4.4 5.5"`

+ a NumPy array of complex numbers - as complex numbers with a space before every item (including the first entry):
  `info(np.array([complex(1.1, 2.2), complex(3.3, 4.4)])) -> " (1.1,2.2) (3.3,4.4)"`
+ in other cases, the argument would be converted to string representation by Python's built-in function `str()` accordingly.

2. For multiple arguments, the function works as Python's built-in function `str.format()`: the first argument *`format`* is used as the format string. Every other argument is represented according to the rules specific to Python and substituted in the format string. There are three ways to substitute values in the format string which also can be combined, as shown below:
   + in the order of *`val`*-arguments with empty braces:
     `info("Count: {}. Distance: {}.", 8, 2000) -> "Count: 8. Distance: 2000."`
   + according to the indexes of the *`val`*-arguments (starting from 0) in braces:
     `info("Count: {0}. Distance: {1}.", 8, 2000) -> "Count: 8. Distance: 2000."`
   + according to the keys of *`key`*-arguments in braces:
     `info("Count: {cnt}. Distance: {dst}.", dst=2000, cnt=8) -> "Count: 8. Distance: 2000."`

Parameters:

`val` Single argument that will be represented as a string.

`format` Format string (identical to the built-in Python's function `str.format()`).

`val1, val2, ...` Unnamed values for substitution to the format string.

`key1=kval1, key2=kval2, ...` The keyword arguments for substitution in the format string.

```
warning(val)
warning(format, val1[, val2, ...][, key1=kval1[, key2=kval2, ...]])
```

Purpose:

Displays a warning message in the **Message Log**. Warnings will be visible when **Warnings** (or **All**) is checked in the **Messages** option of the **Message Log**.

Formatting rules:

Rules of formatting are the same as `info()`.

Parameters:

`val` Single argument that will be represented as a string.

`format` Format string (identical to the built-in Python's function `str.format()`).

`val1, val2, ...` Unnamed values for substitution to the format string.

`key1=kval1, key2=kval2, ...` The keyword arguments for substitution in the format string.

```
progress(val)
progress(format, val1[, val2, ...][, key1=kval1[, key2=kval2, ...]])
```

Purpose:

Displays a progress message in the **Message Log**. Progress messages are produced by the simulation engine when the **Report Progress** checkbox is selected in the **Submit Simulation Job** dialog. They are shown in the log when **Progress** (or **All**) is checked in the **Messages** option of the **Message Log**.

Formatting rules:

Rules of formatting are the same as `info()`.

Parameters:

val Single argument that will be represented as a string.

format Format string (identical to the built-in Python's function `str.format()`).

val1, val2, ... Unnamed values for substitution to the format string.

key1=kval1, key2=kval2, ... The keyword arguments for substitution in the format string.

`visualizedata(`*window_id, data1*`[, `*data2*`[, `*data3*`[ ...]]][, `*options*`][, `*sweepinfo*`])`

Purpose:

Visualizes the data from cosimulation using VPIphotonicsAnalyzer (1D, 2D, or 3D).

For details on this function and usage examples, see "Visualizing Data and Data Files" on page 270 and "Simulation Control Functions" on page 294 in Chapter 3, "Simulation Scripts" (both simulation scripting and cosimulation share the same implementation for data visualization).

When using this function in cosimulation, please observe the following:

- The parameter *window_id* is used by VPIphotonicsAnalyzer to decide on which window the data is to be sent. To avoid the mixing of data produced by different instances of the cosimulation modules, consider using the global variable *ifc_name*, as shown in the example below.
- The parameter *sweepinfo* should normally be *omitted* in the cosimulation code as the simulation engine will provide the actual sweep information automatically.

Example:

```
import vpi_tc_ptcl as tc

# Prepare data
D2_1 = np.array([[1, 10], [2, 20], [3, 30], [4, 40]])
D2_2 = np.array([[1, 20], [2, 30], [3, 40], [4, 50]])

# Remove CoSimInterface id
baseId = ifc_name[:ifc_name.rfind(".")]

tc.visualizedata(baseId+".Test2D", D2_1, D2_2, options='legend1="RZ" legend2="NRZ"
title="Test 2D"')
```

`get_sweepinfo()`

Purpose:

Returns a tuple of `SweepLevel` classes (this class is also defined in the `vpi_tc_ptcl` Python module), containing information about the current and total number of module iterations, see "Iteration (Sweep) Information" on page 90.

Examples:

- Access the total number of runs:
  ```
  total_runs = vpi_tc_ptcl.get_sweepinfo()[1].total
  ```
- Access the current sweep step an depth 1 (assuming 2-level sweep is used):
  ```
  sw_info = vpi_tc_ptcl.get_sweepinfo()
  sw_num = sw_info[3].current
  ```

Parameters:

The function accepts no parameters.

`get_context_id()`

Purpose:

Returns the full topology path in the universe to the module instance that contains the Python script with this function. For example (pseudocode):

```
star M1
{
      float B = 0
}

galaxy G1
{
      init.py
      M1_vtms1
      M1_vtms1.B = user_function() [Py]
}

galaxy G2
{
      user_script.py
      CoSimInterface_vtms1
      CoSimInterface_vtms1.IfcFile = user_script.py
      CoSimInterface_vtms1.RunCommand = output=function1(input)
}

init.py
{
      import vpi_tc_ptcl
      ...
      def user_function():
            vpi_tc_ptcl.message(vpi_tc_ptcl.get_context_id())
            ...
}

user_script.py
{
      import vpi_tc_ptcl
      ...
      def function1(input):
            vpi_tc_ptcl.message(vpi_tc_ptcl.get_context_id())
            ...
}

universe U1
{
      G1_vtmg1
      G2_vtmg1
}
```

In the above example, `get_context_id()` function returns `U1_vtmu.G2_vtmg1.CoSimInterface_vtms1` and `U1_vtmu.G1_vtmg1` from `user_script.py` and from `init.py`, respectively.

Parameters:

The function accepts no parameters.

`get_inputs_dir()`

Purpose:

If called from the cosimulation script, returns the full path to the `Inputs` folder of a galaxy or universe that contain an instance of the ***CoSimInterface*** module. If called from the initialization script (`init.py`), returns the full path to the `Inputs` folder containing this script.

Note: The `Inputs` folder with its content is copied to the temporary location as the simulation is started. The `get_inputs_dir()` function returns the path to this temporary located copy.

Parameters:

The function accepts no parameters.

`isremotesimulation()`

Purpose:

Returns `true` if a simulation is running at the Remote Simulation Server.

Parameters:

None

`mux(`*`signal_array`*`)`

Purpose:

The function multiplexes a number of input optical or electrical signals. The type of input signal can be `osample`, `osignal`, `esample`, or `esignal`. The types are described in tables Table 2-1, Table 2-2, Table 2-6, and Table 2-8 of the “Mapping Signals to Data Structures in the Target Environment” section. The function reuses the multiplexing algorithm realized in the simulation engine. It takes into account the features of signal representation described in Chapter 5, “Advanced Signal Modes and Representations” of the *VPItransmissionMaker™Optical Systems User’s Manual.* For instance, the function creates a single frequency band as output signal if input signal bands overlap. The multiple frequency band representation is used for output signal when input signal bands do not overlap.

Parameters:

`signal_array` an array of input signals.

Return value:

Returns multiplexed output signal of the same type as input signals.

Example:

```
#output = ideal_mux_3x1(input1, input2, input3) using CoSimInputOpt
```

```
import vpi_tc_ptcl as tc

def ideal_mux_3x1(in1, in2, in3):
  return tc.mux([in1, in2, in3])
```

`filter_opt(`*`input, f_tf`* `[,` *`nb_threshold=-120, nb_dynamic=3, nb_resolution=1e9`*`])`

Purpose:

Performs filtering of input signal of the **`osignal`** type (see Table 2-2). For details on the filtering features and their application to noise bins, see "Characterization of Optical Filters" in the Optical Filters chapter of the Photonic Modules Reference.

Parameters:

`input` the input optical single-mode signal.

`f_tf` the filter transfer function. It returns a value of transmittance for the given input frequency value.

`nb_threshold` Specifies the lower limit of the transfer function for spectral regions in which the noise bin width is adapted. In dB units. The default value is -120 dB.

`nb_dynamic` Represents the maximum allowed power variation, in dB units, of the filter transfer function within the bandwidth of a single noise bin. The default value is 3 dB.

`nb_resolution` The lower limit for adaptation of noise bin width in Hz units. The default value is 1e9 Hz.

Return value:

Returns the filtered output signal.

Example:

```
import vpi_tc_ptcl as tc
import numpy as np

class MyFilter(object):
  def __init__(self, fc, df):
    self.fc = fc
    self.df = df
  def tf(self, f):
    r=np.exp(-np.log(np.sqrt(2.0)) * pow((2.0*np.abs(f-self.fc)/self.df),2))
    return r

def run(input):
  f = MyFilter(193.1e12, 300.0e9)
  return tc.filter_opt(input, f.tf)
```

`filter_el(`*`input, f_tf`*`)`

Purpose:

Performs filtering of input signal of the **`esignal`** type (see Table 2-6).

Parameters:

`input` the input electrical signal.

`f_tf` the filter transfer function. It returns a value of transmittance for the given input frequency value. Transfer function values at negative frequencies are calculated as complex conjugates of transfer function values at positive frequencies.

Return value:

Returns the filtered output signal.

Example:

```
import vpi_tc_ptcl as tc
import numpy as np

class MyFilter(object):
  def __init__(self, a, b):
    self.a = a
    self.b = b
  def tf(self, f):
    s=1j*2.0*np.pi*f
    r=1.0/(self.a*s**2 + self.b*s + 1.0)
    return r

def run(input):
  f = MyFilter(2.533e-20, 2.251e-10)
  return tc.filter_el(input, f.tf)
```

`resample(`*`input, fs`* `[,` *`fc=193.1e12, nb_mode='Skip', noise_bandwidth=10e12, noise_fc=193.1e12, nb_threshold=1e-19, nb_resolution =100e9, nb_grid_ref =193.1e12, nb_dynamic=3.0, pol_tolerance=1.0, gpf_limit =GreatestPrimeFactorLimit`*`])`

Purpose:

The function modifies the sample rate and center frequency of the sampled band signals and/or changes the spectral width of the noise bins. It can be applied to a signal of the `osignal` or *`esignal`* type (see Table 2-2 and Table 2-6 respectively). The behavior of the `resample()` function is similar to the features of the ***Resample*** module described in the "Signal Conversion" chapter of the Photonic Modules Reference.

Parameters:

`input` the input optical single-mode signal.

`fs` Defines new sample rate of the output signal. The actual sample rate may be adjusted to satisfy the GreatestPrimeFactorLimit and ensure an integer number of samples per symbol. In Hz units.

`fc` Sets a new center frequency of the output signal (ignored for electrical signals). In Hz units.

`noise_fc` Center frequency of the spectral range where the noise bins will be modified. Noise bins falling outside this range will be discarded. The default value is 193.1e12 Hz.

`noise_bandwidth` Represents the width of the spectral range around center frequency where the noise bins will be modified. Noise bins falling outside this range will be discarded. In Hz units.

`nb_mode` Selects the operating mode for the noise bins. For the Uniform mode, the spectral width of the noise bins will be changed to the specified value and will be equal for all output noise bins. For the Adaptive mode, the neighboring noise bins that satisfy the specified noise dynamics will be joined. For the Skip mode, the noise bins will remain unmodified. The default value is 'Skip'.

`nb_resolution` Defines the spectral width of the output noise bins. Hz units.The default value is 100e9 Hz.

nb_threshold Sets a power spectral density threshold to discard low-power noise bins. The default value is 1e-19 W/Hz.

nb_grid_ref The parameter is used for alignment of the frequencies of noise bins. The center frequency of one of the generated noise bins is equal to this parameter. The default value is 193.1e12 Hz.

nb_dynamic Specifies the maximum allowed difference of the power spectral density between two neighboring noise bins in order they could be joined. The default value is 3 dB.

pol_tolerance Sets the maximum allowed difference of the polarization states (normalized) of two neighboring noise bins in order they could be joined. The default value is 1.

gpf_limit Makes sure that the number of samples of the output signal has the greatest prime factor that does not exceed the specified limit. For example, for 150 samples the GPF is equal to 5 (150 = 2x3x5x5). The specified sample rate *fs* may be automatically adjusted (increased) to satisfy the above condition. Holding the GPF limit is important to ensure high speed of the fast Fourier transform in the output signal path. The highest speed is achieved with the default setting 2, which is equivalent to the condition that the number of samples is an integer power of two. It is recommended to set the GPF limit not larger than 13. Setting it to -1 will deactivate the GPF limiting, which allows to simulate an arbitrary number of samples in the signal. If not specified, the value of the schematic global parameter is used.

Return value:

Returns resampled output signal.

Example:

```
import vpi_tc_ptcl as tc

def run(input, SampleRate, CenterFrequency, NoiseBinMode, NoiseBandwidth,
NoiseCenterFrequency, NoiseBinThreshold,
FreqResolutionNB, GridReferenceFrequency, NoiseBinDynamic, PolarizationTolerance,
GreatestPrimeFactorLimit):

  return tc.resample(input, SampleRate, fc = CenterFrequency,
    nb_mode = NoiseBinMode, noise_bandwidth = NoiseBandwidth, noise_fc =
NoiseCenterFrequency, nb_threshold = NoiseBinThreshold, nb_resolution =
FreqResolutionNB, nb_grid_ref = GridReferenceFrequency, nb_dynamic =
NoiseBinDynamic, pol_tolerance = PolarizationTolerance, gpf_limit =
GreatestPrimeFactorLimit)
```

Non-blocking Simulation Package vpi_cosim_nonblock

The Python package `vpi_cosim_nonblock` allows you to perform cosimulation with Matplotlib without blocking the simulation when a Matplotlib visualization is shown. The package runs Matplotlib visualization in a separate process, and thus simulation blocking is avoided. See an example of using this package in "Example 5 — Plot an Optical Signal Spectrogram Using Matplotlib".

Import:

```
from vpi_cosim_nonblock.nonblocker import non_blocker as nb
```

The package provides the following functions:

`nb.init(ifc_name, visualize_func)`

Purpose:

Initializes visualization with a Python function object.

Parameters:

`ifc_name(str)` ID of the cosimulation module which performs visualization.

`visualize_func(callable)` a function in which drawing algorithms are specified. This function must be present in the cosimulation script.

`nb.get_data(ifc_name, run=None, index=0)`

Purpose:

Gets or initializes the data container for a specified module ID and runs the key (if there are several simulation runs) and figure (Matplotlib window) index if there are several figures.

Parameters:

`ifc_name(str)` ID of the cosimulation module which performs visualization.

`run(any type)` simulation run key if needed. It serves to distinguish several different simulation runs.

`index(int)` figure index within a given *`ifc_name-run`* context.

Returns:

Container with data for visualization. This container can be used in the drawing algorithm.

Note: To fill the container with a highly nested data structure using the `get_data()` method, the `deepcopy()` method from the module `copy` of the Python standard library can be used. As an alternative, the `set_data()` method can also be applied (described below).

`nb.set_data(data, ifc_name=None, run=None, index=0)`

Purpose:

Fills the data container with data for a specified module ID, the run key (if there are several simulation runs), and the Matplotlib figure index (if there are several figures).

Parameters:

`data(dict)` data to be stored in the container.

`ifc_name(str)` ID of the cosimulation module which performs visualization.

`run(any type)` simulation run key if needed. It serves to distinguish several different simulation runs.

`index(int)` figure index within a given *`ifc_name-run`* context.

`nb.get_figure(`*`index=0`*`)`

Purpose:

Gets a Matplotlib figure for the specified figure index.

Parameters:

`index(int)` figure

Returns:

Matplotlib figure object

`nb.get_next_data(`*`ifc_name, run=None`*`)`

Purpose:

Initializes the next data container for a specified module ID and the run key.

Parameters:

`ifc_name(str)` ID of the cosimulation module which performs visualization.

`run(any type)` simulation run key if needed. It serves to distinguish several different simulation runs.

Returns:

Container with data for visualization. This container can be used in the drawing algorithm.

`nb.has_data(`*`ifc_name, run=None, index=0`*`)`

Purpose:

Checks if the module with a specified ID has data for a given run key and figure index.

Parameters:

`ifc_name(str)` ID of the cosimulation module which performs visualization.

`run(any type)` simulation run key if needed. It serves to distinguish several different simulation runs.

`index(int)` figure index within a given *`ifc_name-run`* context.

Returns:

`True` if the module has data and `False` otherwise.

`nb.show(`*`ifc_name, run=None`*`)`

Purpose:

Launches a separate process which shows a Matplotlib visualization without blocking the simulation.

Parameters:

`ifc_name(str)` ID of the cosimulation module which performs visualization.

run(any type) simulation run key if needed. It serves to distinguish several different simulation runs.

As visualization is performed in a separate process, exceptions will be thrown in that separate process (if there are errors in the drawing algorithm). The package `vpi_cosim_nonblock` provides two methods to see the error messages.

`nb.set_debug_console(`*`_debug_console, ifc_name=ifc_name`*`)`

Purpose:

Shows the console window during the execution of the visualization script. All visualization error messages will be shown in this console.

Parameters:

ifc_name(str) ID of the cosimulation module which performs visualization. Default value is the current module ID.

_debug_console(bool) if `True` then the console will be shown; if `False` the console will not be displayed.

`nb.set_debug_output(`*`_debug_output, ifc_name=ifc_name`*`)`

Purpose:

Outputs information about performed visualization.

Parameters:

ifc_name(str) ID of the cosimulation module that performs visualization. The default value is the current module ID.

_debug_output(bool) if `True` then information will be shown, if `False` then information will not be displayed.

The following pieces of information will be shown in the VPI Design Suite Message Log once visualization is complete:

- path to a temporary file with collected data (DATA FILE)
- path to a temporary Python script (SCRIPT FILE) which contains the drawing algorithm
- system process ID (PID) of Matplotlib visualization
- visualization module ID (IFC_NAME)
- run key (RUN)

`nb.add_inputs_to_pythonpath`

Purpose:

If this attribute is set to `True`, the `Inputs` folder of the current cosimulation galaxy is added to *PYTHONPATH*. The default value is `False`.

Handling Parameters

Parameters are passed to the Python cosimulation environment as global variables. Regular (non-function) parameters are mapped to native Python types like `float`. Function parameters are represented with objects of a special type. Those objects have attributes `type`, `expr_type`, `value` and a method `eval()`.

`type` returns the type of the parameter. The relation between the type of the function parameter and the value of the `type` property is illustrated in Table 2-19.

Table 2-19 *Relation between 'type' property and type of a function parameter in VPI Design Suite*

Parameter type	Value of 'type' property
Integer	int
Integer array	intarray
Float	float
Float array	floatarray
Complex	complex
Complex array	complexarray

`expr_type` returns an expression type of the parameter. Expression type and value follows rules explained under "Function Parameters Evaluation" on page 91.

`value` returns a value of the parameter if parameter expression type `expr_type` = '`standard`'. Otherwise (`expr_type` = '`python`' or '`file`') it returns a string with a Python expression or a full file name.

If expression type of the function parameter is not '`file`', parameters expression could be evaluated using a special method `eval(dict[,globals[,locals]])`. Where *`dict`* is a dictionary of parameters and their values, for example:

```
{'$x': 1.0, '$y': 1+1j}
```

`eval()` returns the result of the expression evaluation for the values specified in *`dict`* if `expr_type` = '`python`'. If `expr_type` = '`standard`', the `value` is returned. If `expr_type` = '`file`' an exception is thrown.

The dimension of type of the variable could be increased by one dimension for simultaneous evaluation of the Python expression for several values of this variable. Thus a single call of the `eval` method could be performed instead of multiple calls. Correspondingly, the dimension of the result value of the `eval` method is equal to or greater by one dimension than the dimension of the type of the evaluated parameter. For example, if a Python expression takes a float argument `$x` and returns a complex value, then, if an array of floats is passed to `$x` in *`dict`*, the result of evaluation will be an array of complex values corresponding to the evaluation results for each value in the array `$x`.

If you need to evaluate a galaxy parameter in a custom context, you can use `globals` and `locals` optional parameters of the `eval` method, which are similar to the parameters of the Python built-in `eval()` function.

For a more elaborate example, see the demonstration *Simulation Techniques > Cosimulation > Python > Function Parameters in Cosimulation (Python)*.

Mapping VPI Signals to Python Data Types

The Python cosimulation interfaces uses the *dictionary* data type to map VPI's internal signals. The arrays of sampled bands, parameterized signals and noise bins in the structure representing an optical block signal are implemented as *lists*. Arrays of numbers, such as arrays of samples, are implemented as NumPy arrays. Hence, for instance the second component of the Stokes vector of the 42nd noise bin in the optical signal **`sig`** is qualified as

```
sig['noise'][41]['S'][1]
```

When using multiple port input/output modules to pass optical signals to and from the cosimulation environment (as described on page 84), the corresponding structure type is the Python *list*, and the same component of noise bin in the *third input* can be referenced as

```
sig[2]['noise'][41]['S'][1]
```

In contrast with the equivalent MATLAB expression described above under "Mapping VPI Signals to Matlab Data Types" on page 96, indices here are less by one. This is because array indices in Python are numbered starting at 0 rather than at 1, as in MATLAB.

The numbers of input and output signals can be accessed via the global variable (dictionary) **`interface`**. For example, the lines

```
NumberOfInputs = interface['input_buswidth']['input1']
NumberOfOutputs = interface['output_buswidth']['output3']
```

give the number of inputs and number of outputs to the modules *CoSimInputOpt_N* and *CoSimOutputOpt_N*, respectively with parameter **Name** set to `input1` and `output3`.

To access the logical channels, the global variable ***`lchannels`*** is available without any previous actions. For example, it is possible to get the label of modulator channel via:

```
labels=lchannels.keys()
      for i in labels:
            sequenceNumbers = lchannels[i].keys()
            for seqno in sequenceNumbers:
                  channel = lchannels[i][seqno]
                  modulator = channel['modulatorlabel']
                  if len(modulator) > 0:
                        ML=modulator[0]['label']
```

Debugging Python Cosimulation Code

When developing Python cosimulation scripts, it is frequently necessary to debug Python code to spot and fix errors of the algorithm. For more detailed information on how to debug Python scripts inside VPI Design Suite, refer to "Debugging Python Code" on page 24.

Using the Python Cosimulation Interface

A Simple Example: Adding Two Floating-Point Values

A very simple example is a cosimulation module that takes two floating-point value inputs, adds them and scales the result by a gain that is provided as a parameter. The output is then passed out of the cosimulation interface. This same example was presented previously, for the case of the MATLAB interface, and the setup was shown in Figure 2-6. As in the MATLAB example, we assume that the parameter **`Name`** of the upper input declaration module has been set to `input1`, that of the lower module to `input2`. The **`Name`** of the output module is `output`. The galaxy has a parameter **`Gain`**, which is set by default to 1.0

As it happens, the syntax in Python for adding two simple scalar values, and assigning them to a variable, is exactly the same as that in MATLAB:

```
output = Gain * (input1 + input2)
```

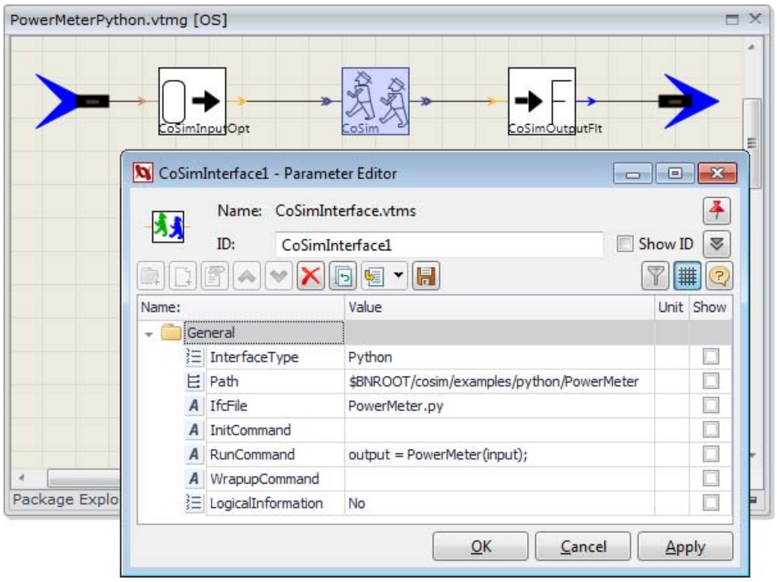
Thus the only change required to the setup shown in Figure 2-6 to perform cosimulation with Python instead of MATLAB is to set the **`InterfaceType`** parameter of the *CoSimInterface* module to Python.

Example 1 — Calculating the Total Optical Power of an Input Signal Using Python

The equivalent in Python of the MATLAB power meter example presented in "Example 1 — Calculating the Total Optical Power of an Input Signal using MATLAB" on page 100 is shown in Figure 2-22. The example schematic is called *Power Meter (Python)*, which can be found in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Python.* This setup is identical to the MATLAB setup shown in Figure 2-7, except for the settings of the parameters of the *CoSimInterface* module. In this case, the setting of the **`InterfaceType`** parameter is `Python`, the **`Path`** is set to the directory where the Python power meter script resides, the **`IfcFile`** is set to the name of the script itself (`PowerMeter.py`), and the **`RunCommand`** is set to

```
output = PowerMeter(input)
```

Note: The syntax of the Python command is very similar to that of the MATLAB command.



Name: CoSimInterface.vtms
ID: CoSimInterface1

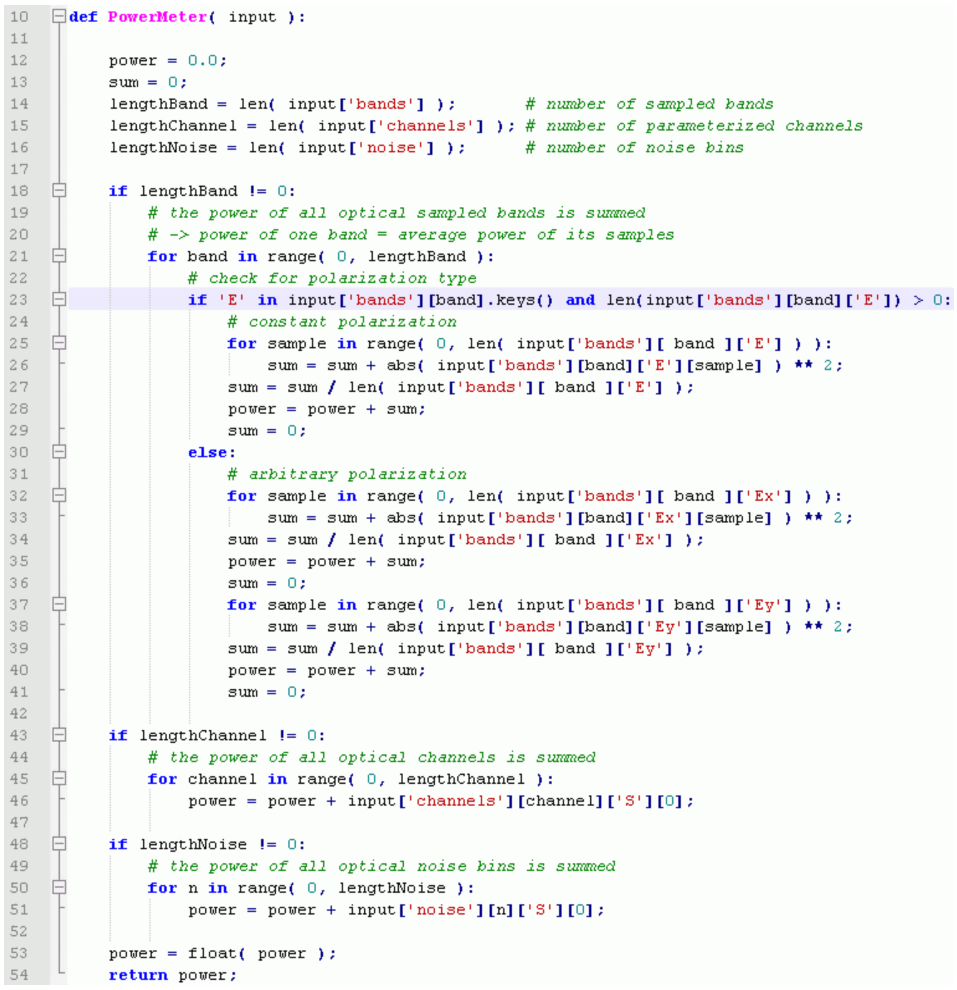
Name:	Value	Unit	Show
General			
InterfaceType	Python		☐
Path	$BNROOT/cosim/examples/python/PowerMeter		☐
IfcFile	PowerMeter.py		☐
InitCommand			☐
RunCommand	output = PowerMeter(input);		☐
WrapupCommand			☐
LogicalInformation	No		☐

Figure 2-22 *Example of the cosimulation interface galaxy for implementing a power meter in Python. The parameter settings of the CoSimInterface are shown.*

The complete definition of the Python `PowerMeter` function is shown in Figure 2-23. The script can be found in the `simeng\cosim\examples\python\PowerMeter` subfolder of the VPI Design Suite installation. It is very similar in structure and content to the MATLAB example shown in Figure 2-8. One notable difference is the way in which the various fields of the structures are tested and accessed. In the case of the MATLAB interface, all fields and arrays will be present — if any array is empty, it will still exist, but will have zero size. This situation is somewhat different in the Python interface, because Python dictionaries are used to hold the signal structures. The expression that tests for the presence of the E array, to distinguish between a signal of fixed polarization and one of arbitrary polarization, is (line 23)

```
if 'E' in input['bands'][band].keys() and
        len(input['bands'][band]['E'])>0:
```

There are two tests in this expression. The first checks to see if the key `'E'` exists at all in the dictionary holding the sampled band structure. If so, then the second test verifies that it is nonempty. Thus, in Python cosimulation code, you should not assume that all fields will be present. Unused or empty elements may be absent completely from the dictionary, and you should always use a test similar to the above.



```
def PowerMeter( input ):

    power = 0.0;
    sum = 0;
    lengthBand = len( input['bands'] );       # number of sampled bands
    lengthChannel = len( input['channels'] ); # number of parameterized channels
    lengthNoise = len( input['noise'] );      # number of noise bins

    if lengthBand != 0:
        # the power of all optical sampled bands is summed
        # -> power of one band = average power of its samples
        for band in range( 0, lengthBand ):
            # check for polarization type
            if 'E' in input['bands'][band].keys() and len(input['bands'][band]['E']) > 0:
                # constant polarization
                for sample in range( 0, len( input['bands'][ band ]['E'] ) ):
                    sum = sum + abs( input['bands'][band]['E'][sample] ) ** 2;
                sum = sum / len( input['bands'][ band ]['E'] );
                power = power + sum;
                sum = 0;
            else:
                # arbitrary polarization
                for sample in range( 0, len( input['bands'][ band ]['Ex'] ) ):
                    sum = sum + abs( input['bands'][band]['Ex'][sample] ) ** 2;
                sum = sum / len( input['bands'][ band ]['Ex'] );
                power = power + sum;
                sum = 0;
                for sample in range( 0, len( input['bands'][ band ]['Ey'] ) ):
                    sum = sum + abs( input['bands'][band]['Ey'][sample] ) ** 2;
                sum = sum / len( input['bands'][ band ]['Ey'] );
                power = power + sum;
                sum = 0;

    if lengthChannel != 0:
        # the power of all optical channels is summed
        for channel in range( 0, lengthChannel ):
            power = power + input['channels'][channel]['S'][0];

    if lengthNoise != 0:
        # the power of all optical noise bins is summed
        for n in range( 0, lengthNoise ):
            power = power + input['noise'][n]['S'][0];

    power = float( power );
    return power;
```

Figure 2-23 *The PowerMeter function in the Python source PowerMeter.py. It calculates the total power of all used optical bands, channels and noise.*

Example 2 — Implementing an Optical Bandpass Filter Using Python

The demonstration *Filter (Python)* in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Python* is equivalent to the MATLAB example described in "Example 2 — Implementing an Optical Bandpass Filter Using MATLAB" on page 103. This demonstration implements a Gaussian bandpass filter for optical signals using a Python script.

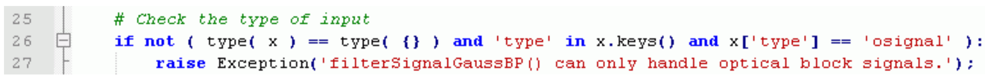
The setup of the Python optical filter demonstration, and the corresponding cosimulation galaxy, are identical to those of the MATLAB example, as shown in Figure 2-10 and Figure 2-11, so they are not repeated here.

The Python file `filter.py` that implements the optical filter can be found in the subfolder `simeng\cosim\examples\python\GaussBPFilter` of the VPI Design Suite installation directory (typically `C:\Program Files\VPI\VPIdesignSuite 11.5` under Windows, for example). The main cosimulation entry point is the function `filterSignalGaussBP`, which is executed by the *CoSimInterface* module using the **`RunCommand`**

```
output = filterSignalGaussBP(input,centerFrequency,Bandwidth)
```

Note that in this function call, `input` is the data structure representing the optical signal defined by the *CoSimInputOpt* module, `centerFrequency` is the Python variable defined by the *CoSimInputFlt* module, while `Bandwidth` is the Python variable that is created from the galaxy parameter **`Bandwidth`**. As in the case of MATLAB cosimulation, there is no discernible difference in the Python code between variables created from inputs, and variables created from galaxy parameters.

The fragment of Python code that verifies that the input signal is of the expected type is shown in Figure 2-24. The check consists of three tests (at line 26). The first test confirms that the input `x` (which is the optical signal input) is of the Python dictionary type — the expression '`{}`' defines an empty dictionary, so the test literally checks that the type of `x` is the same as the type of an empty dictionary. The second test confirms that the dictionary `x` contains an entry with the key `'type'`. Finally, if both these tests are passed, the `'type'` entry is compared with the string `'osignal'` to confirm that the structure is in fact an optical signal.



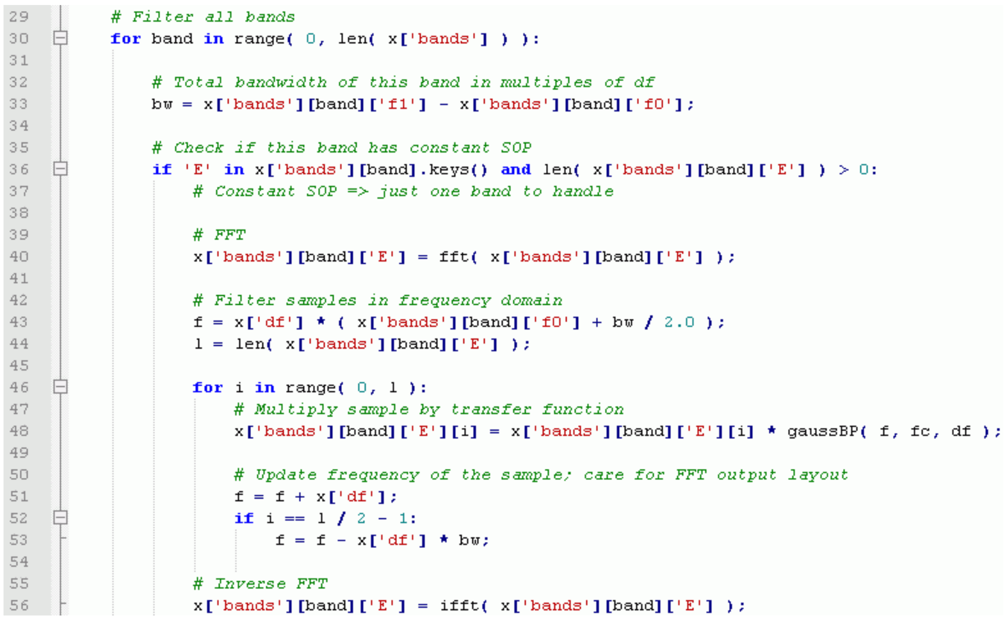
```
25      # Check the type of input
26      if not ( type( x ) == type( {} ) and 'type' in x.keys() and x['type'] == 'osignal' ):
27          raise Exception('filterSignalGaussBP() can only handle optical block signals.');
```

Figure 2-24 *Optical block signal type checking code in Python*

The fragment of Python code that implements the filtering of fully polarized sampled bands is shown in Figure 2-25. This code performs exactly the same function, in the same way, as the MATLAB code, so the description of the MATLAB example shown in Figure 2-13 applies in general here also. Note the use of the FFT and inverse FFT functions, at lines 40 and 56 respectively. These functions are provided by the NumPy package, and are imported at line 6 of the script (not shown) using the command

```
from numpy.fft import *
```

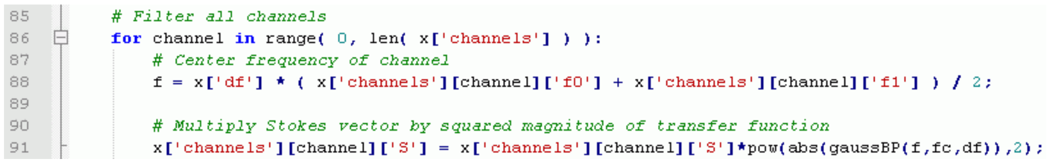
Note: Parameter passing and assignment in Python is by *reference*, and not by value. Thus the `E` array is not copied by the function calls or assignments in lines 40 and 56. It is only a reference to the array that is passed and assigned. This is a common pitfall for programmers new to Python, and who are familiar with languages like MATLAB that perform assignment by value. In Python, the statement `a=b` does not make a copy of `b`, but just assigns a reference of `b` to `a`. Hence any changes subsequently applied to `b`, also affect `a`.



```
# Filter all bands
for band in range( 0, len( x['bands'] ) ):

    # Total bandwidth of this band in multiples of df
    bw = x['bands'][band]['f1'] - x['bands'][band]['f0'];

    # Check if this band has constant SOP
    if 'E' in x['bands'][band].keys() and len( x['bands'][band]['E'] ) > 0:
        # Constant SOP => just one band to handle

        # FFT
        x['bands'][band]['E'] = fft( x['bands'][band]['E'] );

        # Filter samples in frequency domain
        f = x['df'] * ( x['bands'][band]['f0'] + bw / 2.0 );
        l = len( x['bands'][band]['E'] );

        for i in range( 0, l ):
            # Multiply sample by transfer function
            x['bands'][band]['E'][i] = x['bands'][band]['E'][i] * gaussBP( f, fc, df );

            # Update frequency of the sample; care for FFT output layout
            f = f + x['df'];
            if i == l / 2 - 1:
                f = f - x['df'] * bw;

        # Inverse FFT
        x['bands'][band]['E'] = ifft( x['bands'][band]['E'] );
```

Figure 2-25 *Python code fragment that filters fully polarized optical sampled bands*

The Python code fragment that filters parameterized signals is shown in Figure 2-26. Once again, there are obvious similarities with the equivalent MATLAB code shown in Figure 2-14. As in the MATLAB code, multiplication of the Stokes array by a scalar value (the magnitude-squared of the filter transfer function) results in each element of the array being scaled.



```
# Filter all channels
for channel in range( 0, len( x['channels'] ) ):
    # Center frequency of channel
    f = x['df'] * ( x['channels'][channel]['f0'] + x['channels'][channel]['f1'] ) / 2;

    # Multiply Stokes vector by squared magnitude of transfer function
    x['channels'][channel]['S'] = x['channels'][channel]['S']*pow(abs(gaussBP(f,fc,df)),2);
```

Figure 2-26 *Python code fragment that filters parameterized signals*

As in the MATLAB example, the filtering of the optical signal in Python is done in-place, that is, the input data itself is modified in the course of filtering. The result is returned using the statement `return x` which returns the value by reference (that is, again, there is no separate copy made). Thus in the Python example no memory containing optical signal data is allocated or copied throughout the entire program.

Python's semantics of parameter passing and assignment by reference can be very convenient for memory-intensive applications. It ensures that memory is generally not allocated or copied inadvertently. However, there will be occasions when you may need to make a copy of a complex structure, such as an optical signal. This can easily be done by using the `deepcopy()` function in the `copy` module. For example, if `signal` is an optical signal, a copy called `signalCopy` can easily be created, as follows

```
import copy

signalCopy = copy.deepcopy(signal)
```

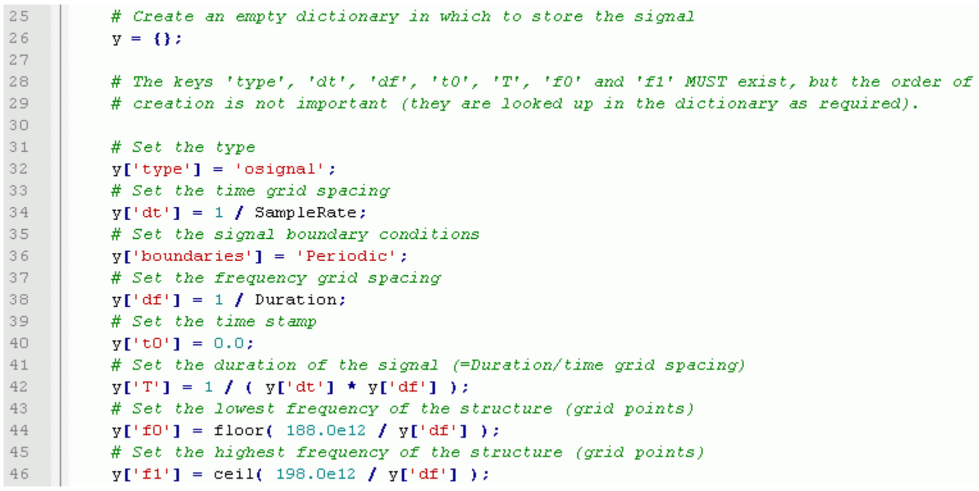
Example 3 — Creating New ONSL Signals Using Python

It is possible to create new optical or electrical signals in Python, or to add or remove components from signals passed into the cosimulation module, so long as some simple rules are followed.

- As previously mentioned, all structures are implemented as Python dictionaries (in some languages, data structures of this kind are known as associative arrays, or maps). An ONSL signal (for example, an optical or electrical signal) is a dictionary whose keys are strings representing the names of the member variables `type`, `dt`, `df`, `t0`, `T`, `f0`, `f1`, and optionally `noise`, `channels` and `bands`. Similarly, noise bins, parameterized signals, and sampled bands are represented as arrays of dictionaries whose keys are strings representing the member variables of the corresponding structures.
- Unlike the MATLAB interface, the order in which the elements in each dictionary is created does not matter. Each member is looked up by name (just like a word in a dictionary), so its location within the structure is unimportant.
- Also unlike the MATLAB interface, elements that are not required may be omitted. For example, if an optical signal contains no noise bins, then the 'noise' array need not be created. It is not necessary to create an empty array as a placeholder.
- All arrays must be created as Python NumPy arrays, that is, using the array functions of the NumPy module. Arrays of real numbers, such as Stokes vectors and electrical sampled bands, must be of the type `numpy.float` (that is, double precision real numbers, which is the default if the array is created using ordinary floating-point values in Python). Similarly arrays of complex numbers, such as optical sampled bands, must be of the type `numpy.complex`. Ordinary Python arrays, or NumPy arrays of other types, will be rejected by the cosimulation interface.

The demonstration *Optical Signal Generation (Python)* in the demonstrations *folder Optical Systems Demos > Simulation Techniques > Cosimulation > Python gives an example of how to create optical signals from scratch in* Python. *The* Python *source file can be found in* the `simeng\cosim\examples\python\CreateOpticalSignal` subfolder of the VPI Design Suite installation directory.

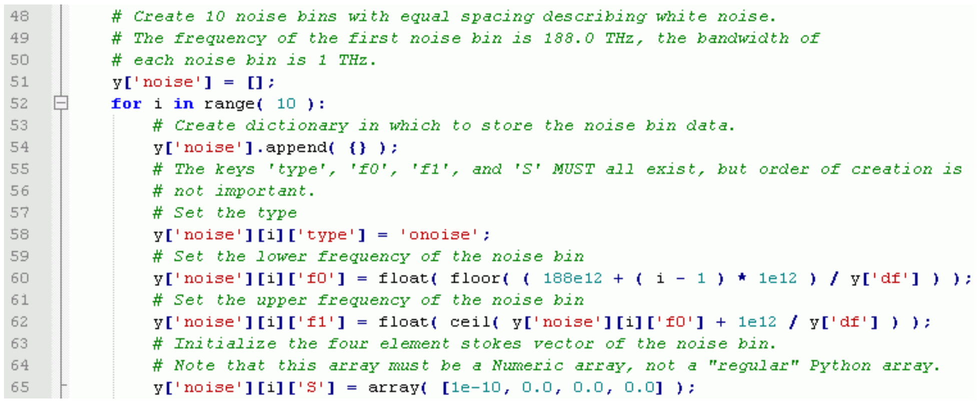
Figure 2-27 illustrates the process of creating the main parameters of a new optical signal in Python. It is necessary to first create the dictionary `y` that will contain the output signal. The parameters `type`, `dt`, `df`, `t0`, `T`, `f0`, and `f1` are all required, and can be created in any order. In the example, they are created in the conventional order defined by the top table of Table 2-2. Note that `y['t0']`, `y['T']`, `y['f0']` and `y['f1']` are initialized using floating-point values or expressions. If they are assigned integer values (for example, if `y['t0']` was set equal to `0` instead of `0.0`), then the corresponding variables would be of integer type, and would be rejected by the simulator according to the fourth rule above. The rounding functions `floor` and `ceil` are used when defining the frequency limits `f0` and `f1` in order to quantize the corresponding physical frequencies (188.0 THz and 198.0 THz) onto the global frequency grid defined by `df`. Although the values returned by these functions are integers, their type is floating point.



```
# Create an empty dictionary in which to store the signal
y = {};

# The keys 'type', 'dt', 'df', 't0', 'T', 'f0' and 'f1' MUST exist, but the order of
# creation is not important (they are looked up in the dictionary as required).

# Set the type
y['type'] = 'osignal';
# Set the time grid spacing
y['dt'] = 1 / SampleRate;
# Set the signal boundary conditions
y['boundaries'] = 'Periodic';
# Set the frequency grid spacing
y['df'] = 1 / Duration;
# Set the time stamp
y['t0'] = 0.0;
# Set the duration of the signal (=Duration/time grid spacing)
y['T'] = 1 / ( y['dt'] * y['df'] );
# Set the lowest frequency of the structure (grid points)
y['f0'] = floor( 188.0e12 / y['df'] );
# Set the highest frequency of the structure (grid points)
y['f1'] = ceil( 198.0e12 / y['df'] );
```

Figure 2-27 *Example of creating the parameters of an optical signal structure in Python*

Following the declaration of the main parameters, noise bins, parameterized signals, and sampled bands are created, in any order. Figure 2-28 illustrates the creation of ten noise bins. First, an empty array is created to hold the noise bin structures. Note that if no noise bins were to be created, it is not necessary to create an empty array. Then each noise bin is created in a loop. Each noise bin is represented by another Python dictionary, using the same procedure as before. Note the creation of the Stokes vector as a NumPy array, which will have the default type `numpy.float` since the members of the initializing array are floating-point values.



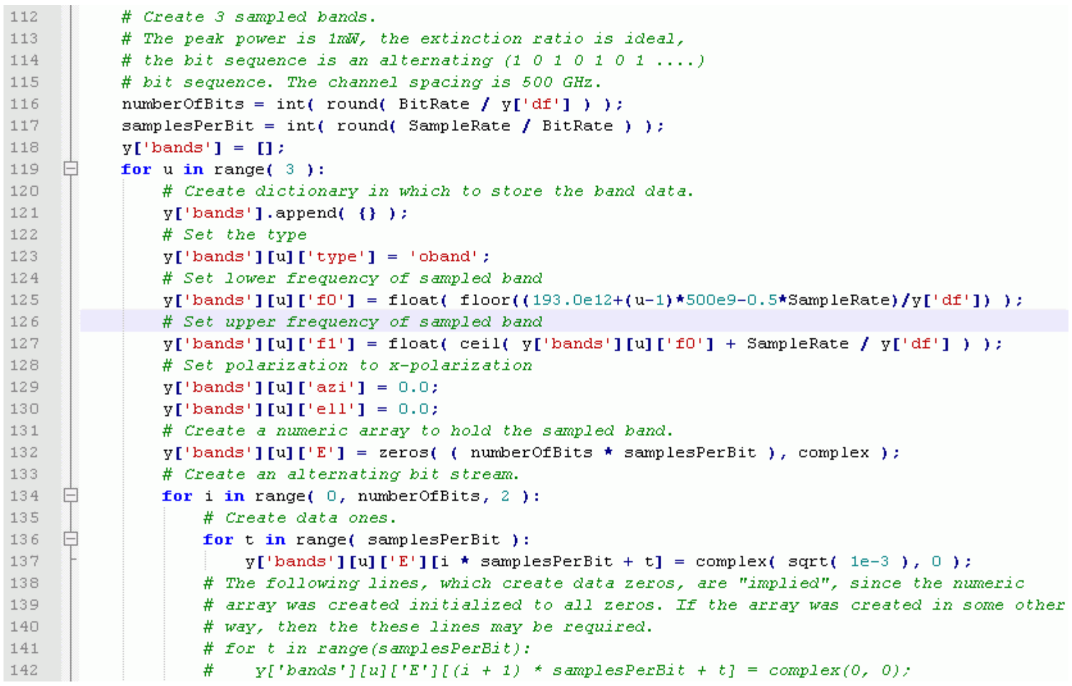
```
# Create 10 noise bins with equal spacing describing white noise.
# The frequency of the first noise bin is 188.0 THz, the bandwidth of
# each noise bin is 1 THz.
y['noise'] = [];
for i in range( 10 ):
    # Create dictionary in which to store the noise bin data.
    y['noise'].append( {} );
    # The keys 'type', 'f0', 'f1', and 'S' MUST all exist, but order of creation is
    # not important.
    # Set the type
    y['noise'][i]['type'] = 'onoise';
    # Set the lower frequency of the noise bin
    y['noise'][i]['f0'] = float( floor( ( 188e12 + ( i - 1 ) * 1e12 ) / y['df'] ) );
    # Set the upper frequency of the noise bin
    y['noise'][i]['f1'] = float( ceil( y['noise'][i]['f0'] + 1e12 / y['df'] ) );
    # Initialize the four element stokes vector of the noise bin.
    # Note that this array must be a Numeric array, not a "regular" Python array.
    y['noise'][i]['S'] = array( [1e-10, 0.0, 0.0, 0.0] );
```

Figure 2-28 *Example of creating noise bins in Python*

After creating noise bins, parameterized signals are created. The process is almost identical to that for creating noise bins — see the example code in `simeng\cosim\examples\python\CreateOpticalSignal` for details.

Finally, sampled bands are created. This is illustrated in Figure 2-29. Again, an empty array is created first, and then all elements of the band structures are created in a loop. The arbitrary polarization sampled band arrays `Ex` and `Ey` are not used, so they need not be created. If the sampled bands were represented as arbitrarily polarized signals, the two arrays `Ex` and `Ey` would be created, but the array `E`, and the polarization parameters `azi` and `ell` would not be required.

Note: In fact, if an optical sampled band is represented as an arbitrarily polarized signal, then the array E *must not* be created. If this array is found to exist by the Python cosimulation interface, then it will assume that the signal is fully polarized, and it will not look for the `Ex` and `Ey` arrays at all.



```
# Create 3 sampled bands.
# The peak power is 1mW, the extinction ratio is ideal,
# the bit sequence is an alternating (1 0 1 0 1 0 1 ....)
# bit sequence. The channel spacing is 500 GHz.
numberOfBits = int( round( BitRate / y['df'] ) );
samplesPerBit = int( round( SampleRate / BitRate ) );
y['bands'] = [];
for u in range( 3 ):
    # Create dictionary in which to store the band data.
    y['bands'].append( {} );
    # Set the type
    y['bands'][u]['type'] = 'oband';
    # Set lower frequency of sampled band
    y['bands'][u]['f0'] = float( floor((193.0e12+(u-1)*500e9-0.5*SampleRate)/y['df']) );
    # Set upper frequency of sampled band
    y['bands'][u]['f1'] = float( ceil( y['bands'][u]['f0'] + SampleRate / y['df'] ) );
    # Set polarization to x-polarization
    y['bands'][u]['azi'] = 0.0;
    y['bands'][u]['ell'] = 0.0;
    # Create a numeric array to hold the sampled band.
    y['bands'][u]['E'] = zeros( ( numberOfBits * samplesPerBit ), complex );
    # Create an alternating bit stream.
    for i in range( 0, numberOfBits, 2 ):
        # Create data ones.
        for t in range( samplesPerBit ):
            y['bands'][u]['E'][i * samplesPerBit + t] = complex( sqrt( 1e-3 ), 0 );
        # The following lines, which create data zeros, are "implied", since the numeric
        # array was created initialized to all zeros. If the array was created in some other
        # way, then the these lines may be required.
        # for t in range(samplesPerBit):
        #    y['bands'][u]['E'][(i + 1) * samplesPerBit + t] = complex(0, 0);
```

Figure 2-29 *Example of creating sampled bands in Python*

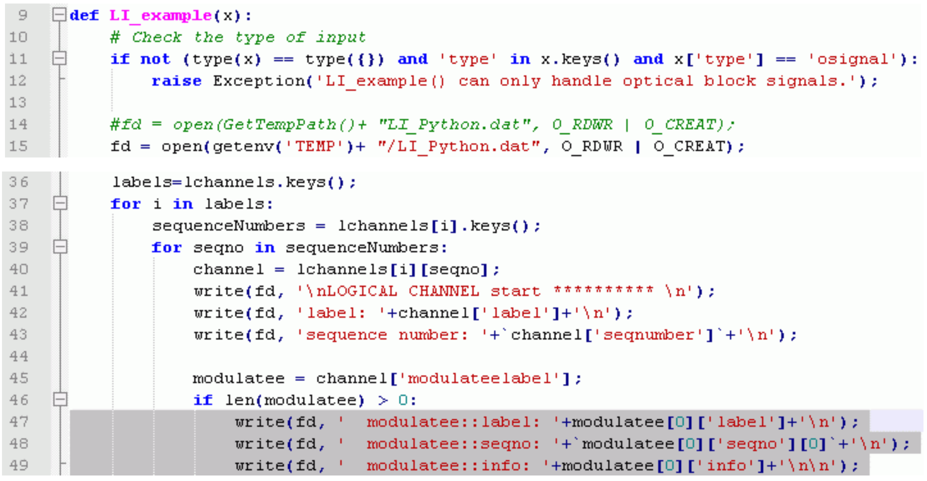
For a new signal source, the logical information shown in Table 2-9 can also be added if it is required. Logical channels should at least contain a label string, sequence number (which is usually equal to current run number), SignalSource structure (`f0`, `f1`, `t0`, `T` should be equal to an appropriate data of the physical signal) and Label (label string and seqnumber of the logical channel should be also added to the labelset of the new physical signal).

Example 4 — Handling the Logical Information Using Python

Another example of using Python functions in VPI Design Suite is handling of logical information. The cosimulation galaxy is similar to that shown in Figure 2-18. The figure shows the cosimulation galaxy used for transferring a bit stream stored in a logical channel. The galaxy has one optical input and returns a float matrix output. Therefore the data interface modules *CoSimInputOpt* and *CoSimOutputMxFlt* have been used to convert the VPI Design Suite signal representation into the appropriate Python form and vice versa. The RunCommand of the *CoSimInterface* is set to

```
output=LI_example(input)
```

The IfcFile parameter should contain the name of the Python file `LI_python.py`. Hence, the Python function `LI_python.py` located in the directory `Inputs` of *Logical Info (Python) Demo* defined by the parameter *CoSimInterface* > **`Path`** will be executed with the parameter **`input`**. The variable ***`input`*** represents the optical signal passed in to the *CoSimInterface* module by the module *CoSimInputOpt.* The output of the Python function is written into the variable ***`output`***. The ***`output`*** is then passed on to the module *CoSimOutputMxFlt,* where it is converted into VPI Design Suite float values.



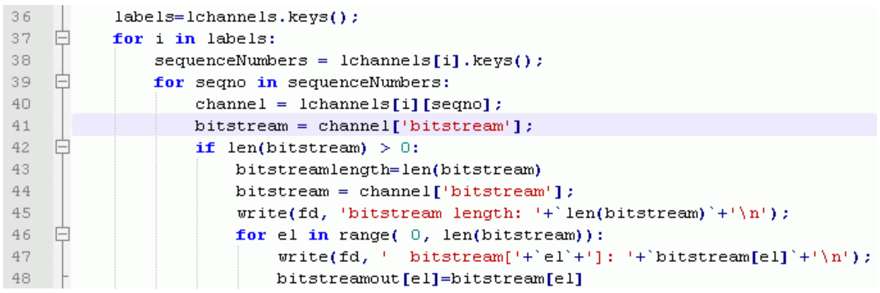
```
def LI_example(x):
    # Check the type of input
    if not (type(x) == type({}) and 'type' in x.keys() and x['type'] == 'osignal'):
        raise Exception('LI_example() can only handle optical block signals.');

    #fd = open(GetTempPath()+ "LI_Python.dat", O_RDWR | O_CREAT);
    fd = open(getenv('TEMP')+ "/LI_Python.dat", O_RDWR | O_CREAT);

    labels=lchannels.keys();
    for i in labels:
        sequenceNumbers = lchannels[i].keys();
        for seqno in sequenceNumbers:
            channel = lchannels[i][seqno];
            write(fd, '\nLOGICAL CHANNEL start ********** \n');
            write(fd, 'label: '+channel['label']+'\n');
            write(fd, 'sequence number: '+`channel['seqnumber']`+'\n');

            modulatee = channel['modulateelabel'];
            if len(modulatee) > 0:
                write(fd, '  modulatee::label: '+modulatee[0]['label']+'\n');
                write(fd, '  modulatee::seqno: '+`modulatee[0]['seqno'][0]`+'\n');
                write(fd, '  modulatee::info: '+modulatee[0]['info']+'\n\n');
```

Figure 2-30 *Example opening file for writing and accessing modulatee info*

The Python function `LI_python.py` writes the logical information stored in global variable `lchannels` to a file in the temporary directory (see Figure 2-30).

Logical information is accessed through the variable ***`lchannels`***, for example, modulatee information is written to file.

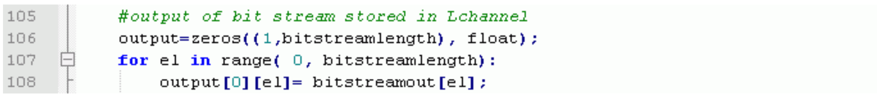
To get the bit stream from the logical channel, the following code is written (see Figure 2-31):



```
labels=lchannels.keys();
for i in labels:
    sequenceNumbers = lchannels[i].keys();
    for seqno in sequenceNumbers:
        channel = lchannels[i][seqno];
        bitstream = channel['bitstream'];
        if len(bitstream) > 0:
            bitstreamlength=len(bitstream)
            bitstream = channel['bitstream'];
            write(fd, 'bitstream length: '+`len(bitstream)`+'\n');
            for el in range( 0, len(bitstream)):
                write(fd, '  bitstream['+`el`+']: '+`bitstream[el]`+'\n');
                bitstreamout[el]=bitstream[el]
```

Figure 2-31 *Example of getting bit stream (Python)*

Then, the matrix of size `1 x BitStreamLength` is created and filled in by the bitstream derived from the logical channel. The last string in the code transfers this matrix to the output.



```
#output of bit stream stored in Lchannel
output=zeros((1,bitstreamlength), float);
for el in range( 0, bitstreamlength):
    output[0][el]= bitstreamout[el];
```

Figure 2-32 *Writing the bit stream to output (Python)*

Example 5 — Plot an Optical Signal Spectrogram Using Matplotlib

While the *SignalAnalyzer* module provides advanced signal visualization capabilities, there are cases when it is required to use another data visualization tool for design. This may include examples with visualization of non-signal data, for example, a structure of a device or some functional attributes of a model. Also, specific plot types such as vector-field visualization, contour plots, density plots, or time-frequency plots (see Figure 2-33) can be realized using external visualization tools.

In this example, the Matplotlib library is used for visualization of an optical signal spectrogram.

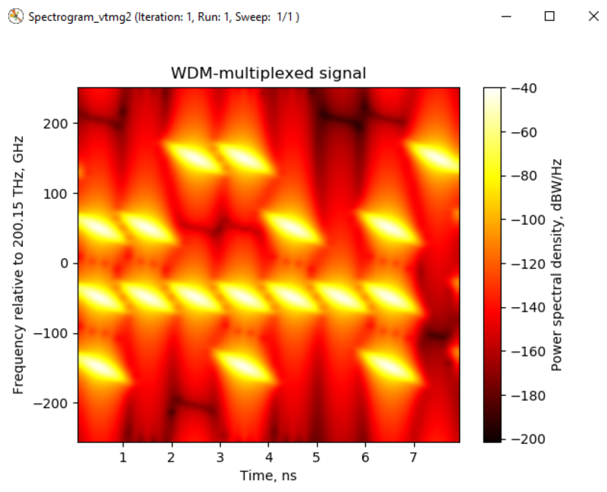


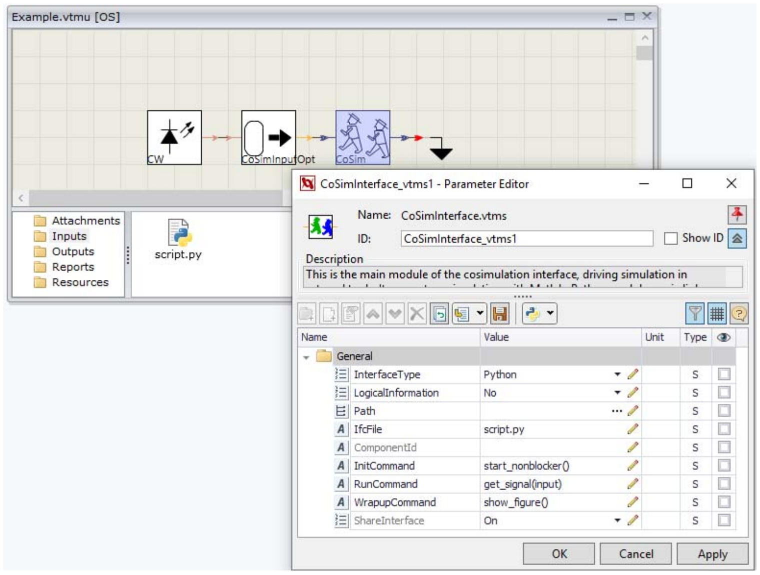
Figure 2-33 *Time-frequency plot of the WDM multiplexed signal in cosimulation with Matplotlib*

One peculiarity of this cosimulation example is that it shows how to start data visualization with Matplotlib as a separate process. Such decoupling of data visualization from the main process allows users to draw plots without blocking the simulation. The dedicated ***vpi_cosim_nonblock*** Python package of VPI Design Suite realizes this capability. See "Non-blocking Simulation Package vpi_cosim_nonblock" on page 125 for details.

Two examples are considered below: the first representing a basic illustration of how to apply the `vpi_cosim_nonblock` package for showing charts and the second containing the code of the ***Spectrogram*** module to provide more details about the application of the Matplotlib library within VPI Design Suite.

A Basic Example of Using the vpi_cosim_nonblock Package

The setup analyzed in this example is shown in Figure 2-34. The output signal from the ***LaserCW*** module is received by the ***CosimInputOpt*** module and then processed by a cosimulation script specified with the **`IfcFile`** parameter of the ***CosimInterface*** module. The script `script.py` is located in the `Inputs` folder of the schematic. Three cosimulation commands ***InitCommand***, ***RunCommand***, and ***WrapupCommand*** call the functions that are presented in the cosimulation script: `start_nonblocker()`, `get_signal(`*`input`*`)`, and `show_figure()` respectively.



Name: CoSimInterface.vtms

ID: CoSimInterface_vtms1 Show ID

Description

This is the main module of the cosimulation interface, driving simulation in

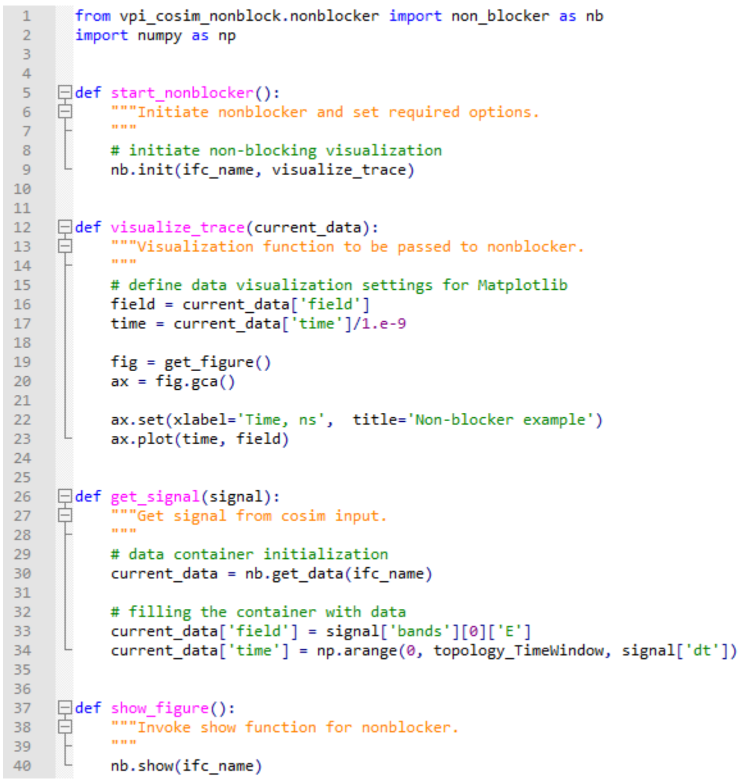
Name	Value	Unit	Type
General			
InterfaceType	Python		S
LogicalInformation	No		S
Path			S
IfcFile	script.py		S
ComponentId			S
InitCommand	start_nonblocker()		S
RunCommand	get_signal(input)		S
WrapupCommand	show_figure()		S
ShareInterface	On		S

Figure 2-34 *Parameter settings for the* ***CosimInterface***

Next Figure 2-35 shows the usage of the `vpi_cosim_nonblock` package within a cosimulation script. In the `start_nonblocker()` function, we initialize a non-blocking visualization. To start visualization as a separate process, a temporary script is created in which the Matplotlib algorithm is specified. For this reason, the drawing algorithm must be presented in the cosimulation script as a separate function. Then an object of this function is passed to the nb object. In our case, the `visualize_trace(`*`current_data`*`)` function defines the Matplotlib drawing algorithm that will be applied to the data provided as an argument of the function.

Note: The actual show command is performed within the `show_figure()` function by invocation of the `nb.show()` function that is responsible for displaying Matplotlib charts in a separate process.

The function `get_signal(`*`signal`*`)` serves to create a proper `current_data` data container and retrieve signal properties from the input signal.



```python
from vpi_cosim_nonblock.nonblocker import non_blocker as nb
import numpy as np


def start_nonblocker():
    """Initiate nonblocker and set required options.
    """
    # initiate non-blocking visualization
    nb.init(ifc_name, visualize_trace)


def visualize_trace(current_data):
    """Visualization function to be passed to nonblocker.
    """
    # define data visualization settings for Matplotlib
    field = current_data['field']
    time = current_data['time']/1.e-9

    fig = get_figure()
    ax = fig.gca()

    ax.set(xlabel='Time, ns',  title='Non-blocker example')
    ax.plot(time, field)


def get_signal(signal):
    """Get signal from cosim input.
    """
    # data container initialization
    current_data = nb.get_data(ifc_name)

    # filling the container with data
    current_data['field'] = signal['bands'][0]['E']
    current_data['time'] = np.arange(0, topology_TimeWindow, signal['dt'])


def show_figure():
    """Invoke show function for nonblocker.
    """
    nb.show(ifc_name)
```

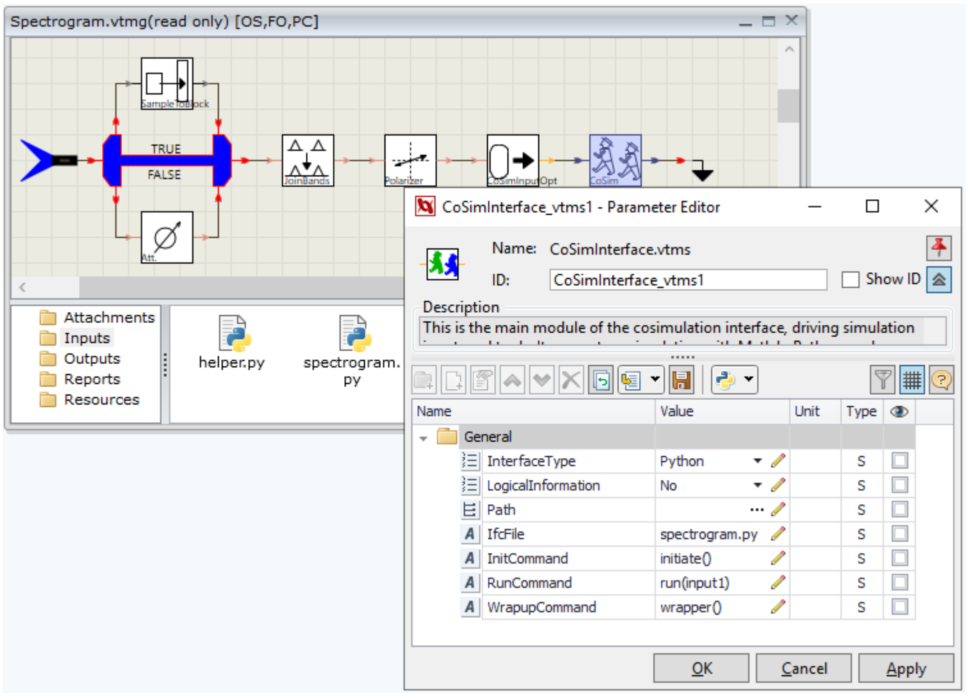
Figure 2-35 *A basic example of using the* `vpi_cosim_nonblock` *package for Matplotlib visualization*

Execution of this cosimulation setup shows that the simulation was not blocked by the Matplotlib visualization and the plot does not disappear after simulation completion. This is possible due to the decoupling of data visualization from the main simulation process.

Application of the Matplotlib Library in the Spectrogram Module

The *Spectrogram Of Multiplexed Signal.vtmu* demo can be found in the demonstrations folder *Simulation Techniques > Cosimulation > Python*. It shows the application of a joint time-frequency analysis that is based on the short-time Fourier transform for building a spectrogram (a time-frequency plot) of an optical WDM signal, as shown in Figure 2-33. Also, the ***Spectrogram*** galaxy can be found in the *Analyzers* folder of the TC Modules Library.

Figure 2-36 shows the details of cosimulation interface settings for the ***Spectrogram*** module. The main script `spectrogram.py` provides implementation of the `initiate()`, `run(`*`input`*`)`, and `wrapper()` functions.



Name: CoSimInterface.vtms

ID: CoSimInterface_vtms1 Show ID

Description

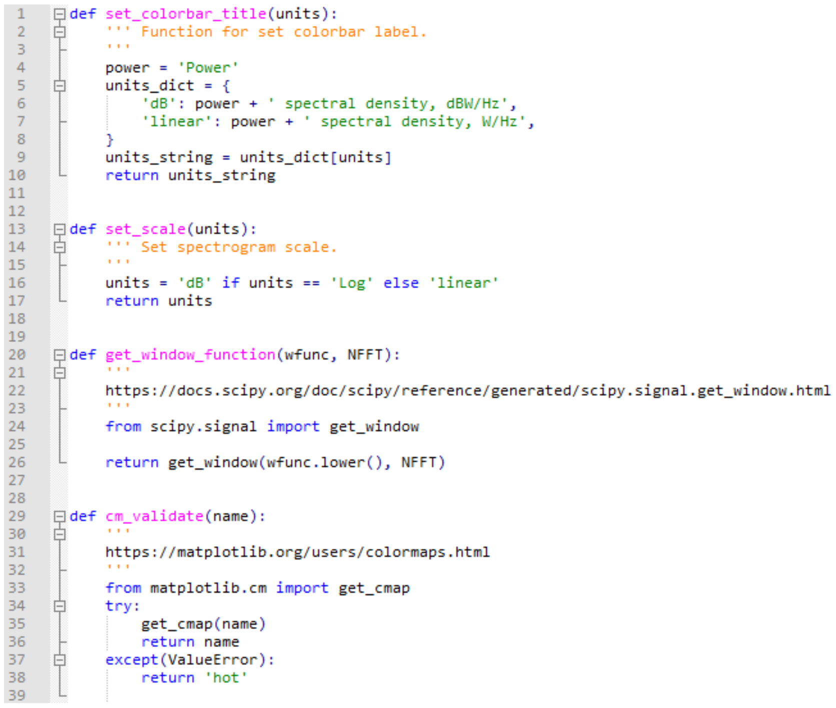
This is the main module of the cosimulation interface, driving simulation

Name	Value	Unit	Type
General			
InterfaceType	Python		S
LogicalInformation	No		S
Path			S
IfcFile	spectrogram.py		S
InitCommand	initiate()		S
RunCommand	run(input1)		S
WrapupCommand	wrapper()		S

Figure 2-36 *Cosimulation interface in the **Spectrogram** galaxy*

An additional `helper.py` script provides implementation of some auxiliary functions required for proper mapping of the ***Spectrogram*** module parameters to the Matplotlib commands and attributes.

This helper script, shown in Figure 2-37, is imported within the `visualize_trace(`*`current_data`*`)` function of the main `spectrogram.py` script. As in the previous example, the `visualize_trace` function is responsible for capturing the drawing algorithm for Matplotlib. Further you can find some specific details for the key functions of the script.

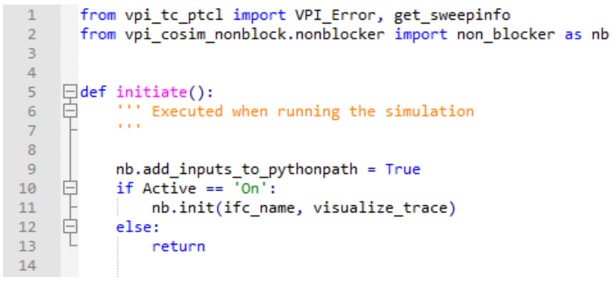


```
def set_colorbar_title(units):
    ''' Function for set colorbar label.
    '''
    power = 'Power'
    units_dict = {
        'dB': power + ' spectral density, dBW/Hz',
        'linear': power + ' spectral density, W/Hz',
    }
    units_string = units_dict[units]
    return units_string


def set_scale(units):
    ''' Set spectrogram scale.
    '''
    units = 'dB' if units == 'Log' else 'linear'
    return units


def get_window_function(wfunc, NFFT):
    '''
    https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.get_window.html
    '''
    from scipy.signal import get_window

    return get_window(wfunc.lower(), NFFT)


def cm_validate(name):
    '''
    https://matplotlib.org/users/colormaps.html
    '''
    from matplotlib.cm import get_cmap
    try:
        get_cmap(name)
        return name
    except(ValueError):
        return 'hot'
```

Figure 2-37 *The* `helper.py` *module*

initiate()

The *`nb.add_inputs_to_pythonpath`* attribute (shown in Figure 2-38) adds the `Inputs` folder to the *`PYTHONPATH`* variable. This mechanism allows us to import the `helper.py` Python modules from the `Inputs` folder to the `visualize_trace(`*`current data`*`)` function.



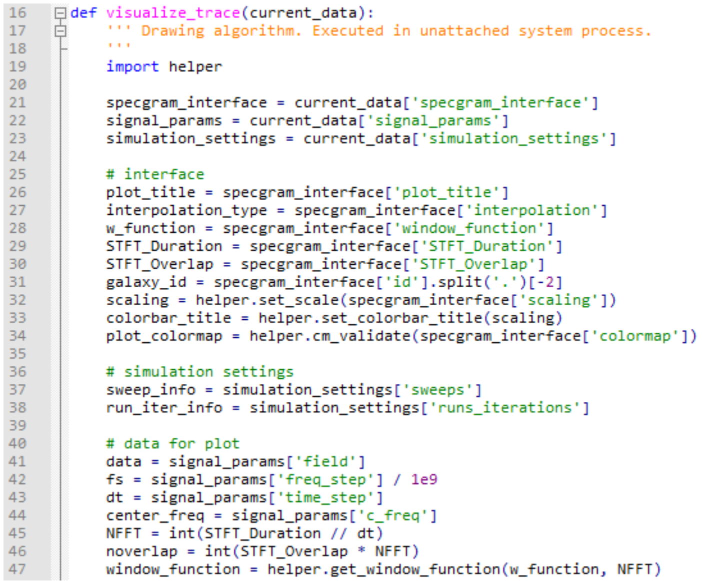
```
from vpi_tc_ptcl import VPI_Error, get_sweepinfo
from vpi_cosim_nonblock.nonblocker import non_blocker as nb


def initiate():
    ''' Executed when running the simulation
    '''

    nb.add_inputs_to_pythonpath = True
    if Active == 'On':
        nb.init(ifc_name, visualize_trace)
    else:
        return
```

Figure 2-38 `initiate()` *function defines additional nonblocker options*

visualize_trace()

The `visualize_trace()` function consists of two parts:

- unpacking of the collected data container
- the drawing algorithm

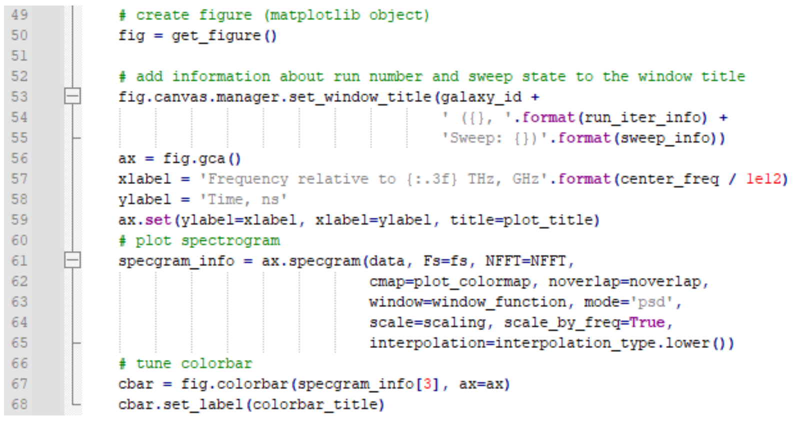


```
def visualize_trace(current_data):
    ''' Drawing algorithm. Executed in unattached system process.
    '''
    import helper

    specgram_interface = current_data['specgram_interface']
    signal_params = current_data['signal_params']
    simulation_settings = current_data['simulation_settings']

    # interface
    plot_title = specgram_interface['plot_title']
    interpolation_type = specgram_interface['interpolation']
    w_function = specgram_interface['window_function']
    STFT_Duration = specgram_interface['STFT_Duration']
    STFT_Overlap = specgram_interface['STFT_Overlap']
    galaxy_id = specgram_interface['id'].split('.')[-2]
    scaling = helper.set_scale(specgram_interface['scaling'])
    colorbar_title = helper.set_colorbar_title(scaling)
    plot_colormap = helper.cm_validate(specgram_interface['colormap'])

    # simulation settings
    sweep_info = simulation_settings['sweeps']
    run_iter_info = simulation_settings['runs_iterations']

    # data for plot
    data = signal_params['field']
    fs = signal_params['freq_step'] / 1e9
    dt = signal_params['time_step']
    center_freq = signal_params['c_freq']
    NFFT = int(STFT_Duration // dt)
    noverlap = int(STFT_Overlap * NFFT)
    window_function = helper.get_window_function(w_function, NFFT)
```

Figure 2-39 *Unpacking and processing collected data in the* `visualize_trace()` *function*

Note: All necessary imports for the drawing algorithm must be performed inside the `visualize_trace()` function because the temporary Python script based on this function does not contain the global imports located in the header of the cosimulation script. Only those modules that were added to the *PYTHONPATH* variable (e.g., NumPy and SciPy) can be accessed within the `visualize_trace()` function.

The drawing algorithm represents the second part of the visualization function (see Figure 2-40).



```
    # create figure (matplotlib object)
    fig = get_figure()

    # add information about run number and sweep state to the window title
    fig.canvas.manager.set_window_title(galaxy_id +
                                        ' ({}, '.format(run_iter_info) +
                                        'Sweep: {})'.format(sweep_info))
    ax = fig.gca()
    xlabel = 'Frequency relative to {:.3f} THz, GHz'.format(center_freq / 1e12)
    ylabel = 'Time, ns'
    ax.set(ylabel=xlabel, xlabel=ylabel, title=plot_title)
    # plot spectrogram
    specgram_info = ax.specgram(data, Fs=fs, NFFT=NFFT,
                                cmap=plot_colormap, noverlap=noverlap,
                                window=window_function, mode='psd',
                                scale=scaling, scale_by_freq=True,
                                interpolation=interpolation_type.lower())
    # tune colorbar
    cbar = fig.colorbar(specgram_info[3], ax=ax)
    cbar.set_label(colorbar_title)
```

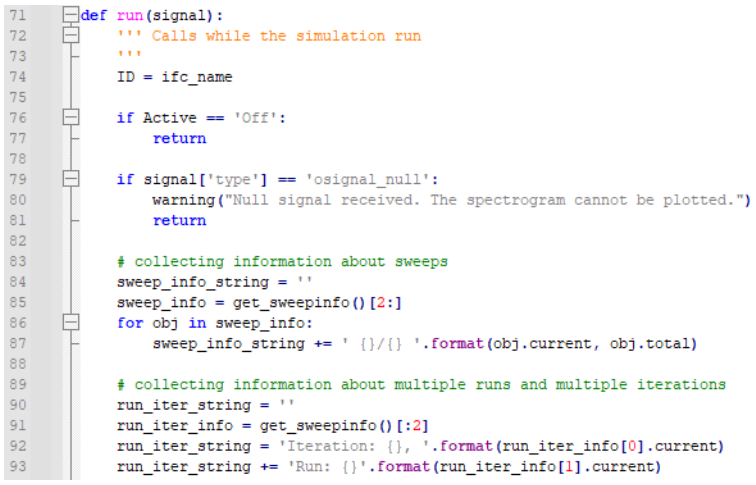
Figure 2-40 *The drawing algorithm*

run()

The `run()` function can be logically divided into three parts:

- collecting information about the simulation (iterations, runs, and sweeps) and about the cosimulation module (its ID)
- initializing a special data container
- filling the container with data, handling errors, and showing the visualization

At the first part of the `run()` function, information about the cosimulation module, iterations, runs, and sweeps is collected (see Figure 2-41).

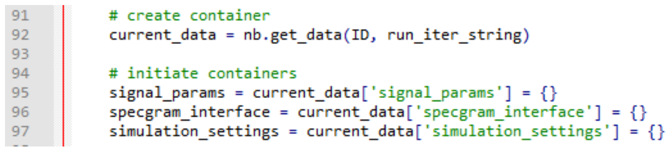


```
def run(signal):
    ''' Calls while the simulation run
    '''
    ID = ifc_name

    if Active == 'Off':
        return

    if signal['type'] == 'osignal_null':
        warning("Null signal received. The spectrogram cannot be plotted.")
        return

    # collecting information about sweeps
    sweep_info_string = ''
    sweep_info = get_sweepinfo()[2:]
    for obj in sweep_info:
        sweep_info_string += ' {}/{} '.format(obj.current, obj.total)

    # collecting information about multiple runs and multiple iterations
    run_iter_string = ''
    run_iter_info = get_sweepinfo()[:2]
    run_iter_string = 'Iteration: {}, '.format(run_iter_info[0].current)
    run_iter_string += 'Run: {}'.format(run_iter_info[1].current)
```

Figure 2-41 *Collecting information about the module and simulation*

The module ID and information about iterations, runs, and sweeps will be displayed in the header of the Matplotlib window.

The second part includes initialization of a container in which the collected data will be stored (see Figure 2-42).

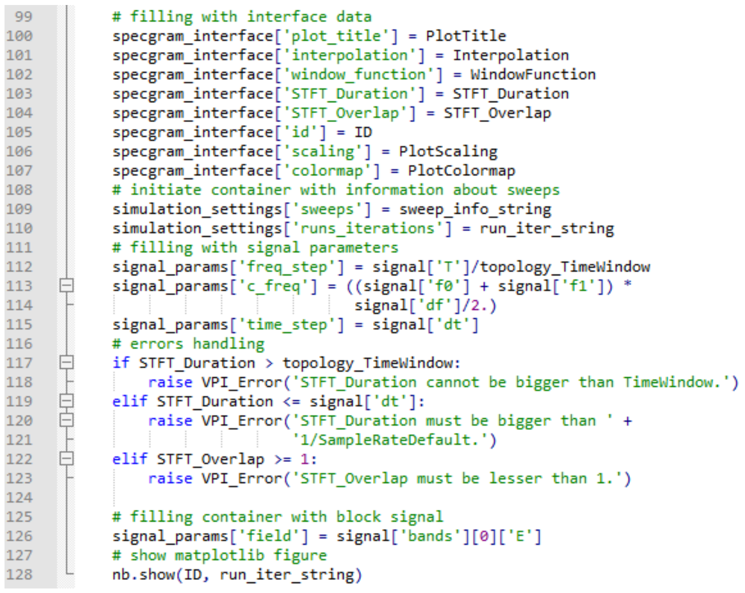


```
# create container
current_data = nb.get_data(ID, run_iter_string)

# initiate containers
signal_params = current_data['signal_params'] = {}
specgram_interface = current_data['specgram_interface'] = {}
simulation_settings = current_data['simulation_settings'] = {}
```

Figure 2-42 *Initialization of a special data container*

Three different types of data are collected in the `run()` function: input signal parameters (*`signal_params`* key), ***Spectrogram*** galaxy interface (*`specgram_interface`* key), and simulation settings (*`simulation_settings`* key).

The third part of the `run()` function is responsible for populating the data container, handling errors, and showing the resulting plot (see Figure 2-43).



```
# filling with interface data
specgram_interface['plot_title'] = PlotTitle
specgram_interface['interpolation'] = Interpolation
specgram_interface['window_function'] = WindowFunction
specgram_interface['STFT_Duration'] = STFT_Duration
specgram_interface['STFT_Overlap'] = STFT_Overlap
specgram_interface['id'] = ID
specgram_interface['scaling'] = PlotScaling
specgram_interface['colormap'] = PlotColormap
# initiate container with information about sweeps
simulation_settings['sweeps'] = sweep_info_string
simulation_settings['runs_iterations'] = run_iter_string
# filling with signal parameters
signal_params['freq_step'] = signal['T']/topology_TimeWindow
signal_params['c_freq'] = ((signal['f0'] + signal['f1']) *
                           signal['df']/2.)
signal_params['time_step'] = signal['dt']
# errors handling
if STFT_Duration > topology_TimeWindow:
    raise VPI_Error('STFT_Duration cannot be bigger than TimeWindow.')
elif STFT_Duration <= signal['dt']:
    raise VPI_Error('STFT_Duration must be bigger than ' +
                    '1/SampleRateDefault.')
elif STFT_Overlap >= 1:
    raise VPI_Error('STFT_Overlap must be lesser than 1.')

# filling container with block signal
signal_params['field'] = signal['bands'][0]['E']
# show matplotlib figure
nb.show(ID, run_iter_string)
```

Figure 2-43 *Filling the data container, handling errors, and showing the visualization*

As we can see, in the last line of this code the `nb.show()` command is executed. Showing the plot in the `run()` function provides a simple way of accessing information about the different iterations, runs, and sweeps and visualizing it immediately.

wrapper()

The `wrapper()` function that was called by the **`WrapupCommand`** is empty. There is no need to perform any actions while this function is being executed as the visualization is already shown.

Cosimulation with Dynamic-Link Libraries

This section describes the use of the cosimulation interface using a compiled dynamic-link library (DLL) as the target environment.

System Requirements

To perform cosimulation with dynamic link libraries, you will need to have a suitable compiler installed on your system. In principle, any compiler capable of generating dynamically linkable object files for the host operating system (that is, DLLs on Windows platforms, or shared objects on Unix platforms) is suitable. In practice, the differences between compilers, operating system environments, and development environments can be substantial, and it is beyond the scope of this section to provide a detailed tutorial on creating dynamic link libraries using all compilers in all environments. The sections below describe how to create DLLs using a Microsoft Visual C++ project in Microsoft Visual Studio.

Setting Up the Cosimulation Interface

To choose a dynamic-link library as the external program, set parameter **`InterfaceType`** of the *CoSimInterface* module to `Library`. In this mode, the interpretation of the various parameters of the cosimulation module is as follows:

- **`Path`**: the path to a directory containing the dynamic-link library object file.
- **`IfcFile`**: set this parameter to the name of the dynamic-link library object file. This file is linked to the running simulation executable during initialization of the cosimulation interface. It must contain exported definitions of all the required interface functions, as described below.

Note: It is possible to attach the DLL object file to the cosimulation galaxy package and run it from the `Inputs` package folder. To use an attached DLL, leave the **`Path`** parameter blank and specify only the DLL name in the **`IfcFile`** parameter.

- **`LogicalInformation`**: set this parameter to `Yes` to access the logical channels during the cosimulation. Note that the global parameter **`LogicalInformation`** should also be `On` or `CurrentRun`.

Dynamic-Link Library Interface

Unlike the MATLAB and Python interfaces, the library interface does not use the **`InitCommand`**, **`RunCommand`** and **`WrapupCommand`** parameters to define the functions to be called during cosimulation. Instead, a simple convention is used to define the names of the interface functions.

The interface between the simulation environment and the dynamic-link library consists of three structure definitions, four function definitions and two variables. They are used to call into the dynamic-link library.

The names of the interface functions and structures are derived from the instance ID of the cosimulation galaxy or the file name of the universe (if the *CoSimInterface* module is placed directly onto the universe). This may seem like an unusual convention, providing less flexibility than is the case with the MATLAB and Python interfaces; however, there is an important benefit in coupling the dynamic-link library to its associated galaxy. When cosimulating with native compiled code, any errors in the library, and especially errors which result in illegal memory accesses, will cause the simulator to crash — sometimes with quite dramatic results. As will become apparent, such problems are quite likely to occur if the parameters of the galaxy, or the number and type of input and output ports, are different from those expected by the compiled code. Thus tying the interface of the library to the associated galaxy provides an extra level of security against accidentally executing a dynamic-link library from an incompatible cosimulation galaxy.

To construct the base name for functions and structures, trailing digits are removed from the instance ID. This allows the same DLL to be used for instances which have instance IDs that differ only by the numeric suffix *(which is the standard approach for naming different instances of the same module)*.

Due to a limitation on identifier names in C/C++ instances, the ID of the cosimulation galaxy should not begin with a number.

Note: To simplify maintenance, it is best to use instance IDs that adhere to the standard naming schema: the instance ID for galaxy instances is constructed as `*GalaxyName*_vtmg*X*`, where *`GalaxyName`* is the name of the galaxy file and *`X`* is a unique number. The numeric suffix is incremented automatically each time a module is placed on the schematic or copied and pasted.
For example, if a DLL has been compiled for a galaxy named `CosimFilter.vtmg` with an ID `StopBandFilter_vtmg1`, after a copy-paste procedure, a new galaxy instance will be added to the schematic with an instance ID `CosimFilter_vtmg1`. This ID would need to be changed to `StopBandFilter_vtmgX` in order for the cosimulation to start *(here X is a numeric suffix which will be ignored)*.

Assuming that the instance ID of the cosimulation galaxy is CosimLib_vtmg1, the structures are:

- `struct CosimLib_vtmg_params` which is used to pass the values of the galaxy parameters from the simulation environment to the library;
- `struct CosimLib_vtmg_in` which is used to pass the cosimulation inputs from the simulation environment to the library;
- `struct CosimLib_vtmg_out` which is used to pass the cosimulation outputs from the library back to the simulation environment.

The functions are:

- `int CosimLib_vtmg_init(struct CosimLib_vtmg_params *)`
  This function is called when the interface is initialized, prior to the start of

simulation. It receives a pointer to a structure containing the initial parameter settings, and it must return `1` if it executed successfully, or `0` if an error occurred and simulation should be aborted.

- `int CosimLib_vtmg_run(struct CosimLib_vtmg_in *, struct CosimLib_vtmg_out *, struct CosimLib_vtmg_params *)`
  This function is called each time the cosimulation module is fired during the simulation run. It receives pointers to a structure containing the current inputs, a structure which must be used to store the outputs, and a structure containing the current parameter settings. You should note that parameter settings can change in the course of a simulation run (for example, by using the *ParameterController* modules); however, you are free to ignore such changes by making copies of the initial parameter settings in the initialization function. Thus cosimulation parameters may be "volatile" or "nonvolatile" according to how you implement the module (however, there is no way for the simulator to know, and to generate an error if an attempt is made to change a "nonvolatile" parameter).
  The run function must return `1` if it executed successfully, or `0` if an error occurred and simulation should be aborted. In case the global parameter **`LogicalInformation`** is set to On or `CurrentRun` and the corresponding parameter of the *CosimInterface* module is switched on — when logical information is transferred to cosimulation — additional structure appears in this function:
- `int CosimLib_vtmg_run(struct CosimLib_vtmg_in *, struct CosimLib_vtmg_out *, struct CosimLib_vtmg_params *,struct ONSL_LogicalInformation *)`
  `struct ONSL_LogicalInformation *` is a pointer to `ONSL_LogicalInformation` structure which holds the list of all logical channels. This structure also contains the "runnumber" member, which defines the current run number of given cosimulation module.
- `int CosimLib_vtmg_wrapup(struct CosimLib_vtmg_params *)`
  This function is called when the simulation is complete, prior to releasing all the resources used by the cosimulation interface. It receives a pointer to a structure containing the final parameter settings, and it must return `1` if it executed successfully, or `0` if an error occurred and simulation should be aborted.
- `const char *CosimLib_vtmg_lastError(void)`
  This function is called if any of the previous three functions return `0`, that is, it indicates that an error occurred. It can be used to return a string describing the cause of the error, which will be displayed in an error dialog, and written to the messages window. The pointer returned must point to memory that will remain allocated until the cosimulation interface is destroyed.

The two variables, which must be provided and exported by the cosimulation library, are

- `const char *ifc_name`
  This variable will be initialized by the cosimulation interface to point to a string uniquely identifying the instance of the cosimulation galaxy. This provides one mechanism that can be used to identify the specific instance of the cosimulation module in the case where there are multiple instances on a single schematic.

- `int ifc_CRCnew`
  This variable is initialized by the cosimulation interface to hold a checksum calculated over the interface structures (that is, the inputs, outputs and parameters). This should be checked at run-time against a checksum calculated when the galaxy was first created (provided in the generated header).

Generating a C Include File

You do not need to declare the interface functions, or define the input, output and parameter data structures for yourself. A special cosimulation target, `GenerateHeader`, is provided which automatically generates a C header file containing the necessary declarations and definitions.

To use this feature, set the parameter **`InterfaceType`** of the *CoSimInterface* module to `GenerateHeader`. The remaining parameters have the following meanings:

- **`Path`**: the path to the directory in which the generated header file should be placed.
- **`IfcFile`**: set this parameter to the desired name of the header file, for example `CosimLib.h`.
- **`LogicalInformation`**: set this parameter to `Yes` to access the logical channels during the cosimulation. Note that the global parameter **`LogicalInformation`** should also be `On` or `CurrentRun`.

C linkage is used for the interface functions. Thus, if you are using a C++ compiler, you must ensure that the functions are explicitly declared to use C linkage, by enclosing the declarations in an `extern ”C” { ... }` directive. The include files created by the `GenerateHeader` target already contain a suitable directive that should work on all standard C++ compilers.

Note: The GenerateHeader target uses the Python cosimulation interface to run a script which creates the header file. Thus, header generation will not work unless Python is correctly installed, as described in the section “System Requirements” on page 114. The script itself is called `genHeader.py`, located in the subdirectory `simeng/misc` of the VPI Design Suite installation. It is an interesting example of a script that discovers its own inputs, outputs and parameters by searching the dictionaries used to pass these variables into the Python cosimulation environment.

Handling Parameters

As mentioned above, parameters are passed to the init, run and wrapup functions as fields of the structure of the type `XXX_params`. Regular (non-function) parameters are mapped to native C types like `double`. Function parameters of the type `T` are represented with a structures of a type like the following:

```
COSIM_FuncParam_T {
        unit16_t type;
        unit16_t expr_type;
```

```
            char const* expr;
            void const* reserved;
            T value;
    };
```

Where `COSIM_FuncParam_T` stands for the corresponding parameter type: `COSIM_FuncParam_Double`, `COSIM_FuncParam_ComplexArray`, etc.

`type` contains the type of the parameter. The relation between type of the function parameter and value of the `type` property is illustrated in Table 2-20.

Table 2-20 *Relation between 'type' member and type of a function parameter in VPI Design Suite*

Parameter type	**Value of 'type' field**
Integer	`CSPT_INT`
Integer array	`CSPT_INT|CSPT_ARRAY`
Float	`CSPT_FLOAT`
Float array	`CSPT_FLOAT|CSPT_ARRAY`
Complex	`CSPT_COMPLEX`
Complex array	`CSPT_COMPLEX|CSPT_ARRAY`

`expr_type` contains an expression type of the parameter. The relation between type of the function parameter and value of the `type` property is illustrated in Table 2-21.

Table 2-21 *Relation between 'expr_type' member and expression type of a function parameter in VPI Design Suite*

Parameter type	**Value of 'expr_type' field**
Standard	`CSET_STANDARD`
Python	`CSET_PYTHON`
File	`CSET_FILE`

Expression type and value follow rules explained under "Function Parameters Evaluation" on page 91.

If expression type of the parameter was '`Python`' and the specified Python expression was successfully calculated (for example, it does not contain variables like `$x`) and its result is numerical, then the expression type of this parameter is transformed to '`Standard`'.

`expr` contains an original function parameter expression or file path. Not available for parameters with Standard expression type.

`value` contains a value of the parameter if parameter expression type is '`Standard`'. Otherwise (expression type is '`Python`' or '`File`') it returns a default value of the corresponding parameter.

If expression type of the function parameter is not '`File`', parameters expression can be evaluated using the method `COSIM_Interface::evalFuncExpr(`*`funcParam, args, result`*`)` described under "Simulation Environment Interface" on page 163.

For a more elaborate example, see the demonstration: *Simulation Techniques > Cosimulation > Library > Function Parameters in Cosimulation (Lib)*.

Mapping VPI Signals to C Data Types

The library cosimulation interfaces uses C structures to map VPI ONSL signals. The arrays of sampled bands, parameterized signals and noise bins in the structure representing an optical block signal are implemented as ordinary C arrays. As C arrays require explicit storage of the array length, the C mapping of the structures contains additional integer variables defining the length of each array. The matrix data types are also represented as structures in the C interface.

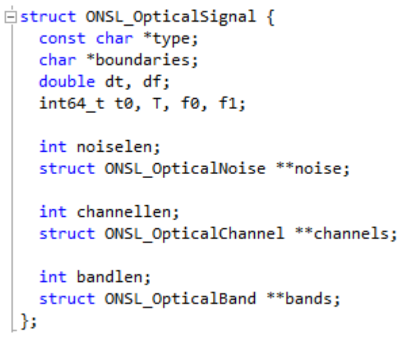
Note: The header file `cosim.h` contains the definitions of all the mapped data structures. This header file is required by, and automatically included in, header files created using the `GenerateHeader` target. You must ensure that the directory containing this header file is added to the path your compiler uses to search for include files. It is located in the subdirectory `simeng\cosim\include` relative to the installation directory.

What follows is a summary of the data structures defined in `cosim.h`. Except in a few cases of particular importance, or to demonstrate an important principle, full details are not provided. You should refer to the header file itself for a full definition of each structure.

- `struct ONSL_Complex`
  A data structure representing a complex number, consisting simply of the real and imaginary parts (as `double`). The examples presented later illustrate that this structure can usually be safely cast to the standard C++ type `complex<double>`, or to the double precision complex types defined by the MIT FFTW package.
- `struct ONSL_IntArray`
  Integer array type.

-
  ```
  struct ONSL_IntMatrix {
  int **data;
  int rows, cols;
  };
  ```
  Integer matrix type. Matrices are defined as an array of pointers to ordinary single dimensional arrays. Each element of the `data` array is a pointer to an array of `cols` integers representing a single row of the matrix. The `data` array itself contains `rows` elements. Thus the element at the sixth column of the fourth row is `data[3][5]`.

- `struct ONSL_FloatArray`
  Double precision floating-point array type.
- `struct ONSL_FloatMatrix`
  Double precision floating-point matrix type.
- `struct ONSL_ComplexArray`
  Complex array type.
- `struct ONSL_ComplexMatrix`
  Complex matrix type.
- `struct ONSL_StringArray`
  String array type.
- `struct ONSL_OpticalSignal`
  The structure defining an optical signal, shown in Figure 2-44. This structure closely follows the general optical signal format shown in Table 2-2, with a few minor differences. The integers `noiselen`, `channellen` and `bandlen` contain the number of noise bins, parameterized channels, and sampled bands respectively, in the `noise`, `channels` and `bands` arrays. These array are themselves arrays of pointers to instances of the corresponding structures, discussed below. The `type` field should be a pointer to the string "`osignal`", indicating that the structure contains optical signal data. To test this field, compare it (using `strcmp()`) with the provided string constant `ONSL_OpticalSignalId_c`. If you are constructing your own optical signals, you should set `type` equal to this string constant. The string array `ONSL_OpticalSignalFields_c[]` that is also defined in the `cosim.h` file (not shown in Figure 2-44) may change or be removed in future releases, and should not be relied upon in your own code. This warning applies to all similar string arrays in `cosim.h`.

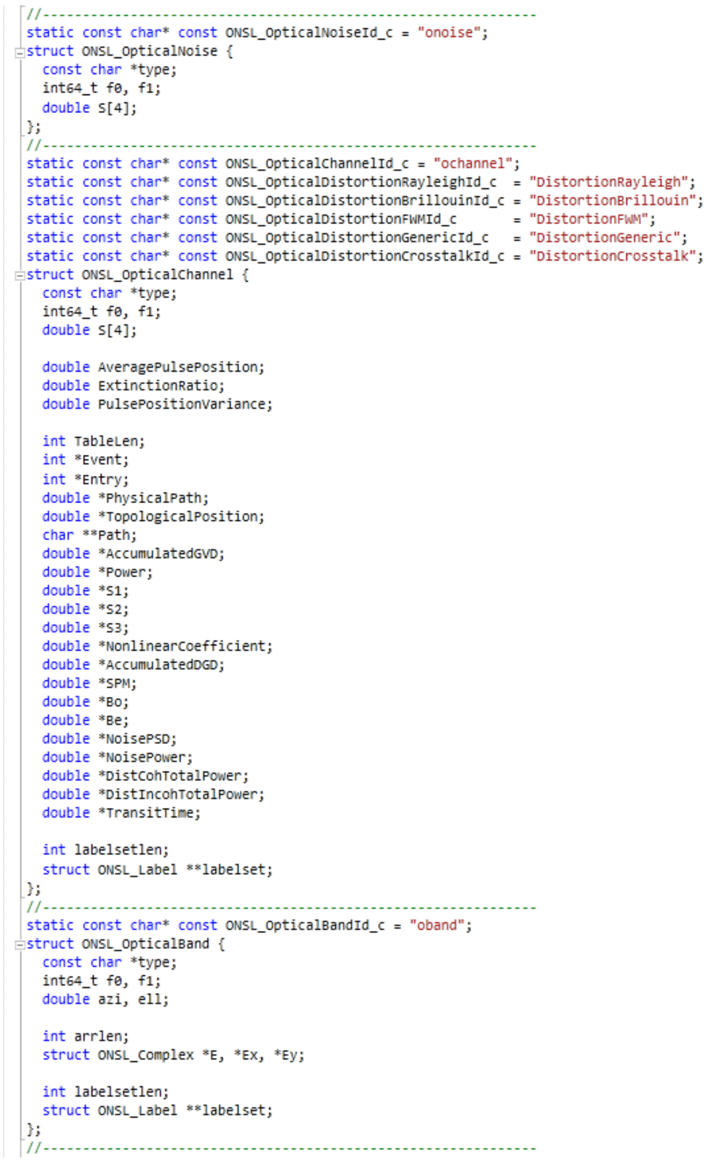


```
struct ONSL_OpticalSignal {
   const char *type;
   char *boundaries;
   double dt, df;
   int64_t t0, T, f0, f1;

   int noiselen;
   struct ONSL_OpticalNoise **noise;

   int channellen;
   struct ONSL_OpticalChannel **channels;

   int bandlen;
   struct ONSL_OpticalBand **bands;
};
```

Figure 2-44 *C structure defining an optical signal*

- `struct ONSL_OpticalNoise`
  `struct ONSL_OpticalChannel`
  `struct ONSL_OpticalBand`
  The structures representing noise bins, parameterized channels, and sampled bands are shown in Figure 2-45. These have the format you would probably expect, based on the tables in Table 2-2. Note that the Stokes parameters arrays are part of the `ONSL_OpticalNoise` and `ONSL_OpticalChannel` structures, and do not have

separate memory allocated for them. The arrays containing the sampled bands, on the other hand, are represented by pointers in the `ONSL_OpticalBand` structure, and reside in separately allocated memory. The field `arrlen` contains the number of samples in the sampled band. Any particular sampled band will comprise either a single array `E` representing a fully-polarized signal, or two arrays `Ex` and `Ey` representing the X and Y polarization components of an arbitrarily polarized signal. Unused arrays are represented by null pointers, and the field `arrlen` applies to whichever arrays are non-null. As with the `ONSL_OpticalSignal` structure, the `type` field in each case is a pointer to a constant string, which should be initialized using the constant provided if you are creating your own signal structures.

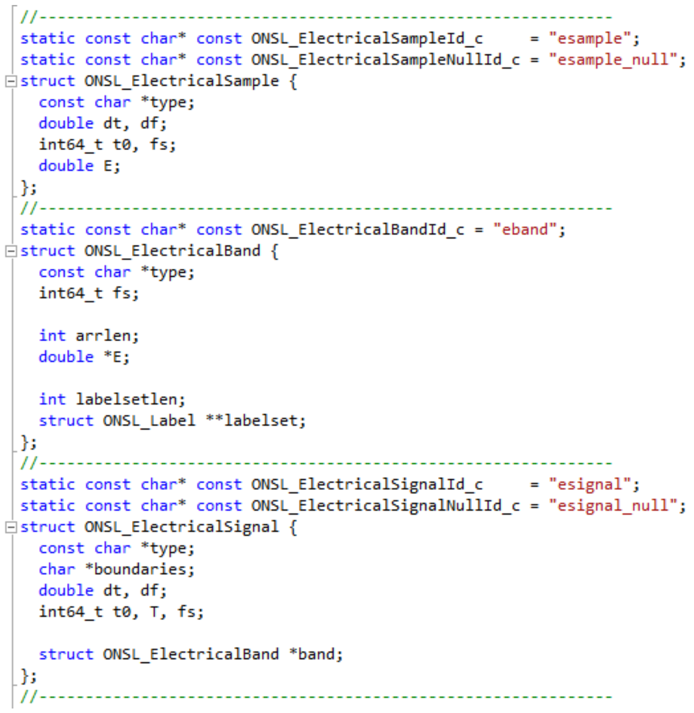


```
//-------------------------------------------------------------
static const char* const ONSL_OpticalNoiseId_c = "onoise";
struct ONSL_OpticalNoise {
  const char *type;
  int64_t f0, f1;
  double S[4];
};
//-------------------------------------------------------------
static const char* const ONSL_OpticalChannelId_c = "ochannel";
static const char* const ONSL_OpticalDistortionRayleighId_c  = "DistortionRayleigh";
static const char* const ONSL_OpticalDistortionBrillouinId_c = "DistortionBrillouin";
static const char* const ONSL_OpticalDistortionFWMId_c       = "DistortionFWM";
static const char* const ONSL_OpticalDistortionGenericId_c   = "DistortionGeneric";
static const char* const ONSL_OpticalDistortionCrosstalkId_c = "DistortionCrosstalk";
struct ONSL_OpticalChannel {
  const char *type;
  int64_t f0, f1;
  double S[4];

  double AveragePulsePosition;
  double ExtinctionRatio;
  double PulsePositionVariance;

  int TableLen;
  int *Event;
  int *Entry;
  double *PhysicalPath;
  double *TopologicalPosition;
  char **Path;
  double *AccumulatedGVD;
  double *Power;
  double *S1;
  double *S2;
  double *S3;
  double *NonlinearCoefficient;
  double *AccumulatedDGD;
  double *SPM;
  double *Bo;
  double *Be;
  double *NoisePSD;
  double *NoisePower;
  double *DistCohTotalPower;
  double *DistIncohTotalPower;
  double *TransitTime;

  int labelsetlen;
  struct ONSL_Label **labelset;
};
//-------------------------------------------------------------
static const char* const ONSL_OpticalBandId_c = "oband";
struct ONSL_OpticalBand {
  const char *type;
  int64_t f0, f1;
  double azi, ell;

  int arrlen;
  struct ONSL_Complex *E, *Ex, *Ey;

  int labelsetlen;
  struct ONSL_Label **labelset;
};
//-------------------------------------------------------------
```

Figure 2-45 *C structures representing optical signal components*

- `struct ONSL_OpticalSample`
  Represents a single optical sample, and has exactly the form shown in Table 2-1. It also has a constant "`osample`" provided to initialize or test the `type` field.
- `struct ONSL_ElectricalSignal` (electrical signal definition).
  `struct ONSL_ElectricalBand` (electrical sampled band definition).
  `struct ONSL_ElectricalSample` (electrical sample definition).
  The structures representing electrical signals and samples are shown in Figure 2-46. They are defined and operate according to the same principles as the

equivalent optical signal. Electrical signals are simpler because electrical noise bins and parameterized signals are not supported, only a single sampled band is allowed, and all electrical samples are real-valued.



```
//-------------------------------------------------------------
static const char* const ONSL_ElectricalSampleId_c     = "esample";
static const char* const ONSL_ElectricalSampleNullId_c = "esample_null";
struct ONSL_ElectricalSample {
  const char *type;
  double dt, df;
  int64_t t0, fs;
  double E;
};
//-------------------------------------------------------------
static const char* const ONSL_ElectricalBandId_c = "eband";
struct ONSL_ElectricalBand {
  const char *type;
  int64_t fs;

  int arrlen;
  double *E;

  int labelsetlen;
  struct ONSL_Label **labelset;
};
//-------------------------------------------------------------
static const char* const ONSL_ElectricalSignalId_c     = "esignal";
static const char* const ONSL_ElectricalSignalNullId_c = "esignal_null";
struct ONSL_ElectricalSignal {
  const char *type;
  char *boundaries;
  double dt, df;
  int64_t t0, T, fs;

  struct ONSL_ElectricalBand *band;
};
//-------------------------------------------------------------
```

Figure 2-46 *Structure definitions for electrical signal types*

Additionally, the library interface uses special data types to pass optical and electrical signals in and out of the dynamic-link library. These are `unions`, which can be used to represent a pointer to any optical type (signal or sample), or any electrical type (signal or sample) respectively. They also enable direct access to the `type` field of the signal or sample, so that it is possible to determine what kind of pointer is actually passed. The union for optical types is defined as

```
union ONSL_Optical {
char **type;
struct ONSL_OpticalSample *sample;
struct ONSL_OpticalSignal *signal;
};
```

The `type` field of the union can be used to check if the structure is actually representing a block or a sample in the following manner

```
union ONSL_Optical sig;
if ( !strcmp( *sig.type, "osignal" ) ) {
      // Is block signal
      // Do whatever is required for
optical blocks...
}
else if ( !strcmp( *sig.type, "osample" )
) {
      // Is a sample
      // Do whatever is required for
samples...
}
else {
// Some problem has occurred. Arrange for
error
             // handling...
}
```

A similar union is defined for electrical signals and sampled signals.

As an example of how components of an optical signal carried within these structures can be accessed, the second component of the Stokes vector of the 42nd noise bin in the optical signal `sig` would be referenced using

```
sig.signal->noise[41]->S[1]
```

In cases when logical information is cosimulating, the library interface uses another structure to describe optical and electrical signals with an additional field labelset for optical bands and channels:

```
struct ONSL_OpticalChannelLI {
      const char *type;
      int64_t f0, f1;
      double S[4];
      double AveragePulsePosition;
      double ExtinctionRatio;
      double PulsePositionVariance;
      int TableLen;
      int *Event;
      int *Entry;
      double *PhysicalPath;
      double *TopologicalPosition;
      char **Path;
      double *AccumulatedGVD;
      double *Power;
      double *S1;
      double *S2;
      double *S3;
```

```
        double *NonlinearCoefficient;
        double *AccumulatedDGD;
        double *SPM;
        double *Bo;
        double *Be;
        double *NoisePSD;
        double *NoisePower;
        double *DistCohTotalPower;
        double *DistIncohTotalPower;
        double *TransitTime;
        int labelsetlen;
        struct ONSL_Label **labelset;
};

struct ONSL_OpticalBandLI {
        const char *type;
        int64_t f0, f1;
        double azi, ell;
        int arrlen;
        struct ONSL_Complex *E, *Ex, *Ey;
        int labelsetlen;
        struct ONSL_Label **labelset;
};

struct ONSL_OpticalSignalLI {
        const char *type;
        char *boundaries;
        double dt, df;
        int64_t t0, T, f0, f1;
        int noiselen;
        struct ONSL_OpticalNoise **noise;
        int channellen;
        struct ONSL_OpticalChannelLI **channels;
        int bandlen;
        struct ONSL_OpticalBandLI **bands;
};

union ONSL_OpticalLI {
        char **type;
        struct ONSL_OpticalSample *sample;
        struct ONSL_OpticalSignalLI *signal;
};
```

The structures describing electrical signals are modified also:

```
struct ONSL_ElectricalBandLI {
        const char *type;
        int64_t fs;
        int arrlen;
        double *E;
        int labelsetlen;
        struct ONSL_Label **labelset;
};

struct ONSL_ElectricalSignalLI {
        const char *type;
        char *boundaries;
        double dt, df;
        int64_t t0, T, fs;
        struct ONSL_ElectricalBandLI *band;
};

union ONSL_ElectricalLI {
        char **type;
        struct ONSL_ElectricalSample
*sample;
        struct ONSL_ElectricalSignalLI
*signal;
};
```

Additional structures are available when logical information is transferred to the cosimulation interface (see Tables 2-9–2-13):

```
struct ONSL_SignalSource {
        const char *type;
        char *name;
        int64_t f0, f1, t0, T;
        double dt, df;
};

struct ONSL_Modulation {
        const char *type;
        char *subtype;
        int bitrate;
        double df;

        double modulationindex;
        double frequencydeviation;
        double phasedeviation;

        char *usersubtype;
        int  numberofuserparameters;
        struct Strings_Array
**userparameters;
};

struct ONSL_PulseShape {
        const char *type;
        char *subtype;
//arbitrary parameter
        double bias;
//subtype dependant parameters
```

```
        double dutycycle;
        double shift;
        double amplitude;
        double centerpos;
        double t_fwhm;
        double extrabits;
        double order;
        double chirp;
        double pulsewidth;
        double alpha;
        char* pulsename;
        int pulselen;
        ONSL_Complex *pulse;
        int duration;
        double dt;
};

struct ONSL_LineCoding {
        const char *type;
        char *subtype;
        double lastbit;
        int isdifferential;
};

struct ONSL_LogicalChannel {
        const char *type;
        char *label;
        int seqnumber;
        struct ONSL_Label *modulateelabel;
        struct ONSL_Label *modulatorlabel;
        struct ONSL_SignalSource *signalsource;
        struct ONSL_Modulation *modulation;
        int bitstreamlen;
        long *bitstream;
        struct ONSL_PulseShape *pulseshape;
        struct ONSL_LineCoding *linecoding;
};

struct ONSL_LogicalInformation {
        const char *type;
        int runnumber;
        int channelslen;
        struct ONSL_LogicalChannel **channel;
};
```

Simulation Environment Interface

The dynamic-link library interface is used by the simulation environment to call the cosimulation implementation. The simulation environment interface provides access to functions implemented by the simulation environment, such as memory allocation and message/warning reporting routines.

The following set of structures, classes, macros and functions are provided to work with simulation environment interface:

Types

`COSIM_Interface`

Purpose:

This type is defined differently for C and C++. In C, it is a structure which contains pointers to functions provided by the cosimulation environment. In C++, it is a class which wraps the C structure to simplify access to those functions.

In most cases, there is no need to use objects of this type directly. For example, memory allocation functions and message reporting routines are provided as more convenient macros (e.g., `COSIM_CREATE_OBJECT()` and `COSIM_INFO()`).

Example:

```
/* In C use this form */
(*pIface)->writeMessage(pIface, CSMT_WARNING, "Warning!!!");

// In C++ use this form
pIface->writeMessage(CSMT_WARNING, "Warning!!!");
```

`COSIM_Interface::allocMem(size)`

Purpose:

Allocates a block of memory of the specified size in bytes.

Use of this function is **vital** for objects which are passed back to the simulation environment (for example as output signal). But in practice, it is more convenient to use either the `COSIM_CREATE_OBJECT()/COSIM_CREATE_ARRAY()` macros or `cosim::createObject()/cosim::createArray()` functions.

Parameters:

`size` size of block in bytes to allocate

Returns:

A pointer to the newly allocated memory.

Example:

```
out->output.signal = (ONSL_OpticalSignal*)pIface->allocMem(
sizeof(ONSL_OpticalSignal));
```

`COSIM_Interface::evalFuncExpr(funcParam, args, result)`

Purpose:

Evaluates a Python expression of a function parameter.

Parameters:

`funcParam` a pointer to a structure of type `COSIM_FuncParam`.

args a map (represented as an array of name-value pairs) of arguments that must be used as values of parameters during the expression evaluation. For example, if the expression contains the parameter `$V`, then the name of the corresponding argument in the *args* map needs to be `$V` too.

result a pointer to a structure of type `COSIM_Variant`.

The dimension of type of the variable could be increased by one dimension for simultaneous evaluation of the Python expression for several values of this variable. Thus a single call of the `evalFuncExpr(...)` method could be performed instead of multiple calls. Correspondingly, the dimension of the result value of the `evalFuncExpr(...)` method is equal to or greater by one dimension than the dimension of the type of the evaluated parameter. For example, if a Python expression takes a float argument `$x` and returns a complex value, then if an array of floats is passed to `$x` in *args*, the result of evaluation will be an array of complex values corresponding to the evaluation results for each value in the array `$x`.

`COSIM_Interface::freeMem(ptr)`

Purpose:

Deallocates a memory buffer that was allocated using `allocMem` function.

If a memory buffer was allocated using other functions (malloc, operator new, etc.) and is deallocated using the `freeMem` function, program behavior is undefined. Memory for structures (and their members) that are passed from the simulation engine can be deallocated using the `freeMem` function. After memory is deallocated, it is necessary to set the pointer to `NULL`.

Use of this function is **vital** for objects which are passed by the simulation environment (for example as input signal). But in practice, it is more convenient to use either `COSIM_DESTROY_OBJECT()/COSIM_DESTROY_ARRAY()` macros or `cosim::destroyObject()/cosim::destroyArray()` functions.

Parameters:

ptr pointer to memory to deallocate

Returns:

None.

Example:

```
pIface->freeMem(in->input.signal->noise[i]);
in->input.signal->noise[i] = NULL;
```

`COSIM_Interface::getSweepInfo()`

Purpose:

This function returns the iteration (sweep) information, see "Iteration (Sweep) Information" on page 90, in a variable of type `COSIM_SweepInfo`.

Parameters:

None.

Example:

The following code gets COSIM_SweepInfo from the simulation engine and checks the number of iteration levels, the current iteration number, and the total number of steps for 0 sweep level:

```
COSIM_SweepInfo* swInfo = getSweepInfo();
int len = swInfo->len;
int iter = swInfo->levels[0].current;
int sw0_total = swInfo->levels[2].total;
```

COSIM_Interface::isRemoteSimulation()

Purpose:

This function returns `true` if simulation is being executed at the Remote Simulation Server.

Parameters:

None.

Example:

```
const bool isRemoteSimulation = pIface->isRemoteSimulation();
```

COSIM_Interface::writeMessage(*type*, *msg*)

Purpose:

This function may be used to send a message of the specified type to the VPI Design Suite Message Log.

Supported message types are info, warnings and progress messages. Progress messages are processed if Report Progress is switched on in the Submit Simulation Job dialog.

In practice, it is more convenient to use the `COSIM_MESSAGE_EX()/ COSIM_INFO()/COSIM_WARNING()` or `COSIM_PROGRESS()` macro.

Parameters:

type type of message to send (`CSMT_INFO`, `CSMT_WARNING`, `CSMT_PROGRESS` for info, warning or progress respectively).

msg message to be sent.

Note: The type `CSMT_MESSAGE` is a deprecated alias for the `CSMT_INFO`.

Example:

```
pIface->writeMessage(CSMT_WARNING, "Warning!!!");
```

COSIM_SweepInfo

Purpose:

This structure contains an array of `COSIM_SweepLevel` structures, `levels`, and the array length, `len`, providing all the iteration levels of cosimulated module.

`COSIM_SweepLevel`

Purpose:

This structure contains information on the `current` iteration on a specific level and the `total` number of iterations.

Macros

`IMPLEMENT_COSIM()`

Purpose:

This macro implements the infrastructure necessary to access the simulation environment interface. It must be put into one (and only one) of the source files.

`GET_COSIM_INTERFACE()`

Purpose:

This macro should be used to acquire a pointer to the simulation environment interface.

Returns:

Pointer to `COSIM_Interface` or `NULL` if the interface cannot be acquired.

Example:

```
COSIM_Interface* pIface = GET_COSIM_INTERFACE();
```

`COSIM_CREATE_OBJECT(`*`T`*`)`

Purpose:

This macro should be used to create ONSL structures which are going to be sent back to the simulation environment via output ports.

Parameters:

T type of object to create

Returns:

Pointer to the object.

Example:

```
out->output.signal = COSIM_CREATE_OBJECT(ONSL_OpticalSignal);
```

`COSIM_DESTROY_OBJECT(`*`ptr`*`)`

Purpose:

This macro should be used to free memory occupied by ONSL structures which are sent by the simulation environment via input ports.

Parameters:

ptr pointer to object to destroy

Example:

```
COSIM_DESTROY_OBJECT(in->input.signal->noise[i]);
in->input.signal->noise[i] = NULL;
```

COSIM_CREATE_ARRAY(*T, n*)

Purpose:

This macro should be used to create an array of ONSL structures or primitive types which are going to be sent back to the simulation environment via output ports.

Parameters:

T type of objects to store in the array

n number of object to store

Returns:

Pointer to the array.

Example:

```
out->output.signal->bands = COSIM_CREATE_ARRAY(ONSL_OpticalBand*, 1);
```

COSIM_DESTROY_ARRAY(*ptr*)

Purpose:

This macro should be used to free memory occupied by an array of ONSL structures or primitive types which are sent by the simulation environment via input ports.

Parameters:

ptr pointer to the array

Example:

```
COSIM_DESTROY_OBJECT(in->input.signal->noise);
in->input.signal->noise = NULL;
```

COSIM_IS_REMOTE_SIMULATION()

Purpose:

This macro returns **`true`** if a simulation is being executed at the Remote Simulation Server.

Parameters:

None.

Example:

```
const bool isRemoteSimulation = COSIM_IS_REMOTE_SIMULATION();
```

`COSIM_MESSAGE_EX(`*`type, msg`*`)`

Purpose:

This macro may be used to send a message of the specified type to the VPI Design Suite **Message Log**.

Supported message types are info, warnings and progress messages. Progress messages are processed if **Report Progress** is switched on in the **Submit Simulation Job** dialog.

Parameters:

`type` type of message to send (`CSMT_INFO`, `CSMT_WARNING`, `CSMT_PROGRESS` for message, warning or progress, respectively).

`text` message to be sent.

Note: The type `CSMT_MESSAGE` is a deprecated alias for the `CSMT_INFO`.

Example:

```
COSIM_MESSAGE_EX(CSMT_WARNING, "Warning!!!");
```

`COSIM_INFO(`*`msg`*`)`
`COSIM_WARNING(`*`msg`*`)`
`COSIM_PROGRESS(`*`msg`*`)`

Purpose:

Shorthand for `COSIM_MESSAGE_EX()` with appropriate message type.

Note: The macro `COSIM_MESSAGE` is a deprecated alias for the `COSIM_INFO`.

Parameters:

`msg` message to be sent.

Example:

```
COSIM_PROGRESS("Running...");
```

Functions

`cosim::createObject<`*`T`*`>()`

Purpose:

This template function is a C++ alternative to the `COSIM_CREATE_OBJECT()`macro and should be used to create ONSL structures which are going to be sent back to the simulation environment via output ports.

Parameters:

T type of object to create.

Returns:

Pointer to the object.

Example:

```
out->output.signal = createObject<ONSL_OpticalSignal>();
```

`cosim::destroyObject(`*ptr*`)`

Purpose:

This function is a C++ alternative to the `COSIM_DESTROY_OBJECT()`macro and should be used to to free memory occupied by ONSL structures which are sent by the simulation environment via input ports.

Parameters:

ptr pointer to object to destroy

Example:

```
destroyObject(in->input.signal->noise[i]);
in->input.signal->noise[i] = NULL;
```

`cosim::createArray<`*T*`>(`*n*`)`

Purpose:

This template function is a C++ alternative to the `COSIM_CREATE_ARRAY()`macro and should be used to create an array of ONSL structures or primitive types which are going to be sent back to the simulation environment via output ports.

Parameters:

T type of objects to store in the array

n number of object to store

Returns:

Pointer to the array.

Example:

```
out->output.signal->bands = createArray<ONSL_OpticalBand*>(1);
```

`cosim::destroyArray(`*ptr*`)`

Purpose:

This template function is a C++ alternative to the `COSIM_DESTROY_ARRAY()` macro and should be used to free memory occupied by an array of ONSL structures or primitive types which are sent by the simulation environment via input ports.

Parameters:

`ptr` pointer to the array

Example:

```
destroyArray(in->input.signal->noise);
in->input.signal->noise = NULL;
```

Additional Compiler Considerations

Memory Alignment

When compiling cosimulation modules, it is important to ensure that the data structures generated by your compiler have exactly the same layout in memory as those constructed by the VPI Design Suite simulator. This is because the structures are passed between the simulator and the cosimulation environment at runtime in binary form. If your code has a different idea of where it will find each element within a data structure, obviously it will not function correctly.

The structures that are generated by the cosimulation interface use a memory alignment of eight bytes. This may require that you set the appropriate flag for your compiler accordingly. However, this kind of alignment is default for most compilers on x86 architectures. If you experience problems with access of data passed into the interface, check this setting in your compiler.

Memory Management

Further considerations are necessary if you wish to allocate memory yourself that will be passed back to the simulation environment or free memory which was passed by the simulation environment. The simulator is responsible for freeing all memory passed across the interface from the target environment. Thus, if you allocate memory you must ensure that you do so in such a way that the system knows how to release it correctly. If you deallocate memory, you should ensure that all objects are freed and pointers set to `NULL` to avoid recurrent deallocation.

Several macros and functions are provided to meet the above requirement:

- `COSIM_CREATE_OBJECT()`
- `COSIM_CREATE_ARRAY()`
- `COSIM_DESTROY_OBJECT()`
- `COSIM_DESTROY_ARRAY()`
- `cosim::createObject()`
- `cosim::createArray()`
- `cosim::destroyObject()`

- `cosim::destroyArray()`

These functions must be used instead of the `new` operator and the `malloc()` function whenever memory is allocated for passing to the simulation environment or the `delete` operator and the `free()` function when memory needs to be deallocated in the cosimulation code.

Naming Convention

It is good practice to name the resulting dynamic-link library file in accordance with the following template: *`<base_name>_<os>_<cpu_arch>.<ext>`*. Where *`<base_name>`* is the desired name of the .dll file without suffixes *(usually, the name of the project file)*, *`<os>`* is the abbreviation of target operating system (`win` for Windows, `lnx` for Linux), *`<cpu_arch>`* is the abbreviation of CPU architecture (`x86` or `x64`) and *`<ext>`* is the file name extension of the library file specific for the operating system (`dll` for Windows and `so` for Linux). For instance, the name of the .dll file for a project called "`Filter`" for 64-bit Windows would be `Filter_win_x64.dll`.

This special naming technique is very useful because it allows you to create a cosimulation galaxy which is independent of the target operating system.

There are several variables defined in the simulation engine: `PTCL_OS`, `PTCL_ARCH`, `LIB_EXT`. These variables can be used in expressions defined in the **IfcFile** parameter of the **CoSimInterface** module.

`PTCL_OS` is evaluated to abbreviation of target operating system:

- `win` - for Windows
- `lnx` - for Linux

`PTCL_ARCH` is evaluated to abbreviation of CPU architecture of target operating system:

- `x86` - for Intel and AMD 32-bit processors
- `x64` - for Intel EM64T and AMD64 processors.

`LIB_EXT` is evaluated to extension of dynamic-link library specific to target platform:

- `dll` - for Windows
- `so` - for Linux

For the above example, the `IfcFile` parameter for the **Filter** module can be set to `!"Filter_${PTCL_OS}_${PTCL_ARCH}.${LIB_EXT}"`.

Using the Dynamic-Link Library Cosimulation Interface

A Simple Example: Adding Two Floating-Point Values

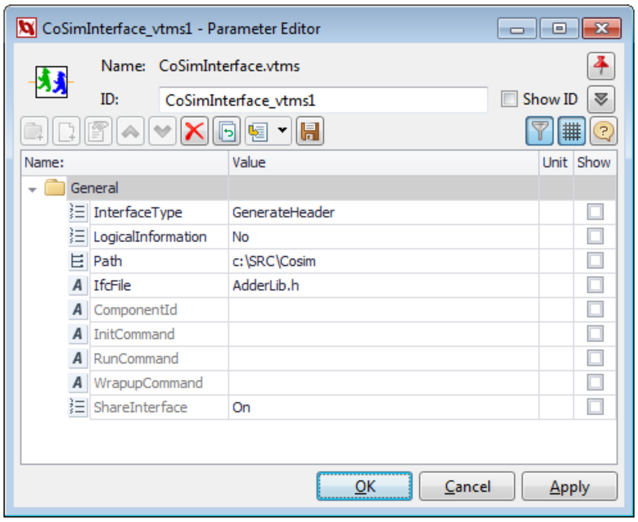
The simple adder example previously presented for MATLAB and Python cosimulation requires a great deal more effort to set up as a library cosimulation module. Normally it would not be worthwhile to use compiled C code for such a simple application, in which there is little to be gained in terms of efficiency. However, for the sake of completeness, and to illustrate the steps involved, we will develop the example here.

Let us start by setting out the objectives and requirements for the development. The goal is to develop a dynamic-link library cosimulation module with the following characteristics and constraints:

- The module will have two floating-point inputs, *input1* and *input2*, and one floating-point output, *output*.
- The module will have a single floating-point parameter, **`Gain`**.
- The output shall be equal to the sum of the two inputs, multiplied by the **`Gain`**.
- The module will be called *AdderLib*.
- The dynamic-link library will be developed under Windows, and the source and object files will be stored in the directory `C:\SRC\Cosim\AdderLib`.

The first step is to build the cosimulation galaxy that will implement the module. The galaxy is identical to that used in the MATLAB and Python adder examples (see Figure 2-6). The same galaxy may be reused, but will now be saved under the name *AdderLib*, because that is the name we have chosen for our module.

The next step is to generate the C header file. This is achieved by setting the parameters of the *CoSimInterface* module in the *AdderLib* galaxy as shown in Figure 2-47.



Name: CoSimInterface.vtms
ID: CoSimInterface_vtms1

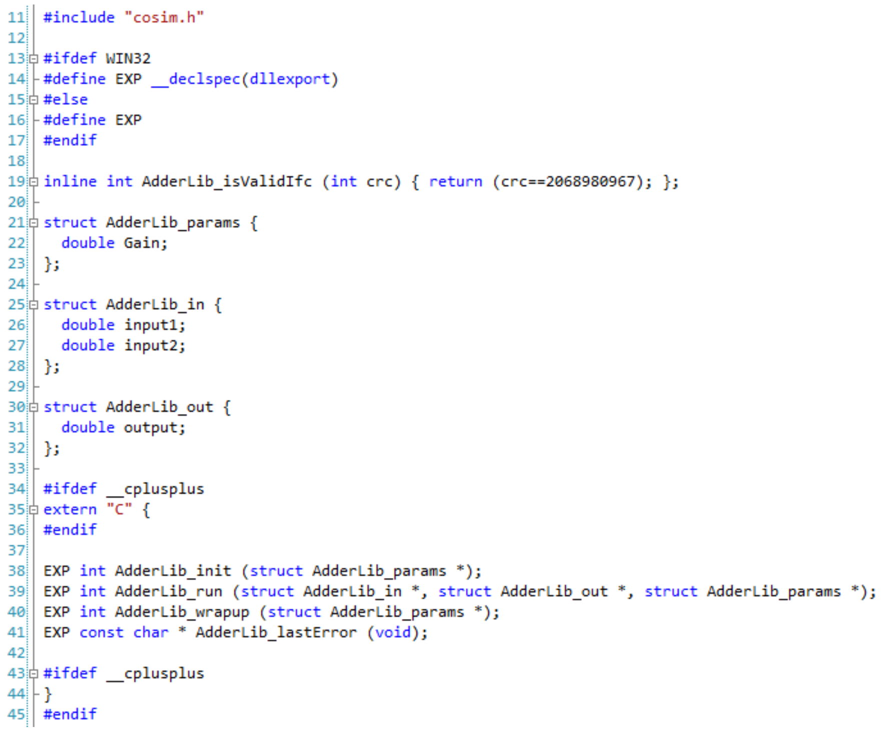
Name:	Value	Unit	Show
General			
InterfaceType	GenerateHeader		☐
LogicalInformation	No		☐
Path	c:\SRC\Cosim		☐
IfcFile	AdderLib.h		☐
ComponentId			☐
InitCommand			☐
RunCommand			☐
WrapupCommand			☐
ShareInterface	On		☐

Figure 2-47 *CoSimInterface parameters for generation of the AdderLib.h file*

The header file is generated by creating and running a schematic containing the ***AdderLib*** galaxy. When this is done, the file `AdderLib.h` is created in the directory `C:\SRC\Cosim\AdderLib`. The body of the file generated (excluding heading comments and the `#define` "guard" against multiple inclusion) is shown in Figure 2-48. The following structures have been defined:

- The parameter structure `AdderLib_params`, containing the floating-point parameter **`Gain`** as a `double`.
- The input structure `AdderLib_in`, containing the inputs *input1* and *input2* as `doubles`.
- The output structure `AdderLib_out`, containing the output *output* as a `double`.

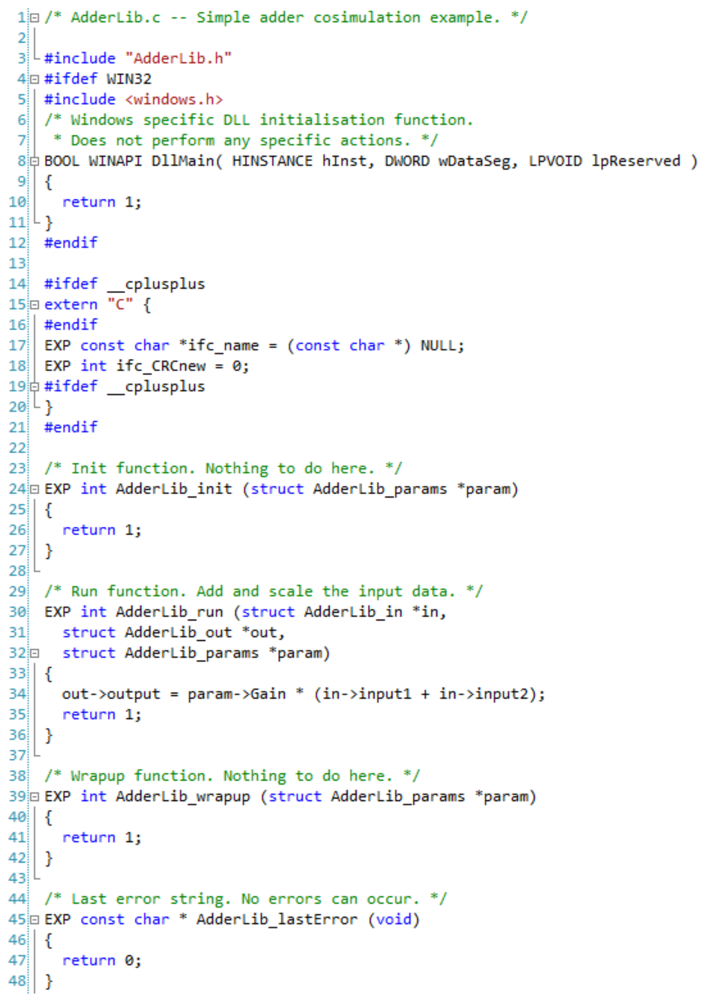
Additionally, the four interface functions `AdderLib_init()`, `AdderLib_run()`, `AdderLib_wrapup()` and `AdderLib_lastError()` have been declared. Note that the header also defines a macro `EXP` (for "export"), which is a platform dependent type modifier. Under Windows it is necessary to use the modifier `__declspec(dllexport)` on any external function or variable which is to be available to the application loading the DLL at runtime, whereas this is not necessary on Unix platforms. By using the macro `EXP` when declaring exported functions and variables, it is possible to write portable code. Note, however, that there are typically many other issues to consider when writing portable code, and in many cases (especially when platform-specific libraries and system calls are used), it may be very difficult to achieve full portability. Further discussion of this topic is beyond the scope of this manual, and if you need to develop fully portable code you should consult the relevant documentation of all target systems.



```
#include "cosim.h"

#ifdef WIN32
#define EXP __declspec(dllexport)
#else
#define EXP
#endif

inline int AdderLib_isValidIfc (int crc) { return (crc==2068980967); };

struct AdderLib_params {
  double Gain;
};

struct AdderLib_in {
  double input1;
  double input2;
};

struct AdderLib_out {
  double output;
};

#ifdef __cplusplus
extern "C" {
#endif

EXP int AdderLib_init (struct AdderLib_params *);
EXP int AdderLib_run (struct AdderLib_in *, struct AdderLib_out *, struct AdderLib_params *);
EXP int AdderLib_wrapup (struct AdderLib_params *);
EXP const char * AdderLib_lastError (void);

#ifdef __cplusplus
}
#endif
```

Figure 2-48 *Body of the AdderLib.h file generated*

C/C++ source code implementing the adder is shown in Figure 2-49. This code has been written for maximum portability. It should compile on Windows or Unix systems, using either an ANSI C or C++ compiler.



```
/* AdderLib.c -- Simple adder cosimulation example. */

#include "AdderLib.h"
#ifdef WIN32
#include <windows.h>
/* Windows specific DLL initialisation function.
 * Does not perform any specific actions. */
BOOL WINAPI DllMain( HINSTANCE hInst, DWORD wDataSeg, LPVOID lpReserved )
{
  return 1;
}
#endif

#ifdef __cplusplus
extern "C" {
#endif
EXP const char *ifc_name = (const char *) NULL;
EXP int ifc_CRCnew = 0;
#ifdef __cplusplus
}
#endif

/* Init function. Nothing to do here. */
EXP int AdderLib_init (struct AdderLib_params *param)
{
  return 1;
}

/* Run function. Add and scale the input data. */
EXP int AdderLib_run (struct AdderLib_in *in,
  struct AdderLib_out *out,
  struct AdderLib_params *param)
{
  out->output = param->Gain * (in->input1 + in->input2);
  return 1;
}

/* Wrapup function. Nothing to do here. */
EXP int AdderLib_wrapup (struct AdderLib_params *param)
{
  return 1;
}

/* Last error string. No errors can occur. */
EXP const char * AdderLib_lastError (void)
{
  return 0;
}
```

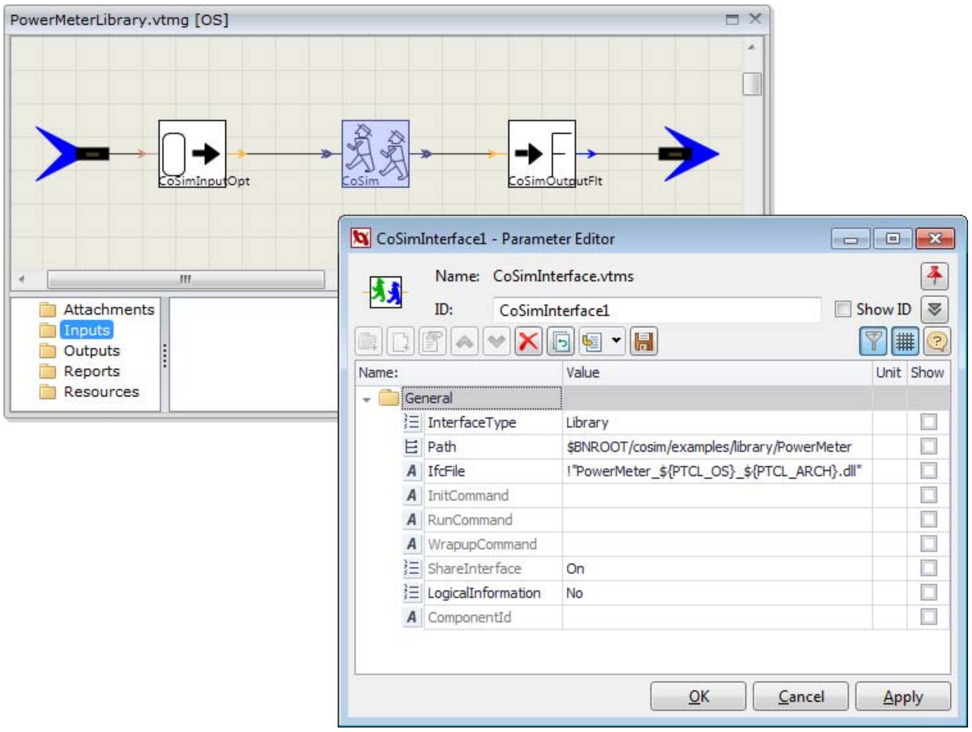
Figure 2-49 *Main source code file AdderLib.c for the adder example*

The implementation of the adder in the `AdderLib_run()` function is straightforward. The main points to note in this simple example are the following.

- Under Windows, it is necessary to implement the function `DLLMain()`. This function is called by Windows when various events affecting the DLL occur (for example, just after the DLL is loaded, and just before it is unloaded). In most cases, cosimulation modules will not need to do anything in this function, and can simply return `1` (or the Windows defined macro `TRUE`). The Windows standard header file `<windows.h>` must be included to declare this function, and the various types it depends upon.
- The two variables `ifc_name` and `ifc_CRCnew` must be exported (thus they are declared using `EXP`), and must have C naming semantics (thus they have been enclosed in an `extern "C" { ... }` block when used with a C++ compiler).
- If you do not wish to return an error string from the `lastError` function (whether or not errors can occur), you may simply return a null pointer.

Example 1 — Calculating the Total Optical Power of an Input Signal Using C/C++

The equivalent in C/C++ of the MATLAB power meter example presented in "Example 1 — Calculating the Total Optical Power of an Input Signal using MATLAB" on page 100 is shown in Figure 2-23. This setup is identical to the MATLAB setup shown in Figure 2-7, except for the settings of the parameters of the *CoSimInterface* module. In this case, the setting of the **InterfaceType** parameter is `Library`.



Name: CoSimInterface.vtms

ID: CoSimInterface1

Name:	Value	Unit	Show
General			
InterfaceType	Library		☐
Path	$BNROOT/cosim/examples/library/PowerMeter		☐
IfcFile	!"PowerMeter_${PTCL_OS}_${PTCL_ARCH}.dll"		☐
InitCommand			☐
RunCommand			☐
WrapupCommand			☐
ShareInterface	On		☐
LogicalInformation	No		☐
ComponentId			☐

Figure 2-50 *Example of the cosimulation interface galaxy for implementing a power meter in a dynamic-link library. The parameter settings of the CoSimInterface are shown.*

The complete definition of the power meter run function in C is shown in Figure 2-51. It is very similar in structure and content to the MATLAB and Python examples shown in Figure 2-8 and Figure 2-23. The main difference is that the inputs, outputs and parameters are accessed via the structures that are passed into the function, whereas in Python and MATLAB these are available as variables automatically created within the target environment. The structures created by the GenerateHeader cosimulation target for the power meter example are:

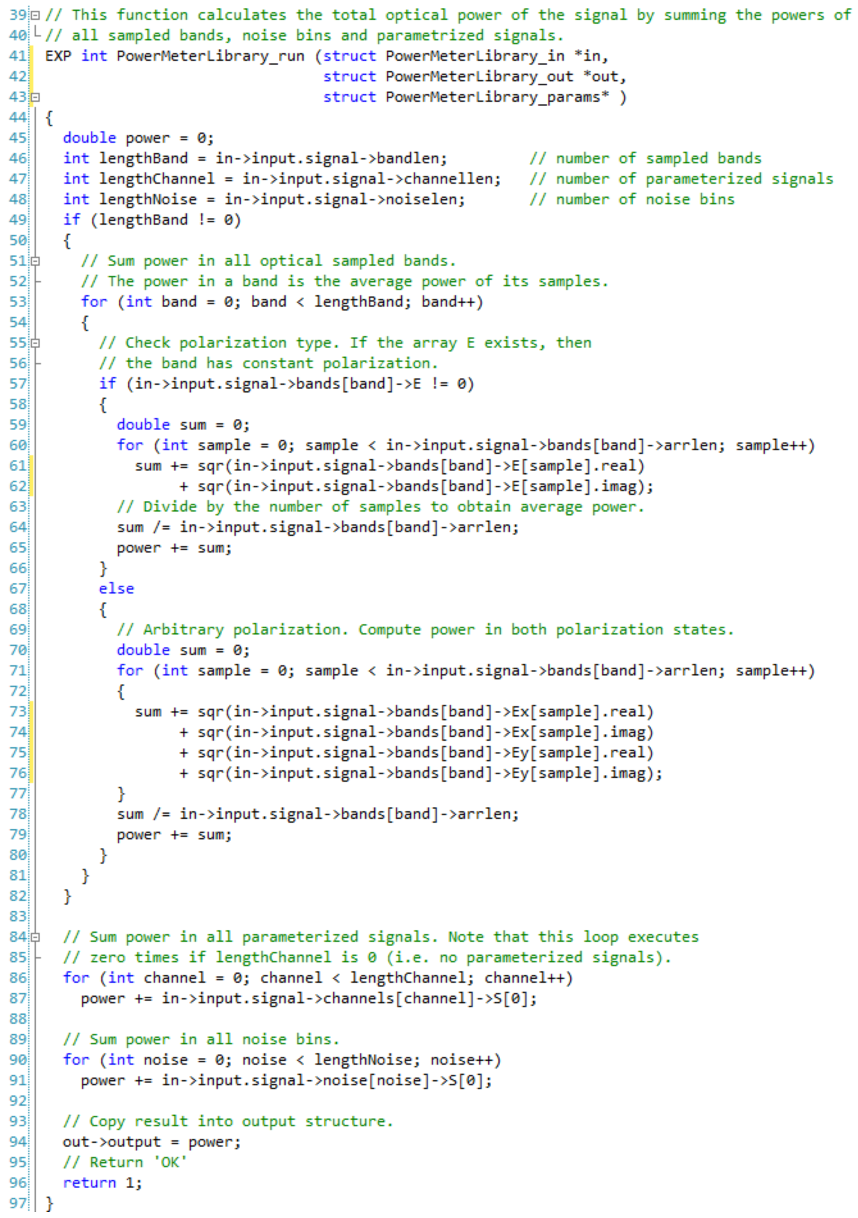
```
struct PowerMeterLibrary_params {
};

struct PowerMeterLibrary_in {
      union ONSL_Optical input;
};

struct PowerMeterLibrary_out {
        double output;
};
```

There are no parameters in this example. The input is an optical signal, called `input`, and the output is a floating-point value, called `output`.

Arrays that are not used are identified by null pointers — this indicates that no memory has been allocated. Thus the cosimulation code can determine whether a sampled band represents a signal with constant polarization or arbitrary polarization by testing the pointer to the `E`-array:

```
if (in->input.signal->bands[band]->E != 0) {
        // Constant polarization Ex, Ey == 0
        ...
}
else {
        // Arbitrary polarization, Ex != 0, Ey != 0
        ...
}
```

The complete source code for the power meter DLL can be found in the subfolder `simeng\cosim\examples\library\PowerMeter` of the VPI Design Suite installation directory.



```
// This function calculates the total optical power of the signal by summing the powers of
// all sampled bands, noise bins and parametrized signals.
EXP int PowerMeterLibrary_run (struct PowerMeterLibrary_in *in,
                               struct PowerMeterLibrary_out *out,
                               struct PowerMeterLibrary_params* )
{
  double power = 0;
  int lengthBand = in->input.signal->bandlen;         // number of sampled bands
  int lengthChannel = in->input.signal->channellen;   // number of parameterized signals
  int lengthNoise = in->input.signal->noiselen;       // number of noise bins
  if (lengthBand != 0)
  {
    // Sum power in all optical sampled bands.
    // The power in a band is the average power of its samples.
    for (int band = 0; band < lengthBand; band++)
    {
      // Check polarization type. If the array E exists, then
      // the band has constant polarization.
      if (in->input.signal->bands[band]->E != 0)
      {
        double sum = 0;
        for (int sample = 0; sample < in->input.signal->bands[band]->arrlen; sample++)
          sum += sqr(in->input.signal->bands[band]->E[sample].real)
               + sqr(in->input.signal->bands[band]->E[sample].imag);
        // Divide by the number of samples to obtain average power.
        sum /= in->input.signal->bands[band]->arrlen;
        power += sum;
      }
      else
      {
        // Arbitrary polarization. Compute power in both polarization states.
        double sum = 0;
        for (int sample = 0; sample < in->input.signal->bands[band]->arrlen; sample++)
        {
          sum += sqr(in->input.signal->bands[band]->Ex[sample].real)
               + sqr(in->input.signal->bands[band]->Ex[sample].imag)
               + sqr(in->input.signal->bands[band]->Ey[sample].real)
               + sqr(in->input.signal->bands[band]->Ey[sample].imag);
        }
        sum /= in->input.signal->bands[band]->arrlen;
        power += sum;
      }
    }
  }

  // Sum power in all parameterized signals. Note that this loop executes
  // zero times if lengthChannel is 0 (i.e. no parameterized signals).
  for (int channel = 0; channel < lengthChannel; channel++)
    power += in->input.signal->channels[channel]->S[0];

  // Sum power in all noise bins.
  for (int noise = 0; noise < lengthNoise; noise++)
    power += in->input.signal->noise[noise]->S[0];

  // Copy result into output structure.
  out->output = power;
  // Return 'OK'
  return 1;
}
```

Figure 2-51 *The PowerMeter "run" function in the C source PowerMeter.cpp. It calculates the total power of all used optical bands, channels and noise.*

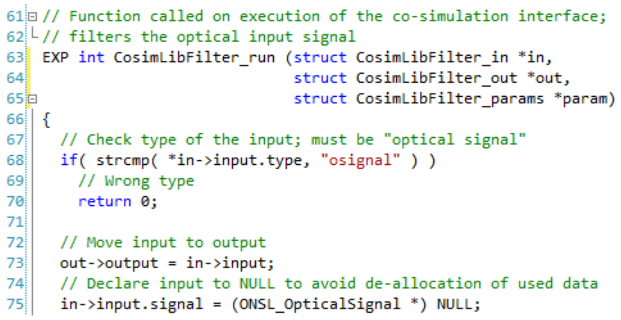
Example 2 — Implementing an Optical Bandpass Filter Using C/C++

The demonstration *Filter (Library)* in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Library* is equivalent to the MATLAB example described in "Example 2 — Implementing an Optical Bandpass Filter Using MATLAB" on page 103 and the Python example described in "Example 2 — Implementing an Optical Bandpass Filter Using Python" on page 133. This demonstration implements a Gaussian bandpass filter for optical signals using C code. The setup of the C optical filter demonstration, and the corresponding cosimulation galaxy, are identical to those of the MATLAB and Python examples, and they are not repeated here.

The filter example introduces a number of new features and concepts, which will be described in this section. You will find the complete source code for the DLL in the subdirectory `simeng\cosim\examples\library\GaussBPFilter` of the VPI Design Suite installation directory.

Like the MATLAB and Python filter examples, the DLL filter uses the fast Fourier transform (FFT) to filter the input sampled bands in the frequency domain. The VPI Design Suite installation includes the MIT FFTW library, which can be used to perform the required transforms. To access the functions in this library, the `fftw3.h` header file must be included.

The start of the "run" function is shown in Figure 2-52. First the function checks that the received signal is of the expected type. The lines (72-75) following this "move" the input signal to the output. They illustrate a very common and useful technique in C/C++ cosimulation modules. To understand how and why they work, it is necessary to understand how the library cosimulation interface treats structures passed in and out of the DLL.



```
61 // Function called on execution of the co-simulation interface;
62 // filters the optical input signal
63 EXP int CosimLibFilter_run (struct CosimLibFilter_in *in,
64                              struct CosimLibFilter_out *out,
65                              struct CosimLibFilter_params *param)
66 {
67   // Check type of the input; must be "optical signal"
68   if( strcmp( *in->input.type, "osignal" ) )
69     // Wrong type
70     return 0;
71
72   // Move input to output
73   out->output = in->input;
74   // Declare input to NULL to avoid de-allocation of used data
75   in->input.signal = (ONSL_OpticalSignal *) NULL;
```

Figure 2-52 *Checking the signal type, and using the input signal as the output*

Before calling the "run" function, the cosimulation interface creates the input structure by allocating memory for all of the inputs, such as the optical signal in this example, and copying the corresponding data into this allocated memory. It also creates an empty output structure, in which all values are initialized to zero (including null pointers to structures such as optical signals).

After the "run" function returns, the cosimulation interface copies all of the data held in the output structure into corresponding objects within the simulator, so that simulation can continue with the outputs being passed onto the following module.

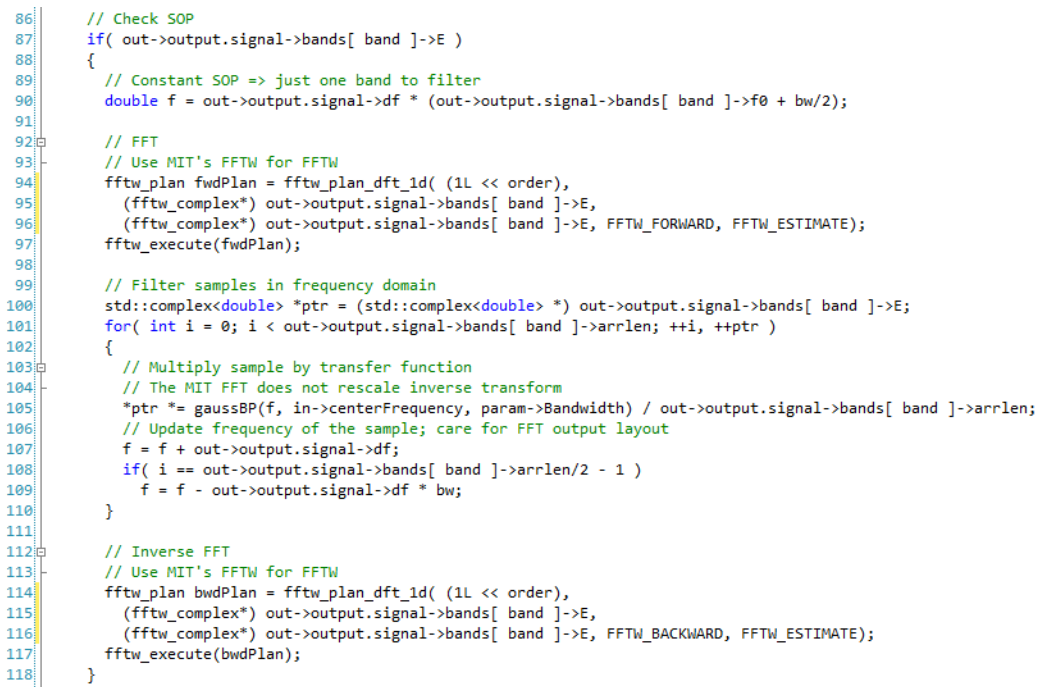
Once this has been done, the cosimulation interface releases all of the memory occupied by the input and output structures. In doing this, it works its way through each structure freeing memory whenever it finds a non-null pointer. In particular, it does *not* assume that the DLL has not modified the input structure. This has the following important consequences:

- You must not "carelessly" copy pointers to arrays or other data structures so that the inputs and outputs of your cosimulation module contain multiple pointers to the same structure in memory. If you do this, the cosimulation interface will inevitably attempt to free the same memory multiple times, with (usually) disastrous consequences! If you need to reference the same data multiple times in your outputs, you must make multiple copies of the data itself.
- As long as you ensure that there is only one pointer referencing any given structure, you may freely move this pointer around, and hence avoid unnecessary allocation of memory or copying of data.
- You must not assign null to any pointer unless you assigned its value to another pointer in the input or output structures. Otherwise memory will not be freed.

It is these principles that are applied in lines 72 to 75 of the code in Figure 2-52. By copying the pointer `in->input` to the pointer `out->output`, the code effectively moves the input signal to the output structure. And by then setting the original pointer to null, it ensures that the cosimulation interface will not subsequently attempt to release the memory occupied by the signal twice.

Note: As the example above illustrates, "assignment by reference" is simple in C/C++ cosimulation code, although there is a constraint that upon return there must only be one reference to any object. "Assignment by value", or "copying" signals, is much less convenient, since ONSL signals are represented by nested structures, that must be copied recursively. See the section "The VPI Cosimulation Library" on page 188, and the example "Three Port Fabry-Perot Filter" on page 210 for information on a more convenient approach to copying ONSL signals.

Figure 2-53 shows the C code that is used to filter a single sampled band that has a constant state of polarization (that is, where the `E` pointer is valid, and `Ex` and `Ey` are null). Aside from the different operations required to use FFTW library, this code is very similar in structure to the MATLAB and Python examples discussed previously. The main point to note from this sample is that the FFTW library defines their own complex data type (`FFTW_COMPLEX`), this structure is compatible with the `ONSL_Complex` type. Thus it is safe to cast pointers to one complex data type to pointers of a different complex data type, that is done on lines 95, 96, 115 and 116 of the sample code. Furthermore, all of these types are compatible with the C++ standard `complex<double>` type, so that the cast on line 100 is also valid.



```
// Check SOP
if( out->output.signal->bands[ band ]->E )
{
  // Constant SOP => just one band to filter
  double f = out->output.signal->df * (out->output.signal->bands[ band ]->f0 + bw/2);

  // FFT
  // Use MIT's FFTW for FFTW
  fftw_plan fwdPlan = fftw_plan_dft_1d( (1L << order),
    (fftw_complex*) out->output.signal->bands[ band ]->E,
    (fftw_complex*) out->output.signal->bands[ band ]->E, FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(fwdPlan);

  // Filter samples in frequency domain
  std::complex<double> *ptr = (std::complex<double> *) out->output.signal->bands[ band ]->E;
  for( int i = 0; i < out->output.signal->bands[ band ]->arrlen; ++i, ++ptr )
  {
    // Multiply sample by transfer function
    // The MIT FFT does not rescale inverse transform
    *ptr *= gaussBP(f, in->centerFrequency, param->Bandwidth) / out->output.signal->bands[ band ]->arrlen;
    // Update frequency of the sample; care for FFT output layout
    f = f + out->output.signal->df;
    if( i == out->output.signal->bands[ band ]->arrlen/2 - 1 )
      f = f - out->output.signal->df * bw;
  }

  // Inverse FFT
  // Use MIT's FFTW for FFTW
  fftw_plan bwdPlan = fftw_plan_dft_1d( (1L << order),
    (fftw_complex*) out->output.signal->bands[ band ]->E,
    (fftw_complex*) out->output.signal->bands[ band ]->E, FFTW_BACKWARD, FFTW_ESTIMATE);
  fftw_execute(bwdPlan);
}
```

Figure 2-53 *C code that filters a single sampled band with constant state of polarization*

The remainder of the code in the filter example is quite straightforward, and does not introduce any new features or techniques beyond those discussed above, and previously in the power meter example.

Example 3 — Creating New ONSL Signals Using C/C++

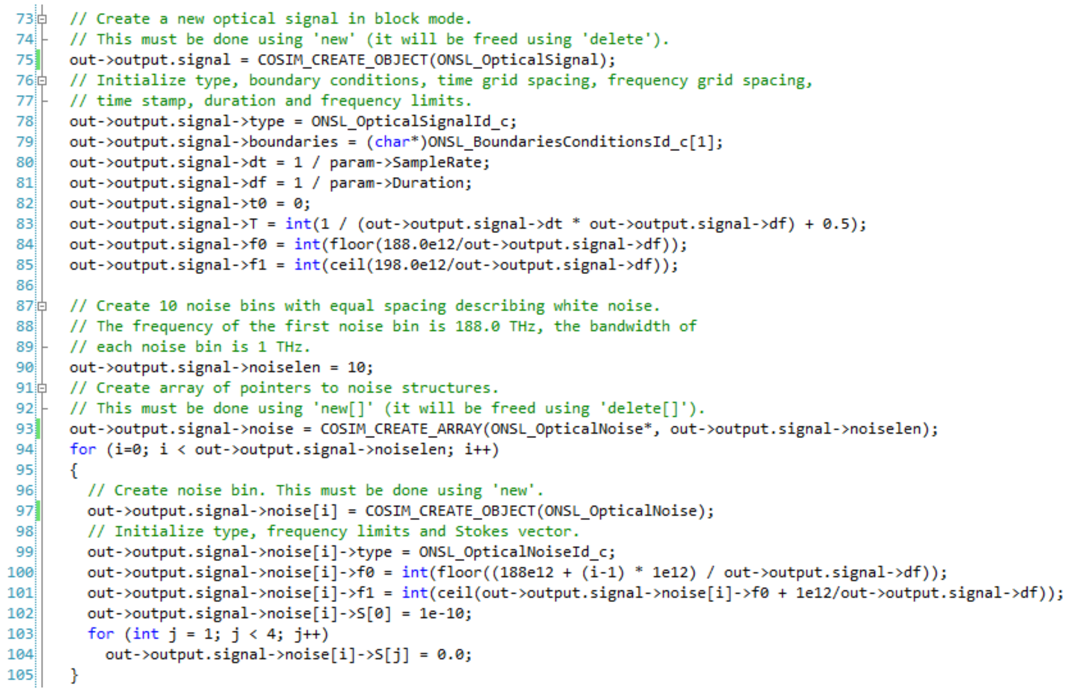
Creating or deleting signals or signal components in library cosimulation modules is possible, so long as you understand the way in which memory is managed by the cosimulation interface, and follow some simple rules. Some of the main issues have already been mentioned in the course of the previous example. The interface allocates and initializes all required memory before calling your "run" function, and releases all memory resources that are still in use once it has retrieved the returned data. You may wish to deallocate memory yourself that was allocated by the interface (for example, you might want to delete a sampled band in a specific type of filter module). Or you may wish to allocate memory for new signals that will subsequently be deallocated by the cosimulation interface. The rules you must follow in order to perform these tasks successfully are as follows:

- You must use memory management functions provided by the Simulation Environment Interface for memory allocation and deallocation. See "Simulation Environment Interface" on page 163 for details.
- If you deallocate memory corresponding to an object or array that was passed into the cosimulation module, then you must set the corresponding pointer to zero. Upon

return from your “run” function the cosimulation interface will not attempt to follow null pointers, or to delete memory when a null pointer is encountered.

- If you are going to set some pointer to zero without coping it to another pointer, you must first deallocate memory referenced by it.
- Upon return from your “run” function, the input and output structures must not contain multiple references to any single allocated object or array. Otherwise, the cosimulation interface will attempt to deallocate the corresponding memory multiple times.

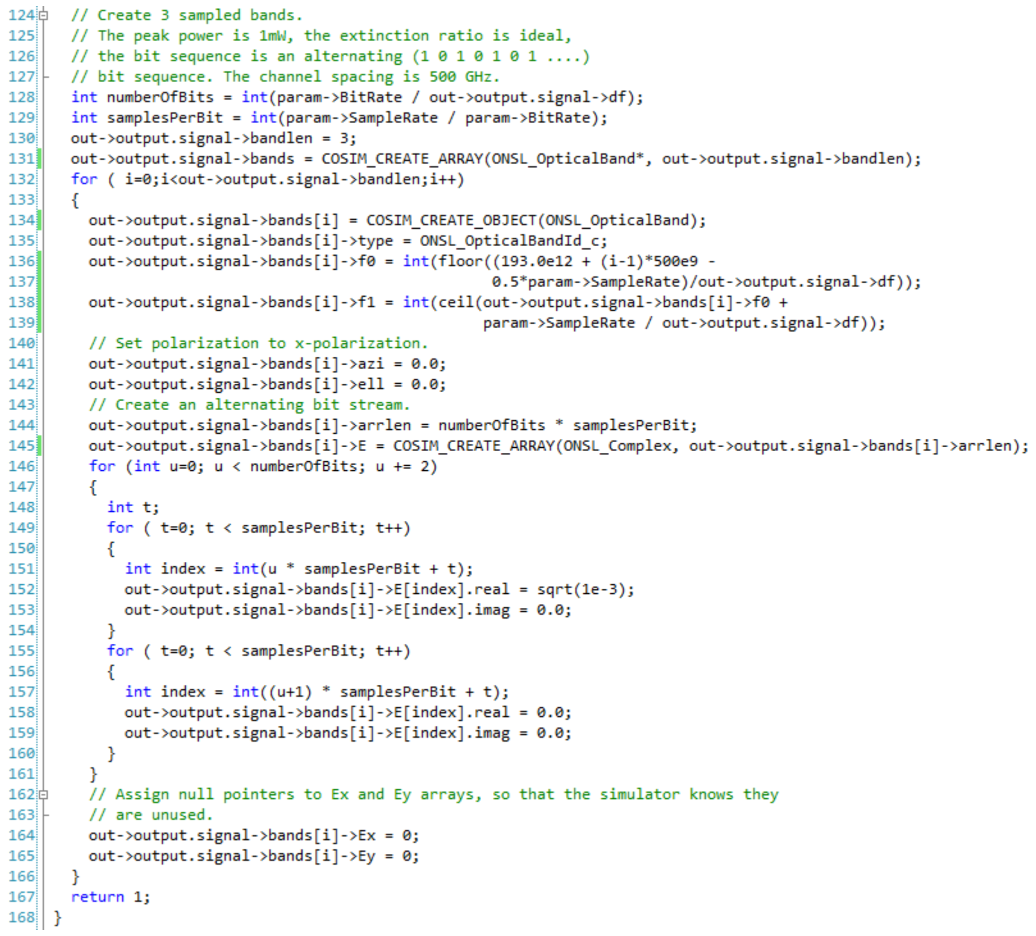
Given these requirements, the code used to create an optical signal and 10 noise bins is shown in Figure 2-54. This code is taken from the demonstration *Optical Signal Generation (Library)* in the demonstrations folder *Optical Systems Demos > Simulation Techniques > Cosimulation > Library*. The complete source code can be found in the subfolder `simeng\cosim\examples\library\CreateOpticalSignal` of the VPI Design Suite installation directory. Most of this code is very similar in structure to the corresponding MATLAB and Python examples discussed previously. The key new features are the creation of the optical signal structure on line 75, the noise bin array on line 93, and the noise bin structures on line 97.



```
 73  // Create a new optical signal in block mode.
 74  // This must be done using 'new' (it will be freed using 'delete').
 75  out->output.signal = COSIM_CREATE_OBJECT(ONSL_OpticalSignal);
 76  // Initialize type, boundary conditions, time grid spacing, frequency grid spacing,
 77  // time stamp, duration and frequency limits.
 78  out->output.signal->type = ONSL_OpticalSignalId_c;
 79  out->output.signal->boundaries = (char*)ONSL_BoundariesConditionsId_c[1];
 80  out->output.signal->dt = 1 / param->SampleRate;
 81  out->output.signal->df = 1 / param->Duration;
 82  out->output.signal->t0 = 0;
 83  out->output.signal->T = int(1 / (out->output.signal->dt * out->output.signal->df) + 0.5);
 84  out->output.signal->f0 = int(floor(188.0e12/out->output.signal->df));
 85  out->output.signal->f1 = int(ceil(198.0e12/out->output.signal->df));
 86
 87  // Create 10 noise bins with equal spacing describing white noise.
 88  // The frequency of the first noise bin is 188.0 THz, the bandwidth of
 89  // each noise bin is 1 THz.
 90  out->output.signal->noiselen = 10;
 91  // Create array of pointers to noise structures.
 92  // This must be done using 'new[]' (it will be freed using 'delete[]').
 93  out->output.signal->noise = COSIM_CREATE_ARRAY(ONSL_OpticalNoise*, out->output.signal->noiselen);
 94  for (i=0; i < out->output.signal->noiselen; i++)
 95  {
 96    // Create noise bin. This must be done using 'new'.
 97    out->output.signal->noise[i] = COSIM_CREATE_OBJECT(ONSL_OpticalNoise);
 98    // Initialize type, frequency limits and Stokes vector.
 99    out->output.signal->noise[i]->type = ONSL_OpticalNoiseId_c;
100    out->output.signal->noise[i]->f0 = int(floor((188e12 + (i-1) * 1e12) / out->output.signal->df));
101    out->output.signal->noise[i]->f1 = int(ceil(out->output.signal->noise[i]->f0 + 1e12/out->output.signal->df));
102    out->output.signal->noise[i]->S[0] = 1e-10;
103    for (int j = 1; j < 4; j++)
104      out->output.signal->noise[i]->S[j] = 0.0;
105  }
```

Figure 2-54 *Creation of an optical signal and 10 noise bins in C++ cosimulation code*

Figure 2-55 shows the code that is used to generate three sampled bands. Again, you can see how the optical band array is created on line 131, the band structures themselves are created on line 134, and the sample array is created on line 145. Finally, the pointers to the unused sample arrays for arbitrarily polarized signals are set to zero on lines 164 and 165. This is essential to ensure that the cosimulation interface does not attempt to delete memory associated with these arrays.



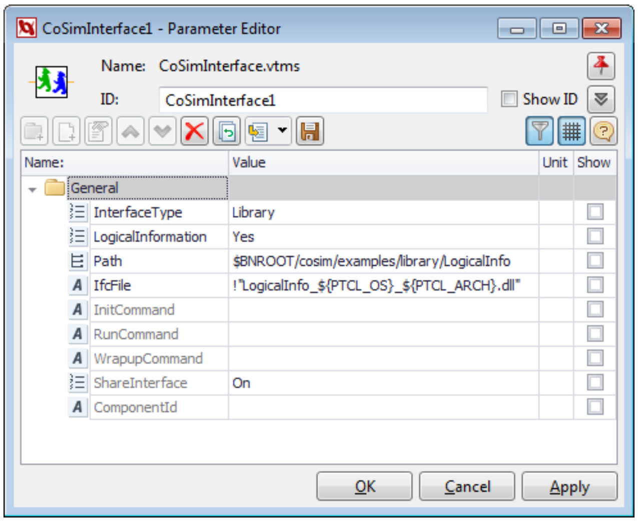
```
124   // Create 3 sampled bands.
125   // The peak power is 1mW, the extinction ratio is ideal,
126   // the bit sequence is an alternating (1 0 1 0 1 0 1 ....)
127   // bit sequence. The channel spacing is 500 GHz.
128   int numberOfBits = int(param->BitRate / out->output.signal->df);
129   int samplesPerBit = int(param->SampleRate / param->BitRate);
130   out->output.signal->bandlen = 3;
131   out->output.signal->bands = COSIM_CREATE_ARRAY(ONSL_OpticalBand*, out->output.signal->bandlen);
132   for ( i=0;i<out->output.signal->bandlen;i++)
133   {
134     out->output.signal->bands[i] = COSIM_CREATE_OBJECT(ONSL_OpticalBand);
135     out->output.signal->bands[i]->type = ONSL_OpticalBandId_c;
136     out->output.signal->bands[i]->f0 = int(floor((193.0e12 + (i-1)*500e9 -
137                                             0.5*param->SampleRate)/out->output.signal->df));
138     out->output.signal->bands[i]->f1 = int(ceil(out->output.signal->bands[i]->f0 +
139                                            param->SampleRate / out->output.signal->df));
140     // Set polarization to x-polarization.
141     out->output.signal->bands[i]->azi = 0.0;
142     out->output.signal->bands[i]->ell = 0.0;
143     // Create an alternating bit stream.
144     out->output.signal->bands[i]->arrlen = numberOfBits * samplesPerBit;
145     out->output.signal->bands[i]->E = COSIM_CREATE_ARRAY(ONSL_Complex, out->output.signal->bands[i]->arrlen);
146     for (int u=0; u < numberOfBits; u += 2)
147     {
148       int t;
149       for ( t=0; t < samplesPerBit; t++)
150       {
151         int index = int(u * samplesPerBit + t);
152         out->output.signal->bands[i]->E[index].real = sqrt(1e-3);
153         out->output.signal->bands[i]->E[index].imag = 0.0;
154       }
155       for ( t=0; t < samplesPerBit; t++)
156       {
157         int index = int((u+1) * samplesPerBit + t);
158         out->output.signal->bands[i]->E[index].real = 0.0;
159         out->output.signal->bands[i]->E[index].imag = 0.0;
160       }
161     }
162     // Assign null pointers to Ex and Ey arrays, so that the simulator knows they
163     // are unused.
164     out->output.signal->bands[i]->Ex = 0;
165     out->output.signal->bands[i]->Ey = 0;
166   }
167   return 1;
168 }
```

Figure 2-55 *Creation of three sampled bands in C++ cosimulation code*

For a new signal source, the logical information shown in Table 2-9 can also be added if it is required. Logical channels should at least contain a label string, sequence number (which is usually equal to the current run number), `ONSL_SignalSource` structure (`f0`, `f1`, `t0`, `T` should be equal to an appropriate data of the physical signal) and `ONSL_Label` (label string and sequence number of the logical channel should also be added to the label set of the new physical signal).

Example 4 — Handling the Logical Information Using C/C++

The equivalent in C/C++ of the MATLAB logical information cosimulation example presented in "Example 4 — Handling the Logical Information using MATLAB" on page 110 is shown in Figure 2-56. This setup is similar to the MATLAB setup shown in Figure 2-18, except for the settings of the parameters of the *CoSimInterface* module. In this case, the setting of the **InterfaceType** parameter is `Library`. Note that the DLL itself has been attached to the schematic. To use the attached file, the **Path** parameter has been set to {Path} - parameter of schematic, and **IfcFile** has been set to the expression `!"LogicalInfo_${PTCL_OS}_${PTCL_ARCH}.${LIB_EXT}"` which is evaluated to the appropriate file name depending on the target operating system.


CoSimInterface1 - Parameter Editor

Name: CoSimInterface.vtms

ID: CoSimInterface1

Show ID

Name:	Value	Unit	Show
General			
InterfaceType	Library		☐
LogicalInformation	Yes		☐
Path	$BNROOT/cosim/examples/library/LogicalInfo		☐
IfcFile	!"LogicalInfo_${PTCL_OS}_${PTCL_ARCH}.dll"		☐
InitCommand			☐
RunCommand			☐
WrapupCommand			☐
ShareInterface	On		☐
ComponentId			☐

Figure 2-56 *Parameter setting of the cosimulation interface galaxy for implementing a logical info in a dynamic-link library.*

The complete definition of the `run` function in C can be found in the `LI_example.cpp` file in the subfolder `simeng\cosim\examples\library\LogicalInfo` of the VPI Design Suite installation directory. The header file is also stored there.

The `run` function is very similar in structure and content to the MATLAB and Python examples shown in Figure 2-19 and Figure 2-30. The main difference is that the inputs, outputs and parameters are accessed via the structures that are passed into the function, whereas in Python and MATLAB, these are available as variables automatically created within the target environment. The structures created by the `GenerateHeader` cosimulation target for the power meter example are:

```
struct CosimLib_params {
        char * Path;
};

struct CosimLib_in {
        union ONSL_OpticalLI input;
};
```

```
struct CosimLib_out {
      ONSL_FloatMatrix output;
};
```

There are no parameters in this example. The input is an optical signal, called `input`, and the output is a floating-point matrix, called `output`. The run function implemented transfers to output the bit stream stored in logical channels. All other logical information available in this setup is written to the `LI_library.dat` stored in the temporary directory.

Developing Cosimulation Modules using Microsoft Visual C++

The information and examples in the previous section were intended to be as generic as possible, so that they can be applied to multiple platforms, and to different compilers. In practice, the demands of maintaining binary compatibility, and differences in the availability of libraries on different platforms (such as the MIT FFTW library, as well as "standard" system libraries) make writing generally portable code very difficult. This section therefore focuses on the mechanics of developing C++ cosimulation modules on the most widespread platform (Windows), using the most popular Windows development environment, Microsoft Visual Studio. Some additional support provided by VPI Design Suite for development of C++ cosimulation modules in this environment is also introduced.

This section assumes that you are familiar with the main features of the Microsoft Visual Studio development environment.

Project Settings

When you start work on developing a new cosimulation application using Microsoft Visual Studio, you will typically begin by creating a new Project and Solution. The type of project you should create is a **Visual C++\Win32\Win32 Project** with Application type - **DLL**.

The recommended project settings (accessed via the **Properties** item on the **Project** menu) are as follows:

- The **Struct member alignment**, set under the **Code Generation** options of the C/C++ settings tab, must be set to Default or 8 bytes (which are identical).
- In the **General** options of the C/C++ settings tab, you will need to add the VPI Design Suite include directory to the list of Additional include directories (typically `C:\Program Files\VPI\VPIdesignSuite 11.5\simeng\cosim\include`). You may also need to add other include paths here, depending upon your application.
- In the **General** options of the **Linker** settings tab, you will need to add VPI Design Suite library directory (typically `C:\Program Files\VPI\VPIdesignSuite 11.5\simeng\cosim\lib\win`). You may also need to add other library paths here, depending upon your application.
- If you are using the MIT FFTW library (as does, for example, the optical bandpass filter example discussed earlier), you will need to add `fftw3.lib` to the Additional Dependencies list under the **Input** options of the **Linker** tab.
- The target platform should be the same as selected for the simulation engine in the VPI Design Suite **Preferences** dialog. Refer to the Microsoft Visual Studio help for information on changing the target platform of a dynamic-link library.

The default settings for all other options are typically acceptable.

Creating Cosimulation Modules as Self-Contained C++ Classes

The standard VPI Design Suite library cosimulation interface described in the previous sections is a C-language interface. It uses C calling conventions, and provides only one entry point for each of the supported user functions. This provides a simple interface to the cosimulation DLL, and ensures maximum support for different system configurations, since any standard C compiler should be able to produce code that is compatible with the interface.

However, the C-language interface presents some complications when multiple instances of a particular cosimulation module are required in a single simulation. For example, if each instance must maintain some private state information, it is necessary to manage the storage of this data. It is not possible, for example, to simply use global or static variables, because the state data is different for each instance, but only one copy of the DLL can be loaded.

It would clearly be much more convenient, when using a C++ compiler, to implement the cosimulation module as a C++ class. The most obvious general interface for such a class would consist of:

- A public constructor, taking no arguments, that is executed when each instance of the cosimulation module in the simulation schematic is created.
- A public method `initialize`, taking a single argument (a pointer to the parameter structure), that is executed when the simulation is initialized.
- A public method `run`, taking three arguments (pointers to the parameter, input and output structures), that is executed each time the module is fired during the simulation run.
- A public method `wrapup`, taking a single argument (a pointer to the parameter structure), that is executed when the simulation terminates.
- A public destructor, that is executed after the simulation has terminated and all resources are being released.

The class could then additionally contain whatever private data and methods it requires to perform its functions. All required information would be contained within the class definition, and it would not be necessary for the developer to be concerned with managing this data on a per-instance basis.

A template for such an object-oriented model of cosimulation is provided in VPI Design Suite. It consists of three files, that can be found in the installation subfolder `simeng\cosim\examples\library\Generic`. These three files are:

- `DLLMain.cpp`: This file is a generic C-cosimulation interface file, containing the standard “initialize”, “run”, “wrapup” and “lastError” functions. It manages all instances of the cosimulation module, creating and destroying the cosimulation module objects as required, and dispatching calls to their `initialize`, `run` and `wrapup` methods. The “initialize” function also performs the CRC check, and returns an error if the cosimulation interface has changed.

- `CosimModule.hpp`: This file is a generic header file for a cosimulation module class. It declares the class containing the `initialize`, `run` and `wrapup` methods.
- `CosimModule.cpp`: This file is a generic main source file for a cosimulation module class. It contains empty definitions of the `initialize`, `run` and `wrapup` methods.

To use these files as the basis for your own cosimulation modules, simply copy the generic templates to your development directory, and rename `CosimModule.hpp` and `CosimModule.cpp` to something more appropriate to your application. Renaming the files is not strictly necessary, but most people will find it more convenient to use filenames that reflect the contents.

The templates themselves contain the string `##CosimModuleName##` wherever the actual instance ID of the cosimulation module is required (it is derived from the instance ID of the galaxy, as described in "Dynamic-Link Library Interface" on page 150). An automatic procedure to customize these files is not provided. Open each file in a text editor such as the Microsoft Visual Studio source code editor, and replace all occurrences of the string `##CosimModuleName##` with the actual instance ID of your cosimulation module.

You may have noticed that in the above description of the C++ module interface, no mention was made of any error reporting method in the class interface. This is because errors are not reported using a function-style interface. Instead, the preferred C++ technique of throwing an exception to indicate an error condition is used. The VPI Cosimulation Library (described in detail in the next section) provides the exception class `cosim::ErrorMessage`. This class provides constructors accepting both C-style and standard C++ strings as parameters, which are used to set the message string. `ErrorMessage` objects thrown by methods of the cosimulation module class are caught by the interface code in `DLLMain.cpp`, and used to provide an error string to the simulator via the `lastError()` mechanism. Exceptions thrown by the C++ standard template library (STL) are also caught and converted into messages to be displayed by the simulator.

In summary, use of the DLL interface and class templates, in combination with the VPI Cosimulation Library described in the next section, greatly simplifies the development of cosimulation modules, enables the reuse of one standard implementation of the C-language interface, and facilitates the implementation of cosimulation modules using object-oriented techniques in C++.

The VPI Cosimulation Library

A collection of support functions and classes for C++ cosimulation modules is provided, referred to as the VPI Cosimulation Library. The initial version includes a small collection of commonly needed functions, and it is expected that the size of the library will grow in future releases. VPIphotonics welcomes feedback from users on this new feature, as well as suggestions, or even contributions, for future additions. The Cosimulation Library is currently provided in source form, and you are welcome to study the code, or adapt it to your own specific requirements. If you wish to do this, we recommend that you make a copy, and place your adapted versions in a separate namespace so that you do not create conflicts with future revisions of the library.

The source for the VPI Cosimulation Library is in the subfolder `simeng\cosim\examples\library\CosimLib` of the VPI Design Suite installation. It includes a Visual C++ project file, which allows it to be inserted as a separate project into your development solution. This is illustrated in the step-by-step example presented in the next section.

The Cosimulation Library defines a namespace called `cosim` to hold its global functions and classes. You can bring all the cosimulation functions and classes into scope with a `using namespace cosim` directive in your source files, or you can refer to each explicitly with the `cosim::` prefix.

Header Files

Declarations of the functions and classes in the VPI Cosimulation Library reside in two header files:

- `CosimLib.hpp` contains declarations for general global functions, and the `ErrorMessage` class.
- `CosimFilter.hpp` contains declarations of the generic optical and electrical filter base classes `CosimFilterOpt` and `CosimFilterElec`.

Libraries

The Cosimulation Library is called `CosimLib_win_x64.lib`. A compiled version of this library file is provided in the `simeng\cosim\examples\library\CosimLib` subfolder of the VPI Design Suite installation. You can link with this binary version directly, or you can add the CosimLib project to your solution (as described in the step-by-step example below), and allow Microsoft Visual Studio to build and link the library automatically.

The optical and electrical filter classes require the MIT FFTW library, so you must also ensure that your project is linked with `fftw3.lib`, as described in the section "Project Settings" on page 186.

Functions

The following general functions are provided by the VPI Cosimulation library, and are declared in the header file `CosimLib.hpp`.

`int cosim::strcmp_nocase(const char *s1, const char *s2)`

Purpose:

Compare two C strings in a case-insensitive manner.

Uses the same conventions as the standard `strcmp()`function.

Parameters:

s1	pointer to first zero terminated character string
s2	pointer to second zero terminated character string

Returns:

0 if strings are equal; -1 if first is "less than" second; 1 if first is "greater than" second.

Application example:

This function is useful for testing the value of string and enumeration parameters passed to cosimulation modules. The "Three Port Fabry-Perot Filter" demo (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > FP Filter (T&R)*) shows how to implement an `Active` parameter in a module.

`std::string cosim::strip_spc(const char *`*`s`*`)`

Strip all leading and trailing spaces from the C-style string passed, and return a C++ string.

Parameters:

`s` pointer to a C-style string

Returns:

A string containing the input string with all leading and trailing spaces removed.

Application example:

This function is useful for testing the value of string parameters passed to cosimulation modules, where the user may have inadvertently added spaces before, after or between keywords.

`int enumerate(const char *`*`name`*`, const char *`*`val`*`, const char **`*`vallist`*`)`

Purpose:

Search an array of C-style strings for a given string, and return the index if found, intended for converting enumerated parameters into equivalent integers.

The comparison is performed after stripping any leading and trailing spaces from the string, and is case-insensitive.

Parameters:

`name` a pointer to a C-style string giving the parameter name.

`val` a pointer to a C-style string holding the parameter value.

`vallist` a pointer to an array of C-style strings holding the allowed values, terminated by a zero (null) pointer.

Returns:

The index of the parameter value in the array.

Exceptions:

`ErrorMessage` exception if no match is found.

Application example:

This function is useful for converting enumeration parameters passed to cosimulation modules into corresponding integer values. For example:

```
const char *selection[] = {"FirstChoice", "SecondChoice", 0};
int choice = enumerate("ChoiceParam", params->ChoiceParam,
             selection);
```

`union ONSL_Optical cosim::dup(const union ONSL_Optical &`*`orig`*`)`

Purpose:

Duplicate an optical signal or sample.

Parameters:

orig the original optical signal or sample.

Returns:

An exact duplicate of the input.

Application example:

This function performs a recursive copy of the input optical signal or sample, allocating memory to contain structures and arrays as required. It is used whenever a duplicate signal is required. The "Three Port Fabry-Perot Filter" demo (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > FP Filter (T&R)*) shows how to generate a second output signal for the additional reflected output port of the filter.

`union ONSL_Electrical cosim::dup(const union ONSL_Electrical &`*`orig`*`)`

Purpose:

Duplicate an electrical signal or sample.

Parameters:

orig the original electrical signal or sample.

Returns:

An exact duplicate of the input.

Comment:

This function overloads the optical signal duplication function of the same name. Thus the function `cosim::dup()` can be used with either an optical or electrical signal as an argument.

`union ONSL_Optical cosim::zero(const union ONSL_Optical &`*`orig`*`)`

Purpose:

Create a zero optical signal or sample using the input as a model.

The resulting signal or sample is equivalent to the input signal with all components multiplied by zero.

Parameters:

orig the signal or sample to use as a model.

Returns:

A copy of the input with all components set to zero.

Application example:

The returned signal has exactly the same structure as the input. For example, in the case of an optical block signal, it contains the same number of parameterized signals, noise bins, and sampled bands. However, all of the components have zero power, as if the signals had all been completely attenuated. Memory is allocated to contain the structures and arrays as required. The "Three Port Fabry-Perot Filter" demo (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > FP Filter (T&R)*) shows how to generate the reflected output signal when the filter is inactive.

`union ONSL_Electrical cosim::zero(const union ONSL_Electrical &`*`orig`*`)`

Purpose:

Create a zero electrical signal or sample using the input as a model.

The resulting signal or sample is equivalent to the input signal with all components multiplied by zero.

Parameters:

orig the signal or sample to use as a model

Returns:

A copy of the input with all components set to zero

Comment:

This function overloads the optical signal duplication function of the same name. Thus the function `cosim::zero()` can be used with either an optical or electrical signal as an argument.

`void cosim::resample(ONSL_OpticalBand *`*`band,`*` int `*`newLength`*`)`

Purpose:

Resample an optical sampled band.

Up or down sampling will be performed. The resampling factor must be a power of two.

When upsampling, the function performs ideal, periodic, interpolation. When downsampling, the function performs ideal, periodic, "brick-wall" anti-alias filtering.

Parameters:

band pointer to the band to be resampled.

newLength the number of samples in the resampled band.

Exceptions:

`ErrorMessage` if the resampling factor is not a power of two.

Application example:

The *Simple Optical Signal Resampling* (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > Optical Signal Resampling*) demo uses this function to resample all input bands to the same array length. Note that making the array lengths the same is equivalent to making the sampling rate the same, since the time window is a global property of the signals in a simulation. Thus a more practical application of this function could be to make two or more signals have the same sampling rate so that some common processing can be performed on them. For example, you could generate an output representing the total instantaneous power in all sampled bands as a function of time. In order to generate the correct values for all sample times, you would first need to upsample all lower rate bands to the same sample rate as the highest rate band.

`void cosim::resample(ONSL_ElectricalBand *`*`band`*`, int `*`newLength`*`)`

Purpose:

Resample an electrical sampled band.

Up or down sampling will be performed. The resampling factor must be a power of two.

When upsampling, the function performs ideal, periodic, interpolation. When downsampling, the function effectively performs ideal, periodic, “brick-wall” anti-alias filtering.

Parameters:

band pointer to the band to be resampled.

newLength the number of samples in the resampled band.

Exceptions:

`ErrorMessage` if the resampling factor is not a power of two.

Application example:

The *Low Pass Electrical Filtering* (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > Electrical LP Filter*) demo uses this function to downsample the filtered band under the condition that the high frequency components have all been set to zero by the filter. This reduces the output sample rate, and hence the amount of processing that subsequent modules must perform.

Classes

The following class provided by the VPI Cosimulation library is declared in the header file `CosimLib.hpp`.

`ErrorMessage` class.

Class providing a standard mechanism to pass error messages back to the functions that provide the direct interface to the simulator.

The `initialize()`, `run()` and `wrapup()` functions can generate an error by throwing an `ErrorMessage` exception containing an appropriate string.

Constructors.

`cosim::ErrorMessage::ErrorMessage() [inline]`

Purpose:

Default constructor.

Sets the error string "Unknown error."

`cosim::ErrorMessage::ErrorMessage(const char *s) [inline]`

Purpose:

Constructor taking a C-style string.

Parameters:

s pointer to a zero-terminated character string.

`cosim::ErrorMessage::ErrorMessage(const std::string &s) [inline]`

Purpose:

Constructor taking a C++ string.

Parameters:

s reference to a string.

Member Functions.

`std::string cosim::ErrorMessage::asString() const [inline]`

Purpose:

Get the error message as a C++ string.

Returns:

A string containing the error message.

`const char* cosim::ErrorMessage::c_str() const [inline]`

Purpose:

Get the error message as a C-style string.

Returns:

A pointer to a zero terminated character string that is valid for as long as the ErrorMessage object exists.

The following classes provided by the VPI Cosimulation library are declared in the header file `CosimFilter.hpp`.

`CosimFilterOpt` class.

Base class for optical filters.

Provides a standard periodic filter implementation for block mode signals. The frequency domain transfer function is defined by deriving a specific filter class from this class, and overriding the default implementation of the private member function `transferFunction()`. A basic noise bin adaptation implementation is provided.

This class also provides a framework for sample-by-sample time domain filtering for sample mode signals. The time domain transfer function is defined by deriving a specific filter class from this class, and overriding the default implementation of the private member function `timeDomainResponse()`.

Note that in principle a single derived filter class can provide both time and frequency domain implementations, and thus be able to filter either sample or block mode signals.

Application example:

The "Three Port Fabry-Perot Filter" demo (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > FP Filter (T&R)*) uses this class to implement both the transmission and reflection filter functions.

Constructors.

`cosim::CosimFilterOpt::CosimFilterOpt(double dyn = 2.0, double thr = 1.0e-10)`

Parameters:

`dyn` the maximum variation of the filter transfer function over a single noise bin for noise bin adaptation (default = 3dB).

`thr` the threshold level for noise bin adaptation — if the magnitude of the filter response is below this level, noise bins are not adapted (default = -100dB).

Public Member Functions.

`void cosim::CosimFilterOpt::filter(union ONSL_Optical &`*`osig`*`) const`

Purpose:

Apply the filter transfer function to a complete optical signal or apply the time domain filter function to a single optical sample.

Parameters:

`osig` reference to the optical signal or sample to be filtered.

Returns:

Optical signal modified by the filter transfer function.

`void cosim::CosimFilterOpt::filterChannel(ONSL_OpticalChannel *`*`channel`*`, double df) const`

Purpose:

Apply the filter transfer function to a single optical channel.

Parameters:

channel	pointer to the channel to be filtered.
df	the frequency resolution of the optical signal.

`void cosim::CosimFilterOpt::filterNoiseBin(ONSL_OpticalNoise *`*`noise`*`, double df) const`

Purpose:

Apply the filter transfer function to a single noise bin.

Parameters:

noise	pointer to the noise bin to be filtered.
df	the frequency resolution of the optical signal.

`void cosim::CosimFilterOpt::filterNoiseBins (ONSL_OpticalSignal *signal, bool adapt = true )  const`

Purpose:

Apply the filter transfer function to all noise bins within an optical signal, adapting the noise bins if required.

Parameters:

signal	pointer to the optical signal.
adapt	true if noise bins should be adapted (default), false otherwise.

`void cosim::CosimFilterOpt::filterBand(ONSL_OpticalBand *band, double df)  const`

Purpose:

Apply the filter transfer function to a single optical band.

Parameters:

band	pointer to the band to be filtered.
df	the frequency resolution of the optical signal.

Protected Member Functions.

`double cosim::CosimFilterOpt::getThreshold() const [inline]`

Purpose:

Get the current noise threshold setting.

Useful for derived classes implementing their own noise bin adapters.

Returns:

The current threshold.

`double cosim::CosimFilterOpt::getDynamic() const [inline]`

Get the current noise 'dynamic' setting.

Useful for derived classes implementing their own noise bin adapters.

Returns:

The current noise 'dynamic'.

Private Member Functions.

`complex<double> cosim::CosimFilterOpt::transferFunction(double f) const [virtual]`

Purpose:

Frequency domain transfer function, returns the complex amplitude of the filter.

The default implementation in the base class throws an `cosim::CosimFilterOpt::UnsupportedSignalType` exception. This function must be overridden in the derived class to implement frequency domain (that is, optical signal) filtering.

Parameters:

f frequency in Hz.

Returns:

The complex filter amplitude at the input frequency.

Exceptions:

`cosim::CosimFilterOpt::UnsupportedSignalType` if not implemented.

`void cosim::CosimFilterOpt::timeDomainResponse(ONSL_OpticalSample &s) const [virtual]`

Purpose:

Time domain transfer function.

Modifies the input sample according to the time response of the filter, typically dependent also on previous samples. The default implementation in the base class throws a `cosim::CosimFilterOpt::UnsupportedSignalType` exception. This function must be overridden in the derived class to implement time domain (that is, optical sample) filtering.

Parameters:

s the current optical input sample, modified by the function to produce the corresponding optical output sample.

Exceptions:

`cosim::CosimFilterOpt::UnsupportedSignalType` if not implemented.

`void cosim::CosimFilterOpt::adaptNoiseBins(ONSL_OpticalSignal *signal) const [virtual]`

Purpose:

Noise bin adapter, used when performing frequency domain filtering of noise bins.

The default implementation is reasonably general and straightforward. It uses some basic optimizations to try to keep calculation time to a reasonable level. This function may be overridden in derived classes, for example to provide an implementation optimized for a specific filter response.

Parameters:

osig reference to the optical signal containing noise bins to be adapted. Modified accordingly upon return.

`CosimFilterElec` class.

Base class for electrical filters.

Provides a standard periodic filter implementation for block mode signals. The frequency domain transfer function is defined by deriving a specific filter class from this class, and overriding the default implementation of the private member function `transferFunction()`.

This class also provides a framework for sample-by-sample time domain filtering for sample mode signals. The time domain transfer function is defined by deriving a specific filter class from this class, and overriding the default implementation of the private member function `timeDomainResponse()`.

Note that in principle a single derived filter class can provide both time and frequency domain implementations, and thus be able to filter either sample or block mode signals.

Application examples:

The *Low Pass Electrical Filtering* (*Optical Systems Demos > Simulation Techniques > Cosimulation > Library > Electrical LP Filter*) demo uses this class to implement the electrical low pass filter in the frequency domain.

Public Member Functions.

`void cosim::CosimFilterElec::filter(union ONSL_Electrical &esig) const`

Purpose:

Apply the filter transfer function to the electrical signal.

Parameters:

esig reference to the electrical signal to be filtered.

Returns:

Electrical signal modified by the filter transfer function.

Private Member Functions.

`complex<double> cosim::CosimFilterElec::transferFunction(double f ) const [virtual]`

Purpose:

Frequency domain transfer function, returns the complex amplitude of the filter.

The default implementation in the base class throws a `cosim::CosimFilterElec::UnsupportedSignalType` exception. This function must be overridden in the derived class to implement frequency domain (that is, electrical signal) filtering.

Parameters:

f frequency in Hz.

Returns:

The complex filter amplitude at the input frequency.

Exceptions:

`cosim::CosimFilterElec::UnsupportedSignalType` if not implemented.

`void cosim::CosimFilterElec::timeDomainResponse(ONSL_ElectricalSample &s) const [virtual]`

Purpose:

Time domain transfer function.

Modifies the input sample according to the time response of the filter, typically dependent also on previous samples. The default implementation in the base class throws a `cosim::CosimFilterElec::UnsupportedSignalType` exception. This function must be overridden in the derived class to implement time domain (that is, electrical sample) filtering.

Parameters:

s the current electrical input sample, modified by the function to produce the corresponding electrical output sample.

Exceptions:

`cosim::CosimFilterElec::UnsupportedSignalType` if not implemented.

Step-By-Step Example: The Gaussian Bandpass Filter as a C++ Class Module

In this section, as an example of the development process using Visual C++, and the use of the VPI Cosimulation Library, we will work through a complete implementation step-by-step. The example we have chosen is a familiar one — the simple Gaussian bandpass optical filter previously used as an example of MATLAB, Python and C cosimulation. Since this version will use the `CosimFilterOpt` class to implement the frequency domain filtering, it will additionally support noise bin adaptation. Two new parameters are required to support this feature, the standard filter parameters **`NoiseDynamic`** and **`NoiseThreshold`**.

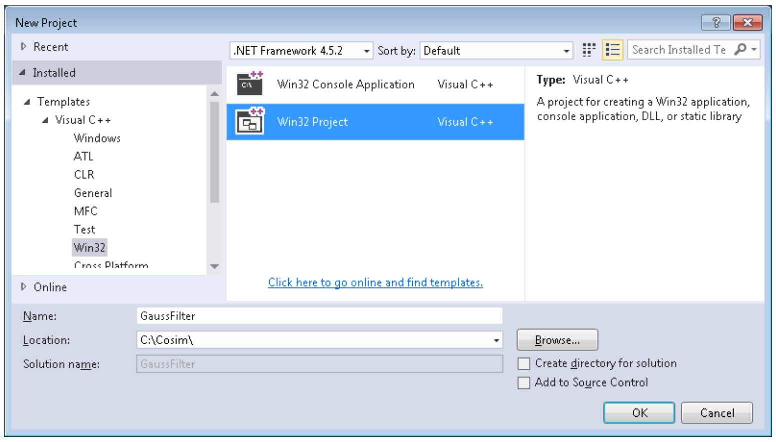
We will call our cosimulation module *GaussFilter*. The final source files are provided in the subfolder `simeng\cosim\examples\library\GaussFilter` of the VPI Design Suite installation, to save you from having to type out all the code if you wish to follow the example. However, no example schematic, or Visual C++ project or solution files are provided, so if you want to test the filter you will have to follow the steps in the example for yourself!

Step 1 — Create the Main DLL and Cosimulation Module Source Files

1. Copy the three files `DLLMain.cpp`, `MODULENAME.hpp` and `MODULENAME.cpp` from the subfolder `simeng\cosim\examples\library\Generic` of the installation into the development working directory. We shall use the directory `C:\Cosim\GaussFilter` in this example.
2. Optionally rename the files `MODULENAME.hpp` and `MODULENAME.cpp` to `GaussFilter.hpp` and `GaussFilter.cpp` respectively, if you would like them to be easily identifiable.
3. Open all three files in a text editor, and perform a global find-and-replace, replacing the string `##CosimModuleName##` with the string `GaussFilter`.

Step 2 — Create the Visual C++ Project

1. In Microsoft Visual Studio, open the New Project wizard by selecting New > Project from the File menu.
2. Choose the “Win32 Project” template. Set the location to `C:\Cosim`, and the project name to `GaussFilter`. Figure 2-57 shows the wizard.


New Project

Recent

Installed

Templates

Visual C++

Windows

ATL

CLR

General

MFC

Test

Win32

Cross Platform

Online

.NET Framework 4.5.2 Sort by: Default Search Installed Te

Win32 Console Application Visual C++

Win32 Project Visual C++

Type: Visual C++

A project for creating a Win32 application, console application, DLL, or static library

Click here to go online and find templates.

Name: GaussFilter

Location: C:\Cosim\

Solution name: GaussFilter

Browse...

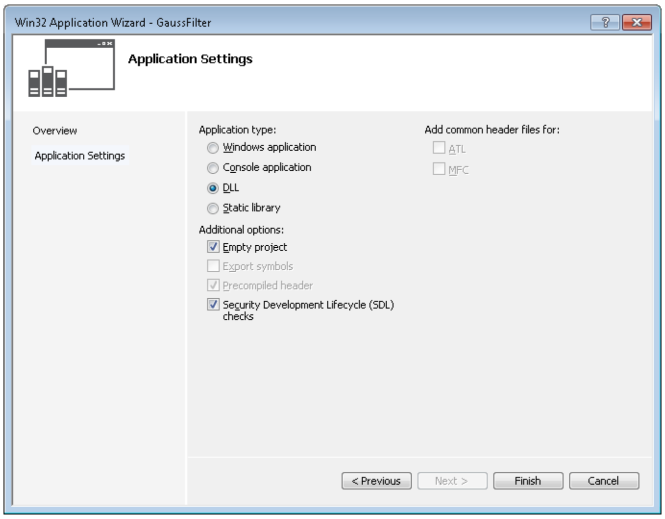
Create directory for solution

Add to Source Control

OK Cancel

Figure 2-57 *Creating the new project*

3. In the next window of the wizard, select DLL under Application type, make sure that the Empty project checkbox is selected, and click Finish.


Win32 Application Wizard - GaussFilter

Application Settings

Overview

Application Settings

Application type:

- Windows application
- Console application
- ● DLL
- Static library

Add common header files for:

- ☐ ATL
- ☐ MFC

Additional options:

- ☑ Empty project
- ☐ Export symbols
- ☑ Precompiled header
- ☑ Security Development Lifecycle (SDL) checks

< Previous Next > Finish Cancel

Figure 2-58 *Project settings dialog*

Step 3 — Add Files to the Project

1. From the Project menu, select Add Existing Item.
2. Select all three files created in Step 1, and add them to the project, as shown in Figure 2-59.

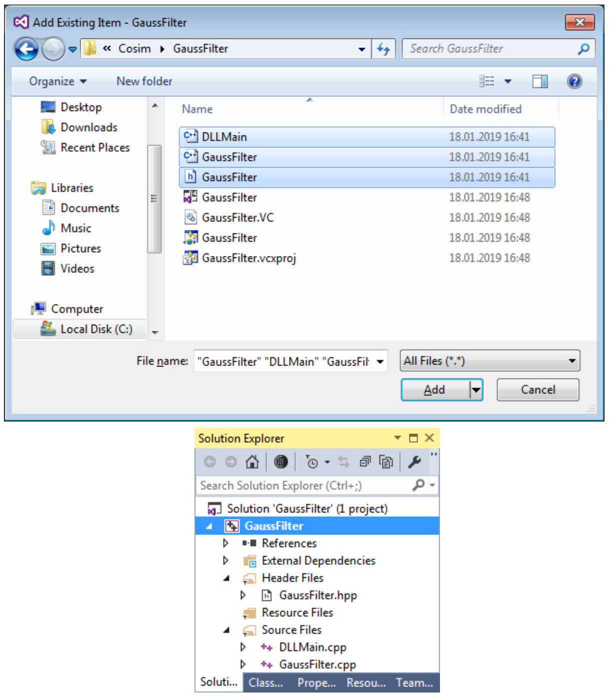

Add Existing Item - GaussFilter
« Cosim ▸ GaussFilter
Search GaussFilter
Organize
New folder
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Pictures
Videos
Computer
Local Disk (C:)
Name
Date modified
DLLMain 18.01.2019 16:41
GaussFilter 18.01.2019 16:41
GaussFilter 18.01.2019 16:41
GaussFilter 18.01.2019 16:48
GaussFilter.VC 18.01.2019 16:48
GaussFilter 18.01.2019 16:48
GaussFilter.vcxproj 18.01.2019 16:48
File name: "GaussFilter" "DLLMain" "GaussFil
All Files (*.*)
Add
Cancel
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'GaussFilter' (1 project)
GaussFilter
References
External Dependencies
Header Files
GaussFilter.hpp
Resource Files
Source Files
DLLMain.cpp
GaussFilter.cpp
Soluti... Class... Prope... Resou... Team...

Figure 2-59 *Adding source files to the project: selecting the files (top); and the resulting Solution File View (bottom)*

Step 4 — Insert the CosimLib Project into your Solution

By inserting the *CosimLib* project into your solution, you will have easy access to the source and header files for reference, and Microsoft Visual Studio will take care of building the `CosimLib.lib` file, and linking it with your cosimulation DLL.

1. From the File menu, select Add > Existing Project.
2. Navigate to the `simeng\cosim\examples\library\CosimLib\msvc` directory, and select the file `CosimLib.vcproj`.
3. From the Project menu, select Add Reference.
4. Select CosimLib project on the Add Reference dialog and click OK.
   This tells Microsoft Visual C++ that the resulting `CosimLib.lib` file will be linked into the GaussFilter DLL.

Step 5 — Set the Active Project and Active Configuration

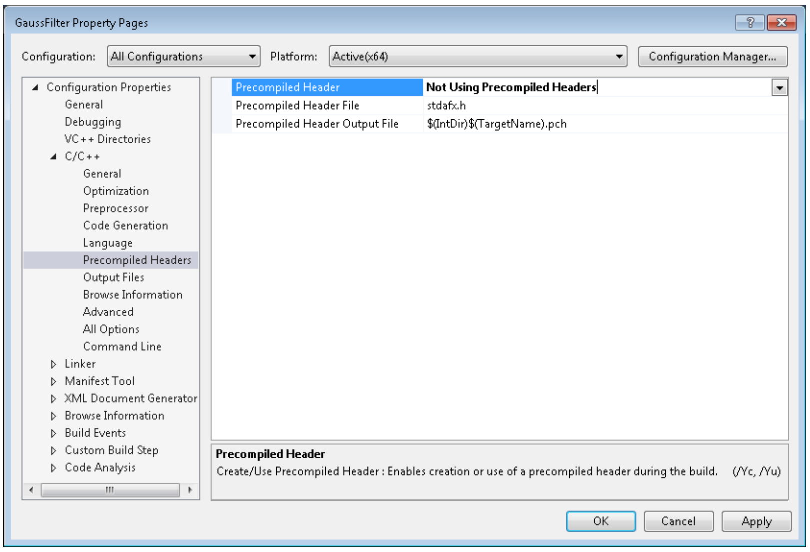
1. Select GaussFilter node in the Solution Explorer and select Set as StartUp Project from the Project menu.
2. From the Build menu, select Configuration Manager, and in the dialog set Active solution configuration to `Release`, and set Active solution platform to `x64`.

Step 6 — Define the Project Settings

1. From the Project menu, select Properties.
2. Choose to apply your settings to `All Configurations`.

3. Following the guidelines in the section “Project Settings” on page 186, make the following changes to the default configuration:

   + Disable the use of precompiled headers (see Figure 2-60).



GaussFilter Property Pages

Configuration: All Configurations Platform: Active(x64) Configuration Manager...

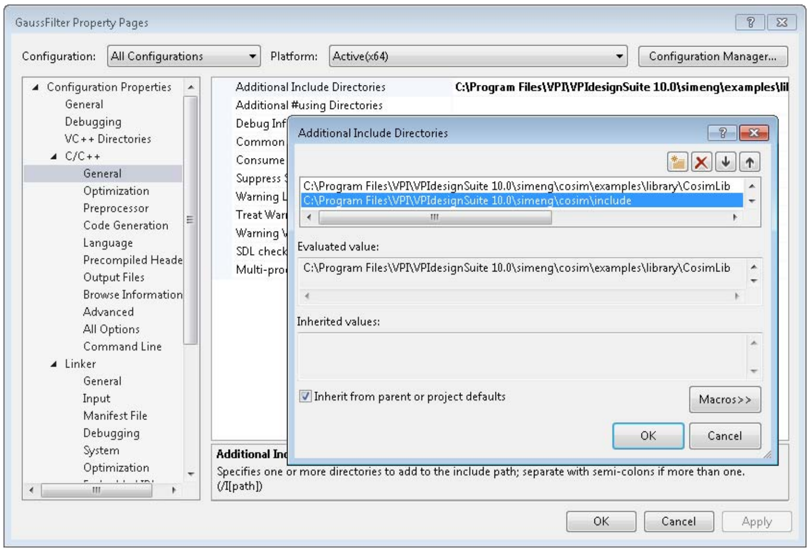
Precompiled Header	Not Using Precompiled Headers
Precompiled Header File	stdafx.h
Precompiled Header Output File	$(IntDir)$(TargetName).pch

Precompiled Header
Create/Use Precompiled Header : Enables creation or use of a precompiled header during the build. (/Yc, /Yu)

Figure 2-60 *Disabling precompiled headers*

   + Add the paths to the directories containing the include files `cosim.h` and `CosimLib.hpp` to the additional include paths list. For a typical installation, the list will be:

     `C:\Program Files\VPI\VPIdesignSuite 11.5\simeng\cosim\include,`
     `C:\Program Files\VPI\VPIdesignSuite 11.5\simeng\cosim\examples\library\CosimLib` (see Figure 2-61).


GaussFilter Property Pages

Configuration: All Configurations Platform: Active(x64) Configuration Manager...

- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
    - General
    - Optimization
    - Preprocessor
    - Code Generation
    - Language
    - Precompiled Heade
    - Output Files
    - Browse Information
    - Advanced
    - All Options
    - Command Line
  - Linker
    - General
    - Input
    - Manifest File
    - Debugging
    - System
    - Optimization

Additional Include Directories: C:\Program Files\VPI\VPIdesignSuite 10.0\simeng\examples\lil

Additional #using Directories

Additional Include Directories

C:\Program Files\VPI\VPIdesignSuite 10.0\simeng\cosim\examples\library\CosimLib

C:\Program Files\VPI\VPIdesignSuite 10.0\simeng\cosim\include

Evaluated value:

C:\Program Files\VPI\VPIdesignSuite 10.0\simeng\cosim\examples\library\CosimLib

Inherited values:

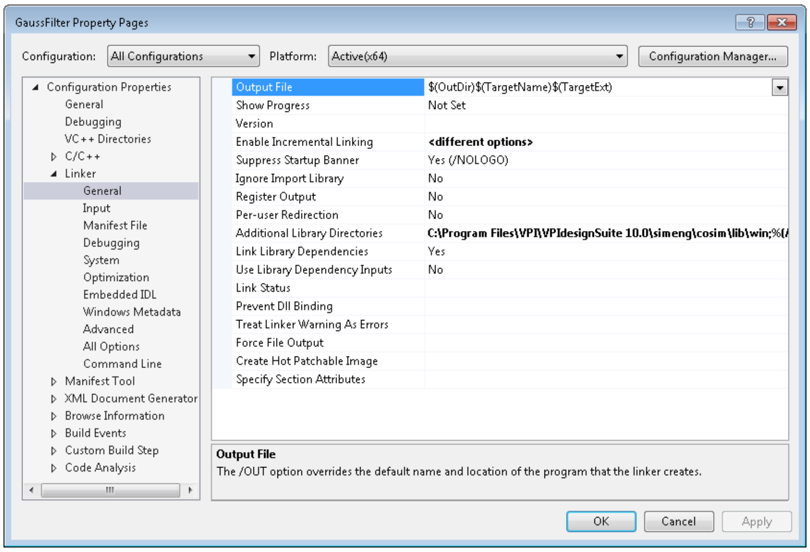
[✓] Inherit from parent or project defaults

Additional Inc

Specifies one or more directories to add to the include path; separate with semi-colons if more than one. (/I[path])

Figure 2-61 *Set the include file directories*

+ The Cosimulation Library requires the MIT FFTW library, so you must add the library file `fftw3_x64.lib` to the list of libraries to link with, and the path to the directory containing the file (typically `C:\Program Files\VPI\VPIdesignSuite 11.5\simeng\cosim\lib\win`) to the library search path (see Figure 2-62).



GaussFilter Property Pages

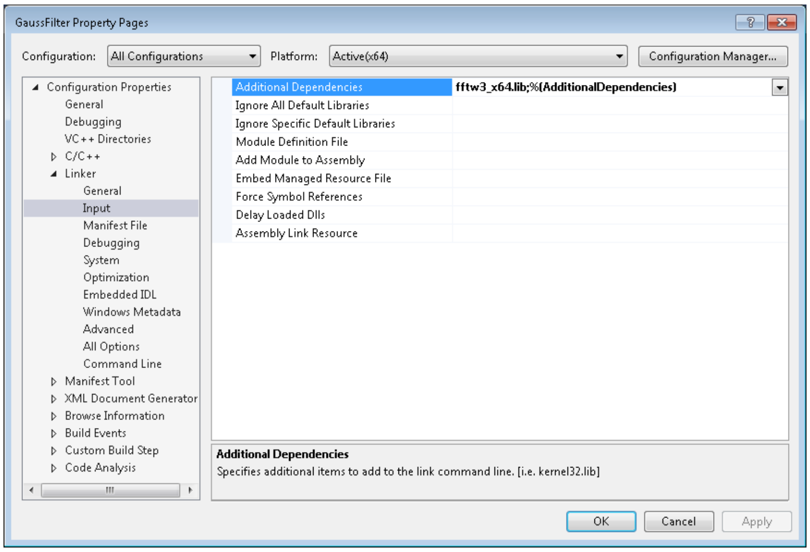
Configuration: All Configurations Platform: Active(x64) Configuration Manager...

- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File
    - Debugging
    - System
    - Optimization
    - Embedded IDL
    - Windows Metadata
    - Advanced
    - All Options
    - Command Line
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis

Output File	$(OutDir)$(TargetName)$(TargetExt)
Show Progress	Not Set
Version	
Enable Incremental Linking	**<different options>**
Suppress Startup Banner	Yes (/NOLOGO)
Ignore Import Library	No
Register Output	No
Per-user Redirection	No
Additional Library Directories	**C:\Program Files\VPI\VPIdesignSuite 10.0\simeng\cosim\lib\win;%(A**
Link Library Dependencies	Yes
Use Library Dependency Inputs	No
Link Status	
Prevent Dll Binding	
Treat Linker Warning As Errors	
Force File Output	
Create Hot Patchable Image	
Specify Section Attributes	

Output File
The /OUT option overrides the default name and location of the program that the linker creates.

OK Cancel Apply



GaussFilter Property Pages

Configuration: All Configurations Platform: Active(x64) Configuration Manager...

- Configuration Properties
  - General
  - Debugging
  - VC++ Directories
  - C/C++
  - Linker
    - General
    - Input
    - Manifest File
    - Debugging
    - System
    - Optimization
    - Embedded IDL
    - Windows Metadata
    - Advanced
    - All Options
    - Command Line
  - Manifest Tool
  - XML Document Generator
  - Browse Information
  - Build Events
  - Custom Build Step
  - Code Analysis

Additional Dependencies	**fftw3_x64.lib;%(AdditionalDependencies)**
Ignore All Default Libraries	
Ignore Specific Default Libraries	
Module Definition File	
Add Module to Assembly	
Embed Managed Resource File	
Force Symbol References	
Delay Loaded Dlls	
Assembly Link Resource	

Additional Dependencies
Specifies additional items to add to the link command line. [i.e. kernel32.lib]

OK Cancel Apply

Figure 2-62 *Adding the MIT FFTW library*

Step 7 — Create the Cosimulation Galaxy Schematic and Generate GaussFilter.h

1. Within VPI Design Suite, create the cosimulation galaxy and add the parameters (you can most easily do this by adapting the existing filter cosimulation galaxy from "Example 2 — Implementing an Optical Bandpass Filter Using C/C++" on page 179).
2. Two additional float-valued parameters, `NoiseDynamic` and `NoiseThreshold`, are required. When creating them, set their default values to 3 dB and -100 dBm respectively.

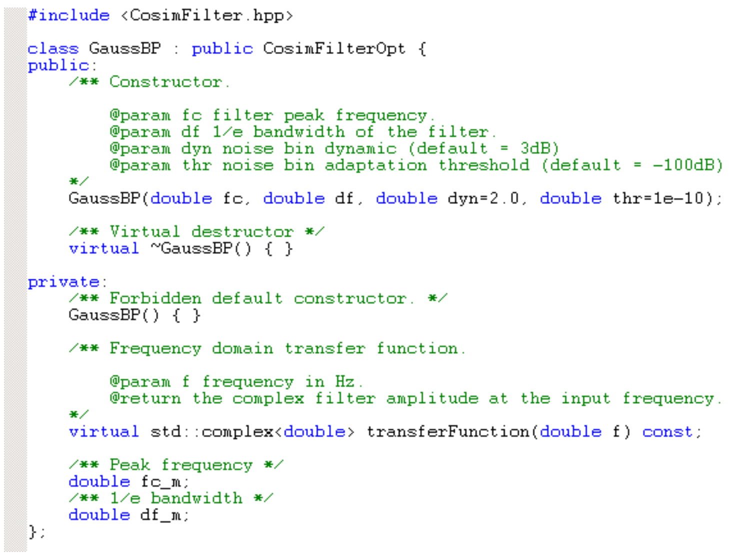
3. Save the cosimulation galaxy under the name *GaussFilter*, place it in a suitable schematic (again, you may wish to copy the schematic from Example 2), and then set the **InterfaceType** to GenerateHeader and create the GaussFilter.h header file in the project directory C:\Cosim\GaussFilter.

Note: The instance ID of the *GaussFilter* instance in the schematic should be *GaussFilter*. This instance ID is used by the header generation routine to name in/out and parameter structures in target header file.

4. Add the new header file to the project using **Project > Add Existing Item**, as in Step 3.
5. Open the file, and verify that it contains the structure definitions that you expect.

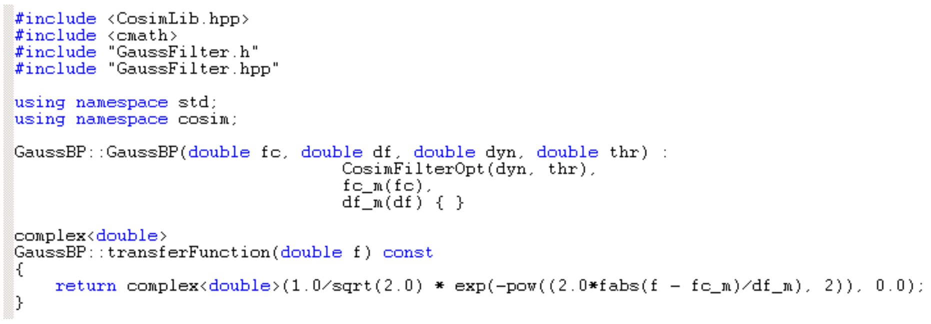
Step 8 — Write the Module Code

The bandpass filter will be created as a class called GaussBP that is derived from the cosim::CosimFilterOpt class. The new class needs only to implement the frequency domain transfer function, and so it is very simple. Since the class is small, we shall not bother to implement it in separate source file. Instead, the class declaration is added to the GaussFilter.hpp header file (see Figure 2-63), and the complete implementation of the class is added to the GaussFilter.cpp source file (see Figure 2-64).



```
#include <CosimFilter.hpp>

class GaussBP : public CosimFilterOpt {
public:
    /** Constructor.

        @param fc filter peak frequency.
        @param df 1/e bandwidth of the filter.
        @param dyn noise bin dynamic (default = 3dB)
        @param thr noise bin adaptation threshold (default = -100dB)
    */
    GaussBP(double fc, double df, double dyn=2.0, double thr=1e-10);

    /** Virtual destructor */
    virtual ~GaussBP() { }

private:
    /** Forbidden default constructor. */
    GaussBP() { }

    /** Frequency domain transfer function.

        @param f frequency in Hz.
        @return the complex filter amplitude at the input frequency.
    */
    virtual std::complex<double> transferFunction(double f) const;

    /** Peak frequency */
    double fc_m;
    /** 1/e bandwidth */
    double df_m;
};
```

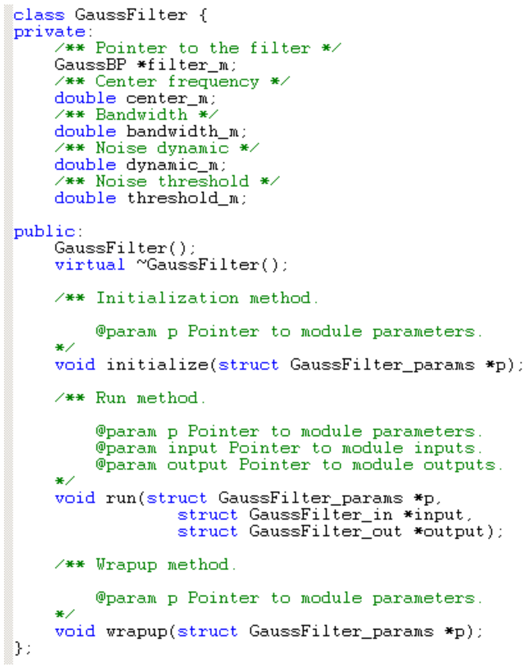
Figure 2-63 *Declaration of the GaussBP filter class*



```
#include <CosimLib.hpp>
#include <cmath>
#include "GaussFilter.h"
#include "GaussFilter.hpp"

using namespace std;
using namespace cosim;

GaussBP::GaussBP(double fc, double df, double dyn, double thr) :
                                CosimFilterOpt(dyn, thr),
                                fc_m(fc),
                                df_m(df) { }

complex<double>
GaussBP::transferFunction(double f) const
{
    return complex<double>(1.0/sqrt(2.0) * exp(-pow((2.0*fabs(f - fc_m)/df_m), 2)), 0.0);
}
```

Figure 2-64 *Implementation of the GaussBP filter class*

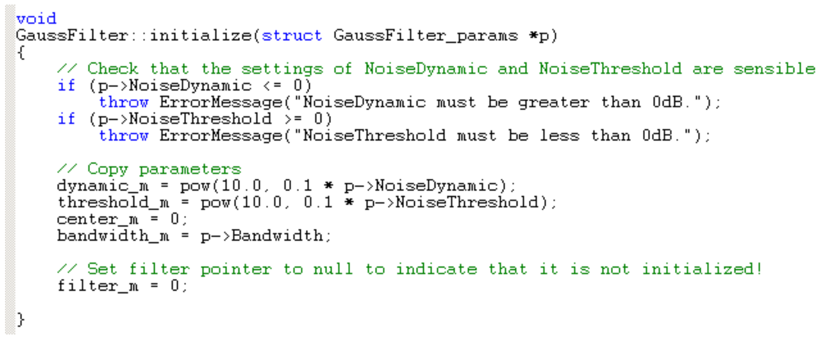
The complete declaration of the `GaussFilter` class is shown in Figure 2-65. Note that it includes a member variable to hold a pointer to its `GaussBP` filter, and that it also keeps private copies of all its parameter settings.



```
class GaussFilter {
private:
    /** Pointer to the filter */
    GaussBP *filter_m;
    /** Center frequency */
    double center_m;
    /** Bandwidth */
    double bandwidth_m;
    /** Noise dynamic */
    double dynamic_m;
    /** Noise threshold */
    double threshold_m;

public:
    GaussFilter();
    virtual ~GaussFilter();

    /** Initialization method.

        @param p Pointer to module parameters.
    */
    void initialize(struct GaussFilter_params *p);

    /** Run method.

        @param p Pointer to module parameters.
        @param input Pointer to module inputs.
        @param output Pointer to module outputs.
    */
    void run(struct GaussFilter_params *p,
             struct GaussFilter_in *input,
             struct GaussFilter_out *output);

    /** Wrapup method.

        @param p Pointer to module parameters.
    */
    void wrapup(struct GaussFilter_params *p);
};
```

Figure 2-65 *Declaration of the GaussFilter cosimulation module class*

The implementation of the initialize function is shown in Figure 2-66. It checks that the input parameters are sensible before making local copies of them (in the process converting from decibels to linear units). This is a rather unusual implementation because the center frequency of the filter is provided by an input to the module, rather than a parameter. Thus the initialize function is unable to create the filter because it does not know all the required parameters. Instead, it sets the filter member pointer to null for now.

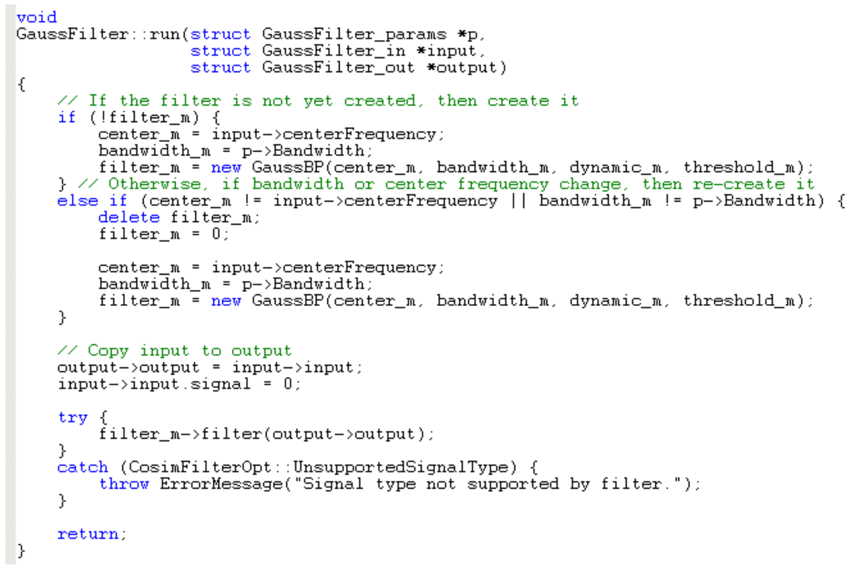


```
void
GaussFilter::initialize(struct GaussFilter_params *p)
{
    // Check that the settings of NoiseDynamic and NoiseThreshold are sensible
    if (p->NoiseDynamic <= 0)
        throw ErrorMessage("NoiseDynamic must be greater than 0dB.");
    if (p->NoiseThreshold >= 0)
        throw ErrorMessage("NoiseThreshold must be less than 0dB.");

    // Copy parameters
    dynamic_m = pow(10.0, 0.1 * p->NoiseDynamic);
    threshold_m = pow(10.0, 0.1 * p->NoiseThreshold);
    center_m = 0;
    bandwidth_m = p->Bandwidth;

    // Set filter pointer to null to indicate that it is not initialized!
    filter_m = 0;

}
```

Figure 2-66 *The GaussFilter initialization method*

The implementation of the run function is shown in Figure 2-67. If it finds that the filter pointer is null, it creates the filter using the input center frequency, which is now known. It also checks to see if the center frequency input, or bandwidth parameter have changed. (This could be achieved during a simulation run using the *ParameterController* module, for example, or the center frequency input could be driven using a *Ramp* module, which would cause it to change on each iteration.) If the center frequency or bandwidth have changed, any existing filter is deleted, and a new one with the updated parameters is created.

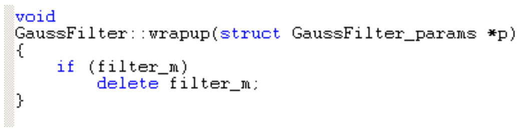
Note that often it is necessary to explicitly write code that can handle changes in parameter settings. In this case, the filter must be recreated if there are any parameter changes. An alternative implementation of the GaussBP filter class might provide member functions for changing the parameters without creating a new filter. It is generally a design decision whether to respond to parameter changes during a simulation run (so called *volatile* parameters), or to use only the parameter settings established at initialization. In this example, changes in the **Bandwidth** parameter cause the bandwidth of the filter to change, but changes in other parameters are ignored.



```
void
GaussFilter::run(struct GaussFilter_params *p,
                 struct GaussFilter_in *input,
                 struct GaussFilter_out *output)
{
    // If the filter is not yet created, then create it
    if (!filter_m) {
        center_m = input->centerFrequency;
        bandwidth_m = p->Bandwidth;
        filter_m = new GaussBP(center_m, bandwidth_m, dynamic_m, threshold_m);
    } // Otherwise, if bandwidth or center frequency change, then re-create it
    else if (center_m != input->centerFrequency || bandwidth_m != p->Bandwidth) {
        delete filter_m;
        filter_m = 0;

        center_m = input->centerFrequency;
        bandwidth_m = p->Bandwidth;
        filter_m = new GaussBP(center_m, bandwidth_m, dynamic_m, threshold_m);
    }

    // Copy input to output
    output->output = input->input;
    input->input.signal = 0;

    try {
        filter_m->filter(output->output);
    }
    catch (CosimFilterOpt::UnsupportedSignalType) {
        throw ErrorMessage("Signal type not supported by filter.");
    }

    return;
}
```

Figure 2-67 *The GaussFilter run method*

The wrap-up function is shown in Figure 2-68. All it has to do is free the memory occupied by the filter.



```
void
GaussFilter::wrapup(struct GaussFilter_params *p)
{
    if (filter_m)
        delete filter_m;
}
```

Figure 2-68 *The GaussFilter wrap-up method*

Additional Examples

A number of additional examples using the VPI Cosimulation Library are provided. Each example consists of source code and Visual C++ project file, found in the `simeng\cosim\examples\library` subfolder of the VPI Design Suite installation, and a demonstration of the example, found in the demonstrations *folder Optical Systems Demos > Simulation Techniques > Cosimulation > Library*. A brief description of each example is provided. One consolidated Microsoft Visual Studio solution for all examples may be found in `simeng\cosim\examples\library`.

Three Port Fabry-Perot Filter

Optical Systems Demos > Simulation Techniques > Cosimulation > Library > FP Filter (T&R)

Cosimulation module name:

FPFilterReflect

Directory (relative to VPI Design Suite installation):

`simeng\cosim\examples\library\FPFilterReflect`

Demonstration name:

FP Filter with Reflection Port

Demonstration Description:

A Fabry-Perot filter with a reflection port as well as a transmission port is implemented as a C++ cosimulation module.

The *FPFilterReflect* module is implemented as a single C++ class encapsulating all the data required to represent an instance of the filter on a schematic.

The module class contains two filters, one representing the transmission characteristics of the Fabry-Perot filter, and the other representing its reflection characteristics. Each filter class is derived from the `CosimFilterOpt` base class in the VPI Cosimulation Library. The features of the filter, including the filtering function itself as well as advanced features such as noise bin adaptation, are provided by the `CosimFilterOpt` base class. The Fabry-Perot filter is defined simply by providing derived classes implementing the frequency domain filter response function.

Additionally, the example demonstrates how the standard **`Active`** parameter can be implemented in a cosimulation module.

Low Pass Electrical Filtering

Optical Systems Demos > Simulation Techniques > Cosimulation > Library > Electrical LP Filter

Cosimulation module name:

ElectricalFilter

Directory (relative to VPI Design Suite installation):

`simeng\cosim\examples\library\ElectricalFilter`

Demonstration name:

Electrical Lowpass Filter with Resampling

Demonstration Description:

This cosimulation example illustrates the application of the base class `CosimFilterElec`, and the function `resample(ONSL_ElectricalBand *band, int newLength)` in the VPI Cosimulation Library.

The *ElectricalFilter* module is implemented as a single C++ class encapsulating all the data required to represent an instance of the filter on a schematic. The main module class uses an instance of a class called `PiecewiseElecFilter` to perform the filtering. This class is derived from the `CosimFilterElec` class, and only needs to provide an implementation of the virtual function `transferFunction(double f)`. In this example, the filter has a piecewise linear frequency response, with flat pass and stop bands and a linear transition region between the two.

Additionally, if the transmission in the stop band is zero, this module will downsample the electrical signal (by a factor that is a power of two) when possible. It does this using the `resample` function in the VPI Cosimulation Library.

This demonstration setup demonstrates the downsampling. The electrical bandwidth of the signal generated by the impulse generator is 20 GHz (**`SampleRateDefault`** = 40e9 Hz). In the upper filter, the stop band transmission is 0.1, so downsampling is not possible since aliasing would occur. In the lower filter the stop band transmission is zero, and the module is able to downsample by a factor of two without aliasing. The difference in the total bandwidth of the resulting signals can be seen on the RF spectrum analyzers.

Simple Optical Signal Resampling

Optical Systems Demos > Simulation Techniques > Cosimulation > Library > Optical Signal Resampling

Cosimulation module name:

ResampleOpt

Directory (relative to VPI Design Suite installation):

`simeng\cosim\examples\library\ResampleOpt`

Demonstration name:

Optical Signal Resampling in a DLL

Demonstration Description:

This cosimulation example illustrates the use of the function `resample(ONSL_OpticalBand *band, int newLength)` in the VPI Cosimulation Library. The optical signal resampler changes the total number of samples in all optical sampled bands of the input signal to the number of samples set by the parameter **`OutputSamples`**. This is a very crude resampling function, as the results of running this schematic demonstrate. However, the `resample` function can be used as the basis for more sophisticated processing requiring changes in sample rate. (Note that more sophisticated resampling functions are provided by the modules *ResampleAbsolute*, *ResampleRelative* and *ResampleAdaptive*, that can be found in the *Signal Representation Conversion* folder of the module library.)

Low Pass Electrical Filtering of Sample Mode Signals

Optical Systems Demos > Simulation Techniques > Cosimulation > Library > Sample Mode Filter

Cosimulation module name:

SMFilterFIRLowpass

Directory (relative to VPI Design Suite installation):

`simeng\cosim\examples\library\SMFilterFIRLowpass`

Demonstration name:

Sample Mode Cosimulation Filter

Demonstration Description:

This cosimulation example illustrates the application of the FFTW library in the VPI cosimulations.

The SMFilterFIRLowpass module is implemented as a single C++ class encapsulating all the data required to represent an instance of the filter on a schematic. In this example, the filter is a finite impulse response (FIR) lowpass filter.The FFTW library is used in order to perform Fourier transformation during generation of the filter coefficients corresponding to the desired filter order and cutoff frequency.

This schematic simply demonstrates the spectral response of the filter by plotting the RF spectrum generated from the filter impulse response.

Cosimulation with COM Components

This section describes the use of the cosimulation interface using a COM component as the target environment.

System Requirements

To perform cosimulation with COM components, you will need a Windows operating system and a suitable compiler or scripting language which supports the creation of COM components. Many modern languages provide such support, including C++, all .NET languages (like C#, VB.NET, etc.), Python, Java, Delphi, etc.

The sections below describe how to create a COM component in different languages.

COM Concepts

The Component Object Model is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or "components" to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems. To support its interoperability features, COM defines and implements mechanisms that allow applications to connect to each other as software objects. A software object is a collection of related functions and associated state.

Interfaces

Interfaces are a key concept of COM. An interface can be understood as a contract between a component and its clients. An interface defines which functions can be called (their names and prototypes). Each interface has unique identifier: the "IID" or Interface Identifier.

Objects

In this document, the terms "coclass", "COM component" and "COM object" are used as synonyms to describe an object which can be created and which supports some set of interfaces. When an object implements only one interface, this interface may be referred to in the documentation simply as an "object" (for instance, the `IQuantizedValue` object).

Each COM component has a unique identifier: the "CLSID" or Class Identifier.

COM Servers

COM servers can be understood as storage for COM components. They can be implemented as Dynamic-Link Libraries (DLL) or executable files (EXE). For languages which do not imply creation of DLLs or EXE files, special wrappers may be provided to meet this requirement (for example for Python or Java).

Identification

To identify different entities (like COM components and interfaces), a special type of identifier is used: the "GUID" or Globally Unique Identifier. It is ensured that each newly generated instance of this identifier is unique across the world. An example of a GUID is `{F943C7F6-1C4C-49FF-995F-223238BAC4E0}`. Both the CLSID and IID are GUIDs.

Each language provides its own tools for generating GUIDs and some development environments even generate them automatically when creating source code from templates.

In addition to the CLSID, COM components may also be identified by their human-readable name: the "ProgID" or Programmatic Identifier. This is a string which must comply with the following requirements:

- has no more than 39 characters
- contains no punctuation (including underscores) except one or more periods
- must not start with a digit

An example of a ProgID is `CosimLibrary.InterfaceModule.1`.

COM components are registered in the Windows registry and are thus available for access by clients regardless of their actual location.

Error Reporting

COM provides a standardized approach to error reporting. Each method of a COM component returns a special code which indicates success or failure of method execution. The standard `IErrorInfo` interface may be used to return additional information about an error (like error message).

All these complications are hidden from the user in high-level languages like C# or Python where the native exception handling mechanisms should be used instead.

Setting Up the Cosimulation Interface

To choose a COM component as the target environment, set the **InterfaceType** parameter of the *CoSimInterface* module to COM. In this mode, the various parameters of the cosimulation module are interpreted as follows:

- **ComponentId**: set this parameter to the ProgID or CLSID of the cosimulation module COM component (see "Identification" on page 215 for more information about ProgID and CLSID).

  For example it can be something like `CosimLibrary.InterfaceModule.1` or `{F943C7F6-1C4C-49FF-995F-223238BAC4E0}`.
- **LogicalInformation**: set this parameter to `Yes` to access the logical channels during the cosimulation. Note that the global parameter **LogicalInformation** should also be `On` or `CurrentRun`.

Module Interface

Similar to the Dynamic-Link Library interface, the COM interface does not use the **InitCommand**, **RunCommand** and **WrapupCommand** parameters to define functions to be called during cosimulation. Instead, the cosimulation module COM object must implement a specific interface:

```
[
	uuid(26657EBF-858F-4F71-88A0-488659EF55D2)
]
interface ICosimModule : IDispatch {
	[id(0x00000001)]
	HRESULT Init(
				[in] ICosimContext* ctx,
				[in] IParameterCollection* prms);
	[id(0x00000002)]
	HRESULT Run(
				[in] ICosimContext* ctx,
				[in] IParameterCollection* prms,
				[in] IPortCollection* input,
				[in] IPortCollection* output);
	[id(0x00000003)]
	HRESULT WrapUp(
				[in] ICosimContext* ctx,
				[in] IParameterCollection* prms);
};
```

This interface contains three functions:

- `Init(ICosimContext* ctx, IParameterCollection* prms)`

  This function is called when the interface is initialized, prior to simulation start. The cosimulation context object and initial parameter settings are passed to the function.

- `Run(ICosimContext* ctx, IParameterCollection* prms, IPortCollection* input, IPortCollection* output)`
  This function is called each time the cosimulation module is fired during the simulation run. The current cosimulation context object, parameter settings, inputs and the object which can be used to provide the outputs are passed to the function. Parameter settings may change in the course of a simulation run (for example, when using the *ParameterController* star modules); however, such changes may be ignored by making copies of the initial parameter settings in the initialization function. Thus cosimulation parameters may be "volatile" or "nonvolatile" according to how the module is implemented (however, there is no way for the simulator to know, and to generate an error if an attempt is made to change a "nonvolatile" parameter).
- `WrapUp(ICosimContext* ctx, IParameterCollection* prms)`
  This function is called when the simulation is complete, prior to releasing all the resources used by the cosimulation interface. The cosimulation context object and the current parameter settings are passed to the function.

Simulation Environment Interface

The module interface is used by the simulation environment to call the cosimulation implementation. The simulation environment interface provides access to simulation information (such as global parameters and logical information) and functions implemented by the simulation environment (such as signal object creation functions and message/warning reporting routines).

Cosimulation Context

The cosimulation context is an object which is sent to all three functions of the cosimulation module and which provides access to various simulation information and useful routines:

```
// Read global parameter
double tw = (double)ctx.SimulationParameters["TimeWindow"].Value;
// Report warning
if( tw < 1.6e-9 )
      ctx.WriteMessageEx(wmtWarning, "Warning!!!");
```

The following interfaces are related to the simulation context:

- `ICosimContext`,
- `ISweepInfo`,
- `ISimulationParameters`,
- `ISignalFactory`,
- `IObjectFactory`.

`ICosimContext`

Purpose:

This interface provides access to global simulation information, signal construction helper and message/warning reporting routines.

Properties:

`InstanceId`

Returns a string that uniquely identifies the instance of the cosimulation module in the schematic. This value is not the same as instance ID of the *CoSimInterface* module, but is derived from it.

`IsRemoteSimulation`

Returns `true` if a simulation is running at the Remote Simulation Server.

`LogicalInfo`

Returns a pointer to the `ILogicalChannelCollection` interface, which provides access to the logical information if the **`LogicalInformation`** parameter of the *CoSimInterface* module is set to Yes. For details on logical information, see "Logical Information" on page 237.

`SignalFactory`

Returns a pointer to the `ISignalFactory` interface, which must be used for creation of signal objects. See "Factory" on page 226.

`SimulationParameters`

Returns a pointer to the `ISimulationParameters` interface, which provides access to global parameters.

`SweepInfo`

Returns a pointer to the `ISweepInfo` interface, which provides information about interactive simulations.

`ObjectFactory`

Returns a pointer to the `IObjectFactory` interface, which must be used to create objects other than signals.

Methods:

`WriteMessage(message)`

This function may be used to send messages to the VPI Design Suite Message Log.

`WriteMessageEx(type, message)`

Similar to `WriteMessage()` but supports several message types such as info, warnings, and progress messages.

Progress messages are processed if Report Progress is switched on in the Submit Simulation Job dialog.

Parameter *type* can be either `wmtInfo`, `wmtWarning` or `wmtProgress` for info, warning or progress messages, respectively.

Note: The type `wmtMessage` is a deprecated alias for the `wmtInfo`.

`ISweepInfo`

Purpose:

This interface provides access to information about the cosimulation module iterations, see "Iteration (Sweep) Information" on page 90.

Properties:

`LevelsCount`

Returns number of sweep levels.

Methods:

`GetCurrent(level)`

Returns current iteration index at specific sweep level. Level is zero-based but iteration index starts from one.

`GetTotal(level)`

Returns total number of iterations at specified sweep level. Level is zero-based.

`ISimulationParameters`

Purpose:

Provides access to global parameters.

Properties:

`BoundaryConditions`

Returns boundary conditions used in simulation. Can be `bcPeriodic`, `bcAperiodic` or `bcMixed`.

`TimeWindow`

Duration of a block of samples in seconds.

`SampleModeBandwidth`

Optical bandwidth of the sampled signals in Hz.

`SampleModeCenterFrequency`

Center frequency of the simulation in Hz.

`GreatestPrimeFactorLimit`

Limit for the greatest prime factor for the signal samples.

`FrequencyStep`

Fundamental frequency step used in simulation : step of the frequency grid in Hz. Equals to `1/TimeWindow`.

IObjectFactory

Purpose:

Creates objects (other than signals) that must be created explicitly.

Methods:

`CreateVariableCollection()`

Returns a new empty `IVariableCollection` object.

Parameters

Parameters of module are also sent to all three methods of the cosimulation module. They are provided as a collection of `ISimulationParameter` objects:

```
// Read parameters
double sampleRate = (double)prms["SampleRate"].Value;
double bitRate = (double)prms["BitRate"].Value;
```

IParameterCollection

Purpose:

Represents a collection of parameters. Used to provide access to parameters of cosimulation galaxy.

This collection supports iteration of its values.

Properties:

`Names`

Returns an array of all parameter names.

Methods:

`Contains(name)`

Returns `true` if a parameter with the specified name exists in the collection.

`Item(name)`

Returns `IParameter` interface if a parameter with the specified name exists in the collection or throws an exception otherwise.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

IParameter

Purpose:

Represents a module parameter.

Properties:

`Name`

Returns name of the parameter.

`Type`

Returns type of the parameter. The following table shows the relation between the value of this property, the underlying type and type of the source parameter:

Table 2-22 *Relation between property Type, parameter type in VPI Design Suite and type which should be used in the target language*

Parameter type	Value of Type property	Target type
Integer	prmInt	int
Integer array	prmIntegerArray	IIntArray
Float	prmFloat	double
Float array	prmFloatArray	IDoubleArray
Complex	prmComplex	IComplex
Complex array	prmComplexArray	IComplexArray
String Enumeration Input/Output file Input/Output file array Input/Output directory	prmString	string
String array	prmStringArray	IStringArray

TypeEx

Returns type of the parameter. The following table shows the relation between the value of this property, the underlying type, and the type of the source parameter:

Table 2-23 *Relation between property TypeEx, parameter type in VPI Design Suite and type which should be used in the target language*

Parameter type	Value of Type property	Target type
Integer	prmInt	int
Integer array	prmIntegerArray	IIntArray
Float	prmFloat	double
Float array	prmFloatArray	IDoubleArray
Complex	prmComplex	IComplex
Complex array	prmComplexArray	IComplexArray
String	prmString	string
Enumeration	prmEnum	string
Input/Output file Input/Output directory	prmPath	string

Table 2-23 *Relation between property TypeEx, parameter type in VPI Design Suite and type which should be used in the target language*

Parameter type	Value of Type property	Target type
Input/Output file array	prmPathArray	string
String array	prmStringArray	IStringArray

`IsFunc`

Indicates if the parameter is a function parameter.

`ExprType`

Returns an expression type of the parameter. Expression type and value follow rules provided under "Function Parameters Evaluation" on page 91.

`Value`

Returns value of the parameter. In strongly typed languages (such as C#), the returned value should be cast to a specific type prior to usage.

Methods:

`Eval(vars)`

Returns the result of the Python expression evaluation for the specified variable values `vars`. The parameter `vars` is an object of the `IVariableCollection` interface. The type of the result value (or the type of elements of the result value if the result value is a vector) of the `Eval` method must be the same as (equivalent to) the type of the evaluated scalar parameter: `int`, `SAFEARRAY<int>`, `double`, `SAFEARRAY<double>`, `Complex`, `SAFEARRAY<Complex>`.

The dimension of type of the variable can be increased by one dimension for simultaneous evaluation of the Python expression for several values of this variable. Thus a single call of the `Eval` method can be performed instead of multiple calls. Correspondingly, the dimension of the result value of the `Eval` method is equal to or greater by one dimension than the dimension of the type of the evaluated parameter. For example, if a Python expression takes a float argument `$x` and returns a complex value, then, if an array of floats is passed to `$x` in *`vars`*, the result of evaluation will be an array of complex values corresponding to the evaluation results for each value in the array `$x`.

```
// Expression evaluation
// Parameter p = a parameter with the expression "$x*$y";
VariableCollection args = ctx.ObjectFactory.CreateVariableCollection();
args.Add("$x", 1.2);
args.Add("$y", -0.7);
double xy = (double)p.Eval(args);
```

IVariableCollection

Purpose:

Represents a collection of variables (objects of the `IVariable` interface). Used as a parameter of the function `Eval` of the `IParameter` interface. Shall be created using `ICosimContext.ObjectFactory.CreateVariableCollection()` function.

This collection supports iteration of its values.

Methods:

`Add(name, value)`

Adds a new variable with the name and the value.

`Item(name)`

Returns `IVariable` interface if a variable with the specified name exists in the collection or throws an exception otherwise.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

IVariable

Purpose:

Represents a variable used in an expression that is evaluated with the function `Eval` of the `IParameter` interface.

Properties:

`Name`

Returns name of the variable.

`Value`

Returns value of the variable.

Ports

Access to input and output information is provided only in `Run()` method through collection of `IPort` objects. For *multiple* ports, several connections may be assigned to one port. This is handled as a collection of `IConnection` objects:

```
Port inPort = input["input"];
Connection conn = inPort.Connections[0];
OptSignal os = (OptSignal)conn.Data;
```

IPortCollection

Purpose:

Represents a collection of module ports. Used to provide access to information at input ports and can be used to set up data on the output ports.

This collection supports iteration of its values.

Properties:

`Names`

Returns an array of all port names in the collection.

Methods:

`Contains(name)`

Returns `true` if a port with the specified name exists in the collection.

`Item(name)`

Returns `IPort` interface if port with the specified name exists in the collection; throws exception otherwise.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

`IPort`

Purpose:

Represents input or output port of the module.

Properties:

`Name`

Returns name of the port.

`Type`

Returns type of the port.

The following table shows the relation between the value of this property, the underlying type and type of the source data:

Table 2-24 *Relation between property Type, port type in VPI Design Suite and type which should be used in the target language*

Port type	Value of Type property	Type of Data property
Float	ptFloat	double
Float matrix	ptFloatMatrix	IDoubleMatrix
Complex	ptComplex	IComplex
Complex matrix	ptComplexMatrix	IComplexMatrix
Optical	ptOptical	IOptSample if SubType is pstSample IOptSignal if SubType is pstBlock
Electrical	ptElectrical	IElSample if SubType is pstSample IElSignal if SubType is pstBlock
Bit Sequence	ptFloatMatrix	IDoubleMatrix

`IsMultiPort`

Returns `true` if it is multiple port (e.g., allows multiple connections).

`SubType`

Returns subtype for optical and electrical ports. Can be one of `pstSample` or `pstBlock`.

`Connections`

Returns a collection of objects representing data at some port. It is a collection because several connection may be made to port (in case it is multiple port).

`IConnectionCollection`

Purpose:

Represents a collection of connections to the port.

This collection supports iteration of its values.

Properties:

`Count`

Returns number of connections to the port.

Methods:

`Item(index)`

Returns `IConnection` object if a connection with the specified index exists in the collection; throws exception otherwise.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

`IConnection`

Purpose:

Represents data at a connection on an input or output port of the module.

Properties:

`IsNull`

Returns `true` if null data is received. Such data can be produced by *NullSource* module or modules in inactive state.

`Data`

Returns value at the connection to the port. It can be of one of the following types: `double`, `IDoubleMatrix`, `IComplex`, `IComplexMatrix`, `IOptSample`, `IElSample`, `IOptSignal`, `IElSignal`. See Table 2-24 on page 224 in the description of the `IPort.Type` property.

In strongly typed languages (such as C#), the returned value should be cast to a specific type prior to usage.

Mapping VPI Signals

COM cosimulation uses COM components and interfaces to map ONSL internal signals. In contrast to MATLAB, Python and Library cosimulations, in COM cosimulation, these are not simple data structures, but complex objects that both store data and provide functions.

Factory

Factory methods are introduced to control creation of objects. It is usually not allowed to create objects directly because in that case they may have the wrong state. Using a factory ensures that objects are created with vital parameters set consistently.

Many objects and collections provide similar factory methods to create subordinate objects.

```
// Create new optical signal and set its properties.
OptSignal os = signalFactory.CreateOpticalSignal();
os.LowerFrequency = signalFactory.CreateQV(188.0e12, df);

// Create new noise bin and add it to the signal
OptNoiseBin noiseBin = noise.CreateNoiseBin(st, cf, bw, sv);
noise.Add(noiseBin);
```

`ISignalFactory`

Purpose:

Create root signal objects.

Methods:

`CreateOpticalSample(startTime, x, y)`

Returns new `IOptSample` object initialized as specified.

`CreateOpticalSignal()`

Returns new `IOptSignal` object.

`CreateElectricalSample(startTime, e)`

Returns new `IElSample` object initialized as specified.

`CreateElectricalSignal()`

Returns new `IElSignal` object.

`CreateQV(value, step)`

Returns new quantized value. Automatically calculates *`factor`* dividing *`value`* by *`step`* and rounding it appropriately.

`CreateQVExact(factor, step)`

Returns new quantized value initialized as specified.

`CreateIntArray(size)`
`CreateDoubleArray(size)`
`CreateDoubleMatrix(rows, cols)`
`CreateComplexArray(size)`
`CreateComplexMatrix(rows, cols)`
`CreateStringArray(size)`

Returns new `IIntArray`, `IDoubleArray`, `IDoubleMatrix`, `IComplexArray`, `IComplexMatrix` or `IStringArray` of specified size.

Collections

There are three types of objects in the signal library which store multiple objects: arrays, matrices and collections.

IXxxArray

Purpose:

Arrays provide access to stored elements via an index. They have a fixed size after initialization and provide methods to convert from/to native COM arrays, which in turn usually have native support in the target language:

```
// Create array of doubles using signals factory
IDoubleArray arr = ctx.SignalFactory.CreateDoubleArray(2);

// Create native array of double values
double[] src = new [] {5.7, 3.6};

// Copy data from native array
arr.FromArray(src);
```

Properties:

`Count`

Number of elements in the array.

Methods:

`Item(index)`

Returns an element with the specified index or throws an exception if the index goes beyond bounds.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

`FromArray(array)`

Initializes an array with elements from a COM array, which is usually represented by an array from the target language.

`ToArray()`

Returns a COM array of contained elements.

IXxxMatrix

Purpose:

Represents a rectangular matrix and provides access to stored elements via a row and column index.

Similar to arrays, matrices have a fixed size after initialization and provide methods to convert from/to native COM two-dimensional arrays, which in turn usually have native support in the target language:

```
// Create complex matrix using signals factory
```

```
ComplexMatrix m =
ctx.SignalFactory.CreateComplexMatrix(2, 2);

// Create native array of Complex values
Complex[,] src = new Complex[2,2];
src[0,0] = new Complex {Real = 1, Imag = 2};
src[0,1] = new Complex {Real = 3, Imag = 4};
src[1,0] = new Complex {Real = 5, Imag = 6};
src[1,1] = new Complex {Real = 7, Imag = 8};
// Copy data to complex matrix
m.FromArray(src);
```

Properties:

`Rows`

Number of rows in matrix.

`Cols`

Number of columns in matrix.

Methods:

`Item(row, col)`

Returns an element with the specified index or throws an exception if the index goes beyond bounds.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

`FromArray(array)`

Initializes a matrix with elements from COM two-dimensional array.

`ToArray()`

Returns a two-dimensional COM array of contained elements.

IXxxCollection

Purpose:

While arrays serve mainly to store values of primitive types and have a fixed size, collections store more comprehensive objects and frequently allow adding and/or removing elements on the fly:

```
// Create noise bin and add it to the collection
OptNoiseBin noiseBin = noise.CreateNoiseBin(st, cf, bw, sv);
noise.Add(noiseBin);
```

Usually collections do not provide access to elements (neither by index nor by key) but they always provide a possibility to enumerate all elements:

```
// Sum power in all optical sampled bands.
// The power in a band is the average power of its samples.
foreach (OptSampledBand band in inputSignal.SampledBands)
{
        ...
}
```

Properties:

`Count`

Number of elements in the collection.

Methods:

`Add(object)`

Adds a new element to the collection.

Generic Types

`IQuantizedValue`

Purpose:

Serves to hold quantities in a quantized manner, where the value is given as a multiple of an atomic unit (step).

Properties:

`Factor`

Multiplier.

`Step`

Grid step.

`Value`

Resulting value which equals the `Step*Factor`.

`IComplex`

Purpose:

Represents a complex number.

Properties:

`Real`

Real part of the complex number.

`Imag`

Imaginary part of the complex number.

`IIntArray`

Purpose:

Array of integer values.

`IDoubleArray`

Purpose:

Array of double precision real numbers.

IDoubleMatrix

Purpose:

Matrix of double precision real numbers.

IComplexArray

Purpose:

Array of complex numbers.

IComplexMatrix

Purpose:

Matrix of complex numbers.

IStringArray

Purpose:

Array of strings.

Optical Signal

There are two representations of optical signal available: *block* and *sample*. Sample mode is represented with only one object – `IOptSample`. Block mode is represented by `IOptSignal`, `IOptSampledBand`, `IOptNoiseBin`, `IStokesVector`, `IOptParamSignal`, `ITrackingData`, `ITrackingDataRecord`.

IOptSampleCollection

Purpose:

Collection of optical samples.

Properties:

`Count`

Number of elements in the collection.

Methods:

`Item(index)`

Returns `IOptSample` object for sample with the specified index or throws an exception if the index goes beyond bounds.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

`FromArray(comp, array)`

Initializes an appropriate component of the sample using an array of double precision values. The component which should be initialized is defined by the value of the *comp* parameter. It can be one of scE, scEx, scEy for optical field or

X/Y polarization. This is the most effective approach to initialize a collection of samples.

Complex values are represented as double precision values, which go one after another.

`ToArray(comp)`

Returns an array of double precision values for specific component of the sample.

IOptSample

Purpose:

Represents either an independent optical sample for sample mode or one sample in a sampled band.

Properties:

`StartTime`

Timestamp of the sample. Available only for independent optical samples.

`Ex`

X-polarization of optical field. Not available for samples from constantly polarized sampled band.

`Ey`

Y-polarization of optical field. Not available for samples from constantly polarized sampled band.

`E`

Optical field of constantly polarized signal. Available only for samples from constantly polarized sampled bands.

IOptSignal

Purpose:

Represents an optical signal in block mode. An optical signal may contain multiple sampled bands, noise bins, and parameterized signals.

Properties:

`BoundaryConditions`

Boundary conditions of the signal. Can be either `bcPeriodic` or `bcAperiodic`.

`StartTime`

Time stamp (start time) of the signal. Makes sense only for signals in aperiodic boundary conditions.

`LowerFrequency`

Lower frequency limit of the signal.

`UpperFrequency`

Upper frequency limit of the signal.

`SampledBands`

Collection of sampled bands (`IOptSampledBand` objects).

`Noise`
: Collection of noise bins (`INoiseBin` objects).

`ParamSignals`
: Collection of parameterized signals (`IParamSignal` objects).

IOptSampledBandCollection

Purpose:

Collection of sampled bands (`ISampledBand` objects).

Methods:

`CreateBand(lowerFreq, upperFreq, startTime, isConstPolarized)`
: Creates a sampled band with the specified parameters. It should then be added to the collection of sampled bands if necessary.

IOptSampledBand

Purpose:

Represents an optical sampled band.

Properties:

`LowerFrequency`
: Lower frequency limit of the band.

`UpperFrequency`
: Upper frequency limit of the band.

`IsConstantlyPolarized`
: Returns true if the signal is constantly polarized.

`Azimuth`
: Azimuth of polarization ellipse in degrees.

`Ellipticity`
: Ellipticity of polarization ellipse in degrees.

`Samples`
: Returns a collection of optical samples (`IOptSampleCollection`).

`Labels`
: Signal labels.

Methods:

`AddLabels(index)`
: Adds a signal label.

IOptParamSignalCollection

Purpose:

Collection of parameterized signals (`IOptParamSignal` objects).

Methods:

`CreateParamSignal(type, lowerFreq, upperFreq, startTime, stokes)`

Creates a parameterized signal with the specified parameters. It should then be added to the collection of parameterized signals if necessary.

IOptParamSignal

Purpose:

Represents an optical parameterized signal.

Properties:

`Type`

Type of the parameterized signal. Can be one of `psParameterized`, `psDistortionGeneric`, `psDistortionRayleigh`, `psDistortionBrillouin`, `psDistortionFWM`, `psDistortionCrosstalk`.

`LowerFrequency`

Lower frequency limit of the parameterized signal.

`UpperFrequency`

Upper frequency limit of the parameterized signal.

`Stokes`

Stokes polarization parameters (not normalized).

`ExtinctionRatio`

Extinction ratio of the signal.

`AveragePulsePosition`

Average position of a pulse.

`PulsePositionVariance`

The variance of a pulse position.

`TrackingData`

Tracking characteristics of the signal.

`Labels`

Signal labels.

Methods:

`AddLabels(labels)`

Adds a signal labels.

IOptNoiseBinCollection

Purpose:

Collection of noise bins (`IOptNoiseBin` objects).

Methods:

`CreateNoiseBin(startTime, centerFreq, bandwidth, stokes)`

Creates a noise bin with the specified parameters. It should then be added to the noise collection if necessary.

`IOptNoiseBin`

Purpose:

Represents an optical noise bin.

Properties:

`LowerFrequency`

Lower frequency limit of the parameterized signal as quantized value.

`UpperFrequency`

Upper frequency limit of the parameterized signal as quantized value.

`Stokes`

Stokes polarization parameters (not normalized).

This property returns an unattached copy of the `IStokesVector` object; changes made to the copy are not propagated to the noise bin automatically. To apply changes, it is necessary to reassign this property with the changed value.

`IStokesVector`

Purpose:

Represents Stokes vector.

Properties:

`S0, S1, S2, S3`

Components of the Stokes vector.

Methods:

`ToArray()`

Returns a COM array with values of Stokes components.

`ITrackingData`

Purpose:

Collection of `ITrackingDataRecord` objects. Stores multiple tracking characteristics of the signal.

Properties:

`Count`

Number of tracking data records.

`Item(index)`

Gets or sets `ITrackingDataRecord` object at specified index.

Methods:

`GetLastTrackingDataRecord()`

Returns a copy of the last tracking data record.

`SetTrackingDataRecord(isDetailed, record)`

Depending on the value of the *`isDetailed`* parameter, this function sets a record as the only record in the tracking data or appends the specified record to the tracking data.

`ITrackingDataRecord`

Purpose:

Stores tracking characteristics of the signal.

Properties:

`Event`

The ordinal number of a module on the signal path.

`Entry`

Number of entry.

`PhysicalPath`

Total length of fiber the signal has passed.

`TransitTime`

Time of signal propagation through a link.

`TopologicalPosition`

Topological position of the signal.

`Path`

Array with IDs of the modules the signal has passed

`AccumulatedGVD`

Accumulated dispersion of the signal.

`Power`

Power of the optical signal.

`S1, S2, S3`

Components of non-normalized Stokes vector.

`NonlinearCoefficient`

Coefficient of fiber nonlinearity.

`AccumulatedDGD`

Accumulated differential group delay of the signal.

`SPM`

Phase changes of the optical signal due to self-phase modulation.

`Bo`

Optical bandwidths.

`Be`

Electrical bandwidths.

`NoisePSD`

Power spectral density of the optical noise at the signal wavelength.

`NoisePower`

Optical noise power in the optical bandwidth `Bo`.

`DistCohTotalPower`

Total power of the coherent distortions located in the electrical bandwidth `Be`.

`DistIncohTotalPower`

Total power of the incoherent distortions in the whole spectral range outside `Be`.

Electrical Signal

There are two representations of electrical signal available: *block* and *sample*. Sample mode is represented with only one object – `IElSample`. Block mode is represented by `IElSignal` and `IElSampledBand`. In contrast to optical signals, in block mode there is no noise and parameterized signals.

`IElSampleCollection`

Purpose:

Collection of electrical samples.

Properties:

`Count`

Number of elements in the collection.

Methods:

`Item(index)`

Returns `IElSample` object for samples with the specified index or throws an exception if the index goes beyond bounds.

This method is usually replaced with an appropriate collection indexing operation for the corresponding language.

`FromArray(array)`

Initializes samples using array of double precision values. This is the most effective approach to initialize a collection of samples.

`ToArray(comp)`

Returns an array of double precision values for samples.

`IElSample`

Purpose:

Represents one independent electrical sample or sample in sampled band.

Properties:

`StartTime`

Timestamp of the sample. Available only for independent samples.

`E`

Real valued sample of signal.

IElSignal

Purpose:

Represents an electrical signal in block mode.

Properties:

`BoundaryConditions`

Boundary conditions of the signal. Can be either `bcPeriodic` or `bcAperiodic`.

`StartTime`

Time stamp (start time) of signal.

`Bandwidth`

Total signal bandwidth.

`SampledBand`

Single sampled band of the electrical signal.

Methods:

`CreateSampledBand(sampleRate)`

Creates a sampled band with the specified sample rate and returns it. `SampledBand` property should be assigned to this value.

IElSampledBand

Purpose:

Represents an electrical sampled band.

Properties:

`SampleRate`

Sample rate of the band in Hz.

`Samples`

Returns a collection of samples (`IElSampleCollection`).

`Labels`

Signal labels.

Methods:

`AddLabels(labels)`

Adds signal labels.

Logical Information

The logical information interfaces can be used to read and change global logical information. They are: `ILogicalChannelCollection`, `ILogicalChannel`, `ILabelCollection`, `ILabel`, `IModulationCollection`, `IModulation`, `ISignalSourceCollection`, `ISignalSource`, `IPulseShapeCollection`, `IPulseShape`, `ILineCodingCollection`, `ILineCoding`.

`ILogicalChannelCollection` object can be acquired from the cosimulation context.

`ILogicalChannelCollection`

Purpose:

Collection of logical channels.

`ILogicalChannel`

Purpose:

Stores logical information for one logical channel.

Properties:

`Label`

Label string of the logical channel.

`SequenceNumber`

Number which is used to discriminate between logical signal blocks belonging to different time instants of one channel.

`ModulateeLabels`

Collection of `ILabel` objects which specify the logical channel representing the signal that was input as modulated signal to modulation of this logical channel (if present).

`SignalSources`

Returns a collection of `ISignalSource` objects which describe source of the signal.

`Modulations`

Returns a collection of `IModulation` objects which describe modulation type.

`PulseShapes`

Returns a collection of `IPulseShape` objects which describe shape of the pulse.

`BitStream`

Returns an array of integer numbers which represent a bitstream or `null` if no bitstream is assigned.

`LineCodings`

Returns a collection of `ILineCoding` objects which describe the employed line coding.

`ILabelCollection`

Purpose:

Collection of `ILabel` objects.

`ILabel`

Purpose:

Represents a signal label.

Properties:

`Label`

String label of the logical channel.

`SequenceNumbers`

Number which is used to discriminate between logical signal blocks belonging to different time instants of one channel.

`Info`

String which can be "modulatee" if the label is of modulatee channel, "modulator" if the label is of modulator channel, or empty "" otherwise.

IModulationCollection

Purpose:

Collection of `IModulation` objects.

IModulation

Purpose:

Stores information about modulation.

Properties:

`Type`

Type of modulation. Can be one of `mtPlain`, `mtASK`, `mtFSK`, `mtPSK`, `mtUser`.

`BitRate`

Bit rate of the modulated signal as quantized value.

`ModulationIndex`

Modulation index of the modulated signal (if `Type` is `mtASK`).

`FrequencyDeviation`

Frequency deviation of the modulated signal as frequency offset between spaces and marks in Hz (if `Type` is `mtFSK`).

`PhaseDeviation`

Phase deviation of the modulated signal as phase offset between spaces and marks in degrees (if `Type` is `mtPSK`).

`UserType`

Name of user-specified modulation (if `Type` is `mtUser`).

`UserParameters`

Array of arbitrary strings representing parameters for user-specified modulation type (if `Type` is `mtUser`).

ISignalSourceCollection

Purpose:

Collection of `ISignalSource` objects.

`ISignalSource`

Purpose:

Stores information about the signal source.

Properties:

`Name`

Source name (ID of source).

`StartTime`

Time stamp (start time) of signal source as quantized value.

`LowerFrequency`

Lower frequency limit of signal source as quantized value.

`UpperFrequency`

Upper frequency limit of signal source as quantized value.

`IPulseShapeCollection`

Purpose:

Collection of `IPulseShape` objects.

`IPulseShape`

Purpose:

Stores information about pulse shape.

Properties:

`Type`

Type of pulse shape. Can be one of `pstArbitrary`, `pstRectangular`, `pstStimulus`, `pstGauss`, `pstSech`, `pstRaisedCosine`, `pstRootRaisedCosine`.

`Bias`

Bias of the pulse.

`DutyCycle`

Duty cycle of the pulse (if `Type` is `pstRectangular`).

`Shift`

Shift of the pulse's high (if `Type` is `pstRectangular`).

`Amplitude`

Amplitude of the pulse (if `Type` is `pstRectangular`, `pstStimulus`, `pstGauss`, `pstSech`, `pstRaisedCosine`, `pstRootRaisedCosine`).

`CenterPos`

Center position of the pulse in seconds (if `Type` is `pstStimulus`, `pstGauss`, `pstSech`, `pstRaisedCosine`, `pstRootRaisedCosine`).

`FWHM`

Full width at half maximum of the pulse in seconds (if `Type` is `pstGauss` or `pstSech`).

`ExtraBits`

Maximum number of the neighboring bits from each side which are taken into account for pulse overlap (if `Type` is `pstGauss`, `pstSech`, `pstRaisedCosine`, `pstRootRaisedCosine`).

`Order`

Order of the Gauss pulse (if `Type` is `pstGauss`).

`Chirp`

Chirp of the pulse (if `Type` is `pstGauss`).

`PulseWidth`

Pulse width (if `Type` is `pstRaisedCosine` or `pstRootRaisedCosine`).

`Alpha`

Excess bandwidth of the pulse (if `Type` is `pstRaisedCosine` or `pstRootRaisedCosine`).

`PulseName`

Pulse name supplied by the user (if `Type` is `pstArbitrary`).

`Pulse`

Pulse supplied by the user (if `Type` is `pstArbitrary`).

`Duration`

Duration of the pulse held in the array as a quantized value. This is interpreted as bit duration (if `Type` is `pstArbitrary`).

ILineCodingCollection

Purpose:

Collection of `ILineCoding` objects.

ILineCoding

Purpose:

Stores information about the coding applied to the signal.

Properties:

`Type`

Type of line coding. Can be `lcManchester`, `lcUnipolar`, `lcBipolar` or `lcAMI`.

`LastBit`

Delay element's contents for differential Manchester coding (if `Type` is `lcManchester` or `lcAMI`).

`IsDifferential`

Indicates if differential Manchester coding is to be employed (if `Type` is `lcManchester`).

Using the COM Cosimulation Interface

A Simple Example: Adding Two Floating-Point Values

The simple adder example previously presented for MATLAB, Python and Library cosimulations requires a bit more effort to set up than the MATLAB or Python examples.

The first step is to build a COM cosimulation component. In this example it will be done using Microsoft Visual C# 2008. The type of project you should create is a **Visual C#\Windows\Class Library**. We will name the project **ComAdder**. This is not strictly necessary, but this name will be used as the name of the default namespace and as the DLL name, which is convenient for the example. The location where the project is saved is not important because the path to the COM component will be written to the Windows registry during registration.

After creating the project, select **Project > Add Reference** and add a reference to `VPI Cosim Library 11.5` from the **COM** tab in the resulting dialog. Then select **Project > ComAdder Properties**, click the **Build** tab and select the **Register for COM interop** checkbox to enable registration of the library for COM interoperability.

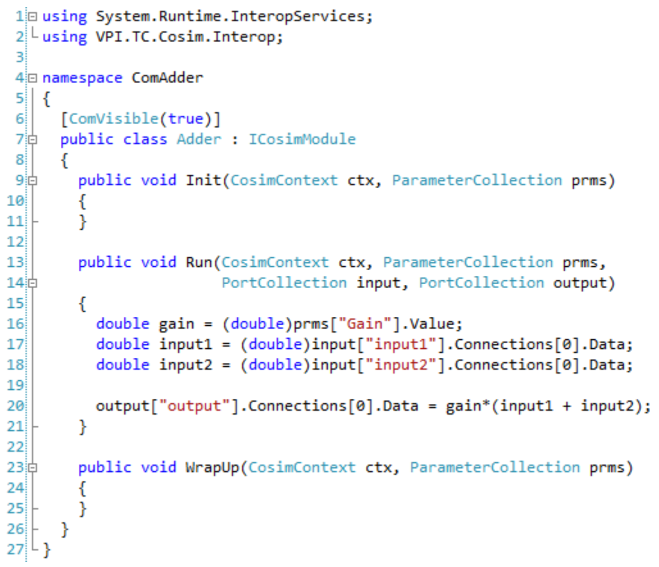
Note: On Windows Vista or Windows 7, Microsoft Visual Studio must be run with administrative privileges in order to add COM components to the registry. (Right-click the Microsoft Visual Studio 2008 shortcut and select **Run as administrator**.)

Another step is required to register the COM component for interoperability with 64-bit applications such as the simulation engine executable. In the project properties dialog, click the **Build Events** tab and add the following string to the **Post-build event command line** (including quotes):

```
"%Windir%\Microsoft.NET\Framework64\v2.0.50727\regasm" /codebase "$(TargetPath)"
```

After setting up the project, the next step is to set up the source code. Visual Studio will add a default class named `Class1` to the project. Rename it to `Adder`, then add a reference to the `VPI.TC.Cosim.Interop` namespace and implement the interface `ICosimModule`. Add the `ComVisible(true)` attribute to the `Adder` class to register it for COM interoperability.

A source code sample implementing the adder is shown in Figure 2-69.

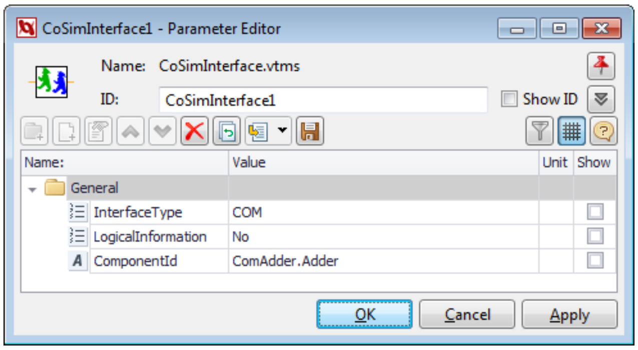


```
using System.Runtime.InteropServices;
using VPI.TC.Cosim.Interop;

namespace ComAdder
{
  [ComVisible(true)]
  public class Adder : ICosimModule
  {
    public void Init(CosimContext ctx, ParameterCollection prms)
    {
    }

    public void Run(CosimContext ctx, ParameterCollection prms,
                    PortCollection input, PortCollection output)
    {
      double gain = (double)prms["Gain"].Value;
      double input1 = (double)input["input1"].Connections[0].Data;
      double input2 = (double)input["input2"].Connections[0].Data;

      output["output"].Connections[0].Data = gain*(input1 + input2);
    }

    public void WrapUp(CosimContext ctx, ParameterCollection prms)
    {
    }
  }
}
```

Figure 2-69 *Main source code file for the adder example*

The implementation of the adder in the Run() function may seem complicated, but it is actually quite straightforward. The most verbose part of the code is access to parameters and ports. As opposed to MATLAB, Python and even Library cosimulations, there are no named objects to access them. Both parameters and ports are accessed by their names as keys in collections. Support for multiple-input ports adds an additional layer of complexity. Since C# is a strongly typed language, the parameter values and port data must be cast to the appropriate type during assignment.

The next step is to build the cosimulation galaxy that will implement the module. The galaxy is identical to that used in the MATLAB and Python adder examples (see Figure 2-6). The parameters of the *CoSimInterface* module in the *AdderCom* galaxy should be modified as shown in Figure 2-70.



CoSimInterface1 - Parameter Editor

Name: CoSimInterface.vtms

ID: CoSimInterface1

Name:	Value	Unit	Show
General			
InterfaceType	COM		☐
LogicalInformation	No		☐
ComponentId	ComAdder.Adder		☐

Figure 2-70 *CoSimInterface parameters COM cosimulation*

The **InterfaceType** parameter is set to COM and **ComponentId** to ComAdder.Adder. This ID is the ProgID of our COM component, which was formed from the names of the namespace and the class of our module.

Example 1 — Calculating the Total Optical Power of an Input Signal Using C#

The COM equivalent of the MATLAB power meter example presented in "Example 1 — Calculating the Total Optical Power of an Input Signal using MATLAB" on page 100 is shown in Figure 2-71. This setup is identical to the MATLAB setup shown in Figure 2-7, except for the settings of the parameters of the *CoSimInterface* module. In this case, the setting of the **InterfaceType** parameter is COM. Also, the **ComponentId** is set to VPI.TC.Examples.CS.PowerMeter.11.5, which is the ProgID of the COM component which implements the module code.

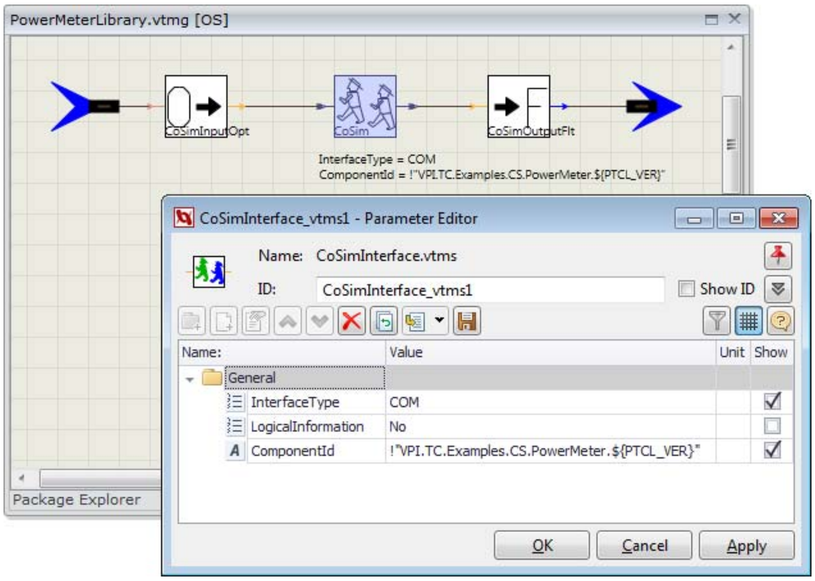
The complete definition of the power meter Run function in C# is shown in Figure 2-72. It is very similar in structure and content to the MATLAB and Python examples shown in Figure 2-8 and Figure 2-23. The main difference is that the inputs, outputs and parameters are accessed via the specialized COM interfaces that are passed into the function, whereas in Python and MATLAB these are available as variables automatically created within the target environment or galaxy-specific structure generated for the dynamic-link library. For example, input data is accessed through code like this:

```
OptSignal inputSignal = input["input"].Connections[0].Data as OptSignal;
```

Here, input is accessed by its name, then data is received from the first connection to the input port (which is the only one in this example). Since C# is a strongly typed language, the `Data` type property must be converted from the generic `Object` to the appropriate type.

Errors can be reported using the standard exception handling mechanism:

```
if (null==inputSignal)
      throw new Exception("Optical signal expected.");
```



InterfaceType = COM
ComponentId = !"VPI.TC.Examples.CS.PowerMeter.${PTCL_VER}"

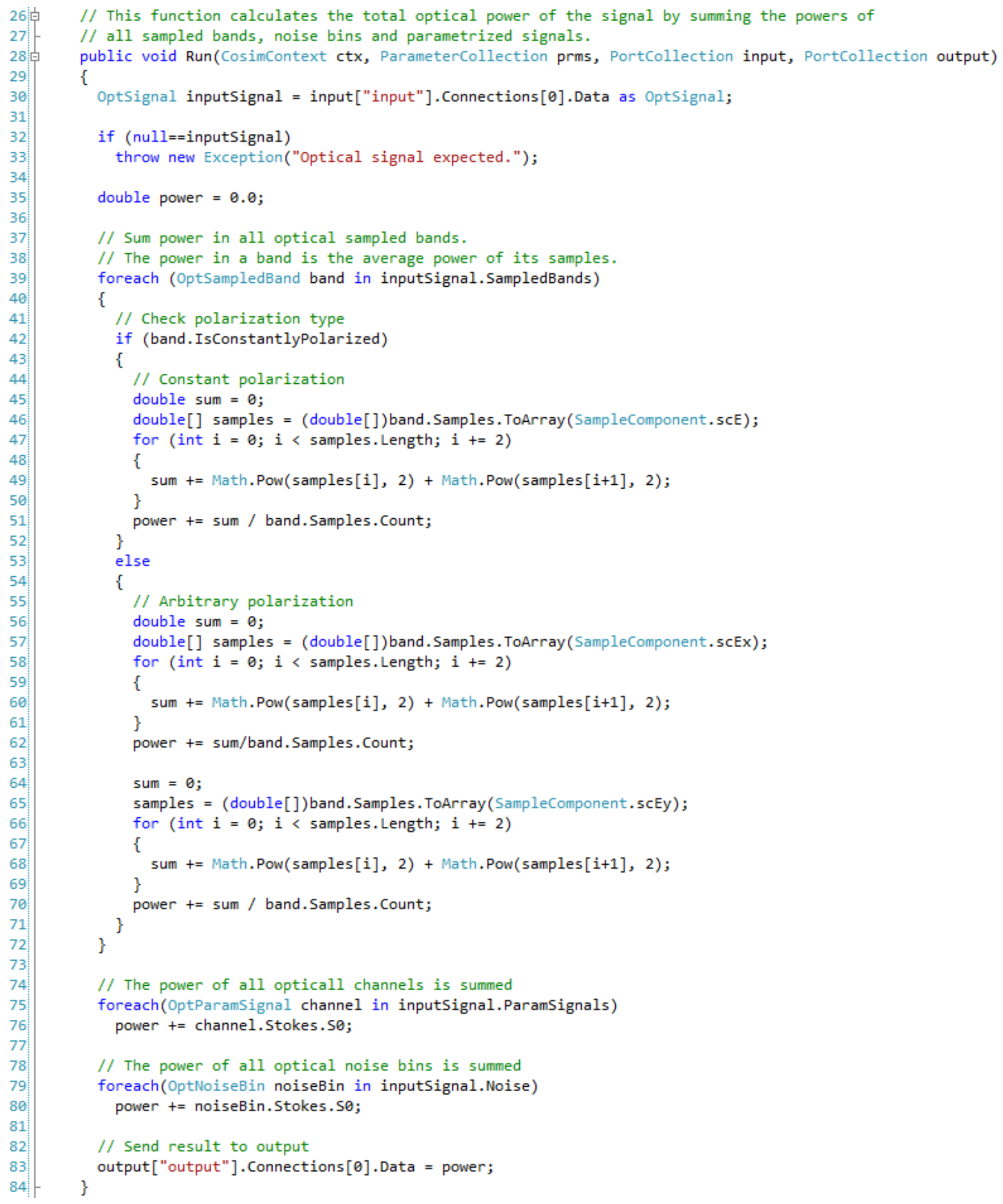
Name: CoSimInterface.vtms
ID: CoSimInterface_vtms1

Name:	Value	Unit	Show
General			
InterfaceType	COM		✓
LogicalInformation	No		☐
ComponentId	!"VPI.TC.Examples.CS.PowerMeter.${PTCL_VER}"		✓

Figure 2-71 *Example of the cosimulation interface galaxy for implementing a power meter as a COM component. The parameter settings of the CoSimInterface are shown.*

COM cosimulation code uses a different mechanism to determine whether a sampled band represents a signal with constant polarization or arbitrary polarization. As opposed to other cosimulation types, there is a special boolean property provided in the sampled band object – `IsConstantlyPolarized`. When this property is `false`, the code may access both X and Y polarizations. Otherwise, only the optical field of a constantly polarized signal may be accessed.

The complete source code for the power meter COM component can be found in the `simeng\cosim\examples\com\PowerMeter` subfolder of the VPI Design Suite installation directory.



```
// This function calculates the total optical power of the signal by summing the powers of
// all sampled bands, noise bins and parametrized signals.
public void Run(CosimContext ctx, ParameterCollection prms, PortCollection input, PortCollection output)
{
  OptSignal inputSignal = input["input"].Connections[0].Data as OptSignal;

  if (null==inputSignal)
    throw new Exception("Optical signal expected.");

  double power = 0.0;

  // Sum power in all optical sampled bands.
  // The power in a band is the average power of its samples.
  foreach (OptSampledBand band in inputSignal.SampledBands)
  {
    // Check polarization type
    if (band.IsConstantlyPolarized)
    {
      // Constant polarization
      double sum = 0;
      double[] samples = (double[])band.Samples.ToArray(SampleComponent.scE);
      for (int i = 0; i < samples.Length; i += 2)
      {
        sum += Math.Pow(samples[i], 2) + Math.Pow(samples[i+1], 2);
      }
      power += sum / band.Samples.Count;
    }
    else
    {
      // Arbitrary polarization
      double sum = 0;
      double[] samples = (double[])band.Samples.ToArray(SampleComponent.scEx);
      for (int i = 0; i < samples.Length; i += 2)
      {
        sum += Math.Pow(samples[i], 2) + Math.Pow(samples[i+1], 2);
      }
      power += sum/band.Samples.Count;

      sum = 0;
      samples = (double[])band.Samples.ToArray(SampleComponent.scEy);
      for (int i = 0; i < samples.Length; i += 2)
      {
        sum += Math.Pow(samples[i], 2) + Math.Pow(samples[i+1], 2);
      }
      power += sum / band.Samples.Count;
    }
  }

  // The power of all opticall channels is summed
  foreach(OptParamSignal channel in inputSignal.ParamSignals)
    power += channel.Stokes.S0;

  // The power of all optical noise bins is summed
  foreach(OptNoiseBin noiseBin in inputSignal.Noise)
    power += noiseBin.Stokes.S0;

  // Send result to output
  output["output"].Connections[0].Data = power;
}
```

Figure 2-72 *The PowerMeter "Run" function in the C# source PowerMeter.cs. It calculates the total power of all used optical bands, channels and noise.*

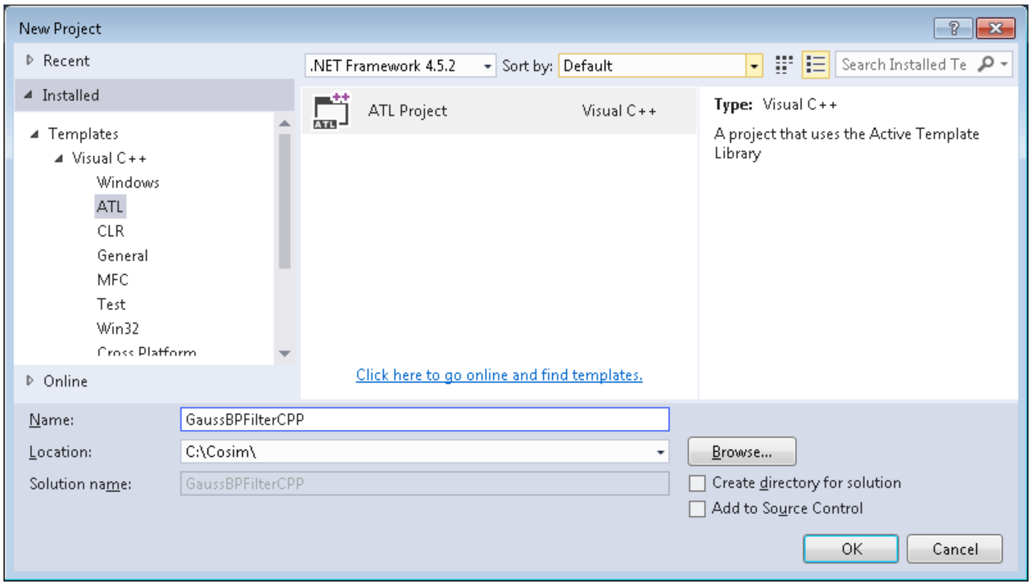
Example 2 — Implementing an Optical Bandpass Filter Using C++

The demonstration *Filter (COM)* under *Optical Systems Demos > Simulation Techniques > Cosimulation > COM is equivalent to the* MATLAB example described in "Example 2 — Implementing an Optical Bandpass Filter Using MATLAB" on page 103. This demonstration implements a Gaussian bandpass filter for optical signals as a COM component using C++ code and Microsoft's ATL library. The setup of the optical filter demonstration, and the corresponding cosimulation galaxy, are identical to those of the MATLAB example, and they are not repeated here.

Writing a COM component in pure C++ may seem like a daunting task, but Microsoft Visual Studio greatly simplifies it by providing different libraries and native compiler support specifically for this task.

The source code structure of the filter example is very similar to that introduced in the Library example, but with a number of new features and concepts, which will be described in this section. You will find the complete source code for the DLL in the `simeng\cosim\examples\com\GaussBPFilterCPP` subdirectory of the VPI Design Suite installation directory.

To create the project for this example, click **File > New > Project**.


New Project

.NET Framework 4.5.2 | Sort by: Default | Search Installed Te

Recent
Installed
Templates
Visual C++
Windows
ATL
CLR
General
MFC
Test
Win32
Cross Platform
Online

ATL Project Visual C++

Type: Visual C++

A project that uses the Active Template Library

Click here to go online and find templates.

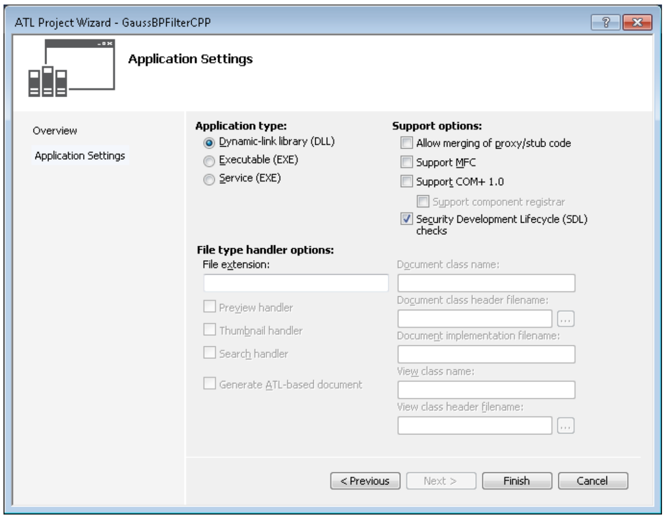
Name: GaussBPFilterCPP
Location: C:\Cosim\
Solution name: GaussBPFilterCPP

Browse...
Create directory for solution
Add to Source Control
OK
Cancel

Figure 2-73 *Creating new project*

In the **Add New Project** window shown in Figure 2-73, select **Templates\Visual C++\ATL\ATL Project**. Specify a **Name** and **Location** and click **OK**.

The ATL Project Wizard shown in Figure 2-74 appears.



Application type:
- [✓ Dynamic-link library (DLL)]
- Executable (EXE)
- Service (EXE)

Support options:
- ☐ Allow merging of proxy/stub code
- ☐ Support MFC
- ☐ Support COM+ 1.0
  - ☐ Support component registrar
- ☑ Security Development Lifecycle (SDL) checks

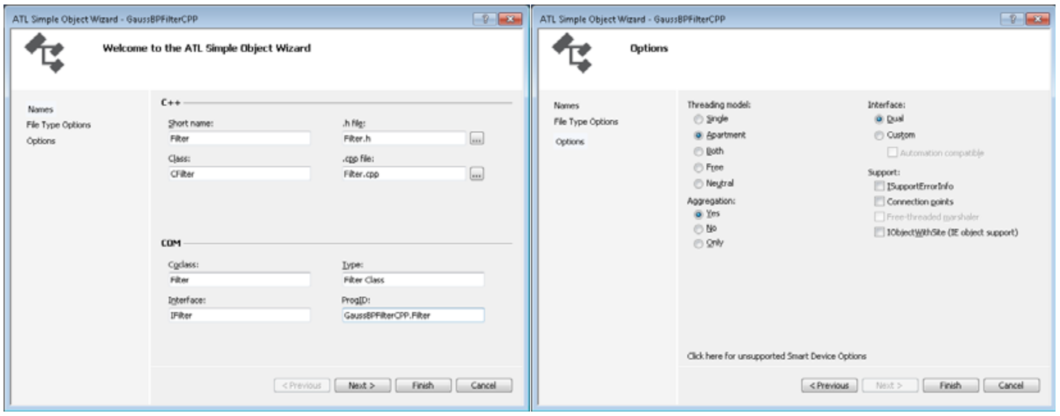
File type handler options:
File extension:
- ☐ Preview handler
- ☐ Thumbnail handler
- ☐ Search handler
- ☐ Generate ATL-based document

Document class name:
Document class header filename:
Document implementation filename:
View class name:
View class header filename:

Figure 2-74 *ATL Project Wizard*

The default application settings can be left as is. Click Finish.

Next, add a COM component class via Project > Add Class. Select Visual C++\ATL\ATL Simple Object in the resulting dialog. The ATL Simple Object Wizard appears as shown in Figure 2-75.


ATL Simple Object Wizard - GaussBPFilterCPP
Welcome to the ATL Simple Object Wizard
Names
File Type Options
Options
< Previous
Next >
Finish
Cancel
ATL Simple Object Wizard - GaussBPFilterCPP
Options
Names
File Type Options
Options
Click here for unsupported Smart Device Options
< Previous
Next >
Finish
Cancel

C++
Short name: Filter
.h file: Filter.h
Class: CFilter
.cpp file: Filter.cpp

COM
Coclass: Filter
Type: Filter Class
Interface: IFilter
ProgID: GaussBPFilterCPP.Filter

Threading model:
- Single
- [✓ Apartment]
- Both
- Free
- Neutral

Aggregation:
- [✓ Yes]
- No
- Only

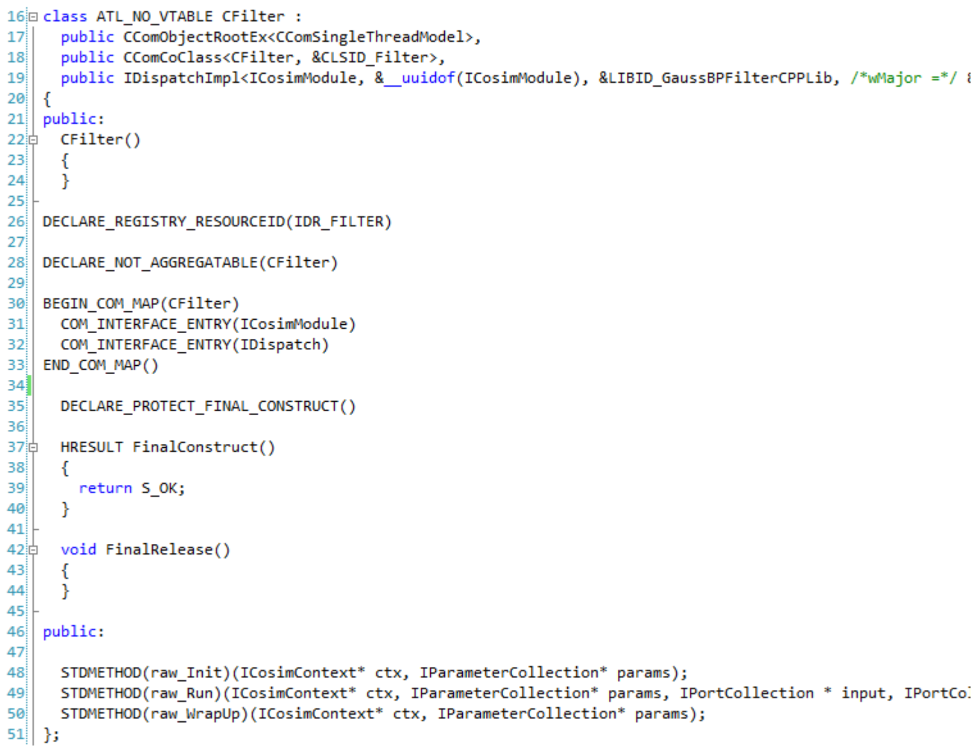
Interface:
- [✓ Dual]
- Custom
  - ☐ Automation compatible

Support:
- ☐ ISupportErrorInfo
- ☐ Connection points
- ☐ Free-threaded marshaler
- ☐ IObjectWithSite (IE object support)

Figure 2-75 *ATL Simple Object Project Wizard*

On the first page, specify the name of the COM component and a ProgID. The default options on the second page may be left as is.

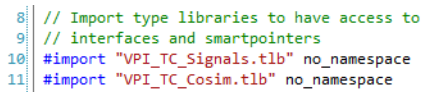
For the example shown in Figure 2-75, both the coclass `Filter` and the interface `IFilter` will be created in the interface definition file (`.idl`), which defines the interfaces of our COM component. But the component must implement the `ICosimModule` interface provided in the type libraries installed with VPI Design Suite. The default interface must be replaced with the `ICosimModule`; the old one may be deleted. The final version of the header file is shown in Figure 2-76.



```
class ATL_NO_VTABLE CFilter :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CFilter, &CLSID_Filter>,
  public IDispatchImpl<ICosimModule, &__uuidof(ICosimModule), &LIBID_GaussBPFilterCPPLib, /*wMajor =*/
{
public:
  CFilter()
  {
  }

DECLARE_REGISTRY_RESOURCEID(IDR_FILTER)

DECLARE_NOT_AGGREGATABLE(CFilter)

BEGIN_COM_MAP(CFilter)
  COM_INTERFACE_ENTRY(ICosimModule)
  COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

  DECLARE_PROTECT_FINAL_CONSTRUCT()

  HRESULT FinalConstruct()
  {
    return S_OK;
  }

  void FinalRelease()
  {
  }

public:

  STDMETHOD(raw_Init)(ICosimContext* ctx, IParameterCollection* params);
  STDMETHOD(raw_Run)(ICosimContext* ctx, IParameterCollection* params, IPortCollection * input, IPortCo
  STDMETHOD(raw_WrapUp)(ICosimContext* ctx, IParameterCollection* params);
};
```

Figure 2-76 *Header file for filter COM component*

To access the `ICosimModule` interface and other cosimulation and signal interfaces, the aforementioned type libraries (`.tlb` files) must be imported into your source code. This can be done with an `#import` directive as shown in Figure 2-77.

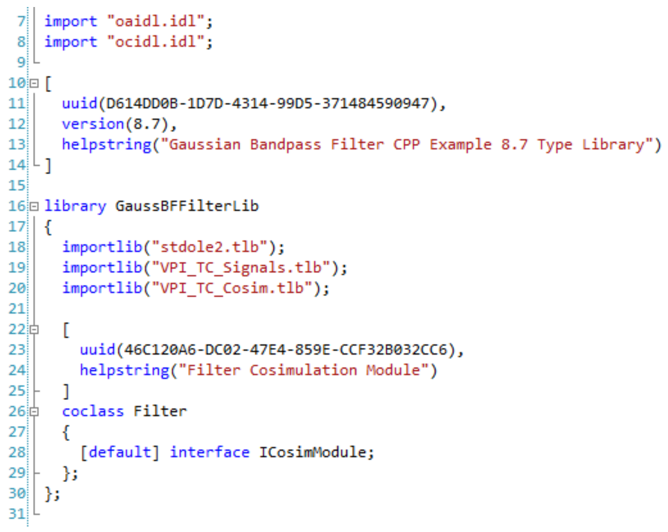


```
// Import type libraries to have access to
// interfaces and smartpointers
#import "VPI_TC_Signals.tlb" no_namespace
#import "VPI_TC_Cosim.tlb" no_namespace
```

Figure 2-77 *Importing type libraries*

The necessary .tlb files can be found in the `simeng\cosim\lib\win` subfolder of the VPI Design Suite installation directory.

The same .tlb files must be imported in the .idl file using the `importlib` directive. The final .idl file is shown in Figure 2-78.



```
import "oaidl.idl";
import "ocidl.idl";

[
  uuid(D614DD0B-1D7D-4314-99D5-371484590947),
  version(8.7),
  helpstring("Gaussian Bandpass Filter CPP Example 8.7 Type Library")
]

library GaussBFFilterLib
{
  importlib("stdole2.tlb");
  importlib("VPI_TC_Signals.tlb");
  importlib("VPI_TC_Cosim.tlb");

  [
    uuid(46C120A6-DC02-47E4-859E-CCF32B032CC6),
    helpstring("Filter Cosimulation Module")
  ]
  coclass Filter
  {
    [default] interface ICosimModule;
  };
};
```

Figure 2-78 *Final interface definition (.idl) file*

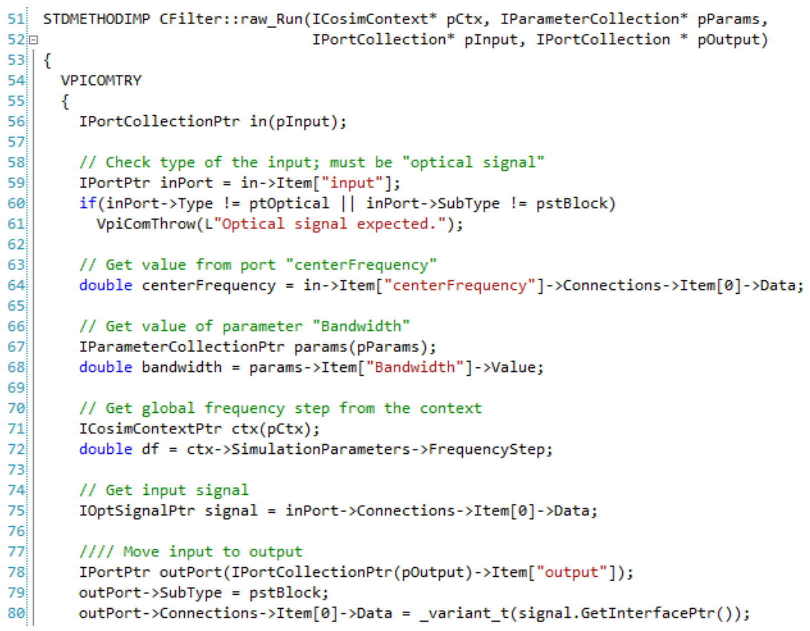
Like the dynamic-link library filter example, the COM filter component uses the MIT FFTW library for the fast Fourier transform (FFT) to filter the input sampled bands in the frequency domain. The VPI Design Suite installation includes this library and in order to access the functions in it, the `fftw3.h` header file must be included. This header file can be found in the `simeng\cosim\include` subfolder, so this path must be added to the list of included directories in the project properties.

The start of the `Run` function is shown in Figure 2-79. The first function checks that the received signal is of the expected type. Special error handling macros and a custom exception class are used to simplify error handling. Their implementation is provided in the `ComErrorHandling.h/ComErrorHandling.cpp` files. These helpers enable the C++ exception handling mechanism instead of the more complex and error-prone standard COM mechanism of error codes and the `IErrorInfo` interface.

Lines 64–75 illustrate how to access parameters and data at module input ports:

```
// Get value from port "centerFrequency"
double centerFrequency = in->Item["centerFrequency"]->Connections->Item[0]->Data;
```

This use of properties and collections is supplied entirely by the special classes generated via the `#import` directive.



```
51 STDMETHODIMP CFilter::raw_Run(ICosimContext* pCtx, IParameterCollection* pParams,
52                              IPortCollection* pInput, IPortCollection * pOutput)
53 {
54   VPICOMTRY
55   {
56     IPortCollectionPtr in(pInput);
57 
58     // Check type of the input; must be "optical signal"
59     IPortPtr inPort = in->Item["input"];
60     if(inPort->Type != ptOptical || inPort->SubType != pstBlock)
61       VpiComThrow(L"Optical signal expected.");
62 
63     // Get value from port "centerFrequency"
64     double centerFrequency = in->Item["centerFrequency"]->Connections->Item[0]->Data;
65 
66     // Get value of parameter "Bandwidth"
67     IParameterCollectionPtr params(pParams);
68     double bandwidth = params->Item["Bandwidth"]->Value;
69 
70     // Get global frequency step from the context
71     ICosimContextPtr ctx(pCtx);
72     double df = ctx->SimulationParameters->FrequencyStep;
73 
74     // Get input signal
75     IOptSignalPtr signal = inPort->Connections->Item[0]->Data;
76 
77     //// Move input to output
78     IPortPtr outPort(IPortCollectionPtr(pOutput)->Item["output"]);
79     outPort->SubType = pstBlock;
80     outPort->Connections->Item[0]->Data = _variant_t(signal.GetInterfacePtr());
```

Figure 2-79 *Checking the signal type, and using the input signal as the output*

Note: Line 51 shows the function name `raw_Run`. The original method name in the `ICosimModule` interface is `Run`, but the `#import` directive renames this and other methods in the interface.

Lines 78–80 illustrate a useful technique which can be employed in COM cosimulation modules to copy a signal object from input to output. To understand how and why it works, it is important to understand how pointers to COM objects work and how the COM cosimulation interface treats objects passed in and out of the COM component.

COM uses reference counting to control the lifetime of objects. So when pointers are used, additional actions are required: the reference counters must be increased and decreased. COM can hold several pointers to one object but requires that the reference counter be equal to the number of those pointer copies. As long as at least one reference is alive, the object is also alive. After all references are released, the object destroys itself.

Reference counting is usually quite tedious, but fortunately the `#import` directive generates special classes, called smart pointers, which take responsibility for reference counting. These classes can be distinguished in the source code of examples by the suffix `Ptr`. For example, the `IPortPtr` smart pointer corresponds to the `IPort` interface. It is highly recommended to use smart pointers instead of raw pointers to COM interfaces.

Before calling the `Run` function, the cosimulation interface creates the input objects and initializes them with the corresponding data. Output objects are not created and the data pointer is initialized with a `null` value.

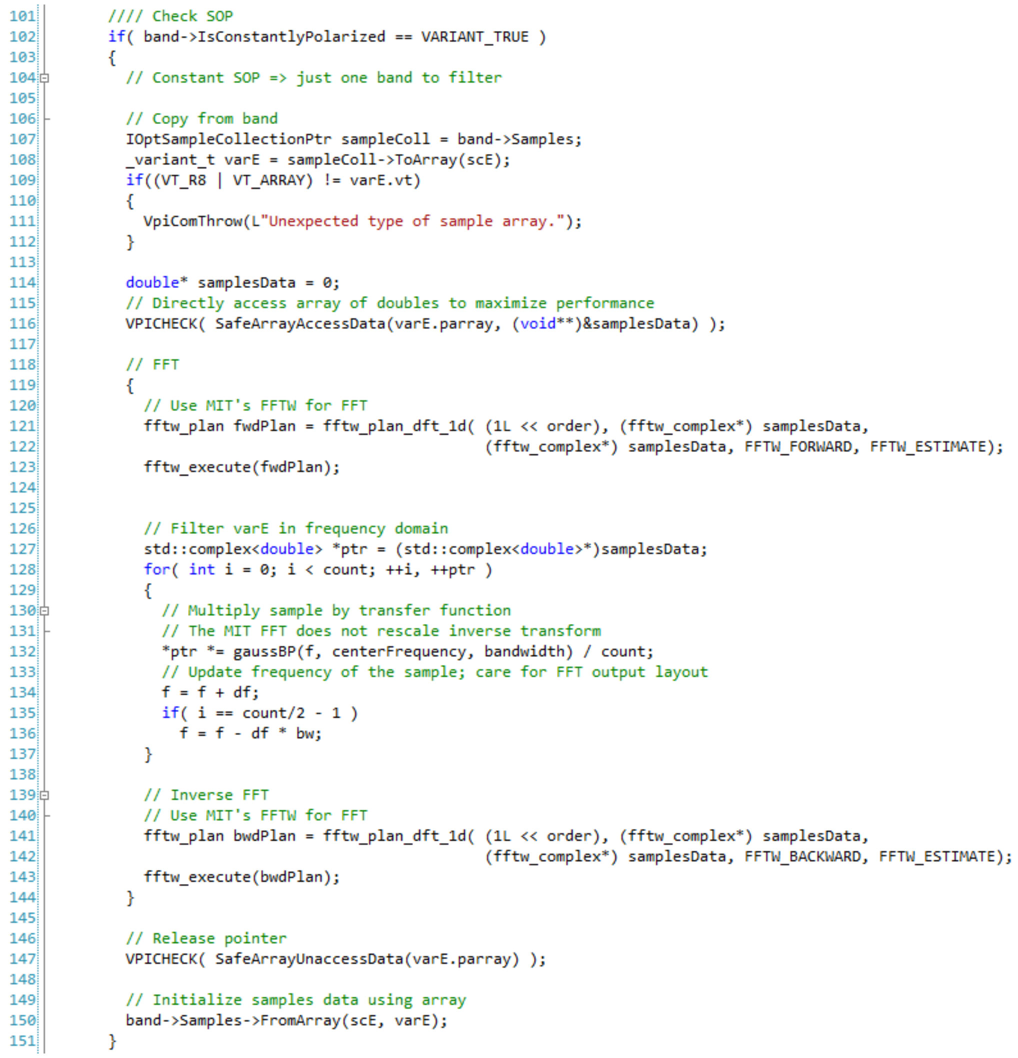
After the Run function returns, and if the cosimulation module has set an output pointer with some value, the cosimulation interface copies all of the data held in the output object into corresponding objects within the simulator, so that simulation can continue with the outputs being passed onto the following module.

This allows references to signal objects to be copied from input to output and perform changes in place. So although the code is written in C++, its behavior is more similar to that which is typical for Python cosimulation than that of library cosimulation.

Figure 2-53 shows the C++ code that is used to filter a single sampled band that has a constant state of polarization. The IsConstantlyPolarized property of the sampled band object is used to check whether the sampled band is constantly polarized. Aside from the different operations required to use the FFTW library, this code is very similar in structure to the dynamic-link library example discussed previously. The main point to note from this sample is that it is not possible to perform FFT in-place. So data is copied from the sampled band object to an array of double precision numbers and copied back after processing is complete. This is done using the ToArray() function of the IOptSampleCollection object on line 108.

A parameter is passed to this function specifying which component is required (Ex, Ey or E). This function returns complex numbers as an array of double precision numbers where the real and imaginary parts go one after another. This array is compatible with an array of complex<double> objects. Backward conversion is performed by the FromArray() function on line 150.

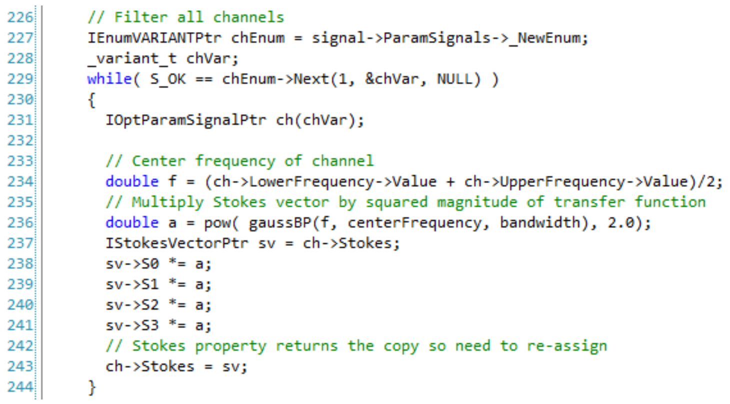
Also note that the FFTW library defines its own complex data type (FFTW_COMPLEX) and this structure is compatible with the complex<double> type. Thus, it is safe to cast pointers to one complex data type to pointers of a different complex data type, that is done on lines 121, 122, 127, 141 and 142 of the sample code.



```
//// Check SOP
if( band->IsConstantlyPolarized == VARIANT_TRUE )
{
  // Constant SOP => just one band to filter

  // Copy from band
  IOptSampleCollectionPtr sampleColl = band->Samples;
  _variant_t varE = sampleColl->ToArray(scE);
  if((VT_R8 | VT_ARRAY) != varE.vt)
  {
    VpiComThrow(L"Unexpected type of sample array.");
  }

  double* samplesData = 0;
  // Directly access array of doubles to maximize performance
  VPICHECK( SafeArrayAccessData(varE.parray, (void**)&samplesData) );

  // FFT
  {
    // Use MIT's FFTW for FFT
    fftw_plan fwdPlan = fftw_plan_dft_1d( (1L << order), (fftw_complex*) samplesData,
                                          (fftw_complex*) samplesData, FFTW_FORWARD, FFTW_ESTIMATE);
    fftw_execute(fwdPlan);


    // Filter varE in frequency domain
    std::complex<double> *ptr = (std::complex<double>*)samplesData;
    for( int i = 0; i < count; ++i, ++ptr )
    {
      // Multiply sample by transfer function
      // The MIT FFT does not rescale inverse transform
      *ptr *= gaussBP(f, centerFrequency, bandwidth) / count;
      // Update frequency of the sample; care for FFT output layout
      f = f + df;
      if( i == count/2 - 1 )
        f = f - df * bw;
    }

    // Inverse FFT
    // Use MIT's FFTW for FFT
    fftw_plan bwdPlan = fftw_plan_dft_1d( (1L << order), (fftw_complex*) samplesData,
                                          (fftw_complex*) samplesData, FFTW_BACKWARD, FFTW_ESTIMATE);
    fftw_execute(bwdPlan);
  }

  // Release pointer
  VPICHECK( SafeArrayUnaccessData(varE.parray) );

  // Initialize samples data using array
  band->Samples->FromArray(scE, varE);
}
```

Figure 2-80 *C++ code that filters a single sampled band with constant state of polarization*

One more important peculiarity of this example is the use of properties which return the `IStokesVector` object. Those properties return an unattached copy of internal data, so when values of the `IStokesVector` object are changed, the new values are not reflected in the source object. To overcome this problem, the entire object must be reassigned to the initial property. This is illustrated in Figure 2-81 in lines 237–243.



```
    // Filter all channels
    IEnumVARIANTPtr chEnum = signal->ParamSignals->_NewEnum;
    _variant_t chVar;
    while( S_OK == chEnum->Next(1, &chVar, NULL) )
    {
      IOptParamSignalPtr ch(chVar);

      // Center frequency of channel
      double f = (ch->LowerFrequency->Value + ch->UpperFrequency->Value)/2;
      // Multiply Stokes vector by squared magnitude of transfer function
      double a = pow( gaussBP(f, centerFrequency, bandwidth), 2.0);
      IStokesVectorPtr sv = ch->Stokes;
      sv->S0 *= a;
      sv->S1 *= a;
      sv->S2 *= a;
      sv->S3 *= a;
      // Stokes property returns the copy so need to re-assign
      ch->Stokes = sv;
    }
```

Figure 2-81 *C++ code that filters all noise bins*

The `IComplex` and `IQuantizedValue` share similar semantics.

The remainder of the code in the filter example is quite straightforward, and does not introduce any new features or techniques beyond those discussed above, and previously in the power meter example.

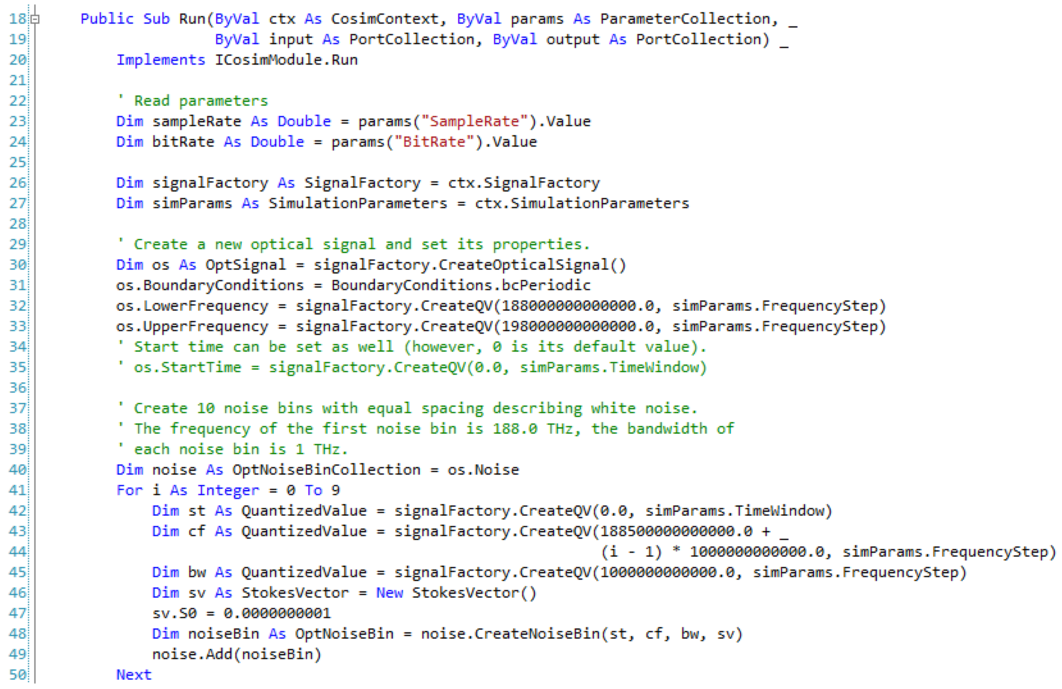
Example 3 — Creating New ONSL Signals using VB.NET

All objects which are used in COM cosimulation are specific and strongly typed. This is probably the most significant benefit of COM cosimulation over MATLAB or Python-based methods, for example. There is no need to refer to documentation or guess which properties a particular object contains. Most modern IDEs provide autocompletion features that suggest properties as you type.

Similar to the other cosimulation interfaces, ONSL signal objects can also be created from scratch in COM. The *Optical Signal Generation (COM VB.NET)* demonstration under *Optical Systems Demos > Simulation Techniques > Cosimulation > COM provides an example of how to create optical signals from scratch in* VB.NET*. The* VB.NET *source file can be found in* the `simeng\cosim\examples\com\CreateOpticalSignalVB` subfolder of the VPI Design Suite installation directory.

However, in contrast to all other cosimulation interfaces, signal objects are not created using the native object instantiation operations of some particular language (such as the `New` operator in VB.NET), but rather using methods provided by a special factory object from the cosimulation context object and similar factory methods from the signal objects themselves.

Figure 2-82 shows examples of signal factory usage on lines 30, 32 and 33. Line 48 shows an example of a factory method in a collection. Factory methods usually take enough parameters to create a fully initialized object, or at least in a consistent state. After the object is created, it should be assigned to some property or added to the collection.



```
    Public Sub Run(ByVal ctx As CosimContext, ByVal params As ParameterCollection, _
                   ByVal input As PortCollection, ByVal output As PortCollection) _
        Implements ICosimModule.Run

        ' Read parameters
        Dim sampleRate As Double = params("SampleRate").Value
        Dim bitRate As Double = params("BitRate").Value

        Dim signalFactory As SignalFactory = ctx.SignalFactory
        Dim simParams As SimulationParameters = ctx.SimulationParameters

        ' Create a new optical signal and set its properties.
        Dim os As OptSignal = signalFactory.CreateOpticalSignal()
        os.BoundaryConditions = BoundaryConditions.bcPeriodic
        os.LowerFrequency = signalFactory.CreateQV(188000000000000.0, simParams.FrequencyStep)
        os.UpperFrequency = signalFactory.CreateQV(198000000000000.0, simParams.FrequencyStep)
        ' Start time can be set as well (however, 0 is its default value).
        ' os.StartTime = signalFactory.CreateQV(0.0, simParams.TimeWindow)

        ' Create 10 noise bins with equal spacing describing white noise.
        ' The frequency of the first noise bin is 188.0 THz, the bandwidth of
        ' each noise bin is 1 THz.
        Dim noise As OptNoiseBinCollection = os.Noise
        For i As Integer = 0 To 9
            Dim st As QuantizedValue = signalFactory.CreateQV(0.0, simParams.TimeWindow)
            Dim cf As QuantizedValue = signalFactory.CreateQV(188500000000000.0 + _
                                                   (i - 1) * 1000000000000.0, simParams.FrequencyStep)
            Dim bw As QuantizedValue = signalFactory.CreateQV(1000000000000.0, simParams.FrequencyStep)
            Dim sv As StokesVector = New StokesVector()
            sv.S0 = 0.0000000001
            Dim noiseBin As OptNoiseBin = noise.CreateNoiseBin(st, cf, bw, sv)
            noise.Add(noiseBin)
        Next
```

Figure 2-82 *VB.NET code that creates sampled band*

A signal object is created with properties initialized to default values, so may be necessary to set them properly. For example, an optical signal should be initialized with its boundary conditions as lower and upper frequency limits.

Most of the frequency and time properties are quantized values (`IQuantizedValue` objects), and should thus be specified using some predefined grid. For example, all frequency values should use the global simulation frequency step as the grid unit. This frequency step (as well as several other useful variables) can be accessed from the cosimulation context object, which is sent to all cosimulation module methods as the `ICosimContext` interface.

Then it is necessary to create, initialize and add subordinate components to the appropriate collections. These components include sampled bands, parameterized signals and noise bins.

Once the signal object is created and initialized, it must be sent to a specific connection on an output port. For the current example, this is achieved with the following statement:

```
output("output").Connections(0).Data = os
```

Here the signal is sent to the first connection of the port with the name "output".

Example 4 — Saving an ONSL Signal to File for Postprocessing

ONSL signals often need to be stored for postprocessing by external tools. While several methods are available, they are not all equally efficient.

The *WriteToFile* module can write an ONSL signal in binary format, but this format is very complicated because it has to support different versions of VPI Design Suite and it is impractical to recommend it for usage.

Data can also be written as a simple text file using a combination of modules such as *UnpackBlockOpt*, *CxToRect* and *WriteDataFile*. But this method is dramatically slow and resulting files will be at least twice as large as those stored in the binary format.

If it is not necessary to ensure backward compatibility and support all possible features of the ONSL signal, it is easy to store the signal to a binary file via cosimulation and later parse the result in some external program.

The *Optical Signal Saving (COM)* demonstration under *Optical Systems Demos > Simulation Techniques > Cosimulation > COM provides an example of how to save optical signals to file and then read them with an external program for postprocessing. The* C# *source files can be found in* the `simeng\cosim\examples\com\SaveOpticalSignal` subfolder of the VPI Design Suite installation directory.

This setup consists of two parts implemented as COM cosimulation modules as shown in Figure 2-83. The first module writes an ONSL signal to a file in some simple binary format and the second module reads this file and displays it as a table.

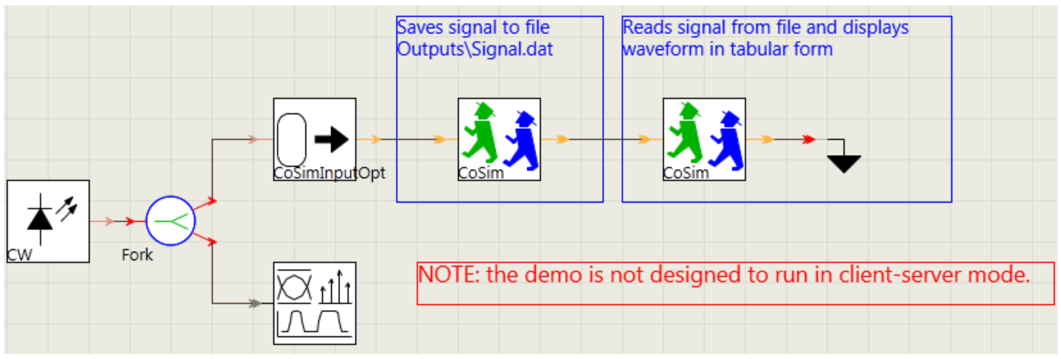
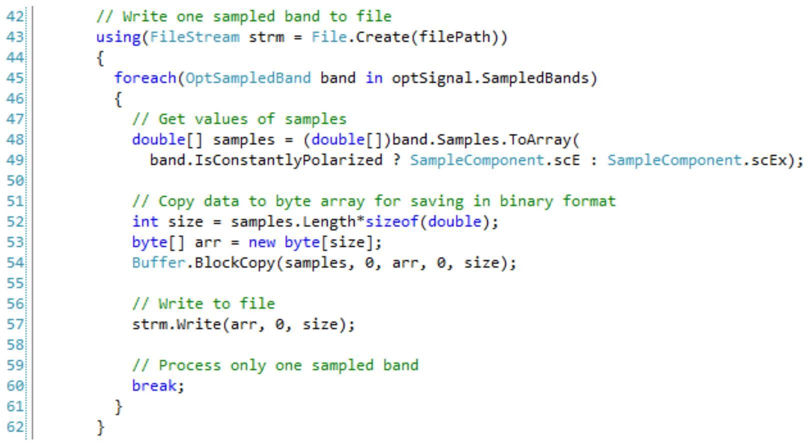

Saves signal to file
Outputs\Signal.dat
Reads signal from file and displays
waveform in tabular form
CoSimInputOpt
CoSim
CoSim
CW
Fork
NOTE: the demo is not designed to run in client-server mode.

Figure 2-83 *Schematic for the signal saving demonstration*

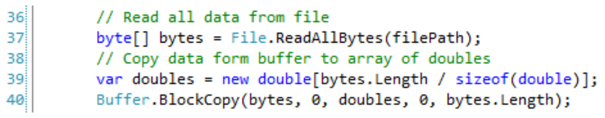
The code used to write the signal to a file is shown in Figure 2-84. It is specific for .NET/C# and the same task can be done differently in another language. For example, in C# there is no effective method to convert an array of double values to the byte array necessary for writing in binary format without copying data. A special utility class `Buffer` is used to perform this conversion.



```
// Write one sampled band to file
using(FileStream strm = File.Create(filePath))
{
  foreach(OptSampledBand band in optSignal.SampledBands)
  {
    // Get values of samples
    double[] samples = (double[])band.Samples.ToArray(
      band.IsConstantlyPolarized ? SampleComponent.scE : SampleComponent.scEx);

    // Copy data to byte array for saving in binary format
    int size = samples.Length*sizeof(double);
    byte[] arr = new byte[size];
    Buffer.BlockCopy(samples, 0, arr, 0, size);

    // Write to file
    strm.Write(arr, 0, size);

    // Process only one sampled band
    break;
  }
}
```

Figure 2-84 *C# code which writes ONSL signal in to file in binary format*

After the file is written, a second module reads and displays it. The code of this module is shown in Figure 2-85. It simply reads a block of bytes and converts them to double values.



```
// Read all data from file
byte[] bytes = File.ReadAllBytes(filePath);
// Copy data form buffer to array of doubles
var doubles = new double[bytes.Length / sizeof(double)];
Buffer.BlockCopy(bytes, 0, doubles, 0, bytes.Length);
```

Figure 2-85 *C# code which reads ONSL signal from the file*

The rest of the module code is specific for the approach used to visualize data and not directly related to the purpose of the demonstration.

This second part is organized as a module just for demonstration purposes. In reality it can be a completely separate program which may read the file produced by the first module when the simulation is finished.

References

[1] H. P. Langtangen, "*Python Scripting for Computational Science*" Springer

Chapter 3

Simulation Scripts

Simulation scripts are used to drive the simulation engine. They are based on the *Tcl* language[1] although they may contain parts written in Python.

A typical simulation script consists of two parts. The first part is generated automatically based on the current schematic content, and the second one depends on the simulation mode: a regular simulation or an interactive simulation (also known as a sweep).

Interactive simulations are driven by some predefined scripts that depend on the mode of the interactive simulation. For regular simulations, the second part either consists of only a single call of the `run` command or executes a *custom* simulation script if it exists. The `run` command just executes the simulation. However, with custom simulation scripts, you can automate far more complex processes, which would save time and provide the ability to run simulations without supervision.

Some examples of applications include:

- Simulations can run until a goal is reached.
- Data from simulations can be sent to a list of files.
- Data for simulations can be extracted from files.
- Parameters can be swept in almost any way.
- Custom optimization routines can be placed around a simulation.
- Data from simulations can be manipulated, for example, to find trends.
- External programs can be called.
- Files can be manipulated.
- Interactive dialogs can be created to control and provide feedback on the status of a multiple-run simulation.

To be taken into account, a custom simulation script should reside in the `Inputs` folder of the executing schematic and be named `simulation_script.tcl`.

1. Thus they are also known as *Ptcl* scripts (or pt-scripts), as they drive the Ptolemy simulation engine using an extension of the *Tcl* language.

In this chapter, we provide some basic examples of custom simulation scripts. However, because simulation scripts can incorporate all *Tcl* (Tool Control Language) commands and also allow building custom GUI elements (such as dialog windows) based on *Tk* (graphical user interface ToolKit), their applications are widespread. Simulation scripts, as with *Tcl*, are the glue to paste simulations, their data, file systems, and computer networks together.

VPI Design Suite includes a built-in Tcl interpreter that allows Tcl commands and expressions to be used in the **Parameter Editor**, simulation scripts, and the PDE Scripting Language that serves as the basis for VPI Design Suite macros. However, the language used for macros has commands to operate on a schematic, rather than to drive the simulation engine after the schematic has been interpreted.

Figure 3-1 shows how different types of scripting relate to the schematic execution. For more information on Tcl, see "Tcl Basics" in Chapter 1, "Macros".

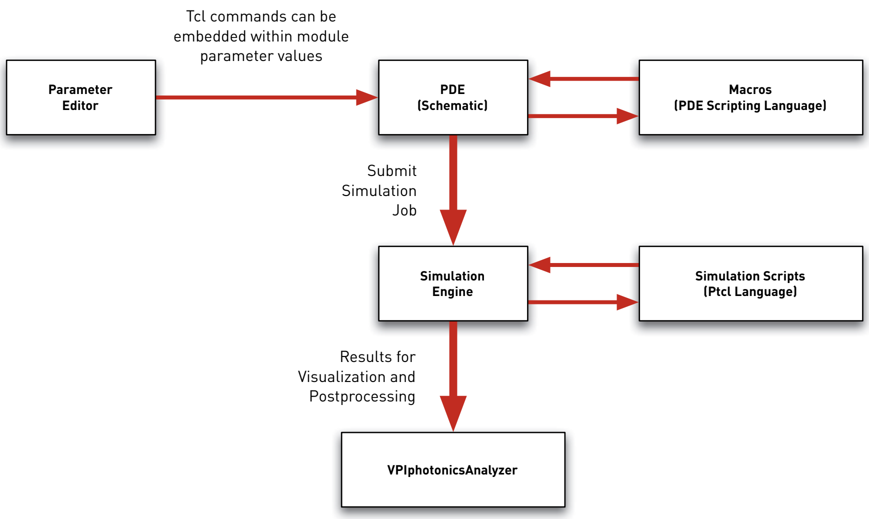

Tcl commands can be embedded within module parameter values
Parameter Editor
PDE (Schematic)
Macros (PDE Scripting Language)
Submit Simulation Job
Simulation Engine
Simulation Scripts (Ptcl Language)
Results for Visualization and Postprocessing
VPIphotonicsAnalyzer

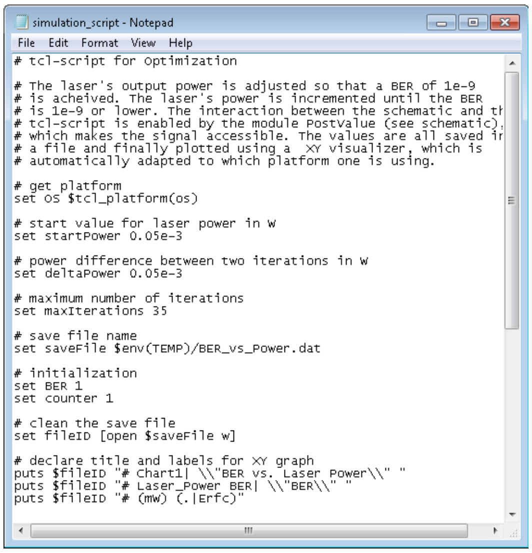
Figure 3-1 *Simulation workflow overview*

Note: In addition to Tcl, VPI Design Suite supports simulation scripting using the Python language. Several demonstrations using Python for multiparameter optimization are available under *Optical Systems Demos > Simulations Techniques > Other Sweep Techniques*. For more information, see "Scripting Using Python" on page 292.

Writing Custom Simulation Scripts

The custom simulation script can be edited directly in the `Inputs` folder or accessed either using the Sim Script button from the Ribbon or through the Submit Simulation Job window.

1. Open the Submit Simulation Job window (by selecting Run from the Run button menu).
2. Click Edit Script in the Job configuration panel.
   If no script existed, a confirmation to create it will be requested. After that, the script will be opened in the external text editor defined on your system (Preferences > General > External Text Editor) or Notepad by default (see Figure 3-2).



```
# tcl-script for Optimization

# The laser's output power is adjusted so that a BER of 1e-9
# is acheived. The laser's power is incremented until the BER
# is 1e-9 or lower. The interaction between the schematic and th
# tcl-script is enabled by the module PostValue (see schematic),
# which makes the signal accessible. The values are all saved in
# a file and finally plotted using a  XY visualizer, which is
# automatically adapted to which platform one is using.

# get platform
set OS $tcl_platform(os)

# start value for laser power in W
set startPower 0.05e-3

# power difference between two iterations in W
set deltaPower 0.05e-3

# maximum number of iterations
set maxIterations 35

# save file name
set saveFile $env(TEMP)/BER_vs_Power.dat

# initialization
set BER 1
set counter 1

# clean the save file
set fileID [open $saveFile w]

# declare title and labels for XY graph
puts $fileID "# Chart1| \\"BER vs. Laser Power\\" "
puts $fileID "# Laser_Power BER| \\"BER\\" "
puts $fileID "# (mW) (.|Erfc)"
```

Figure 3-2 *Editing a script in Notepad*

For more information, see “Running an External Script” on page 265 and “Access to Files from Schematic Packages” on page 272. Also refer to “Using Simulation Scripts” in Chapter 6 and “Simulation Scripts” in Chapter 9 of the *VPIphotonics Design Suite™ Simulation Guide*.

Simulation Script Commands

Before we proceed to more elaborate examples, here is a summary of commands that provide the most basic functions for controlling simulations. Simulation script commands extend the standard *Tcl* commands. The most useful commands are:

```
isremotesimulation
```

```
run
resetSweep
setstate
statevalue
getpostvalue
visualize
calcstate
breakstatus
pathexpand
ptclmessage
extendederrorreporting
init
runscript
runstep
wrapup
signaldb
splitpatharray
transientCheck
SimDataFetch
SimDataReturn
SimOutputReturn
SimOutputReturnEx
SimGetCommand
SimCmdWaiting
SimPaused
SimResumed
```

These commands are described in the following sections. There are several other commands in VPI Design Suite, but they are mainly intended for internal purposes. In this section we also describe basic commands and control structures of *Tcl*.

Note: For more information on standard Tcl commands such as `set`, `for`, etc., see the *Tcl/Tk Documentation* available from the **Help** menu.

Understanding Whether a Simulation is Running on the Remote Server

This information can be retrieved using the `isremotesimulation` command:

```
if [isremotesimulation] {
  ptclmessage "Simulation running on remote server!"
} else {
  ptclmessage "Local simulation"
}
```

It can be used for preventing doing something undesirable at the Remote Simulation Server — for example, showing any GUI windows like charts or input dialogs.

Running a Simulation

```
run n
```

This command executes the current simulation *n* times, and is equivalent to performing a standard simulation with the number of runs set to *n*. The number of runs may be omitted, in which case the default value for *n* is 1. The current universe that is run by this command is the setup from which you started the **Submit Simulation Job** window.

When the `run` command is executed multiple times in a simulation script, it is generally necessary to keep track of the total number of runs, and the maximum number of runs expected. This information is used by the VPIphotonicsAnalyzer Iteration Selector — all the chart types can arrange their input data based on iteration, run, or sweep level. The simple form of the `run` command given above automatically keeps track of the number of runs. However, you can control this explicitly using an extended version of the `run` command that includes an index for the current run number, and a number specifying the total number of runs.

The extended command is

```
run n i m
```

where *i* is the current run number, *n* is the number of executions on this run, and *m* is the total number of runs.

This version of the `run` command is used by parameter sweeps, and is usually placed in a `for` loop. For example, a single-level sweep can be implemented using:

```
for {set i 0} {$i<5} {incr i 1} {
      run 1 $i 5
}
```

Note: The index of the first execution of the `run` command must be zero, otherwise the VPIphotonicsAnalyzer charts that use this information may fail to update correctly.

Typically the loop will include a `setstate` command (see below) to change the value of a parameter that is being swept.

The additional parameters to the `run` command in the above example were optional, since for a single-level sweep the number of runs is automatically tracked. To keep track of the data sorting in a nested sweep it is not sufficient to know only the current run number, it is also necessary to know how many iterations have been performed at each level of the sweep. Additional parameters to the `run` command are used to support nested sweeps, the results of which can be displayed in a single analyzer chart. As always, the first parameter specifies the number of runs to perform per sweep iteration (usually 1). The numbers following specify, in pairs, the position of the current run in the sweep (starting from 0) and the total number of simulations in that sweep. In nested sweeps one number pair has to be specified for every sweep level. The command `resetSweep` may be used to reset the tracking of sweep level information between runs. For example:

```
resetSweep
for {set i1 0} {$i1 < 5} {incr i1 1} {
      for {set i2 0} {$i2 < 3} {incr i2 1} {
         run 1 $i1 5 $i2 3
      }
}
```

that is, perform one run of the simulation per sweep iteration. `i1` is the first sweep level and `i2` is the second sweep level counter. Fifteen runs will be performed in all.

Setting the Value of a Module Parameter

```
setstate modulename statename value -exprtype standard|python|file
```

This command changes the values and expression type of parameters of a module in the current simulation. The default expression type is `standard`. The command can be used before a `run` command to modify a parameter (or parameters) in that run. It can also be used to change the titles of analyzers or the names of files, so you can associate a result with a particular run.

Note: *State* is a term that is borrowed from Ptolemy to denote the parameters of modules and topology parameters. It is used for several command names in simulation scripts.

The command has a special `init` option that is used to set the value of a *volatile* parameter value and reinitialize the module (or the schematic):

```
setstate modulename statename value -exprtype etype init
```

This option is necessary when the command is called after initialization of the simulation, such as in transient simulations as described in "Extended Schematic Iteration Commands" on page 274.

Note: The `setstate` command supports only 'native' units (mostly MKS) of respective parameters. The **System of Units** preference as well the units settings in schematics are ignored.

Note: Parameters can accept only finite number values. Trying to assign `Inf`, `-Inf`, or `NaN` will cause errors.

Setting the Value of a Global Parameter

```
setstate this statename value -exprtype etype
```

By replacing the module name with the keyword `this`, a global parameter of the current universe can be modified.

Before a global parameter can be accessed, it has to be initialized. The syntax for that is:

```
initialize this statename
```

Note: The keyword `this` is a reserved keyword for reference to the current topology (universe) in the simulation scripts. See the following paragraphs for more examples.

Passing Numerical Data from the Simulation

Two modules, the ***PostValue*** and ***PostValueMxFlt***, allow one to probe numerical data circulating in a simulation and use it within a simulation script. A special `getpostvalue` command can be used to get a value received by the ***PostValue*** or ***PostValueMxFlt*** modules. One more option is to use the `statevalue` command together with the ***PostValue*** module.

statevalue

```
statevalue module param ?current|initial?
```

This command returns the value of a specified parameter, for example, the value of any parameter ***param*** of any module ***`module`***. As a special useful case, the *PostValue* module will grab its input data and convert it to a state (parameter value), which can then be read into the script using the `statevalue` command within the simulation script. This is an essential command if the script has to react to the result of a simulation, for example during an optimization loop. The `initial` option should be used in simulation scripts before the `run` command. The default option is `current`, in which case the `statevalue` command returns the value changed during a simulation. If not specified, `current` is assumed.

Note: Optional parameters of the command are enclosed in question marks:
`command ?option?`
The vertical bar divides different values of the parameter value: on|off

The `statevalue` command is often used with the *Tcl* `set` command, so that the returned value is saved in a variable. For example:

```
set outvalue [statevalue PostValue1 InputValue]
```

This command sets a variable `outvalue` to the value of the `InputValue` parameter of the `PostValue1` module.

If it is necessary to access the value of some parameter both before and after the `run` command in an uniform way (for example, in an optimization loop), it is necessary to invoke the `initialize` command in the following form

```
initialize module param
```

before `statevalue` command, for example

```
initialize FiberNLS1 Dispersion
set D [statevalue FiberNLS1 Dispersion] ;#Fiber dispersion
```

After the `run` command has been executed once, `statevalue` can be used without specifying any option (the default option `current` will be used).

Example

A schematic contains a *LaserCW* with **`AveragePower`** set to 1 mW. With the following short script:

```
set power [statevalue LaserCW_vtms1 AveragePower]
```

```
run
```

`power` will be equal to 0 (!)—instead, `statevalue` should be used with the `initial` option:

```
set power [statevalue TxExtModLaser Laser_AveragePower initial]
run
```

In this case, `power=0.001` as expected.

getpostvalue

```
getpostvalue module
```

This command returns the value which arrived to the input port of the specified module (*PostValue* or *PostValueMxFlt*) during the previous simulation run. For the *PostValue* module, it returns a float value, while for the *PostValueMxFlt* module it returns a serialized matrix as a list of the following form:

```
rows cols values
```

where `rows` and `cols` positions represent the number of matrix rows and columns correspondingly, while the `values` are structured as a list of lists representing matrix cells ordered by rows:

```
% set myMatrix [getpostvalue PostValueMxFlt_vtms1]
% ptclmessage myMatrix
2 2 {{3.1 -6.2} {6.2 -6.2}}
```

The command produces an error if no simulation runs were performed and thus no value has arrived to the *PostValue* or *PostValueMxFlt* module yet.

Example

A schematic feeds some float matrix to the *PostValueMxFlt* module. With the following short script[1], the user can access matrix cells, transpose the matrix, or generate a float-matrix signal corresponding to the matrix content.

```
# Load matrix package
package require struct::matrix

# Run simulation to feed data to the PostValueMxFlt module
run 1

# Retrieve data from the PostValueMxFlt_vtms1 instance
set mx_list [getpostvalue PostValueMxFlt_vtms1]

# Print obtained serialized matrix data to message log
ptclmessage [concat Input Matrix = $mx_list]

# Deserialize data to get matrix object
::struct::matrix myMatrix deserialize $mx_list

# Get matrix element from the first column and final row
set col 0
set row end
set element [myMatrix get cell $col $row]
ptclmessage $element
```

1. For details, refer to the documentation of the struct::matrix *Tcl* package: https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/tcllib/files/modules/struct/matrix.md

```
# Transpose matrix and print to message log
myMatrix transpose
ptclmessage [myMatrix format 2string]

# Set parameters of the Matrix_vtms1 module to generate matrix signal
setstate Matrix_vtms1 numRows [myMatrix rows]
setstate Matrix_vtms1 numCols [myMatrix columns]
setstate Matrix_vtms1 FloatMatrixContents [myMatrix format 2string]

run 1
```

Global Recalculation of a Parameter

Schematic parameters (those of universes and galaxies) are usually used indirectly: they are used in other parameters of modules via reference. After setting such parameter values, it is necessary to "tell" the simulation engine to recalculate this state in all modules. This is achieved by the `calcstate` command, that has two forms:

```
calcstate param
```

or

```
calcstate module param
```

where the first variant refers to parameter of the schematic (universe); it is a shorthand notation for `calcstate this` *`param`*.

This command recalculates the specified parameter either in the specified module or globally (for universe parameters). For example:

```
initialize this SampleRateDefault

# Set it globally
setstate this SampleRateDefault $SampleRate

# recalculate SampleRateDefault globally
calcstate SampleRateDefault
```

Running an External Script

It is possible to associate several simulation scripts (or script fragments) with a given schematic and access the scripts that are 'attached' to a schematic (placed in its `Inputs` folder). For this, the following command can be used:

```
runscript path/file
```

This is an enhanced version of the standard command `source` which loads a script from a file and executes it. In addition, `runscript` decodes the path similar to the `pathexpand` command (described in "Access to Files from Schematic Packages" on page 272), and may also execute encrypted scripts.

Clever Sweeps

Clever sweeps, such as piecewise linear sweeps and other interesting sweep functions can be implemented using the *Tcl* conditional evaluation operator '`? :`'.

For more information, see the description of the `expr` command in the *Tcl/Tk Documentation* (available from the **Help** menu), and "Using Expression Controls" in Chapter 7 of the *VPIphotonics Design Suite™ Simulation Guide*, which provides several illustrative examples.

Sweeping a Filename

```
for {set i 1} {$i<5} {incr i} {
      set file [expr $i * 10]
      set filename $env(TEMP)/Filebasename$file.dat
      setstate WriteToFile1 OutputFilename $filename
      run 1
}
```

Every time the `for` loop executes, a new `filename` is generated. In this case the digits 10, 20, 30, etc. are appended to the `Filebasename`. The filename is then used to set the **`OutputFilename`** parameter of the *WriteToFile* module. The *Tcl* expression `$env(TEMP)` is a reference to the global array `env`. This array is created when *Tcl* starts up, and it contains the complete set of environment variables that exists at this time. The index to the array is simply the name of the environment variable. Thus the effect of this expression is to place the files in the user's `TEMP` directory.

To read or write the files that are 'attached' to the schematic (placed in its `Inputs` or `Outputs` folders), use the `pathexpand` command, see "Access to Files from Schematic Packages" on page 272.

Control Structures: For and While loops, If-Then-Else clauses

Control structures are always needed when you program. Here are the most common ones.

The parameter sweep is a special case of creating a for loop. The general syntax for a `For` loop is shown in this example:

```
for {set i 1} {$i<5} {incr i} {
      <statement 1>
      <statement 2>
      <statement N>
}
```

which executes the statements in the loop four times.

Within the loop, any statements can be executed. These can be run commands, for example.

A `while` loop's syntax is:

```
set i 0
while {$i<5} {
      <statement 1>
      <statement 2>
      <statement N>
      set i [expr $i +1]
}
```

which executes until the variable *i* has a value less than 5.

The syntax for `if-then-else` clauses is illustrated with this example:

```
if {$a > 1} {
       <statement a>
} elseif {$a < 0} {
       <statement b>
} else {
       <statement c>
}
```

which executes statement a if the variable *a* is greater than 1, statement b if this variable is less than 0, or statement c otherwise.

Breaking Out of Loops

In addition to analyzing module parameters as described in "Passing Numerical Data from the Simulation" on page 263, it is also possible to check if a module requests a termination of simulation (in sweeps, for example). This possibility is based on a special module, *Break*, which is analogous to the `break` command in programming languages such as *Tcl* or *C++*.

This module can set a flag in the simulation engine to interrupt simulation iterations (either completely or at a certain level). The *Break* module sets this flag according to its input signal and can be used to handle situations when a signal value exceeds some defined safety margins.

The interface between the *Break* module and the simulation script is provided using a special command, `breakstatus`. The usual application of the `breakstatus` command is to check its return value after the `run` command to interrupt the simulation or exit from a certain level of schematic iterations. Additionally, the `breakstatus` command can be used to determine if a simulation interruption was requested by the user using the Break action from the Abort menu at the ribbon. Normally (when no simulation break is requested), the `breakstatus` command returns zero ('false' in *Tcl* terms).

When a termination condition of the ***Break*** module is satisfied during the `run` command, the `breakstatus` command will return either -1 when a termination of the whole simulation is requested, or a positive integer value indicating the level of the schematic iterations within a sweep. When a simulation interruption was requested by the user, the command will return -2. The user's request has a higher priority with respect to the ***Break*** module. For example, if this request was received during the simulation of the ***Break*** module, the command will still return -2.

Caution: The sweep levels (depth) are numbered starting from zero in the VPI Design Suite user interface and modules like *Break* or *Ramp*. However, the `breakstatus` command returns the iteration level to be terminated starting from 1. So if the **`BreakSweepDepth`** parameter is set to 2, the `breakstatus` command will return 3 when the break condition is fulfilled.

Both positive and negative values are treated as 'true' in *Tcl*, therefore it is possible to check if there is **any** termination condition in a single logical expression (see the first example below).

The *Break* module (same as the user's interruption request) neither interrupts the current loop nor exits from simulation scripts by itself. As proper handling of the break status is performed by the standard interactive simulations (like Sweep or Optimization), they would be interrupted by the ***Break*** module or at the user's request. If the schematic contains *Break* module(s), but the script does not use the `breakstatus` command, it will not be aware of the simulation termination.

The following examples illustrate the use of the `breakstatus` command:

- Check whether a simulation stop or break out of iteration level was requested

  ```
  run 1 $i $k

  if {[breakstatus]} {
        exit;
  }
  ```
  (this is the simplest use of the command, and is suitable for most applications such as one-dimensional sweeps)
- Break out of a 'for' loop when the interruption of a certain iteration level of nested loops is requested (below it is assumed that the `sweepLevel` variable contains the current level of schematic iterations)

  ```
  if {[breakstatus] >= $sweepLevel} {
        break;
  }
  ```
- Exit from a simulation script when the ***Break*** module requests stopping the whole simulation

  ```
  run 1 $k $m

  if {[breakstatus] == -1} {
        exit;
  }
  ```
- Exit from a simulation script when a simulation interruption is requested by the user

  ```
  run 1 $k $m

  if {[breakstatus] == -2} {
        exit;
  }
  ```

For more details, please refer to the description of the *Break* module.

File I/O in Tcl Scripts

To write something to a file in a *Tcl* script, you need to open a file, write to it, and then close the file. Below is an example of writing to a file:

```
set var 3
set filename c:/vpi/file.dat
set fileID [open $filename w]
puts $fileID “$var”
close $fileID
```

Here a variable containing the file name is created. Then the file is opened in the write mode, which means that all previous contents are discarded. The open command returns a file ID, that is used to access the file later, such as writing to it or reading its contents. If you wish to append something to a file, open it in the append mode by using

```
open $filename a
```

The `puts` command writes its argument to the file specified by the file ID.

The `close` command closes the file referenced via the file ID supplied to it.

It is possible to write multiple lines to a file. The puts command automatically inserts a new line at the end.

If you want to read something from a file, use

```
set filename c:/vpi/file.txt
set var 0
set fileID [open $filename r]
gets $fileID var
close $fileID
```

The `gets` command reads one line of the file and stores the result in `var`.

To check if the end of the file has been reached, use the `eof` command:

```
if {[eof $fileID]} {
      close $fileID
}
```

`eof` will return 1 if the end of file was reached during the most recent input operation, 0 otherwise.

To read or write files that are 'attached' to a schematic (placed it its `Inputs` or `Outputs` folders), use the `pathexpand` command (see below).

Calling Third-Party Applications in Simulation Scripts

Third-party applications such as MATLAB or Python scripts can be started from simulation scripts using the `exec` command.

The syntax is:

```
exec "application" "$file" ?-sync?
```

Keyword `-sync` defines the regime in which an external application is run. If it is specified (synchronous mode), the script waits the termination of external program and, then, continue working. If it is omitted (asynchronous mode), script runs external application and continues working independently of external application.

For example, MATLAB scripts are started with

```
exec matlab /r mfile /automation
```

which would start the MATLAB script, or `.m` file, `mfile` in the "automation mode", which causes MATLAB to just execute the `.m` file without bringing up a splash screen or the command window. For this to work, the m-file has to reside in the work directory of the MATLAB installation!

Python scripts are started with

```
exec python c:/vpi/vpi_script.py
```

which would start Python, which would execute the Python script `c:/vpi/vpi_script.py`. You have to specify the path to the Python script, unless the path is known to the Python interpreter, which means that it is included in the system variable PYTHONPATH. If you wish to use MATLAB or Python to simulate an optical component (that is, to create a module to place on a schematic), refer to Chapter 2, "Cosimulation".

Visualizing Data and Data Files

To open a new VPIphotonicsAnalyzer window with a chart containing data fetched from a file in the local file system, the command

```
visualize datafile
```

can be used. The ***datafile*** parameter must contain the full path to a data file. Either a plain-text file or a `.vpa` file with the saved results can be visualized by this command.

For details, see "Visualizing Text Files" in the *VPIphotonics Design Suite™ User Interface Reference*.

The data calculated with a simulation script can be visualized in the VPIphotonicsAnalyzer (1D, 2D, or 3D) using the command `visualizedata`:

```
visualizedata window_id dataset1 ?dataset2 ...? ?-options options?
   ?-sweepinfo sweepinfo?
```

Here the ***window_id*** specifies the ID of the window in VPIphotonicsAnalyzer, where the data will be shown (this is similar to the instance ID of the analyzer module in the schematic). It also determines the default title of the analyzer window, which can be changed by specifying necessary title in ***options***. The visualized data can be one of three types: 1D, 2D or 3D. 1D data must be organized as a list of float or integers; 2D/3D data must be a list of lists, where the inner list is a pair/triple of float or integers.

There could be several datasets specified, each of which will be shown as a separate input on the analyzer chart. All datasets shown in one analyzer window must be of the same type (mixing of 1D, 2D and 3D datasets is not allowed).

The `-options` option can be used to specify the plot options like window title, trace legend, trace color, etc. The syntax of the options string is described in the manual pages for the *SignalAnalyzer* module.

The `-sweepinfo` option can be used to assign sweep information (number of iteration, run, or sweep point) to the visualized data. The sweep information attached to the data with the `run` command is ignored in that case. When the command is executed several times with the same sweep information (or if the ***sweepinfo*** option is omitted), the data set(s) from each execution will produce a new iteration.

The general form of the `-sweepinfo` option is as follows:

```
-sweepinfo {
      Iteration {CurrentIteration TotalIterations}
      Run {CurrentRun TotalRuns}
      NameSweep0 {LevelSw0 CurrentSw0 TotalSw0 ValueSw0}
      NameSweep1 {LevelSw1 CurrentSw1 TotalSw1 ValueSw1}
      …
}
```

The lists for `Run` and `Iteration` consist of 2 elements. The 1st element is the number of the current run or iteration (starting from 1), and the 2nd element is the total number of runs or iterations. If the total number of runs or iterations is unknown then the current number of the run or iteration can be used as the 2nd element.

The lists for *`NameSweepN`* consists of 4 elements. The 1st element is the level of the sweep (starting from 0), and the 2nd element is the number of the current sweep point (starting from 1). The 3rd element is the total number of sweep points. The 4th element is the current value of the swept variable (for example, `193.1e12` if the frequency variable is swept). The 4th element is optional. If the total number of sweep points is unknown the current number of the sweep point can be used as the 3rd element.

The entry *`NameSweepN`* is used as the name of the swept variable (in the VPIphotonicsAnalyzer the name of the swept variable is shown in **Data Ordering** controls **Connect** and **Differ by Color** and in tool-tips of **Iteration Selector**).

Here is an example of the usage of the `visualizedata` command:

```
set 2D_1 {{1 10} {2 20} {3 30} {4 40}}
set 2D_2 {{1 20} {2 30} {3 40} {4 50}}
visualizedata "2D_Data" $2D_1 $2D_2 -options {legend1="RZ" legend2="NRZ" title="2D
Data"} -sweepinfo {Iteration {2 3} CenterFreq {0 1 4 2e-3} Power {2 3 7}}
```

When the above code is executed, an analyzer window titled **2D Data** will start. Two straight lines (datasets `2D_1` and `2D_2`) labeled **RZ** and **NRZ** will be plotted. The **Iteration Selector** will show one iteration labeled as iteration 2 of a total of 3 iterations; point 1 of a total of 4 points at sweep named *CenterFreq* of level 0 with swept variable equal to 2e-3; point 3 of a total of 7 points at sweep named *Power* of level 2.

When several datasets need to be plotted in the same window, we recommend using only one `visualizedata` command, rather than multiple commands for each dataset. This will show all data sets together as several inputs rather than showing one dataset per iteration.

To combine several datasets for single `visualizedata` command, the *Tcl* list unpacking can be used:

```
set 2D_1 {{1 10} {2 20} {3 30} {4 40}}
set 2D_2 {{1 20} {2 30} {3 40} {4 50}}
...
set 2D_N {{1 30} {2 40} {3 50} {4 60}}

lappend lst $2D_1
lappend lst $2D_2
# ... Append any required number of datasets here.
# Use loops or any other constructs if necessary. ...
lappend lst $2D_N
visualizedata "2D Data" {*}$lst
```

The `visualizedata` command is also available in Python, see "Simulation Control Functions" on page 294 for details and example.

Displaying Messages from Simulation Scripts

The following command can be used to display a warning (or informational, or progress) message in the VPI Design Suite **Message Log**:

```
ptclmessage text ?Info|Warning|Progress?
```

The *`text`* parameter must contain the text string to be displayed.

Note: Do not forget to enclose text with spaces in quotation marks.

The second optional parameter defines whether the message will be displayed as warning (default), informational or progress message. Note that progress messages are displayed only if the **Report progress** checkbox is selected in the **Submit Simulation Job** dialog.

Note: For more information on the available options, see “Submit Simulation Job Dialog” in the *VPIphotonics Design Suite™ User Interface Reference* (press `F1` in the dialog).

Access to Files from Schematic Packages

To support simulations on different computational platforms, VPI Design Suite uses specially encoded references to the files attached to simulation setups (in the `Inputs` and `Outputs` folders). On the other hand, this encoding prevents ‘direct’ usage of input and output files from simulation scripts.

To access the files attached to a schematic, use the command

```
pathexpand path/filename
```

The command converts the encoded file reference into the absolute path that can be further used with commands such as `open`, `source`, `visualize`, etc.

Note: To use the input (or output) file, please ensure that it is referenced by a schematic or module parameter. Do not forget to use the `initialize` command before accessing the global schematic parameters.

When a filename is specified using an array parameter, it is necessary to parse it with the `splitpatharray` command first and then use standard Tcl commands to manipulate lists.

Example:

```
initialize this OutputDataFiles
set out_file [pathexpand [lindex [splitpatharray [statevalue this OutputDataFiles
current]] 1]]
set fileID [open $out_file w]
```

Debugging Simulation Scripts

If a syntax error occurs, or an undefined variable is encountered, an error message will be displayed, but the line number that is provided may refer **not** to the line of the *custom* simulation script where the error occurred but to the line within the procedure or a loop body.

To find the error source unambiguously, it is recommended to enable the **Extended error reporting** option in the **Submit Simulation Job** dialog. This tells the simulation engine to display an 'extended' error message, containing both the command that actually causes an error, and the 'call stack', which contains information on all procedure calls, loops, etc., that were active when the error was encountered. Usually, this information enables you to find the source of the error more quickly.

The debugging process may require access to the values of internal script variables. This can be achieved by using the `ptclmessage` command, described in "Displaying Messages from Simulation Scripts" on page 272.

Alternatively, you may use the possibility to show some of the variables that are used by your script via Tk-based GUI elements.

For example, the `MessageBox` command (see the code below) can be used to display a script variable during simulation. Additional other examples of Tk-based dialogs are described under "Demonstrations using Simulation Scripts" on page 292.

```
# Reset the Tk window manager

wm withdraw .

#---------------------------------------------------------------------
#   Message Window Procedure
#---------------------------------------------------------------------
proc MessageBox {message} {
toplevel .msgWin
message .msgWin.msg  -width [string length $message]c  -justify left -text $message
button .msgWin.ok -text "OK" -command {destroy .msgWin}
pack .msgWin.msg .msgWin.ok -anchor center  -padx 1m -pady 2m
focus .msgWin
tkwait window .msgWin
}
#---------------------------------------------------------------------
```

Special Scripting Commands for Transient Simulations

This section provides a reference for several commands that are especially implemented for simulating prolonged power transients in fiber networks. See "Signal Database and Simulation Scripting" in Chapter 10 of the *VPItransmissionMaker™Optical Systems User's Manual* for more details and code examples.

Extended Schematic Iteration Commands

To provide flexible control over the schematic iterations in transient simulations, VPI Design Suite provides special commands that implement sweep-like schematic iterations without reinitialization:

```
init n 0 total
```

This initializes the simulation environment, scheduler, signals, etc. Like the `run` command, `init` needs the sweep information, as described in "Running a Simulation" on page 260. For transient simulations, the number of runs, *n*, is normally one.

```
runstep n 0 total
```

The command executes one or several runs of the whole schematics without a reinitialization or freeing of signals.

The `runstep` command resets the logical information (as in standard sweeps, see "Logical Information in Block Mode" in Chapter 4 of the *VPItransmissionMaker™Optical Systems User's Manual* for details). The command supports an option, `-iterate_logic`, that forces the simulation engine to accumulate logical information to newly generated signals (similar to multiple iterations). Setting the global parameter **`LogicalInformation`** to `CurrentRun` will take precedence over the above option (the logical information will be reset).

```
wrapup
```

The command clears the simulation engine memory and destroys all modules.

Note: The commands for database interface and checking schematic status **require** that the respective modules are initialized, Therefore both `signaldb` and `transientCheck` commands must be used **only** when schematic iteration is governed by the `init`, `runstep` and `wrapup` commands.

Together, these three commands perform the same operations as the `run` command.

Caution: To prevent memory leaks, the `init` command should always be accompanied by a corresponding call of the `wrapup` command.

Signal Database Interface Command

Note: Optional parameters of the command are enclosed in question marks:
`command` *`?option value?`*

The signal database (or signal DB) provides a unified mechanism for storing and retrieving signal values (optical, electrical, numerical) and the internal states of modules.

VPI Design Suite provides a special command

```
signaldb
```

with a number of options that allow loading, retrieving, and managing data in the signal database.

Note: If the schematic contains no *active* database-related modules, the `signaldb` command produces an error.

The available options are:

`signaldb setup` *`?database filename? ?modules module_ID? ?simulation sim_ID?`*

This sets up the default parameters of subsequent calls of the `signaldb` command. All of the command parameters (*`database`*, *`modules`*, *`simulation`*) are optional. The set default value(s) will be used if a specific parameter, such as database name or module list is omitted in subsequent calls of other options of the `signaldb` command.

In this and other variants of the `signaldb` command, it is possible to specify a *list* of module IDs. If a module (or module list) is not specified both as default and in the command, the `signaldb` command will interrogate all database-related modules that are present in schematic.

`signaldb save` *`?database filename? ?module module_ID? ?step step_No? ?simulation sim_ID?`*

This forces all database-related modules to save their signal values to the database.

If the `step` option is **not** specified, the command finds the maximal step number in the database (for the selected simulation ID, whenever specified), increases it by one and uses it to save data.

`signaldb load` *`?database filename? ?module module_ID? ?step step_No? ?simulation sim_ID?`*

This forces all respective modules to load the signals (or internal states) from the database.

If the `module` option is specified, only the respective module(s) will load the data. If the `step` option is **not** specified, the modules will load data with the largest step (for the selected simulation ID), otherwise the specified step key value will be used.

If the `simulation` option is specified, the data for the specified simulation ID will be used. If no simulation option is specified and the signal database contains more than one simulation ID, simulation stops with an error message.

`signaldb delete` *`?database filename? ?module module_ID? ?step step_No? ?simulation sim_ID?`*

This deletes some content from the database.

If the `module` option is specified, only data for the respective module(s) is deleted. If `step` option is **not** specified, the command delete all data from the database, otherwise the specified step number is used. If the `simulation` option is specified, data for the specified simulation ID is deleted, otherwise the data for all simulation IDs is removed.

`signaldb modules` *`?database filename? ?step step_No? ?simulation sim_ID?`*

This returns a list of module IDs that are present in database file. If the `step` option is specified, the command returns the list of modules for this particular step, otherwise the list of all modules is returned.

If the `simulation` option is specified, the command returns the module list for this particular simulation ID.

`signaldb steps` *`?database filename? ?module module_ID? ?simulation sim_ID?`*

This returns a list of step numbers that are present in database file. If the ID of a single module is specified, the command returns the list of steps for this particular module. If no module ID or list of modules is specified, the command returns list of all step numbers.

If the `simulation` option is specified, the command returns the step list for this particular simulation ID.

`signaldb get` *`?database filename? ?module module_ID? ?step step_No? ?simulation sim_ID?`*

This returns the value(s) of the module internal parameters or numerical signal(s) that are present in the database file. The command forces the specified module to read the respective numerical signal or parameter value and return this data to the script. Multiple parameter or signal values are returned as a list.

If the `module` option is specified, the command returns the saved value for the particular module(s). If the `step` option is specified, the command returns data for the particular step number, otherwise a list of data values is formed.

If the `simulation` option is specified, the command returns the step list for this particular simulation ID.

Schematic Status Check Commands

Efficient simulation of transient effects requires the evaluation of simulation parameters for several modules and an analysis of schematic contents. VPI Design Suite provides a special command

```
transientCheck
```

with a number of options that provide access to characteristics of the schematic and the simulation process.

The available options are:

`transientCheck steady` *`?module moduleID? ?average?`*

This returns the relative value for the signal change — or internal parameter(s) change — during the last simulation step for the specified module or list of relative signal changes for all registered modules. If the specified module(s) do(es) not support accuracy handling (or cannot calculate it), the command returns -1.

This command should be called on the second or later iteration of the schematic in the transient simulation.

The option ***average*** forces the command to calculate the root-mean-square average of relative signal changes for all registered or specified modules. If at least one of the modules returns -1, the averaging will return -1.

Specification of nonexistent module IDs produces an error.

`transientCheck getmodules`

This returns the list of IDs of all instances of database-related modules in the schematic, including the modules inside hierarchal designs (galaxies).

Special Scripting Commands for Interactive Simulation

This section provides a reference for several commands that previously were intended only for internal usage by interactive simulations (such as sweep, tuning, yield, etc.) but after introduction of the Simulation Engine Driver component, these commands are now also useful in user scenarios (for more information, see Chapter 4, "Simulation Engine Driver").

These commands include: `SimDataFetch`, `SimDataReturn`, `SimOutputReturn`, `SimOutputReturnEx`, `SimCmdWaiting`, `SimGetCommand`, `SimRunning`, `SimDone`, `SimPaused`, `SimResumed`.

Sending and Receiving Data

`SimDataFetch` *`floatValues ?intValues? ?listValues?`*

The `SimDataFetch` command should be used if data needs to be received by the simulation engine from the client that initiated execution of the simulation script.

The command takes up to three names of array variables as arguments. Each array is populated with values of the appropriate type. Although the function has three arguments, only float values are supported by external clients.

`SimDataReturn` *`floatValues ?intValues? ?listValues? ?stringValues?`*

The `SimDataReturn` command should be used if data needs to be returned from the simulation engine to the client.

This command takes up to four names of array variables as arguments, but only float values are supported by external clients.

`SimOutputReturn` *`floatValues`*

`SimOutputReturn` is very similar to `SimDataReturn`, but logically intended for sending results of simulation, not data used during simulation.

It receives only one array of float values.

`SimOutputReturnEx` *`values`*

`SimOutputReturnEx` is very similar to `SimOutputReturn`, but it can send matrix values alongside with float values. It receives a list of lists.

If an inner list has three elements, they are interpreted as follows:

- **i.** key name in the `SimulationData` collection;
- **ii.** value type (`Float`, `FloatMatrix`);
- **iii.** value.

If an inner list has two elements, they are interpreted as follows:

- **i.** key name in the `SimulationData` collection;
- **ii.** ID of the ***PostValue(MxFlt)*** module to take a value from.

If an inner list has only one element, it is interpreted as the ID of the ***PostValue(MxFlt)*** module and a key name.

Commands

`SimCmdWaiting` and `SimGetCommand` are intended for receiving commands which could be used to make a decision on further script execution.

`SimCmdWaiting` has no arguments and returns `1` if a command was received and `0` otherwise. This command is needed because sometimes it is required to know whether a command has been received without actually blocking the operation.

`SimGetCommand` also has no arguments and returns the name of the command. Currently, the following commands are supported: `Pause`, `Resume`, and `Exit`. `SimGetCommand` blocks the operation and waits until a command is received. It will return `Exit` if a simulation interruption is requested by the user via the ***Break*** action at the ribbon (see also "Breaking Out of Loops" on page 267).

State Notifications

The remaining four commands – `SimRunning`, `SimDone`, `SimPaused` and `SimResumed` are intended for notification of clients about simulation engine state transitions. Their usage is absolutely optional and completely up to the simulation script developers. But usually `SimRunning` should be used right before calling the `run` command, whereas `SimDone` is typically used right after the `run` command, `SimPaused` before a script enters the waiting state and `SimResumed` right after it exits the waiting state.

Example

The following listing provides an excerpt of a simulation script which uses all of the aforementioned commands:

```
while { 1 }
{
	# Receive input from client
	SimDataFetch _values

	# Setup module parameters
	setstate "Const_vtms1" "level" $_values(A1)

	# Run simulation
	SimRunning
	run 1
	SimDone

	# Return results of simulation
	set _outputs(Output) [statevalue PostValue_vtms1 InputValue]
	SimOutputReturn _outputs

	# Recalculate input values
	set _values(A1) [expr $_values(A1) + 1]

	# Return data back to client
	SimDataReturn _values

	# Check if pause or exit commanded
	if { [SimCmdWaiting] !=0 }
	{
		switch [SimGetCommand]
		{
			Exit { break }
			Pause
			{
				SimPaused
				switch [SimGetCommand]
				{
					Exit { break }
					Resume { SimResumed }
				}
			}
		}
	}
}
```

Simulation Script Examples

These examples show some of the powerful features of simulation scripts. They can be modified as you desire. They are available in the demonstration folder *Optical Systems Demos > Simulation Techniques > Simulation Scripts.*

Sweep Using a Simulation Script

Optical Systems Demos > Simulation Techniques > Simulation Scripts > Sweep using a Simulation Script

This example shows how a simulation script can set the values of parameters on a schematic, then run the schematic. If the setting and run commands are placed within a `for` loop (or a `while` loop), a sweep is built up.

The schematic is shown in Figure 3-3. The simulation script has been copied to a text box. However, the actual script resides in the **Submit Simulation Job** window. The schematic reads the values of two *Const* modules and sends them to a *NumericalAnalyzer2D* module. The *Const* modules receive their data from the simulation script, which generates two values before it runs the simulation. The setting and running of the schematic is within a `for` loop.

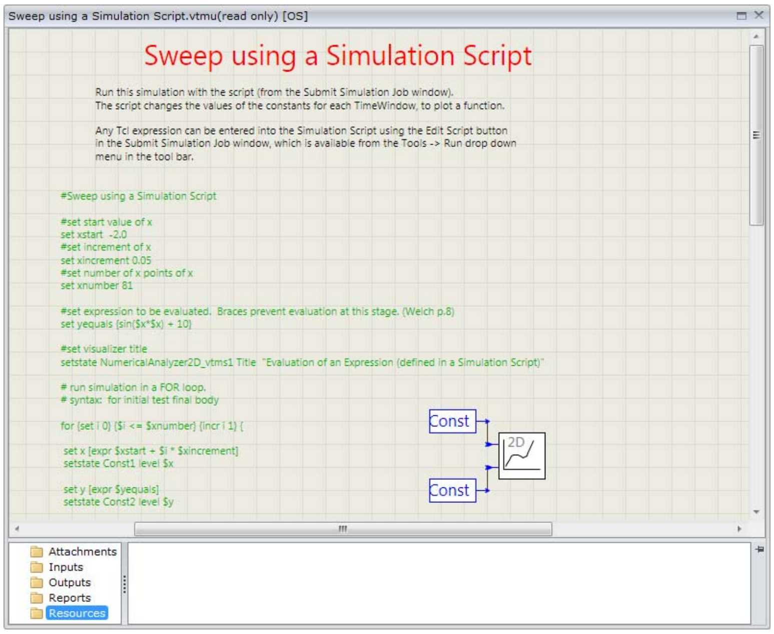
Note: The **Interactive Simulation** function (described in Chapter 7 of the *VPIphotonics Design Suite™ Simulation Guide*) can set up sweeps without having to create a simulation script. However, the simulation script ultimately has more flexibility, in that it can carry out 'housekeeping' operations during the sweep, such as some treatment of data, saving results and creating results files (see the following examples).

The simulation script is reproduced below. The ranges of the sweep are first specified. Then an expression can be entered to provide any form of sweep (this is plotted on the y-axis of the graph). The script also labels the curve of the graph with the expression that is to be evaluated. Within the FOR loop, the simulation script sets the states of the *Const* modules before running the schematic.

```
#Sweep using a Simulation Script

#set start value of x
set xstart  -2.0
#set increment of x
set xincrement 0.05
#set number of x points of x
set xnumber 81
```

```
#set expression to be evaluated.  Braces prevent evaluation at this stage. (Welch p.8)
set yequals {sin($x*$x) + 10}

#set Analyzer title
setstate NumericalAnalyzer2D_vtms1 Title  "Evaluation of an Expression (defined in a
Simulation Script)"

# run simulation in a FOR loop.
# syntax:  for initial test final body
for {set i 0} {$i <= $xnumber} {incr i 1} {

 set x [expr $xstart + $i * $xincrement]
 setstate Const1 level $x

 set y [expr $yequals]
 setstate Const2 level $y

 run 1
}

#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999
Prentice-Hall
```

Note: It is **impossible** to use a simulation script during interactive simulation: the interactive simulation uses a special script by itself. However, it is *possible* to run the same schematic using a simulation script or (alternatively) using an interactive simulation to obtain different types of sweeps, for example.
It is also possible to combine simulation scripts with implicit sweeps as described in Chapter 9 of the *VPIphotonics Design Suite™ Simulation Guide*.


Sweep using a Simulation Script.vtmu(read only) [OS]

Sweep using a Simulation Script

Run this simulation with the script (from the Submit Simulation Job window).
The script changes the values of the constants for each TimeWindow, to plot a function.

Any Tcl expression can be entered into the Simulation Script using the Edit Script button
in the Submit Simulation Job window, which is available from the Tools -> Run drop down
menu in the tool bar.

```
#Sweep using a Simulation Script

#set start value of x
set xstart  -2.0
#set increment of x
set xincrement 0.05
#set number of x points of x
set xnumber 81

#set expression to be evaluated.  Braces prevent evaluation at this stage. (Welch p.8)
set yequals {sin($x*$x) + 10}

#set visualizer title
setstate NumericalAnalyzer2D_vtms1 Title  "Evaluation of an Expression (defined in a Simulation Script)"

# run simulation in a FOR loop.
# syntax:  for initial test final body

for {set i 0} {$i <= $xnumber} {incr i 1} {

 set x [expr $xstart + $i * $xincrement]
 setstate Const1 level $x

 set y [expr $yequals]
 setstate Const2 level $y
```

Figure 3-3 *Schematic demonstrating a parameter sweep using a simulation script*

File Input using a Simulation Script

Optical Systems Demos > Simulation Techniques > Simulation Scripts > File Input using a Simulation Script

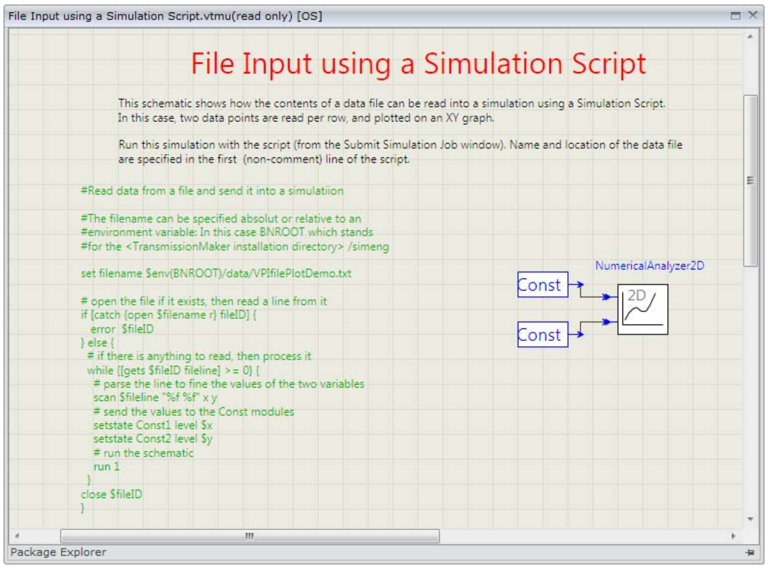
This simulation script reads data from a file and sends it into a simulation. It is a very simple example, but allows many types of data to be input into a simulation, especially if data is converted into VPI ONSL format using the modules in the folder *Signal Conversion*.

The schematic is shown in Figure 3-4. The schematic reads the values of two *Const* modules and sends them to a *NumericalAnalyzer2D* module. The *Const* modules receive their data from the simulation script, which in turn receives its data from a file.

Note: There are also several modules that read data from files into a simulation without using a simulation script, for example *ReadFromFile*, which loads a signal stored to the file system into simulation, or *DopedFiber*, which uses a data file to specify the amplification profile, etc. The advantage of using simulation scripts is that the file name can change from run to run, and the file's contents can be manipulated before it is sent to the simulation.

The simulation script is reproduced below. The file’s location is specified in the third line. Within a `while` loop, the simulation script opens the file outside the loop, reads the file line by line inside the loop, parsing each line and running the simulation. After the loop is ended, the file is closed.

```
#Read data from a file and send it into a simulation

#The filename can be specified absolute or relative to an
#environment variable: In this case BNROOT which stands
#for the <VPIdesignSuite installation directory>/simeng

set filename $env(BNROOT)/data/VPIfilePlotDemo.txt

#open the file if it exists, then read a line from it
if [catch {open $filename r} fileID] {
   error  $fileID
} else {
  # if there is anything to read, then process it
  while {[gets $fileID fileline] >= 0} {
    # parse the line to fine the values of the two variables
    scan $fileline "%f %f" x y
    #send the values to the Const modules
    setstate Const1 level $x
    setstate Const2 level $y
    # run the schematic
    run 1
  }
close $fileID
}
#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999
Prentice-Hall
#See Chapter 9 "Working with Files and Programs"
```



File Input using a Simulation Script

This schematic shows how the contents of a data file can be read into a simulation using a Simulation Script.
In this case, two data points are read per row, and plotted on an XY graph.

Run this simulation with the script (from the Submit Simulation Job window). Name and location of the data file are specified in the first (non-comment) line of the script.

```
#Read data from a file and send it into a simulatiion

#The filename can be specified absolut or relative to an
#environment variable: In this case BNROOT which stands
#for the <TransmissionMaker installation directory> /simeng

set filename $env(BNROOT)/data/VPIfilePlotDemo.txt

# open the file if it exists, then read a line from it
if [catch {open $filename r} fileID] {
  error  $fileID
} else {
 # if there is anything to read, then process it
 while {[gets $fileID fileline] >= 0} {
  # parse the line to fine the values of the two variables
  scan $fileline "%f %f" x y
  # send the values to the Const modules
  setstate Const1 level $x
  setstate Const2 level $y
  # run the schematic
  run 1
 }
close $fileID
}
```

Figure 3-4 *Schematic demonstrating reading data into a simulation from a file*

File Output using a Simulation Script

Optical Systems Demos > Simulation Techniques > Simulation Scripts > File Output using a Simulation Script

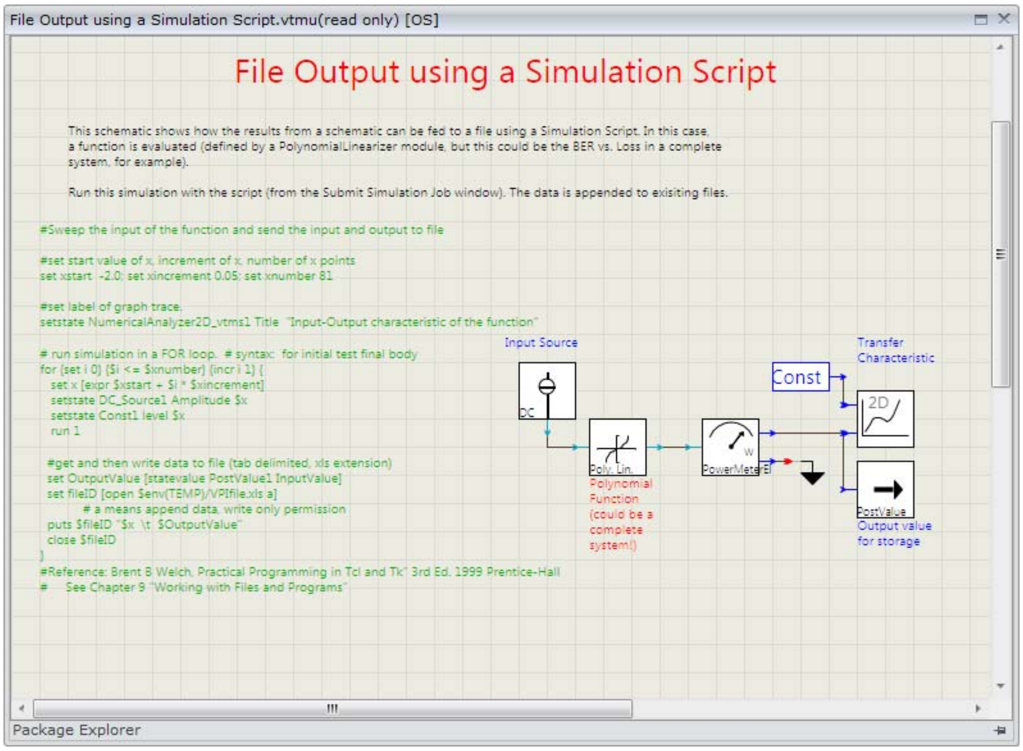
This schematic shows how the results from a schematic can be fed to a file via a simulation script. In this case, a function is evaluated (defined by a *PolynomialLinearizer* module, but this could be the BER vs. Loss in a complete system, for example).

This functionality is useful if you wish to run a large set of simulations, and automatically record the results of each simulation. The advantage is that this process will run overnight without your attention.

The schematic is shown in Figure 3-5. The simulation script runs a FOR loop. On each iteration, the value of the *DC_Source* module is set, then passed to a *PolynomialLinearizer* function as an electrical waveform. The mean value of the output of the polynomial is calculated using a *PowerMeterEl* module, and plotted against the input value using a *NumericalAnalyzer2D* module. The output is also sent back to the simulation script using *PostValue* module. The script then writes the input and output values to a line in a file. This processes is repeated to build up a graph and a file containing the curve produced by the polynomial.

The simulation script is reproduced below. The range of the FOR loop is specified in the third line. The title of the analyzer chart is then set. The simulation script then enters a FOR loop, sets the value of the *DC_Source*, runs the simulation, finds the `statevalue` of the *PostValue* module, opens the file, writes the input-output data to the file, and repeats until the FOR loop is completed.

```
#Sweep the input of the function and send the input and output to file

#set start value of x, increment of x, number of x points
set xstart  -2.0; set xincrement 0.05; set xnumber 81
#set analyzer title.
setstate NumericalAnalyzer2D_vtms1 Title  "Input-Output characteristic of the function"

# run simulation in a FOR loop.  # syntax:  for initial test final body
for {set i 0} {$i <= $xnumber} {incr i 1} {
   set x [expr $xstart + $i * $xincrement]
   setstate DC_Source1 Amplitude $x
   setstate Const1 level $x
   run 1

 #get and then write data to file (tab delimited, xls extension)
  set OutputValue [statevalue PostValue1 InputValue]
  set fileID [open $env(TEMP)/VPIfile.xls a]
            # a means append data, write only permission
  puts $fileID "$x  \t  $OutputValue"
  close $fileID
}
#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999
Prentice-Hall
#     See Chapter 9 "Working with Files and Programs"
```


File Output using a Simulation Script.vtmu(read only) [OS]

File Output using a Simulation Script

This schematic shows how the results from a schematic can be fed to a file using a Simulation Script. In this case, a function is evaluated (defined by a PolynomialLinearizer module, but this could be the BER vs. Loss in a complete system, for example).

Run this simulation with the script (from the Submit Simulation Job window). The data is appended to exisiting files.

```
#Sweep the input of the function and send the input and output to file

#set start value of x, increment of x, number of x points
set xstart  -2.0; set xincrement 0.05; set xnumber 81

#set label of graph trace.
setstate NumericalAnalyzer2D_vtms1 Title  "Input-Output characteristic of the function"

# run simulation in a FOR loop.  # syntax:  for initial test final body
for {set i 0} {$i <= $xnumber} {incr i 1} {
  set x [expr $xstart + $i * $xincrement]
  setstate DC_Source1 Amplitude $x
  setstate Const1 level $x
  run 1

 #get and then write data to file (tab delimited, xls extension)
 set OutputValue [statevalue PostValue1 InputValue]
 set fileID [open $env(TEMP)/VPIfile.xls a]
        # a means append data, write only permission
 puts $fileID "$x  \t  $OutputValue"
 close $fileID
}
#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999 Prentice-Hall
#    See Chapter 9 "Working with Files and Programs"
```

Figure 3-5 *Schematic demonstrating reading data from a simulation into a file*

Retrieve Numerical Data from a Simulation

The “BER vs. Laser Power” schematic shows how results from a schematic can be retrieved by a simulation script applying the ***PostValue*** or ***PostValueMxFlt*** modules.

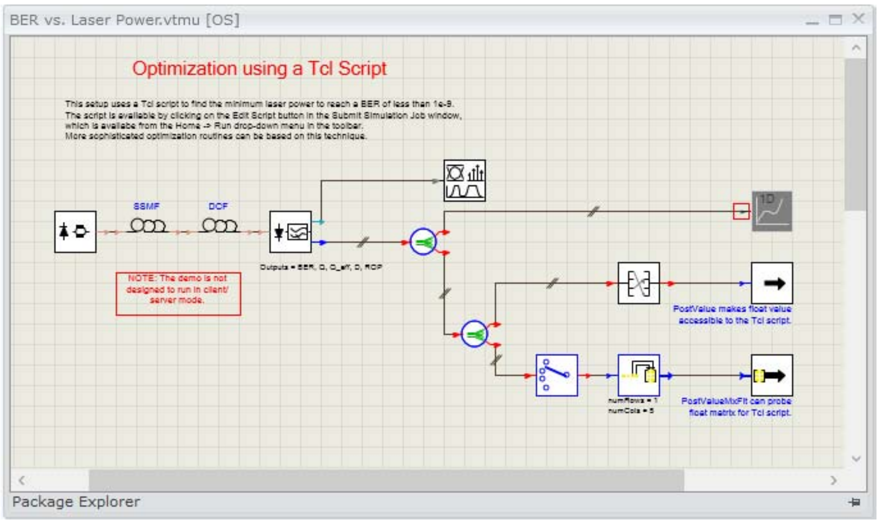
Optical Systems Demos > Simulation Techniques > Simulation Scripts > BER vs. Laser Power

While the ***PostValue*** module gets a float value from a simulation, the ***PostValueMxFlt*** module can access values of a float matrix. This is useful when you need to get multiple signal metrics. Some other features are also demonstrated, such as how to

- use the struct::matrix *Tcl* package for handling matrices,
- save float matrix values to a file, and
- show float matrix data applying the `visualizedata` command.

The schematic is shown in Figure 3-6. It uses a *Tcl* script to find the minimum laser power to reach a BER of less than 1e-9.

To access the script, click the **Edit Script** button in the **Submit Simulation Job** window (available from the **Home** > **Run** drop-down menu in the toolbar).


BER vs. Laser Power.vtmu [OS]
Optimization using a Tcl Script
SSMF
DCF
1D
Package Explorer

This setup uses a Tcl script to find the minimum laser power to reach a BER of less than 1e-9.
The script is available by clicking on the Edit Script button in the Submit Simulation Job window,
which is availabe from the Home -> Run drop-down menu in the toolbar.
More sophisticated optimization routines can be based on this technique.

Outputs = BER, Q, Q_eff, D, ROP

NOTE: The demo is not designed to run in client/ server mode.

PostValue makes float value accessible to the Tcl script.

numRows = 1
numCols = 5

PostValueMxFlt can probe float matrix for Tcl script.

Figure 3-6 *Two ways to access numerical data during a simulation*

Some more sophisticated optimization routines can be based on this technique.

Laser Power Control

Photonic Circuits Demos > Laser Characterization > Laser Power Control

Note: This demonstration works with the VPIcomponentMaker Photonic Circuits product.

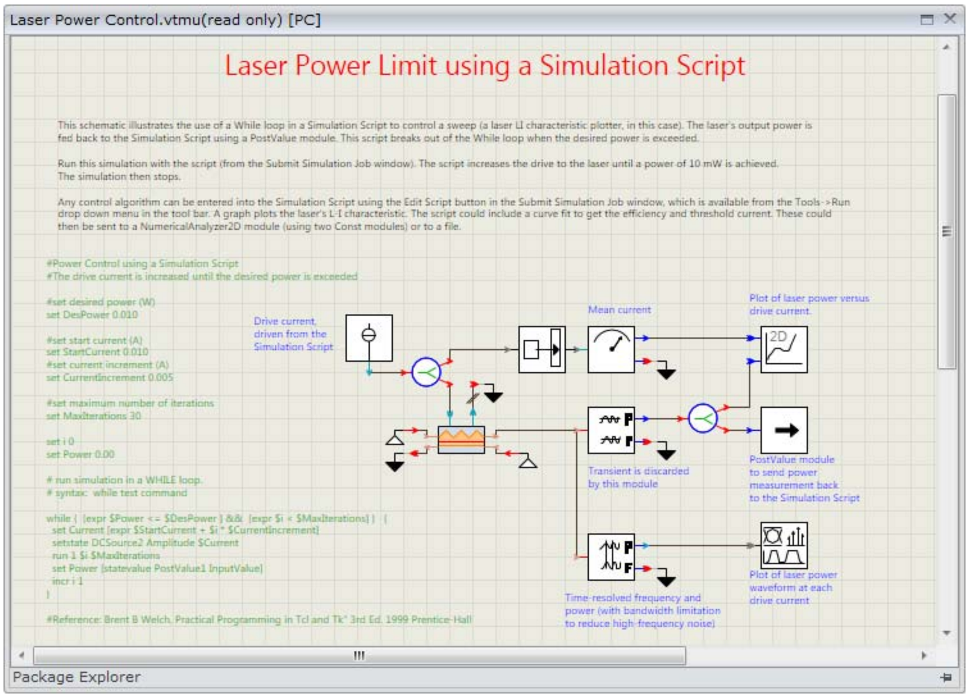
This demonstration illustrates the use of a `while` loop in a simulation script, to control a sweep (a laser LI characteristic plotter, in this case). The laser's output power is fed back to the simulation script using a *PostValue* module. This script breaks out of the `while` loop when the power is exceeded, in this case 10 mW.

The script could be modified to include a curve fit to get the efficiency and threshold current. These could then be sent to a *NumericalAnalyzer1D* module (using two *Const* modules), or to a file.

The schematic is shown in Figure 3-7. The laser is fed from a *DC_Source*, whose output level is set by the simulation script. The output power of the laser is averaged over the second half of the TimeWindow, to discard transients, using the *FreqPowerMeter* module. The averaged output power is plotted against the drive current on a *NumericalAnalyzer2D* module. The power output is also sent to *PostValue* module, that feeds this power measurement to the simulation script.

The simulation script runs the simulation within a `while` loop. The value of the output power is monitored after each run, and used as the test condition for the `while` loop. The drive current to the laser is incremented for each run of the `while` loop.

```
#Power Control using a Simulation Script
#The drive current is increased until the desired power is exceeded

#set desired power (W)
set DesPower 0.010
#set start current (A)
set StartCurrent 0.010
#set current increment (A)
set CurrentIncrement 0.005
#set maximum number of iterations
set MaxIterations 30
set i 0
set Power 0.00

# run simulation in a WHILE loop.
# syntax:  while test command

while {  [expr $Power <= $DesPower ] &&  [expr $i < $MaxIterations] }   {
  set Current [expr $StartCurrent + $i * $CurrentIncrement]
  setstate DCSource2 Amplitude $Current
  run
  set Power [statevalue PostValue1 InputValue]
  incr i 1
}

#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999
Prentice-Hall
```



Laser Power Control.vtmu(read only) [PC]

Laser Power Limit using a Simulation Script

This schematic illustrates the use of a While loop in a Simulation Script to control a sweep (a laser LI characteristic plotter, in this case). The laser's output power is fed back to the Simulation Script using a PostValue module. This script breaks out of the While loop when the desired power is exceeded.

Run this simulation with the script (from the Submit Simulation Job window). The script increases the drive to the laser until a power of 10 mW is achieved. The simulation then stops.

Any control algorithm can be entered into the Simulation Script using the Edit Script button in the Submit Simulation Job window, which is available from the Tools->Run drop down menu in the tool bar. A graph plots the laser's L-I characteristic. The script could include a curve fit to get the efficiency and threshold current. These could then be sent to a NumericalAnalyzer2D module (using two Const modules) or to a file.

```
#Power Control using a Simulation Script
#The drive current is increased until the desired power is exceeded

#set desired power (W)
set DesPower 0.010

#set start current (A)
set StartCurrent 0.010
#set current increment (A)
set CurrentIncrement 0.005

#set maximum number of iterations
set MaxIterations 30

set i 0
set Power 0.00

# run simulation in a WHILE loop.
# syntax:  while test command

while { [expr $Power <= $DesPower ] && [expr $i < $MaxIterations] } {
 set Current [expr $StartCurrent + $i * $CurrentIncrement]
 setstate DCSource2 Amplitude $Current
 run 1 $i $MaxIterations
 set Power [statevalue PostValue1 InputValue]
 incr i 1
}

#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999 Prentice-Hall
```

Drive current, driven from the Simulation Script
Mean current
Plot of laser power versus drive current.
2D
Transient is discarded by this module
PostValue module to send power measurement back to the Simulation Script
Time-resolved frequency and power (with bandwidth limitation to reduce high-frequency noise)
Plot of laser power waveform at each drive current
Package Explorer

Figure 3-7 *Schematic demonstrating a laser power limit using a WHILE loop*

Laser Tuning Control

Photonic Circuits Demos > Laser Characterization > Laser Tuning Control

Note: This demonstration works with the VPIcomponentMaker Photonic Circuits product.

Because a DFB laser emits on one side of the grating's stopband, a control loop is needed to find the grating wavelength parameter that gives the desired lasing frequency. This tuning of the Bragg grating would be achieved thermally in reality.

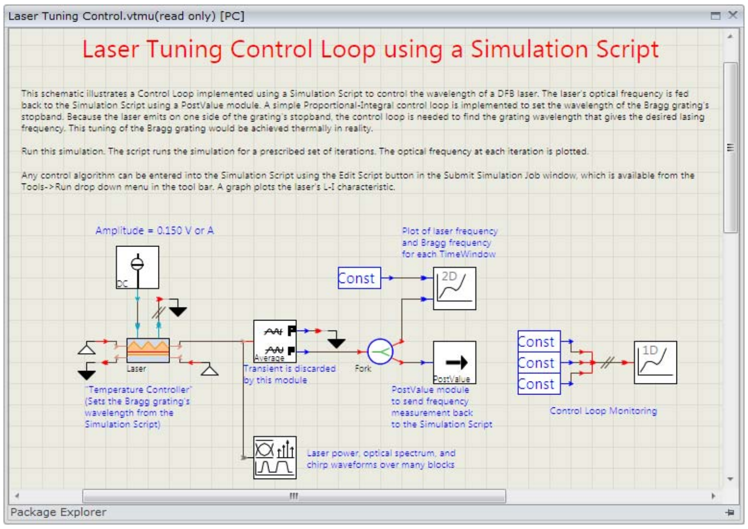
This schematic illustrates a control loop implemented using a simulation script, to control the wavelength of a DFB laser. The laser's optical frequency is fed back to the simulation script using a *PostValue* module. A simple proportional-integral control loop is implemented to set the wavelength of the Bragg grating's stopband.

The schematic is shown in Figure 3-8. The simulation script runs the simulation for a prescribed set of iterations. The optical frequency at each iteration is plotted. Any control algorithm can be entered into the simulation script using the **Edit Script** button in the **Submit Simulation Job** window. The variation of emission frequency with time is plotted on an XY graph. A *NumericalAnalyzer1D* module provides the internal states of the control algorithm. Note how *PostValue* modules have been used to send the values of global variables to the simulation script.

The simulation script is reproduced below. The desired lasing frequency is specified in the fifth line, and the maximum number of iterations is also defined. The control algorithm is then defined as Proportional and Integral gains. The simulation is run once to provide inputs to the control algorithm. The simulation script then enters a `for` loop, sets the value of the **`GratingStopbandOffset`** using the control algorithm, runs the simulation, finds the `statevalue` of the *PostValue* module, plots the emission frequency, and repeats until the `for` loop is completed.

```
#Laser Tuning Control Loop using a Simulation Script
#The Bragg Grating frequency is tuned to obtain the desired Emission Frequency

#set desired laser emission frequency (Hz)
set DesFreq 193.1e12
#set maximum number of iterations
set MaxIterations 20

#set proportional Gain (Bragg Tuning/Error In Frequency)
set PropGain -0.25
#set integral Gain (Bragg Tuning/Error In Frequency)
set IntegGain -0.30
set ThisFreq 0.0
set ErrorFreq 0.0
set IntegratedError 0.0
# run simulation in a WHILE loop until MaxIterations reached.
# syntax:  while test command

#initialize with a single run: set Bragg Grating offset to zero, and graph outputs to
zero
setstate LaserTLM_vtmg1 GratingStopbandFrequency 0.0
setstate Const4 level 0.0
setstate Const5 level 0.0
setstate Const6 level 0.0
setstate Const2 level 0.0
```

```
#get SampleModeCenterFrequency and calculate desired frequency offset
#all calculations are done with offset frequencies for accuracy reasons
set SMCF [statevalue this SampleModeCenterFrequency]
set DesFreqOffset [expr $DesFreq - $SMCF]

#get TimeWindow for graph axes
#set TimeWindow [statevalue PostValue3 InputValue]
set TimeWindow [statevalue this TimeWindow]
#get actual emission frequency of laser after first run
set ThisFreq [statevalue PostValue1 InputValue]

for {set i 1} { $i <= $MaxIterations}  {incr i}  {

       set ErrorFreq [expr  $ThisFreq - $DesFreqOffset]
       set IntegratedError [expr $ErrorFreq + $IntegratedError]
       set BraggFreq [expr $PropGain * $ErrorFreq + $IntegGain * $IntegratedError]

        setstate LaserTLM_vtmg1 GratingStopbandFrequency $BraggFreq

       #monitor graph updates
        setstate Const4 level $ErrorFreq
        setstate Const5 level $IntegratedError
        setstate Const6 level $BraggFreq
        setstate Const2 level [expr $i * $TimeWindow]

        run 1 $i $MaxIterations
        set ThisFreq [statevalue PostValue1 InputValue]

}

#Reference: Brent B Welch, Practical Programming in Tcl and Tk" 3rd Ed. 1999
Prentice-Hall
```



Laser Tuning Control.vtmu(read only) [PC]

Laser Tuning Control Loop using a Simulation Script

This schematic illustrates a Control Loop implemented using a Simulation Script to control the wavelength of a DFB laser. The laser's optical frequency is fed back to the Simulation Script using a PostValue module. A simple Proportional-Integral control loop is implemented to set the wavelength of the Bragg grating's stopband. Because the laser emits on one side of the grating's stopband, the control loop is needed to find the grating wavelength that gives the desired lasing frequency. This tuning of the Bragg grating would be achieved thermally in reality.

Run this simulation. The script runs the simulation for a prescribed set of iterations. The optical frequency at each iteration is plotted.

Any control algorithm can be entered into the Simulation Script using the Edit Script button in the Submit Simulation Job window, which is available from the Tools->Run drop down menu in the tool bar. A graph plots the laser's L-I characteristic.

Amplitude = 0.150 V or A

Plot of laser frequency and Bragg frequency for each TimeWindow

"Temperature Controller" (Sets the Bragg grating's wavelength from the Simulation Script)

Transient is discarded by this module

PostValue module to send frequency measurement back to the Simulation Script

Control Loop Monitoring

Laser power, optical spectrum, and chirp waveforms over many blocks

Figure 3-8 *Schematic demonstrating a control loop implemented as a simulation script*

Demonstrations using Simulation Scripts

For other examples of script usage, see the demonstration folder *Optical Systems Demos > Simulation Techniques > Simulation Scripts* and the following other demos:

- *Optical Systems Demos > Simulation Techniques > Signal Processing Library > Data Post Processing*
- *Fiber Optics Demos > Amplifier Modeling > Black Box Model > Convert to Black-Box*
- *Fiber Optics Demos > Amplifier Modeling > Black Box Model > Verify Black-Box*
- *Optical Systems Demos > Subsystems > Transmitters > Chirp of MZ Modulator*

The *Simulation Scripts > OSNR Pre-Emphasis*, *Convert to Black-Box* and *Verify Black-Box* demonstrations include a Wizard dialog (*Tk*-based GUI elements) that opens when the simulation is started, to provide control and feedback on the simulation's status.

The *Simulation Scripts > Optimization using Bisection Method* demonstration includes another example of implementing GUI elements using *Tk*, including the `MessageBox` command that can be quite useful for debugging purposes.

The *Chirp of MZ Modulator* demonstration includes an example of using simulation scripts and sweeps to implement different modes of simulation in the same setup.

The *Simulation Scripts > BER vs. Laser Power* and *Fiber Optics Demos > Amplifier Modeling > Black Box Model > Convert to Black-Box* demonstrations illustrate the use of the `visualize` command to plot the data from a text file in VPIphotonicsAnalyzer.

Scripting Using Python

As an alternative to *Tcl* scripting, scripts can be written in Python to take advantage of the full power and flexibility provided by this programming language.

Invoking Python in Simulation Scripts

The simulation engine uses the *Tcl* language by default when running simulation scripts.

Simulation scripts can be written in Python using an external editor or an IDE. Python code should be wrapped in the `pythonScript` command as shown in the example below.

```
pythonScript {
import math
x = math.pi
}
```

This command accepts a single argument — the text of the Python script. Multiline scripts should be enclosed in braces or provided by a *Tcl* variable. In most cases, the simulation script will consist of the `pythonScript` command and the Python code itself.

Note: Python uses indentation to define the structure of the program. It is recommended to use an external editor with proper indentation support (instead of the built-in editor) for Python scripts.

Snippets of Python code can also be used to calculate an expression within a *Tcl*-based simulation script, like in the following example:

```
pythonScript {import math}
setstate Const_vtms1 level [pythonExpression {math.pi}]
run 1
```

The *Tcl* command `pythonExpression` estimates its only argument as a Python expression and returns its value to the *Tcl* script.

This command has several important limitations:

- Import statements can not be evaluated, and should be executed within the `pythonScript` command before the `pythonExpression` command. The latter command is primary intended for mixing *Tcl* and Python parts of a complex script;
- The `pythonExpression` command returns a result of type string. Complex structures, like Python lists, are returned using their respective string representations;
- The `pythonExpression` command should not be used in expressions in module parameters.

Debugging Python Simulation Scripts

When developing Python simulation scripts, the Python code wrapped in the `pythonScript` command can import and use external Python modules. It is frequently necessary to debug such Python modules to spot and fix errors of the algorithm. For more detailed information on how to debug Python scripts inside VPI Design Suite, refer to "Debugging Python Code" on page 24.

Simulation Script Commands (Python)

The following simulation script commands are available in Python:

```
isremotesimulation
ptclmessage
setstate
calcstate
statevalue
getpostvalue
useGPU
visualize
visualizedata
run
init
runstep
wrapup
breakstatus
```

```
signaldb
transientCheck
SimCmdWaiting
SimGetCommand
SimPaused
SimResumed
SimDataReturn
SimOutputReturn
SimDataFetch
pathexpand
resetSweep
```

More information on some of the Python commands can be found in the text above where the *Tcl* version of the command is described.

The syntax of Python commands complies with the general syntax of the Python language. For example, the arguments of a function are closed in parentheses, `()`, the arguments are comma delimited, the strings are enclosed in single quotation marks `''`, etc.

To control the simulation, Python simulation scripting provides several functions, which are available in a special Python module: `vpi_ptcl_script`. Normally, the statement `import vpi_ptcl_script` should be present in the beginning of a Python simulation script.

The functions in the `vpi_ptcl_script` module can be divided into two groups:

- Simulation control functions (these are equivalents of the Tcl-based simulation scripting commands as described in under "Simulation Script Commands"), and
- Tcl interoperability functions (provided to call the simulation scripting commands without Python equivalents).

Note: In the sections below, optional parameters are enclosed in square brackets.

The most important commands are described below.

Simulation Control Functions

isremotesimulation

Returns `true` if a simulation is running at the Remote Simulation Server.

`isremotesimulation()`

run

Runs a simulation for the specified number of times, optionally providing sweep (data iteration) information.

`run([num [, iter0, limit0] ...[, iterN, limitN] ...])`

`num`: the number of runs of schematic (default 1). Equivalent to performing a standard simulation with the number of runs set to `num`.

`iter0`: the number of the current sweep iteration on the innermost sweep level (sweep depth is 0), starting from zero.

`limit0`: the maximum number of sweep iterations expected (on sweep depth equal to 0).

`iterN/limitN`: extra optional parameters, specifying the current sweep iteration and the maximal number of iterations for the n-th level of a nested (multidimensional) sweep.

Note: The execution of the `run` command is interrupted if a simulation interruption was requested by the user employing the **Break** action from the **Abort** menu at the ribbon. It means that if the user requests a **Break** action, the current `run` command will return as soon as possible, and all the subsequent executions of the `run` command (if they are expected to be performed) will be skipped.

---

init

Initializes the simulation environment, scheduler, signals, etc., mainly for transient simulations. Like the `run()` function, `init()` needs the sweep (iteration) information.

```
init([num [, iter0, limit0] ...[, iterN, limitN] ...])
```

`num`: the number of runs of schematic (default 1).

`iter0`: the number of the current sweep iteration on the innermost sweep level (the sweep depth is 0), starting from zero.

`limit0`: the maximum number of sweep iterations expected (on the sweep depth equal to 0).

`iterN/limitN`: extra optional parameters, specifying the current sweep iteration and the maximal number of iterations.

Together, three functions, `init()`, `runstep()` and `wrapup()` perform the same operations as the `run()` function.

Refer to Chapter 10 of the *VPItransmissionMaker™Optical Systems User's Manual* and Chapter 7 of the *VPIcomponentMaker™Fiber Optics User's Manual* for more details and code examples of transient simulations.

runstep

Executes one or several runs of the whole schematics without a reinitialization or freeing of signals. Mostly used for transient simulations. Similarly to the `run` command, the execution of the `runstep` command can be interrupted by the **Break** action from the **Abort** menu at the ribbon (see the note on the `run` command for details).

```
runstep([num [, iter0, limit0] ...[, iterN, limitN] ...[, logic]])
```

`num`: the number of runs of the schematic (default 1).

`iter0`: the number of the current sweep iteration on the innermost sweep level (the sweep depth is 0), starting from zero.

`limit0`: the maximum number of sweep iterations expected (on the sweep depth equal to 0).

`iterN/limitN`: extra optional parameters, specifying the current sweep iteration and the maximal number of iterations

`logic`: optional parameter (string, can be '`-iterate_logic`'). If set, forces the simulation engine to accumulate logical information to newly generated signals (similar to multiple iterations). Without this options, the `runstep` command resets the logical information (as in standard sweeps).

Together, three functions, `init()`, `runstep()` and `wrapup()` perform the same operations as the `run()` function.

wrapup

Clears the simulation engine memory and destroys all modules.

```
wrapup()
```

Together, three functions, `init()`, `runstep()` and `wrapup()` perform the same operations as the `run()` function.

To prevent memory leaks, the `init()` function should always be accompanied by a corresponding call of the `wrapup()` function.

setstate

Changes the parameter of a module to a specified value.

```
setstate(module, param, value[, expr_type='standard'])
```

`module`: module ID or the special identifier `this` (for global parameters).

`param`: parameter name.

`value`: new value of the parameter.

`expr_type`: expression type (string '`standard`', '`python`' or '`file`').

Note: The keyword `this` is a reserved keyword for reference to the current topology (universe) in the simulation scripts.

Note: Parameters can accept only finite number values. Trying to assign `Inf`, `-Inf`, or NaN will cause errors.

statevalue

Returns the value of the specified parameter of module or schematic. As a special useful case, the ***PostValue*** module takes its input data and converts it to a parameter value, which can then be read into the script using the `statevalue` function.

`statevalue(module, param[, type])`

> `module`: module ID or the special identifier `this` (for global parameters).
>
> `param`: parameter name.
>
> `type`: optional parameter, (string, can be either '`initial`' or '`current`', the latter is default), specifying the kind of the parameter value to be returned.

The `initial` option should be used in scripts before the `run()` function. The default option is `current`, in which case the function returns the value changed during a simulation (zero or empty string is returned before initialization or simulation run).

The function returns either numerical (float) or a string value.

getpostvalue

Returns the value that was received at the input port of the ***PostValue*** or ***PostValueMxFlt*** module during a simulation run.

`getpostvalue(module)`

> `module`: module ID of the ***PostValue*** or ***PostValueMxFlt*** module

Produces an error if no data was received yet or the wrong module ID was specified.

The function returns either a *scalar (float)* or *NumPy* array value.

calcstate

Sends a request to the simulation engine to recalculate the references to the specified parameters in all modules. The function is intended for schematic parameters (those of universes and galaxies) that are usually have indirect effect: they are used in other parameters of modules via reference.

`calcstate([module,] param)`

> `module`: module ID (can be omitted for global parameters).
>
> `param`: parameter name.

initialize

Initialize the value of a parameter, setting its current value equal to initial (see description of the function `statevalue()` for details). Normally used to access the value of some parameter both before and after the `run` command in an uniform way (for example, in an optimization loop).

`initialize(module, param)`

`module`: module ID or the special identifier `this` (for global parameters).

`param`: parameter name.

breakstatus

Returns a flag that can be analyzed to terminate a simulation or break out of a specific iteration level (in sweeps), as set by the ***Break*** module or at the user's request.

```
breakstatus()
```

Return values can be

`-2`: termination of the whole simulation is requested by the user.

`-1`: termination of the whole simulation is requested by the ***Break*** module.

`0`: no simulation break is requested.

`n`: (positive integer) indicates the level of the sweep iterations for break out.

Caution: The sweep levels (depth) are numbered starting from zero in the VPI Design Suite user interface and modules like ***Break*** or ***Ramp***. However, the `breakstatus()` function returns the iteration level to be terminated starting from 1. So if the **`BreakSweepDepth`** parameter is set to 2, the `breakstatus()` will return 3 when the break condition is fulfilled.

ptclmessage

Displays a warning (or progress) message in the VPI Design Suite **Message Log**.

```
ptclmessage(text[, type])
```

`text`: contains the text string to be displayed.

`type`: optional parameter (string, can be either '`warning`', '`info`' or '`progress`') defines whether the message will be displayed as warning (default), informational or progress message.

Note that progress messages are displayed only if the **Report progress** checkbox is selected in the **Submit Simulation Job** dialog.

resetSweep

Resets the tracking of sweep level information between runs.

```
resetSweep()
```

visualize

Visualizes a file with data using VPIphotonicsAnalyzer. Either a plain-text file or a `.vpa` file with the saved results can be visualized by this function.

`visualize(datafile)`

`datafile`: full path to file with data to be displayed. For details, see "Visualizing Text Files" in the *VPIphotonics Design Suite™ User Interface Reference*.

Note: This command is not available for server simulations.

visualizedata

Visualize the calculated data in the VPIphotonicsAnalyzer (1D, 2D, or 3D).

`visualizedata(window_id, data1[, data2[, data3[, ...]]][, options=''][, sweepinfo=None])`

`window_id`: specifies the ID of the window in VPIphotonicsAnalyzer where the data will be shown. It also determines the default title of the analyzer window, which can be changed by specifying the desired title in `options`.

`data1`, `data2`, `...`: one or several datasets, each of which will be shown as a separate input in the analyzer window.

`options`: used to specify the plot options like window title, trace legend, trace color, etc. The syntax of the options string is described in the documentation for the *SignalAnalyzer* module.

`sweepinfo`: used to specify the sweep information that the visualized dataset belongs to (number of the run, iteration, or sweep point). If some `sweepinfo` is specified, the sweep information attached to the data with the `run` command is ignored.

The visualized data can be one of three types, 1D, 2D, or 3D, and should be a two-dimensional *NumPy* array with the corresponding shape:

- For a 1D chart, N x 1 or simply a one-dimensional array.
- For a 2D chart, N x 2 array.
- For a 3D chart, N x 3 array.
  Note, that only a single dataset can be visualized by a 3D numerical analyzer.

Different datasets in one chart do not necessarily need to be of the same length.

When the command `visualizedata` is used inside the cosimulation module (not in the simulation script) the `sweepinfo` option must not be used, as the full sweep information is already provided inside the cosimulation module and will automatically be shown in VPIphotonicsAnalyzer.

The `sweepinfo` value should be a Python dictionary. The most general form is as follows:

```
sweepinfo={
        'Iteration': [CurrentIteration, TotalIterations],
        'Run': [CurrentRun, TotalRuns],
        NameSweep0: [LevelSw0, CurrentSw0, TotalSw0, ValueSw0],
        NameSweep1: [LevelSw0, CurrentSw1, TotalSw1, ValueSw1],
        …
}
```

The value for the `Run` and `Iteration` keys is a list that consists of 2 elements. The 1st element is the number of the current run or iteration (starting from 1), and the 2nd element is the total number of runs or iterations. If the total number of runs or iterations is unknown then the current number of the run or iteration can be used as the 2nd element.

The value for the *`NameSweepN`* key is a list that consists of 4 elements. The 1st element is the level of the sweep (starting from 0), and the 2nd element is the number of the current sweep point (starting from 1). The 3rd element is the total number of sweep points. The 4th element is the current value of the swept variable (for example, `193.1e12` if the frequency variable is swept). The 4th element is optional. If the total number of sweep points is unknown the current number of the sweep point can be used as the 3rd element.

The key *`NameSweepN`* is used as the name of the swept variable (in the VPIphotonicsAnalyzer the name of the swept variable is shown in **Data Ordering** controls **Connect** and **Differ by Color** and in tool-tips of **Iteration Selector**).

Here is an example of the usage of the `visualizedata` command:

```
pythonScript {
import numpy as np
import vpi_ptcl_script as scr

D2_1 = np.array([[1, 10], [2, 20], [3, 30], [4, 40]])
D2_2 = np.array([[1, 20], [2, 30], [3, 40], [4, 50]])
scr.visualizedata("Test2D", D2_1, D2_2,
   options='legend1="RZ" legend2="NRZ" title="Test 2D"',
   sweepinfo={'Iteration': [2,3], 'CenterFreq' [0,1,4,2e-3], 'Power': [2,3,7]})
}
```

When several datasets need to be plotted in the same window, we recommend using only one `visualizedata` command, rather than multiple commands for each dataset. This prevents from the necessity to distinguish between iterations produced by multiple `visualizedata` commands and iterations produced by the cosimulation module forced by e.g. ***IteratorFixed*** or ***Chop*** modules.

To combine several datasets for single `visualizedata` command, the Python list unpacking can be used:

```
pythonScript {
import numpy as np
import vpi_ptcl_script as scr

D2_1 = np.array([[1, 10], [2, 20], [3, 30], [4, 40]])
D2_2 = np.array([[1, 20], [2, 30], [3, 40], [4, 50]])
D2_N = np.array([[1, 30], [2, 40], [3, 50], [4, 60]])

lst = []
lst.append(D2_1)
lst.append(D2_2)
# ...
# Append any required number of datasets here.
# Use loops or any other constructs if necessary.
# ...
lst.append(D2_N)

scr.visualizedata("Test2D", *lst)
}
```

SimOutputReturn

Unlike Tcl, where two commands, `SimOutputReturn` and `SimOutputReturnEx`, are implemented, in Python there is a single command `SimOutputReturn`, which provides combined functionality. It accepts a list of lists. If an inner list has three elements, they are interpreted as follows:

- **i.** key name in `SimulationData` collection;
- **ii.** value type (use string constants from the `vpi_ptcl_script` module: `FloatType`, `FloatMatrixType`);
- **iii.** value.

```
pythonScript {
import numpy
import vpi_ptcl_script as ptcl

data = ptcl.SimDataFetch()
ptcl.run(1)
ptcl.SimOutputReturn([
    ("floatPar1, ptcl.FloatType, data["top"]),
    ("floatPar2, ptcl.FloatType, ptcl.getpostvalue("PostValue_vtms1")),
    ("PostValueMxFloat_vtms1", ptcl.FloatMatrixType,
ptcl.getpostvalue("PostValueMxFlt_vtms1")),
    ("PostValueMxFloat_vtms2", ptcl.FloatMatrixType, numpy.array([[1, -2], [2, -2],
[5, 7]]))
])
}
```

If an inner list has two elements, they are interpreted as:

- **i.** key name;
- **ii.** ID of *PostValue* module.

```
pythonScript {
import vpi_ptcl_script as ptcl

ptcl.run(1)
ptcl.SimOutputReturn([
    ("PostValue_vtms1", "PostValue_vtms1"),
    ("MatrixPar", "PostValueMxFlt_vtms1")
])
}
```

If an inner list has only one element, it is interpreted as the ID of *PostValue*/*PostValueMxFlt* modules and as a key name.

```
pythonScript {
import vpi_ptcl_script as ptcl

ptcl.run(1)
ptcl.SimOutputReturn([
    ("PostValue_vtms1",),
    ("PostValueMxFlt_vtms1",),
])
}
```

All the above approaches can be used at the same time:

```
pythonScript {
import vpi_ptcl_script as ptcl

ptcl.run(1)
ptcl.SimOutputReturn([
    ("PostValue_vtms1",),
    ("floatPar", "PostValue_vtms1"),
    ("PostValueMxFloat_vtms1", ptcl.FloatMatrixType,
ptcl.getpostvalue("PostValueMxFlt_vtms1")),
])
}
```

It is also possible to use old syntax, similar to the Tcl `SimOutputReturn` command, and pass a dictionary.

```
pythonScript {
import vpi_ptcl_script as ptcl

ptcl.run(1)

ptcl.SimOutputReturn({"PostValueModule" : ptcl.statevalue("PostValueModule",
"InputValue")})
}
```

Tcl Interoperability Functions

The functions below support mixed-style scripting, allowing to combine complex scripts using Python and *Tcl* parts, which is useful to call old scripting commands without Python equivalents.

tcl_expr

Evaluates an expression using *Tcl* interpreter and returns its result to Python.

`tcl_expr(expression)`

`expression`: string containing the expression.

In the present implementation, the `tcl_expr()` function prepends its argument with the *Tcl* `expr` command.

Returns either a numerical (float) or a string result.

tcl_script

Executes a script using *Tcl* interpreter.

`tcl_script(script)`

`script`: a string containing the script.

Example

Optical Systems Demos > Simulation Techniques > Other Sweep Techniques > Rosenbrock Function Minimization

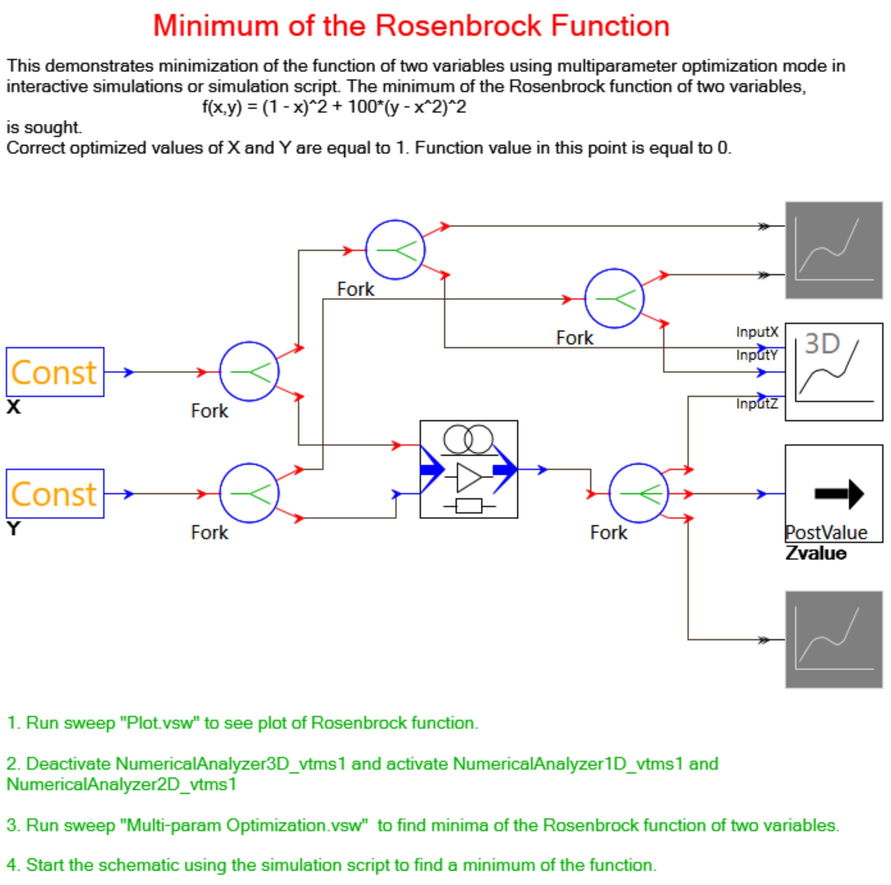
This example demonstrates a two-parameter minimization using the SciPy library [1]. The simulation setup, as presented in Figure 3-9, is typical for this class of simulations. It

- takes inputs (two ***Const*** modules on the left),
- contains a system/device under test (a galaxy in the middle),
- provides the resulting value using the ***PostValue*** module, and
- optionally contains analyzer modules to visualize either a target function (***NumericalAnalyzer3D***, activated by an attached parameter sweep) or the values of independent variables in the course of optimization (***NumericalAnalyzer2D***).

In the demo, the minimum of the Rosenbrock function [2] of two variables,

$$f(x, y) = (1 - x)^2 + 100 \cdot (y - x^2)^2$$

is sought. A galaxy for calculation of the Rosenbrock function is shown in Figure 3-9.


Minimum of the Rosenbrock Function

This demonstrates minimization of the function of two variables using multiparameter optimization mode in interactive simulations or simulation script. The minimum of the Rosenbrock function of two variables,
f(x,y) = (1 - x)^2 + 100*(y - x^2)^2
is sought.
Correct optimized values of X and Y are equal to 1. Function value in this point is equal to 0.

1. Run sweep "Plot.vsw" to see plot of Rosenbrock function.

2. Deactivate NumericalAnalyzer3D_vtms1 and activate NumericalAnalyzer1D_vtms1 and NumericalAnalyzer2D_vtms1

3. Run sweep "Multi-param Optimization.vsw" to find minima of the Rosenbrock function of two variables.

4. Start the schematic using the simulation script to find a minimum of the function.

Figure 3-9 *Example schematic for a two-parameter function minimization*

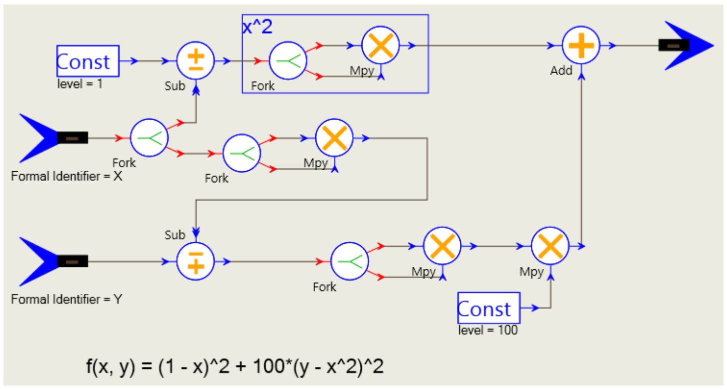


Figure 3-10 *Galaxy to calculate the Rosenbrock function*

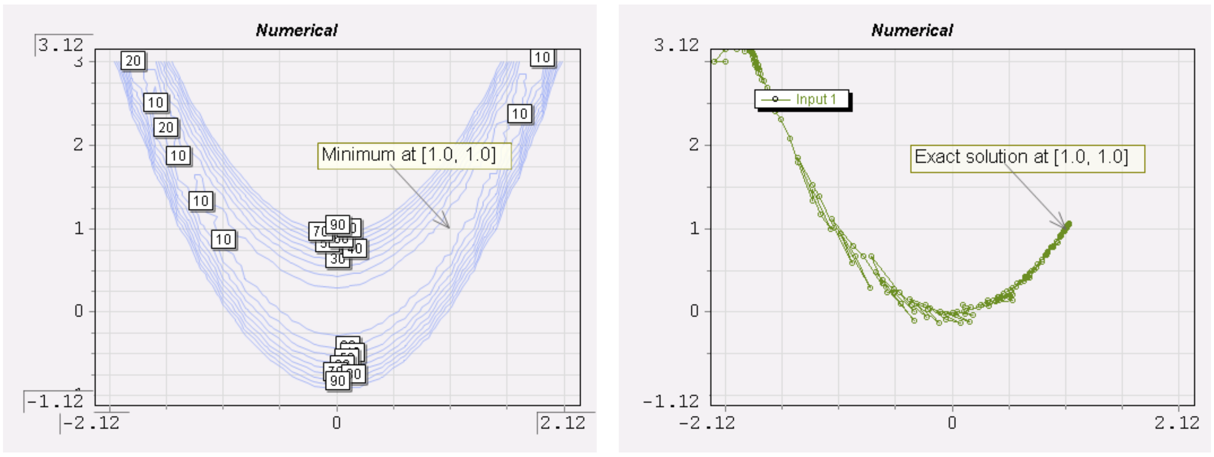


Figure 3-11 *Contour plot of the Rosenbrock function (left) and the series of iterations of the function minimization (right)*

This function belongs to a class of rather complicated optimization problems: the true minimum is in the middle of a narrow curved ‘valley’ (see Figure 3-11, left), and some optimization algorithms are either unstable or require a large number of iterations [1].

The simulation script is presented in Figure 3-12. It consists of the following parts:

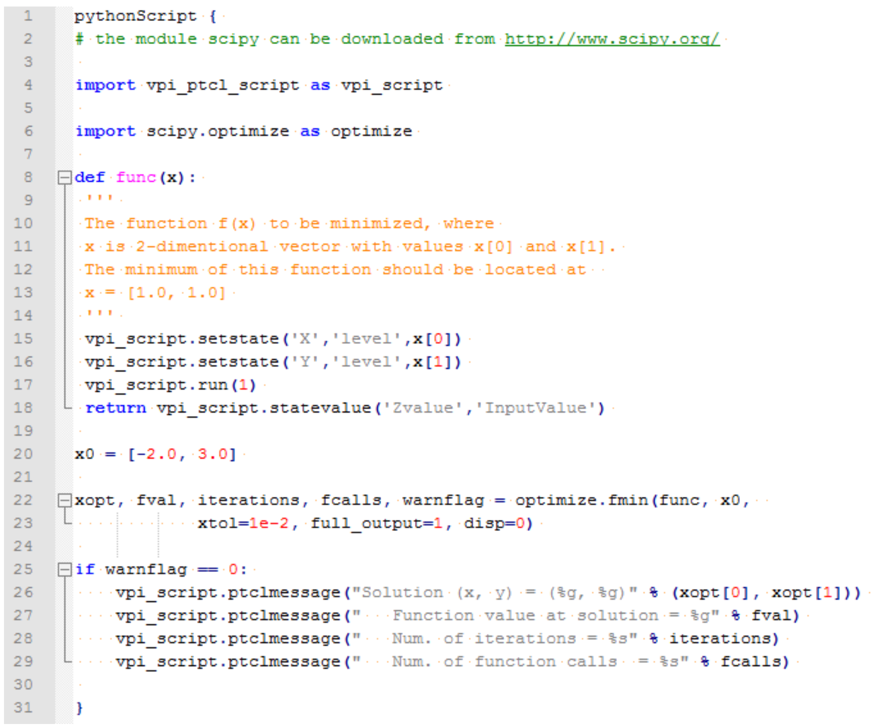
- At the beginning, a `pythonScript` command that switches from the *Tcl* to Python scripting.
- Several `import` statements that make the necessary libraries (`vpi_ptcl_script` and `optimize`) available in the script.

Note: The script first attempts to use the optimization package from the SciPy library (included in the distribution).

- Definition of the `func()` function to be minimized. This function sets the level parameters of two *Const* modules as specified by its input, runs the schematic, and then returns the value taken from the *PostValue* module.
- Specification of the initial guess for independent parameters of the minimization algorithm.

Note: Multidimensional parameters are passed as the respective list components.

- The optimization itself is performed by the `fmin()` function from the optimize package, using the downhill Nelder-Mead simplex algorithm. For details of the optimization algorithm, please consult the SciPy documentation[1].
- In case of successful optimization, the script prints the solution and diagnostic information provided by the `fmin()` function as shown in Figure 3-13. A sequence of trials converging to a minimum is depicted in Figure 3-11 (right).



```
pythonScript {
# the module scipy can be downloaded from http://www.scipy.org/

import vpi_ptcl_script as vpi_script

import scipy.optimize as optimize

def func(x):
 '''
 The function f(x) to be minimized, where
 x is 2-dimentional vector with values x[0] and x[1].
 The minimum of this function should be located at
 x = [1.0, 1.0]
 '''
 vpi_script.setstate('X','level',x[0])
 vpi_script.setstate('Y','level',x[1])
 vpi_script.run(1)
 return vpi_script.statevalue('Zvalue','InputValue')

x0 = [-2.0, 3.0]

xopt, fval, iterations, fcalls, warnflag = optimize.fmin(func, x0,
                xtol=1e-2, full_output=1, disp=0)

if warnflag == 0:
    vpi_script.ptclmessage("Solution (x, y) = (%g, %g)" % (xopt[0], xopt[1]))
    vpi_script.ptclmessage("   Function value at solution = %g" % fval)
    vpi_script.ptclmessage("   Num. of iterations = %s" % iterations)
    vpi_script.ptclmessage("   Num. of function calls  = %s" % fcalls)

}
```

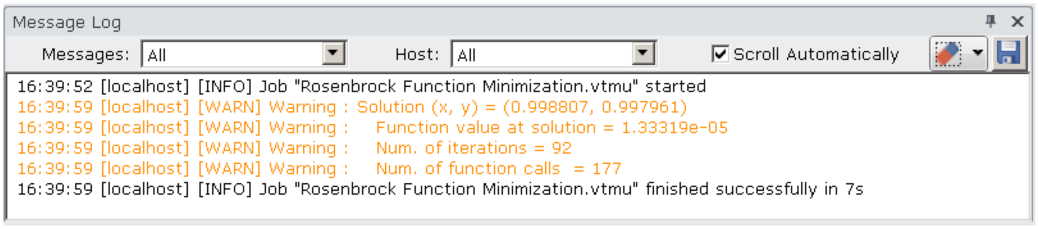
Figure 3-12 *Sample script for the function minimization using Python*

Almost the same script can be used to find an optimal operation conditions of the communication link model.

1. http://docs.scipy.org/doc/scipy/reference/

See the following demos for more details:

- *Optical Systems Demos > Simulation Techniques > Other Sweep Techniques > DPSK-3ASK Optimization*
- *Optical Systems Demos > Simulation Techniques > Other Sweep Techniques > BER Optimization in DPSK System*



```
16:39:52 [localhost] [INFO] Job "Rosenbrock Function Minimization.vtmu" started
16:39:59 [localhost] [WARN] Warning : Solution (x, y) = (0.998807, 0.997961)
16:39:59 [localhost] [WARN] Warning :    Function value at solution = 1.33319e-05
16:39:59 [localhost] [WARN] Warning :    Num. of iterations = 92
16:39:59 [localhost] [WARN] Warning :    Num. of function calls  = 177
16:39:59 [localhost] [INFO] Job "Rosenbrock Function Minimization.vtmu" finished successfully in 7s
```

Figure 3-13 *Solution and diagnostic output of the optimization routine*

References

[1] "*Python Scientific lecture notes*", V. Haenel, E. Gouillart, and G. Varoquaux, Eds., http://scipy-lectures.github.io, (2012)

[2] H. H. Rosenbrock, "*An automatic method for finding the greatest or least value of a function*", Computer Journal, Vol. 3, pp. 175-184 (1960)

Chapter 4

Simulation Engine Driver

VPIphotonics Design Suite™ combines a powerful graphical interface, a sophisticated and robust simulation engine and flexible signal representations to enable efficient modeling of transmission systems, photonic circuits, and advanced semiconductor optical amplifier and laser applications.

At the core of this system is the Ptolemy-based simulation engine, which processes input data, performs the necessary calculations, and prepares output data for interpretation.

While the VPI Design Suite graphical interface provides a convenient means of preparing simulation input in most scenarios, certain applications may require a more automated approach, or the integration of third-party software to enable additional processing options or input sources.

VPI Design Suite includes a dedicated Simulation Engine Driver component, which provides access to the simulation engine for external systems and third-party tools.

This allows simulations to be run and operated without the VPI Design Suite graphical interface, in scenarios such as:

- batch simulations where input is produced by an external system,
- interactive simulations with custom user interfaces,
- simulations controlled from third-party applications such as Microsoft Excel,
- multidimensional optimizations with third-party tools such as MATLAB,
- simulations in third-party tools where a VPI Design Suite simulation acts as a regular module, or
- specialized tools which use a VPI Design Suite simulation to calculate results.

Features

The Simulation Engine Driver can perform the following actions:

- Read and parse simulation scripts (for more information, see Chapter 3, “Simulation Scripts”).
- Read and/or modify the values and expression type of global parameters and module parameters in schematics.
- Run simulations on the local computer.
- Receive scalar numerical values (from modules such as *PostValue*) or matrix numerical values (from modules such as *PostValueMxFlt*) from simulations.
- Receive notifications about simulation engine state changes.
- Send flow control commands which influence simulation script execution.

Simulation Workflow

Figure 4-1 shows the simulation workflow using the Simulation Engine Driver.

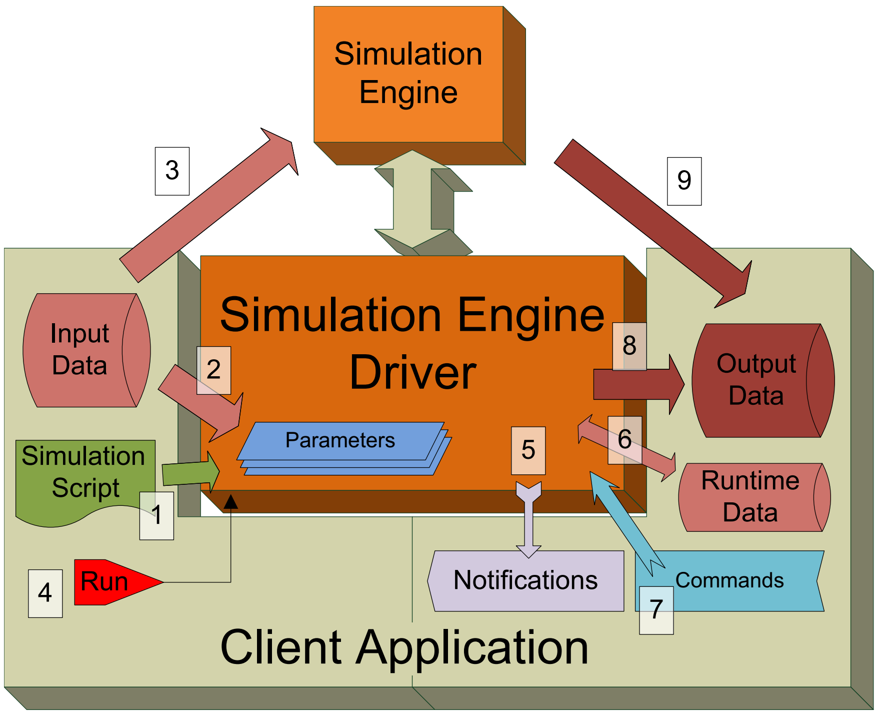


Figure 4-1 *Simulation Engine Driver workflow*

During this workflow, the Simulation Engine Driver progresses through the following steps:

1. Provide a simulation script.
2. Set any parameters whose values should differ from those in the simulation script.
3. Prepare input data files (for example, put an ASCII file with input data into the working folder).
4. Start the simulation.
5. Receive and handle any notifications about simulation state transitions.
6. Send and receive numerical values to and from the simulation.
7. Send commands which influence simulation script execution.
8. Receive output data (scalar or matrix numerical values).
9. Process the output data files that were produced during the simulation.

Using the Simulation Engine Driver

The Simulation Engine Driver is a .NET component with a fairly simple Application Programming Interface (API). It serves as a COM server and provides a COM Automation API which allows it to be used in virtually any modern programming language.

Note: For more information, see Appendix C, "Simulation Engine Driver — API Reference".

In the sections below, all examples are provided in both MATLAB and Python variants. The MATLAB examples which perform optimization require the `Optimization` toolbox.

COM Automation

COM Automation (previously called OLE Automation) is a technology mainly intended for use by scripting languages. It enables client applications (called "automation controllers" or "clients") to access and manipulate (i.e., set properties of or call methods on) third-party components (referred to as "automation objects" or "servers").

Automation is based on the Microsoft Component Object Model (COM), which allows software components to communicate. COM can be used to create reusable software components, link components together to build applications, and take advantage of Windows services.

COM is used in applications such as the Microsoft Office Family of products. For example COM OLE technology allows Word documents to dynamically link to data in Excel spreadsheets and COM Automation allows users to build scripts in their applications to perform repetitive tasks or control one application from another.

COM Automation in MATLAB

MATLAB supports COM Automation out of the box. There are several functions to work with COM Automation objects:

- `actxserver(ProgID)` – creates a COM Automation object via a unique programmatic identifier known as a ProgID
- `object.invoke` – returns a structure with a list of all methods and their signatures defined in a COM Automation object
- `object.invoke('methodname', arg1, arg2, ...)` – calls a method by name
- `object.get` – returns a structure representing all properties and their values
- `object.get('propertyname')` – returns the value of a property
- `object.Method1(arg1, arg2, ...)` – calls a method `Method1` on an object
- `object.Prop1` – accesses the value of an object's property `Prop1`
- `object.delete` – destroys a COM object

Note: Refer to the MATLAB documentation for more information on COM Automation support.

COM Automation in Python

Python has no built-in support of COM Automation, but it can be added with the free `Python for Windows extensions` library. During application installation, all necessary Python packages are installed and special wrappers for the Simulation Engine Driver are generated automatically. Wrappers enable more type-safe access to objects and more elaborate error reporting. Moreover, access to several constants, such as event types and simulation engine statuses, are generated in wrappers (see "Pull Approach" on page 328).

Note: When the Simulation Engine Driver is called from a Python script and the current directory is set to the Python folder, simulations that use Python cosimulation will fail. To resolve this issue, please change the current working directory.

SimulationEngineDriver Object

The `SimulationEngineDriver` object is an entry point for the Simulation Engine Driver component. It provides a means to set up a simulation, access collections of parameters, runtime and output data, run the simulation, send and receive runtime and output data, handle events and send commands to the simulation engine.

Creation

The `SimulationEngineDriver` object can be created in the same manner as any other COM object – using the programmatic identifier or ProgID:

- MATLAB:

```
driver = actxserver('VPI.TC.SED.SimulationEngineDriver');
```

- Python:

```
import win32com.client
driver = win32com.client.Dispatch( "VPI.TC.SED.SimulationEngineDriver")
```

There are two ProgIDs for the `SimulationEngineDriver` object. The first ProgID is version-independent and corresponds to the version of the Simulation Engine Driver component installed last. There is only one version-independent ProgID:

```
VPI.TC.SED.SimulationEngineDriver
```

The second ProgID is used to address the Simulation Engine Driver component from a specific VPI Design Suite release. It begins like the version-independent ProgID, with an additional period and version number appended to the end:

```
VPI.TC.SED.SimulationEngineDriver.11.5
```

The version-specific ProgID should be used in programs which are designed and tested for use with a specific version of VPI Design Suite.

Opening a Package

Prior to running a simulation, the `SimulationEngineDriver` object should be set up. This process implies (at least) opening a predefined design package (`.vtmu`), VPI Module Interchange file (`.vmi`), specialized Simulation Engine Driver package (`.sed`), or raw simulation package:

- MATLAB:

```
% Load design package
driver.OpenPackage
('c:\Program Files\VPI\VPIdesignSuite 11.5\tooldata\
demos\OS\Simulation Techniques\General\
Resampling Options.vtmu');

% Load .sed package
driver.OpenPackage('c:\Test\Test.sed');

% Load raw package
driver.OpenPackage('c:\Test');
```

- Python:

```
# Load design package
driver.OpenPackage
(r"c:\Program Files\VPI\VPIdesignSuite 11.5\tooldata\
demos\OS\Simulation Techniques\General\
Resampling Options.vtmu")

# Load .sed package
driver.OpenPackage(r"c:\Test\Test.sed")

# Load raw package
```

```
driver.OpenPackage(r"c:\Test")
```

Design packages and VPI Module Interchange files are standard document types used to save and distribute schematics. A raw package is the set of files used directly by the simulation engine to run the simulation. A Simulation Engine Driver package is a raw package stored as a single file and complemented with additional information not available in the raw package (such as paths to VPIphotonicsAnalyzer files and `Inputs` and `Outputs` directories).

Note: The simulation script for design packages and VPI Module Interchange files is created during the execution of the `OpenPackage()` function. If those packages are modified afterwards, any changes to the simulation script will not be applied.

It is best to use a design package if frequent changes to the original schematic are still necessary (for example, while debugging the solution). For all other cases where the Simulation Engine Driver is used, the VPI Module Interchange file format (`.vmi`) is recommended. The single-file `.sed` package is also a good choice. A design package can be exported to a Simulation Engine Driver package using **File > Export > Simulation Engine Driver Package**. While VPI Module Interchange file is a fully functional schematic, a Simulation Engine Driver package doesn't contain design information; however, it will contain all the necessary resources and inputs.

Design packages and raw simulation packages are opened in place, and others are unpacked to a temporary location. In all cases, the `PackageDir` property is set to point to the location where the working copy resides.

The `VpaPath` property of the `SimulationEngineDriver` object is set to point to the location where the VPIphotonicsAnalyzer file resides. Design packages and VPI Module Interchange files store VPIphotonicsAnalyzer files in a predefined location. Simulation Engine Driver packages contain this information explicitly. For raw simulation packages, the first VPIphotonicsAnalyzer file found in the root directory of the package is used or this property is left empty if no such files exist.

Two more properties `SchematicInputsDir` and `SchematicOutputsDir` are set to point to the `Inputs` and `Outputs` folder of the schematic. These are left empty for raw simulation packages.

Setting up a Simulation Script Directly

The `SimulationEngineDriver` object can also be set up by specifying a simulation script directly, as a path to a file on disk or as a simple string:

- MATLAB:

```
% Load from file
driver.SimulationScriptPath = 'c:\Test\ptds.pt';

% or specify as string (actual script if omitted for brevity)
driver.SimulationScript = '...'
```

- Python:

```
# Load from file
driver.SimulationScriptPath = r"c:\Test\ptds.pt"

# or specify as string (actual script if omitted for brevity)
driver.SimulationScript = "..."
```

In addition to specifying a simulation script, it is also possible to specify a working directory (which will be used as a base directory for all input and output files consumed or produced by the simulation) and/or set the path to a VPIphotonicsAnalyzer file (`.vpa`):

- MATLAB:

```
driver.WorkingDir = 'c:\Test'
driver.SimulationScriptPath = 'ptds.pt';
driver.VpaPath = 'Test.vpa'
```

- Python:

```
driver.WorkingDir = r"c:\Test"
driver.SimulationScriptPath = "ptds.pt"
driver.VpaPath = "Test.vpa"
```

Setting up a simulation script directly is less convenient in cases when a package is available. However, it may come in handy when the script is generated by some custom tools or libraries.

Setting Parameters

In many cases, it makes sense to alter parameter settings before simulation (either global parameters or the parameters of a certain module). This can be done through objects accessible from the collection returned by the `Parameters` property:

- MATLAB:

```
% Get global parameter TimeWindow
param = driver.Parameters.Item('', 'TimeWindow');
fprintf('%s.%s = %s (%s)\n', param.InstanceId, param.Name, param.Value,
param.ExprType);
% Change value of module’s parameter
param = driver.Parameters.Item('Const_vtms1', 'level');
param.Value = 15;
param.ExprType = 'ParameterExpressionType_Python';
```

- Python:

```
# Get parameter level from module with instance ID Const_vtms1
param = driver.Parameters("Const_vtms1", "level")
print("%s.%s = %s (%s)" % (param.InstanceId, param.Name, param.Value,
param.ExprType))
# Change value of module’s parameter
driver.Parameters('Const_vtms1', 'level').Value = 15
param.ExprType = win32com.client.constants.ParameterExpressionType_Python
```

Note: Similarly to simulation scripts, only ‘native’ units (mostly MKS) of respective parameters are supported. The System of Units preference as well the units settings in schematics are ignored.

Note: Parameters can accept only finite number values. Trying to assign `Inf`, `-Inf`, or `NaN` will cause errors.

64-bit Support

The Simulation Engine Driver can be used only from 64-bit processes (for example MATLAB 64-bit).

Running the Simulation

Simulations can be run in one of two modes: synchronous or asynchronous.

In synchronous simulations, the execution of client code is blocked until the simulation is finished. This mode is very convenient for simple cases where no event handling is required.

Asynchronous mode doesn't block execution of client code and performs simulation in parallel with client code execution. This mode is mostly useful for complex scenarios such as advanced sweeps (see the example in "Multidimensional Optimization" on page 332) or when it is not desirable to block the main execution thread (such as nonbatch applications with a user interface).

To run a simulation in synchronous mode:

- MATLAB:

  ```
  driver.Run;
  ```

- Python:

  ```
  driver.Run()
  ```

To run a simulation in asynchronous mode:

- MATLAB:

  ```
  driver.BeginRun;
  ```

- Python:

  ```
  driver.BeginRun()
  ```

In synchronous mode, any runtime errors of the simulation are reported immediately. In asynchronous mode, error information is accessed via an additional method:

- MATLAB:

  ```
  driver.EndRun(-1);
  ```

- Python:

  ```
  driver.EndRun()
  ```

This is a blocking method that waits until simulation is finished. Once simulation is complete, any errors are reported.

Licensing

The Simulation Engine Driver has several licensing implications. A license is acquired each time a simulation is started, and released when the simulation finishes. The Simulation Engine Driver uses a special set of licenses and shares simulation licenses with VPI Design Suite. So if there are not enough simulation licenses, it may not be possible to run the VPI Design Suite Photonic Design Environment and the Simulation Engine Driver simulation simultaneously.

The license server must be running and reachable for the entire duration of the simulation or the simulation will be aborted.

Implicit license handling is perfectly acceptable in most cases. But sometimes license acquisition/release may bring significant overhead when running VPIlicenseServer on a remote computer. To address these issues, explicit license handling can be used instead:

- MATLAB:

  ```
  % Acquire licenses only once
  driver.CheckoutLicense

  % Run simulation many times
  for i = 1:10
        driver.Run
  end

  % Release licenses at the end
  driver.CheckinLicense
  ```

- Python:

  ```
  # Acquire licenses only once
  driver.CheckoutLicense()

  # Run simulation many times
  for i in range(10):
        driver.Run()

  # Release licenses at the end
  driver.CheckinLicense()
  ```

In addition to the `CheckinLicense()` calls, the license is automatically released when the simulation script is changed, another package is opened, the current package is closed or when the `SimulationEngineDriver` object is destroyed.

Working with Collections

The Simulation Engine Driver provides access to the following collections:

- Collection of parameters (`Parameters`)
- Collections of runtime data (`InputRuntimeData and OutputRuntimeData`)
- Collection of output values (`OutputData`)

For example, to access a collection of parameters:

- MATLAB:

  ```
  parameters = driver.Parameters;
  ```

- Python:

  ```
  parameters = driver.Parameters
  ```

Note: For more information on collections, see “Collection Objects” on page 323.

Handling Input and Output Files

Input and output files may be very useful in many scenarios. This approach may be used to enable interaction with external systems that generate files that are consumed by the simulation or vise versa.

Both absolute and relative paths may be specified for parameters of *Output file/Input file, Output directory/Input directory* and *Output file array/Input file array* types. Relative paths can be specified both in galaxies and at the schematic level. But usually it is more convenient to work with relative paths at the schematic level because they can be configured for a particular schematic independently (e.g., schematics don't influence each other via galaxies) and such setups can be transferred from computer to computer without changes.

If a simulation script is set up manually, it is necessary to manually determine which folders in the raw simulation package correspond to the schematic or specific galaxy.

For all single file package types, the `OpenPackage()` call may be used to determine the location of the schematic's `Inputs` and `Outputs` folders using the `SchematicInputsDir` and `SchematicOutputsDir` properties of the `SimulationEngineDriver` object:

- MATLAB:

```
% Write data to file in schematic Inputs folder
file_name = fullfile(driver.SchematicInputsDir, 'data.dat');
fid = fopen(file_name, 'w');
for i = 1:10
      fprintf(fid, '%f\n', y);
end
fclose(fid);

% Run simulation which consumes file from Inputs folder
driver.Run
```

- Python:

```
# Write data to file in schematic Inputs folder
file_name = os.path.join(driver.SchematicInputsDir, "data.dat")
file = open(file_name, "w")
for i in range(10):
    file.write(str(i))
file.flush()

# Run simulation which consumes file from Inputs folder
driver.Run()
```

When using single file package types, input files are copied to the temporary location before the simulation and output files copied back after the simulation. Also, the `WorkingDir` property is set to point to the temporary location during simulation and restored afterwards.

Handling Output and Runtime Data

Runtime data is split into two instances: Input and Output. Input runtime data can be sent to the simulation engine and output data is received from the simulation engine.

Input runtime data is accessible through the `InputRuntimeData` property. This collection can be modified to prepare data for sending. Values should be explicitly sent to the simulation engine using the `SendRuntimeData` method of the `SimulationEngineDriver` object afterwards. On the simulation engine side, the input runtime data is accessed using the simulation scripting command `SimDataFetch`.

Output runtime data is accessible through the `OutputRuntimeData` property (it is sent from the simulation script using the `SimDataReturn` command). This collection cannot be modified, but only read.

Note: The simulation script commands used for communication with the Simulation Engine Driver are documented in "Special Scripting Commands for Interactive Simulation" on page 277.

- MATLAB:

```
driver.InputRuntimeData.Add('A1', 10.5);
driver.SendRuntimeData;
...
value = driver.OutputRuntimeData.Item('A1').Value;
```

- Python:

```
driver.InputRuntimeData.Add("A1", 10.5)
driver.SendRuntimeData()
...
value = driver.OutputRuntimeData("A1").Value
```

Additionally, there is a method that can be used to synchronize input runtime data with output. This can be useful in some scenarios where runtime data can be modified both by a simulation script and by a client (for instance, in sweep-like simulations).

- MATLAB:

```
driver.CopyOutputRuntimeDataToInput;
driver.SendRuntimeData;
```

- Python:

```
driver.CopyOutputRuntimeDataToInput()
driver.SendRuntimeData()
```

In contrast to runtime data, output data is intended only for receiving but not sending results. Although it is very similar to output runtime data (and in many cases can be replaced with it), it is sometimes useful to separate runtime and output data. Runtime data is mainly intended for sending dynamic data to the simulation engine and receiving any corrections if necessary, whereas output data is generally for receiving the results of simulation.

- MATLAB:

```
result = driver.OutputData.Item('Out1');
```

- Python:

```
result = driver.OutputData("Out1")
```

In the simulation script, the output data is made available for the Simulation Engine Driver client code using the `SimOutputReturn` command.

Along with scalar numerical values, it is possible to get matrix values. This can be done similar to scalar values through objects accessible from the collection returned by the `OutputData`:

- Python:

```
import numpy as np
# Read matrix result
m = driver.OutputData.Item("PostValueMxFloat_vtms1").Value
print("Output = " + str(m))
# Access matrix element
print("Output [0][0] = " + str(m[0][0]))
# Convert to numpy array
print(np.asarray(m))
```

- MATLAB:

```
% Get global parameter TimeWindow
m = driver.OutputData.Item('PostValueMxFloat_vtms1');
% Get matrix dimensions
[mrows, ncols] = size(m);
F = [repmat(' %d',1,ncols),'\n'];
fprintf('Output = \n');
fprintf(F, m.');
% Access matrix element
fprintf('Output [1][1] = %d\n', m(1, 1));
```

In the simulation script, matrix output data is made available for the Simulation Engine Driver client code using the `SimOutputReturnEx` command.

Sending Commands

Three commands are currently supported: `Pause`, `Resume` and `Stop`.

Here is an example of sending the Pause command:

- MATLAB:

```
driver.Pause;
```

- Python:

```
driver.Pause()
```

For more information on commands, see "Commands" on page 329.

Handling Events

The `SimulationEngineDriver` supports COM events through standard mechanisms:

- Excel:

```
' Specify that driver object fires events
Dim WithEvents driver As SimulationEngineDriver
```

It also supports a specific event pumping mechanism through the method `GetNextEvent()`. This mode must be explicitly switched on:

- MATLAB:

```
' Switch pull approach on
```

```
driver.IsEventQueueEnabled = True;
' Wait for the next event infinitely
event = driver.GetNextEvent(-1);
```

- Python:

```
# Switch pull approach on
driver.IsEventQueueEnabled = True
# Wait for the next event infinitely
event = driver.GetNextEvent()
```

For more information on event handling, see "Events" on page 327.

Terminating the Simulation

Sometimes it is necessary to force quit a simulation before it is finished. This may be necessary, for example if a simulation hangs. To stop the simulation, use the `Abort()` or `Kill()` methods.

The `Abort()` method stops the simulation respectfully, allowing the simulation engine to perform any necessary cleanup and finalize pending communications. With this method, the simulation engine process is only force quit if it is not possible to stop the simulation within a specified time:

- MATLAB:

  ```
  % Abort simulation and wait up to 2 seconds
  driver.Abort(2000)
  ```

- Python:

  ```
  # Abort simulation and wait up to 2 seconds
  driver.Abort(2000)
  ```

In contrast, the `Kill()` method terminates the simulation engine process immediately and unconditionally:

- MATLAB:

  ```
  % Terminate simulation engine process immediately
  driver.Kill
  ```

- Python:

  ```
  # Terminate simulation engine process immediately
  driver.Kill()
  ```

Destroying Objects

Many languages manage an object's life cycle themselves and call destruction methods automatically when an object is no longer in use. However, explicit destruction may be necessary to free system resources while the interpreter or compiler still considers the object to be alive.

In addition to the standard actions available for a particular language, the `SimulationEngineDriver` object provides its own explicit cleanup method:

- Python:

```
win32com.client.CastTo(driver, "IDisposable").Dispose()
driver = None
```

It is especially important to call this method if multiple `SimulationEngineDriver` objects are created and the `CheckoutLicense()`/`CheckinLicense()` methods are used.

Simulation Script

A simulation script is the only mandatory input required for the Simulation Engine Driver to run a simulation. This script defines the topology, instance parameters, and other operations that are used to perform the simulation. It may be difficult to write such scripts from scratch. But fortunately, the scripts that are generated automatically are suitable in many cases, and this approach is preferable wherever possible.

The easiest way to acquire a simulation script is to open a predefined design package (`.vtmu`) or VPI Module Interchange file (`.vmi`). A script will be generated automatically.

Another easy way to acquire a simulation script is to export a schematic as a raw package via **File > Export > Simulation Engine Driver Package**.

A simulation script is created whenever a simulation is run in the VPI Design Suite environment. The script is created in the `jobs\`*`N`* subfolder of the temporary directory (typically `C:\Users\`*`<user name>`*`\AppData\Local\Temp\VPIDS115` under Windows 7, for example), where *`N`* is the integer index of the simulation. This index is incremented each time a new simulation is started from VPI Design Suite.

For the Simulation Engine Driver it's possible to specify a script directly.

For a simple simulation run, the script contains the topology definition, initialization of instance parameters, and the `run` command. Usually, the topology and parameter definition blocks in the generated script can be kept intact and the `run` command can be replaced with more sophisticated logic in a modified version of the script.

Note: For more information on simulation scripts, refer to Chapter 3, "Simulation Scripts".

Collection Objects

A collection is an object which may store multiple other objects of the same type and provide access to them in a generic way.

There are two different collections in the Simulation Engine Driver:

- Collection of parameters (`SimulationParameterCollection`)
- Collection of runtime data (`SimulationDataCollection`)

Collections have similar set of functions.

To get the number of objects in a collection:

- MATLAB:

  ```
  count = collection.Count;
  ```

- Python:

  ```
  count = collection.Count
  ```

To get an object from a collection:

- MATLAB:

  ```
  obj = collection.Item('A1');
  ```

- Python:

  ```
  obj = collection.Item("A1")
  # Also contracted form is allowed
  obj = collection("A1")
  ```

Collections in Simulation Engine Driver behave like dictionaries which look up necessary objects by key. Methods are provided to access those keys. Keys may be necessary to enumerate the contents of a collection:

- MATLAB:

  ```
  keys = data.Keys;
  for key_idx = 1:length(keys)
        key = keys{key_idx};
        obj = data.Item(key);
        fprintf('%s\n', obj.Value);
  end
  ```

- Python:

  ```
  keys = data.Keys
  for key in keys:
        print(data(key).Value)
  ```

Some languages are also support simple enumeration of objects in a collection:

- Python:

  ```
  for obj in collection:
        print(obj.Value)
  ```

To check whether an object with a certain key already exists in the collection:

- MATLAB:

```
if collection.ContainsKey('A1')
      % Do something
end
```

- Python:

```
if collection.ContainsKey("A1"):
      # Do something
```

Some collections are also support alteration of their contents.

To add a new object to the collection:

- MATLAB:

```
obj = collection.Add('A1', 55.3);
```

- Python:

```
obj = collection.Add("A1", 55.3)
```

To remove an object from the collection:

- MATLAB:

```
collection.Remove('A1');
```

- Python:

```
collection.Remove("A1")
```

To remove all objects from the collection:

- MATLAB:

```
collection.Clear;
```

- Python:

```
collection.Clear()
```

SimulationParameter Object

This object represents the initial values and expression type for global parameters and parameters of module instances at the schematic level.

It has four properties: `InstanceId`, `Name`, `Value` and `ExprType`. For global parameters, `InstanceId` is an empty string; for module instance parameters, it is an instance ID.

- MATLAB:

```
% Get global parameter TimeWindow
param = driver.Parameters.Item('', 'TimeWindow');
fprintf('%s.%s = %s (%s)\n', param.InstanceId, param.Name, param.Value,
param.ExprType);

% Change value of module’s parameter
driver.Parameters.Item('Const_vtms1', 'level').Value = 15;
% Change expression type
driver.Parameters.Item('Const_vtms1', 'level').ExprType =
'ParameterExpressionType_Python';
```

- Python:

```
# Get parameter level from module with instance ID Const_vtms1
param = driver.Parameters("Const_vtms1", "level")
print("%s.%s = %s (%s)" % (param.InstanceId, param.Name, param.Value,
param.ExprType))

# Change value of module’s parameter
driver.Parameters('Const_vtms1', 'level').Value = 15
# Change expression type
driver.Parameters('Const_vtms1', 'level').ExprType =
win32com.client.constants.ParameterExpressionType_Python
```

Parameter value modifications will be applied only if the changes are made before the simulation is run. This is because the Simulation Engine Driver replaces the values specified in the simulation script with any values specified via this object. The simulation script is sent to the simulation engine only once (at the beginning of simulation execution), so any modifications of the parameter object during simulation do not affect the simulation.

SimulationData Object

In contrast to the `SimulationParameter` object, the `SimulationData` object is used to send/receive data to/from the simulation engine *during* the simulation process. `SimulationData` has only two properties: `Key` and `Value`, where `Key` uniquely identifies the data in the collection.

Runtime data is split into two instances: Input and Output. Input runtime data can be sent to the simulation engine and output is runtime data received from the simulation engine.

Input runtime data shall be explicitly sent by the client to become available in the simulation script. Output runtime data shall be explicitly sent by the simulation script to become available in the client.

Here is an example of sending runtime data from a client and consumption in a simulation script:

- MATLAB:

```
% Set runtime data value
driver.InputRuntimeData.Add('Value1', 234);
driver.InputRuntimeData.Add('Value2', 4.6);
% Send runtime data
driver.SendRuntimeData;
```

- Python:

```
# Set runtime data value
driver.InputRuntimeData.Add("Value1", 100)
driver.InputRuntimeData.Add("Value2", 3.14)
driver.SendRuntimeData()
```

- Simulation script:

```
# Receive runtime data
SimDataFetch runtime_data
# Use values in script
setstate Const_vtms1 $runtime_data(Value1)
setstate Const_vtms2 $runtime_data(Value2)
```

Here is an example of sending runtime data from a simulation script and consumption in a client:

- Simulation script:

```
# Calculate value of runtime data with key Value1
set runtime_data(Value1) [expr $index/3]
# Send runtime data to the client
SimDataReturn runtime_data
```

- MATLAB:

```
% Read runtime data value
value = driver.OutputRuntimeData.Item('Value1').Value;
```

- Python:

```
# Read runtime data value
value = driver.OutputRuntimeData("Value1").Value
```

In addition to runtime data, there is also output data. It represents results explicitly sent by a simulation script to make them available in a client. This is done using the simulation script command `SimOutputReturn` for scalar data. Usually, this command works in conjunction with the *PostValue* module because this is one of the few mechanisms that can be used to get values from a simulation into a simulation script[1].

- Simulation script:

```
# Read current value from PostValue module
set _outputs(PostValue_vtms1) [statevalue PostValue_vtms1 InputValue]
# Send output value
SimOutputReturn _outputs
```

- MATLAB:

```
% Read output value
value = driver.OutputData.Item('PostValue_vtms1').Value;
```

- Python:

```
# Read output value
value = driver.OutputData.Item("PostValue_vtms1").Value
```

The same can be done for matrix data using the `SimOutputReturnEx` script command, which usually works with the *PostValueMxFlt* module.

- Simulation script:

```
# Read matrix data from PostValueMxFlt module and send output value in one line
SimOutputReturnEx [list [list PostValueMxFlt_vtms1]]
# Or for this particular case a shorter syntaxis is also available:
# SimOutputReturnEx PostValueMxFlt_vtms1
```

- MATLAB:

```
% Read output value
mValue = driver.OutputData.Item('PostValueMxFlt_vtms1').Value;
```

- Python:

```
# Read output value
mValue = driver.OutputData.Item("PostValueMxFlt_vtms1").Value
```

1. The *LinkAnalyzer* module can also obtain results from the simulation.

Events

Events are a mechanism widely used in different programming languages to notify clients about various occurrences in another component. Events may include state changes, result availability, etc.

The following events are defined in the Simulation Engine Driver:

- Script execution started/finished
- Output results received
- Runtime data received
- Script waits for command or runtime data
- Message received

The event mechanism is mainly used in asynchronous simulations to synchronize different processes but also may be useful in synchronous simulations to access simulation messages.

There are two mechanisms for handling events: pull and push approaches.

Push Approach

In the "push" approach, a client registers callback functions with the `SimulationEngineDriver` object and those functions are called when a certain event is triggered.

This approach is based on the standard COM events mechanism and supported by many COM clients such as Microsoft Office applications. Unfortunately, it isn't currently supported by MATLAB and the standard Python implementation.

Here are examples for some clients which do support this approach:

- Excel:

```
' Specify that driver object fires events
Dim WithEvents driver As SimulationEngineDriver

' Function with special signature is automatically
' tied with appropriate event
Private Sub driver_OutputDataReceived()
      ' Set named cell with output value
      [Result] = driver.OutputData("PostValue_vtms1")
End Sub
```

- .NET/C#:

```
// Subscribe for event
driver.ScriptStarted += OnScriptStarted;

...

// This function will be called when event is triggered
void OnScriptStarted()
{
      Console.WriteLine("Script execution started");
```

```
        }
```

Pull Approach

The “pull” approach was introduced because some clients (like MATLAB) don’t support push events. The pull approach is widely supported by any client which can call functions on Simulation Engine Driver objects.

The pull approach uses a special code construct called event pump, which is a simple loop in which the `GetNextEvent()` function of `SimulationEngineDriver` is called at each iteration. `GetNextEvent()` returns the `SimulationEvent` object which contains all necessary information to distinguish the event type and additional event information.

Pull events are switched off by default because events are collected in a special event queue, and if this queue isn’t constantly emptied with calls to `GetNextEvent()`, it may soon overflow. To switch it on, set the Boolean property `IsEventQueueEnabled` to `True`.

Here are excerpts of programs which use the pull approach:

- MATLAB:

```
% Constants definition
STATE_CHANGED_EVENT = 0;
IDLE_STATE = 0;

% Switch pull approach on
driver.IsEventQueueEnabled = true;

% Begin asynchronous simulation
driver.BeginRun;

% Event pump
while true
   % Get next event from the queue
   event = driver.GetNextEvent(-1);

   % Exit loop when simulation is finished
   if event.get('Type') == STATE_CHANGED_EVENT
      if event.get('CurState') == IDLE_STATE
         break;
      end
   end
end

% Process errors
driver.EndRun(-1);
```

- Python:

```
import win32com.client
from win32com.client import constants

# Switch pull approach on
driver.IsEventQueueEnabled = True

# Begin asynchronous simulation
driver.BeginRun()

# Event pump
while True:
# Get next event from the queue
event = driver.GetNextEvent()

# Exit loop when simulation is finished
if event.Type ==
constants.SimulationEventType_SimulationEngineStateChanged:
   if event.CurState == constants.SimulationEngineState_Idle:
      break;

# Process errors
driver.EndRun()
```

Note: Constants are generated by the `makepy` utility (e.g., `constants.SimulationEventType_OutputDataReceived`).

Commands

Commands are a mechanism which can be used to control the execution flow of a simulation script. Commands are explicitly sent by clients and consumed by simulation scripts. Three commands are currently supported: `Pause`, `Resume`, and `Stop`.

Here is an example of usage:

- MATLAB:

```
% Run simulation asynchronously
driver.BeginRun

% Pause execution
driver.Pause

% Set runtime data value and send it
driver.InputRuntimeData.Add('Value1', 15);
driver.SendRuntimeData;

% Resume execution
driver.Resume;
```

- Python:

```
# Run simulation asynchronously
driver.BeginRun()

# Pause execution
driver.Pause()

# Set runtime data value and send it
driver.InputRuntimeData.Add("Value1", 15)
driver.SendRuntimeData()

# Resume execution
driver.Resume()
```

- Simulation script:

```
while { 1 } {
   # Receive and interpret command
   switch [SimGetCommand] {
      Exit { break }
      Pause {
         switch [SimGetCommand] {
            Exit { break }
            Resume { }
         }
      }
   }
}
```

Note: The `Stop` command is represented with the `Exit` constant in simulation scripts.

Simulation Engine Driver Examples

The sections on the following pages provide code samples to illustrate the usage of the Simulation Engine Driver.

Examples based on usage of simulation scripts are provided in the `\sed\examples\` subfolder of the installation directory, typically:

`C:\Program Files\VPI\VPIdesignSuite 11.5\sed\examples.`

Additionally, several regular demonstrations are available.

Optical Systems Demos > Simulation Techniques > Simulation Engine Driver

Simple Synchronous Simulation

Synchronous simulation is the simplest approach to use Simulation Engine Driver. It fits many scenarios where simulation engine startup overhead is less important than simplicity of implementation.

The following Python script creates the `SimulationEngineDriver` object, specifies the simulation script, sets parameter values, runs the simulation and prints the acquired results upon completion.

```
import win32com.client
from os.path import *
import sys

# Create driver object
driver =  win32com.client.Dispatch("VPI.TC.SED.SimulationEngineDriver")

# Construct path to pt-script
ptPath = normpath(join(sys.path[0], "ptds.pt"))
# Initialize driver with script
driver.SimulationScriptPath = ptPath

# Set parameters
driver.Parameters.Item("", "A1").Value = 0.6

# Run script synchronously
# This call will be blocked until script finishes
print("Running simulation...")
driver.Run()

# Read results
print("Output = " + str(driver.OutputData.Item("PostValue_vtms1")))
```

Note: For this example to run, the schematic must have a global parameter named A1 and a *PostValue* module with the instance ID `PostValue_vtms1`. Additionally, the following commands should be added to the simulation script after the `run` command (see "SimulationData Object" on page 325):
`set _outputs(PostValue_vtms1) [statevalue PostValue_vtms1 InputValue]`
`SimOutputReturn _outputs`.

A similar script in MATLAB would look like this:.

```
% Create driver object
driver = actxserver('VPI.TC.SED.SimulationEngineDriver');

workingDir = pwd;

% Initialize driver with pt-scrip
driver.SimulationScriptPath = strcat(workingDir, '\ptds.pt');

% Set parameters
driver.Parameters.Item('', 'A1').Value = 0.75;

% Run simulation synchronously
driver.Run;

% Read and print output results
value = driver.OutputData.Item('PostValue_vtms1').Value;
fprintf('Output = %g\n', value);

% Destroy driver object
driver.delete
```

Multidimensional Optimization

This example shows how to implement highly efficient multistep simulations using runtime data, events and commands.

In many cases, simulations can be represented as the following function:

$$O = f(P, I) \tag{4-1}$$

where P is the set of parameters, I is the set of all inputs (e.g., input files) and O is the set of results (e.g., numerical scalar or file) and f is a function which generates specific output for specific sets of parameters and inputs.

As a function, it is well suited for use in multidimensional optimization scenarios.

During optimization, the function is calculated many times, so it is vital to reduce any overhead. The simulation will be fastest when fully implemented in a simulation script. But this is a really difficult task. On the other hand, many modern mathematical packages already include a variety of optimization algorithms that are fully implemented and thoroughly tested. So it is better to split responsibilities between several parties.

Here we demonstrate a two-dimensional optimization using MATLAB. A similar algorithm using Python is provided with the demo:

Optical Systems Demos > Simulation Techniques > Simulation Engine Driver > DPSK-3ASK Optimization

In multistep simulations with the Simulation Engine Driver, work is split between the client (in this case MATLAB) and the Simulation Engine Driver. That is why it is so important to synchronize their work. This synchronization is done by means of events and commands. Besides synchronization it is important to transfer data back and forth. Runtime and output data is used for this purpose.

The flow chart shown in Figure 4-2 illustrates the communication between both parties and the processes executed within each script.

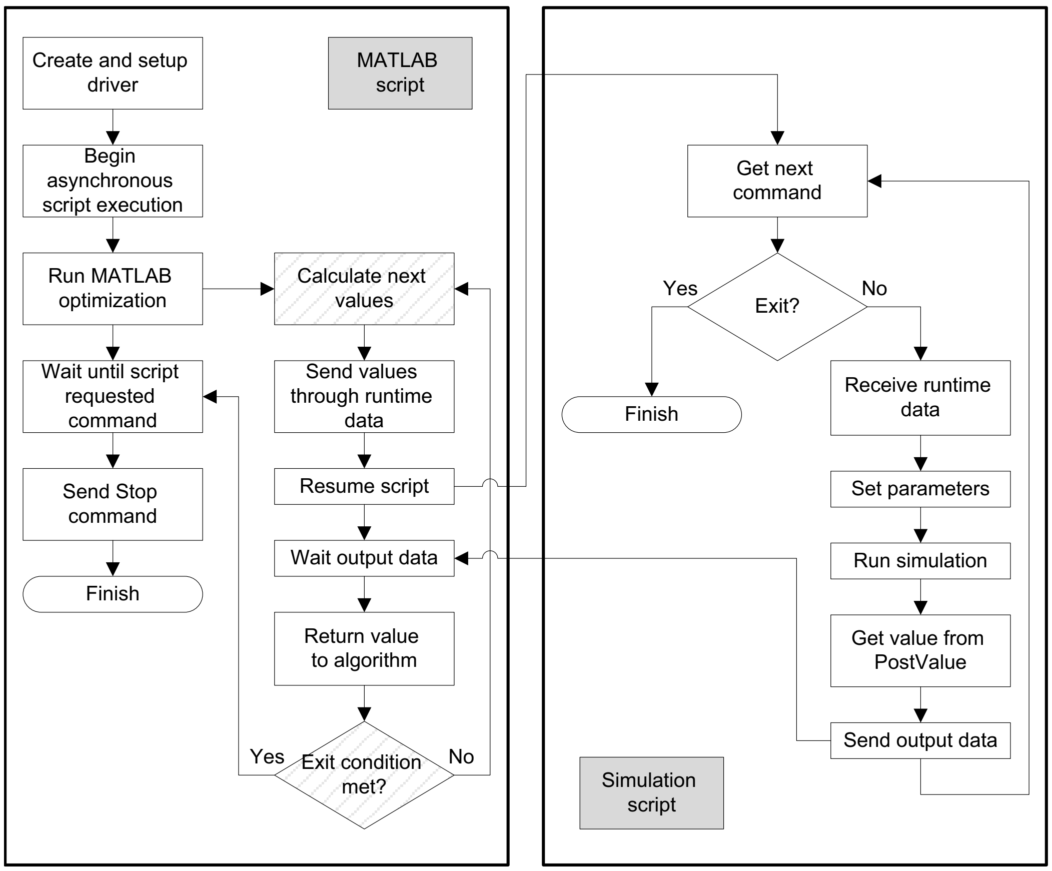


Figure 4-2 *Division of labor between MATLAB and simulation script*

The cross-hatched blocks are performed by MATLAB's `fminsearch` function.

Here is the MATLAB portion of the example:

```
function multidim_optimization()

      skipEndRun = false;

      try
            % Create driver object
            driver = actxserver('VPI.TC.SED.SimulationEngineDriver');
            workingDir = pwd;
            % Initialize driver with simulation scrip
            driver.SimulationScriptPath = strcat(workingDir, '\ptds.pt');
            % Enable event queue
            driver.IsEventQueueEnabled = true;
            % Add runtime data to later use it
            driver.InputRuntimeData.Add('A1', 0);
            driver.InputRuntimeData.Add('A2', 0);
            % Run simulation asynchronously
            driver.BeginRun;

            % Run standard MATLAB optimization routine
            % passing our function into it
            [X,FVAL,EXITFLAG,OUTPUT] = fminsearch(@(x) schematicRun(driver,
x),[0.35;0.75])

            % Wait sync point
            checkState(waitCommandPending(driver));

            % Stop simulation
            driver.Stop

            % Process errors
            skipEndRun = true;
            driver.EndRun(-1);

            % Destroy driver object
            driver.delete
      catch ex
            % Kill simulation engine if still running
            driver.Kill;
            try
                  if ~skipEndRun
                        % Process errors
                        driver.EndRun(-1);
                  end
            catch ex1
                  fprintf('%s\n', ex1.message);
            end
            % Destroy driver object
            driver.delete
            % Propogate exception
            rethrow(ex)
      end
end
```

```
function f = schematicRun(driver, x)
% Function to optimize
% This function is sent to optimization algorithm
% and performs communication of simulation engine with MATLAB

      % Print variables
      fprintf('A1 = %g\n', x(1));
      fprintf('A2 = %g\n', x(2));
      % Wait sync point
      checkState(waitCommandPending(driver));
      % Resume simulation
      driver.Resume
      % Wait until runtime data requested
      waitRuntimeDataPending(driver);
      % Set runtime data
      driver.InputRuntimeData.Item('A1').Value = x(1);
      driver.InputRuntimeData.Item('A2').Value = x(2);
      % Send runtime data
      driver.SendRuntimeData
      % Wait until outputs received
      checkState(waitOutputData(driver));
      % Read outputs
      value = driver.OutputData.Item('PostValue_vtms1').Value;
      fprintf('y = %g\n', value);
      % Return outputs as value of function to optimize
      f = value;
end

function checkState(state)
% Checks that expected state reached
% This method used to break simulation
% if unexpected condition met

      assert(state, 'Unexpected state');
end
function f = getNextEvent(driver)
% Returns next event from driver
% Used for debugging purposes
% Method driver.GetNextEvent() is used to take
% next simulation event from queue of events.
% When there is no event then call is blocked.
% Timeout in milliseconds can be specified to
% wait only specific amount of time.
% -1 used as timeout to signal that it shall wait infinitely.
      event = driver.GetNextEvent(-1);
      printEvent(event);
      f = event;
end

function printEvent(event)
% Prints information about simulation event.
% Used for debugging purposes

      % SimulationEventType_SimulationEngineStateChanged = 0  % from enum
SimulationEventType
```

```
      % SimulationEventType_RuntimeDataReceived = 1
      % SimulationEventType_OutputDataReceived  = 2
      % SimulationEventType_MessageReceived     = 3

      % SimulationEngineState_Idle               = 0 % from enum SimulationEngineState
      % SimulationEngineState_Running            = 1
      % SimulationEngineState_CommandPending     = 2
      % SimulationEngineState_RuntimeDataPending = 3

      if isempty(event)
            fprintf('NullEvent\n');
      elseif event.get('Type') == EV_STATE_CHANGED
            fprintf('EngineStateChanged - %s => %s\n',
decodeState(event.get('PrevState')), decodeState(event.get('CurState')));
      elseif event.get('Type') == EV_RUNTIME_DATA_RECEIVED
            fprintf('RuntimeDataReceived\n');
      elseif event.get('Type') == EV_OUTPUT_DATA_RECEIVED
            fprintf('OutputDataReceived\n');
      else
            fprintf('Unknown event: %s\n', event.get('Type'));
      end
end

function f = decodeState(state)
% Decodes state of simulation driver from numerical to human readable
% format

      switch state
            case 0
                  f = 'Idle';
            case 1
                  f = 'Running';
            case 2
                  f = 'CommandPending';
            case 3
                  f = 'RuntimeDataPending';
      end
end

% Constants
function f = EV_STATE_CHANGED(); f = 0; end
function f = EV_RUNTIME_DATA_RECEIVED(); f = 1; end
function f = EV_OUTPUT_DATA_RECEIVED(); f = 2; end
function f = ST_NONE(); f = -1; end
function f = ST_IDLE(); f = 0; end
function f = ST_RUNNING(); f = 1; end
function f = ST_COMMAND_PENDING(); f = 2; end
function f = ST_RUNTIME_DATA_PENDING(); f = 3; end


function f = waitOutputData(driver)
% Waits until command is requested by simulation script
      f = waitEvent(driver, EV_OUTPUT_DATA_RECEIVED, ST_NONE, []);
end
```

```
function f = waitRuntimeDataPending(driver)
% Waits until command is requested by simulation script
      f = waitEvent(driver, EV_STATE_CHANGED, ST_RUNTIME_DATA_PENDING, [ST_IDLE,
ST_COMMAND_PENDING]);
end

function f = waitCommandPending(driver)
% Waits until command is requested by simulation script
      f = waitEvent(driver, EV_STATE_CHANGED, ST_COMMAND_PENDING, [ST_IDLE,
ST_RUNTIME_DATA_PENDING]);
end


function f = waitEvent(driver, targetEvent, targetState, wrongStates)
% Waits for specific event
% For state event - waits until driver gets into specified state
% Fail if during this wait operation unexpected event received

      f = true;
      while true
            % Get next event from the queue
            event = getNextEvent(driver);
            % Check if event is empty
            if isempty(event)
                  % Indicates error
                  f = false;
                  break;
            end
            ev_type = event.get('Type');
            if ev_type == targetEvent
                  if ev_type == EV_STATE_CHANGED
                        state = event.get('CurState');
                        if state == targetState
                              break;
                        elseif any(state == wrongStates)
                              % Indicates error
                              f = false;
                              break;
                        end
                  else
                        break;
                  end
            end
      end
end
```

The simulation control segment of the simulation script looks like this *(the part of the script which defines the schematic topology is omitted for brevity)*:

```
#=== Simulation control segment ====

# Loop structure
set exit 0
set runCount 0

while { $exit == 0 } {

	# Wait for command
	switch [SimGetCommand] {
		Exit { set exit 1 }
		Pause {
			switch [SimGetCommand] {
				Exit { set exit 1 }
			}
		}
	}

	if { $exit == 0 } {
		# Receive runtime data
		SimDataFetch runtime_data

		# Assign states
		setstate this A1 "$runtime_data(A1)"
		setstate this A2 "$runtime_data(A2)"

		# Execute simulation
		run 1 $runCount
		incr runCount

		# Return result
		set outputs(PostValue_vtms1) [ statevalue PostValue_vtms1 InputValue ]
		SimOutputReturn outputs
	}
}
```

For this example to run, the schematic must have two global parameters A1 and A2 and a *PostValue* module with the instance ID `PostValue_vtms1` (such as the demonstration `Optical Systems Demos\Modulation Multilevel\DPSK-3ASK Level Optimization`).

Receiving Simulation Messages in Synchronous Simulations

The event mechanism described in chapter “Events” on page 327 can be used in synchronous simulations to access messages produced during the simulation.

In order to be able to access events after a simulation is complete, it is necessary first to enable the event queue by setting the Boolean property `IsEventQueueEnabled` to `True`.

The piece of code below illustrates how to enable the event queue, run a synchronous simulation, and print messages received from simulation to the console:

```
import os.path
import sys
import win32com.client
from win32com.client import constants as cnt

try:

    # Create SED object
    driver =\
        win32com.client.Dispatch("VPI.TC.SED.SimulationEngineDriver")

    # Initialize the driver with the '*.vtmu' file
    vtmuPath =\
        os.path.normpath(os.path.join(sys.path[0], "..", "..", "Access Log 
Messages.vtmu"))
    driver.OpenPackage(vtmuPath)

    # DO useful things
    # Set global parameters in the VPI simulation
    # ...
    # Set module parameters in the VPI simulation
    # ...

    # Enable events
    driver.IsEventQueueEnabled = True

    # Run the VPI simulation in the synchronous mode
    driver.Run()

    # Print out events end messages after simulation completion
    print_messages(driver)

except Exception as ex:
    print('Simulation stopped with an error')
    print('Error information: %s' % str(ex))
finally:
    input('Press enter to exit... ')
```

Messages themselves are printed by the `print_messages()` function shown below. This function loops through the queue of events available for the given Simulation Engine Driver object and prints corresponding messages in accordance with their type:

```
def print_messages(driver):
    IDLE= cnt.SimulationEngineState_Idle
    CMD_PENDING = cnt.SimulationEngineState_CommandPending
    RDATA_PENDING = cnt.SimulationEngineState_RuntimeDataPending
    STATE_CHANGED = cnt.SimulationEventType_SimulationEngineStateChanged
    MSG_RECEIVED = cnt.SimulationEventType_MessageReceived

    # Check event queue after simulation completion
    print("====== Simulation events =====")
    while True:
        event = driver.GetNextEvent()
        if event == None:
            print('No more events in the queue.')
            break
        elif event.Type == MSG_RECEIVED:
            messageType = event.MessageType
            if messageType == cnt.MessageType_Info :
                print('Informational message:')
                print(event.Message)
            elif messageType == cnt.MessageType_Warning :
                print('Warning message:')
                print(event.Message)
        elif event.Type == STATE_CHANGED :
            if event.CurState == CMD_PENDING:
                print('Command_pending...')
            elif event.CurState == RDATA_PENDING:
                print('Runtimedata_pending...')
            elif event.CurState == IDLE:
                print('Simulation finished. Sim state is IDLE.')
    print("====== Events finished =====")
```

A similar approach is provided in the following demo:

Optical Systems Demos > Simulation Techniques > SED > Access Log Messages

Appendix A

PDE Scripting Language Reference

This chapter describes the PDE Scripting Language commands in alphabetical order.

Conventions

Commands are represented with a `regular fixed-width` font. Variables and optional arguments are shown in an *`italic fixed-width`* font.

Optional parts of a Tcl command are enclosed in question marks such as *`?something?`*, whereas for Python and the web service they are enclosed in square brackets.

For web service commands, actual parameter values are enclosed in angular braces, like *`<value>`*.

Command Terms

Modules, Types, Instances and Virtual Galaxies

- A *Module* is either a galaxy module or a star module. A *Module Name* is the basic generic name of a module as it appears in the library, for example, *LaserSM_RE*.
- A *Type* is the type of a schematic or module: universe, galaxy, virtual galaxy or star.
- A *Module Instance* is an instance of a module on a schematic. For example, a *LaserSM_RE* module may appear several times and each instance of it may have different parameters.
- A *Virtual Galaxy* is an entity that groups module instances together and has parameters specific to galaxies. In contrast to regular galaxies, it doesn't exist on its own and cannot be reused (e.g., there is only one instance of each virtual galaxy).
- An *Instance ID* (also known as an ID or simply instance) is a unique name given to a module or virtual galaxy every time it appears on the schematic. Each time it requires a different instance ID, because it will be representing a different physical laser. When placing modules using the PDE, the instance ID is generated from the

module name (`LaserSM_RE_vtms1`, `LaserSM_RE_vtms2`, etc.), but can be changed using the parameter editor. When placing modules using the PDE Scripting Language, the instance ID **must** be specified when the module instance is created (the script author is responsible for assigning different IDs for different instances).

Identifying Schematics and Modules

A number of PDE script commands require you to specify a package (universe, galaxy or star) to be opened or retrieved.

URN Paths

A module or schematic is fully specified by its URN. A URN is a string that identifies the schematic's work package. The structure of the URN is described in "Identifying Packages using a URN" on page 31. For schematics and modules (galaxies or stars) in a library, the URN is displayed in the **Properties** dialog, which can be opened by right-clicking on the package in the Library Explorer, and selecting **Properties** from the context menu. For an open schematic, the **Properties** item on the **File** menu will display the **Properties** dialog.

Note: Virtual galaxies don't have URNs.

Examples

The demonstration *Optical Systems Demos > Simulation Techniques > General > Macro Tutorial* has the `URN:VPI_DEMO::OS\Simulation Techniques\General\Macro Tutorial.vtmu:`

The single mode laser rate equation module *LaserSM_RE* in the *Optical Sources* folder of the *Module Library* has the `URN:VPI_LIB::TC Modules\Optical Sources\LaserSM_RE.vtms:`

Selecting Target Context

Many functions that work with modules, parameters or text blocks depend on the current selection context *(the selected schematic or virtual galaxy)*. For example, the `star` command adds a module instance to the current virtual galaxy of the current schematic. If no virtual galaxy is currently selected, the module is added to the main topology. The same approach is used by the `addtext` and `newstate` commands, etc.

Specifying Positions

Positions are specified from the top left corner of the schematic if no virtual galaxy is in context (selected as current). If a virtual galaxy is in context, positions are specified from its top left corner. Each ruled square of the schematic grid is 2 units by 2 units. Positions are quantized to the nearest unit. An icon is usually 4 units by 4 units. The reference point of the module icon is usually in the center.

Commands (in alphabetical order)

Note: Command signatures are given for the Tcl, Python macros, and the web service APIs. First goes a signature for Tcl, then the one for Python, and finally a signature for the web service is provided (if supported).

For the web service, only the last part or the request path and API query parameters are described.

addtext

Adds text to a schematic or virtual galaxy.

```
addtext x y text ?options?
```

```
addtext(x, y, text[, width, height[, font[, fontsize[, textcolor[,
     boxcolor[, bordercolor[, justify]]]]]]])
```

```
addtext?x=<x>&y=<y>&text=<text>[&width=<width>&height=<height>]
     [&font=<font>][&fontsize=<fontsize>][&textcolor=<textcolor>]
     [&boxcolor=<boxcolor>][&bordercolor=<bordercolor>]
     [&justify=<justify>]
```

x: the horizontal position of the center of the text box

y: the vertical position of the center of the text box

text: the text to be displayed.

The following options can be added:

`-size` *width height*: sets the size of the text box

`-font` *font*: sets the font to be used to display the text

`-fontsize` *size*: sets the size of the font

`-textcolor` *color*: sets the color of the text to be displayed

`-boxcolor` *color*: sets the background color of the text box

`-bordercolor` *color*: sets the color of the border of the text box

`-justify` *position*: justifies the text, position must be `left`, `right`, or `center`.

Supported fonts are: `dialog`, `sansserif`, `serif`, `monospaced`, `dialoginput`.

Supported colors are: `white`, `gray25`, `gray50`, `gray75`, `black`, `yellow`, `orange`, `darkorange`, `orangered`, `tomato`, `chocolate`, `brown`, `tan`, `pink`, `violet`, `mediumpurple`, `purple`, `deeppink`, `red`, `blue`, `royalblue`, `royalblue4`, `slateblue`, `cadetblue`, `skyblue`, `cyan2`, `deepskyblue`, `cyan4`, `lightsteelblue1`, `steelblue`, `slategray1`, `slategray3`, `lightslategray`, `darkgreen`, `green`, `limegreen`, `springgreen`, `mediumseagreen`, `greenyellow`, and `olivedrab`.

busconnect

Creates a bus connection between two modules. A bus connection is a wide wire, for example, for carrying several WDM channels before they are multiplexed.

```
busconnect inst1 port1[#term1] inst2 port2[#term2] width ?delay?
     ?-route route?
```

```
busconnect(inst1, port1[#term1], inst2, port2[#term2], width[, delay[,
     route]])
```

```
busconnect?inst1=<inst1>&port1=<port1>[%23<term1>]&inst2=<inst2>&port
     2=<port2>[%23<term2>]
     &width=<width>[&delay=<delay>][&route=<route>]
```

inst1:	instance ID of the first instance
port1:	port name of the first instance (output port)
term1:	terminal index of the port of the first instance when the port is a multiport
inst2:	instance ID of the second instance
port2:	port name of the second instance (input port)
term2:	terminal index of the port of the second instance when the port is a multiport
width:	bus width
delay:	delay of the bus (optional, default = `0`)
route:	set of points that describes how the visual connection should be routed.

Note: Currently, only *uninitialized* delays are supported when using the `busconnect` command.

Note: The route option format and rules are described in the linkroute command section.

cancelmacro

Aborts the current macro without raising an error. That is, the macro will be treated as canceled and not as finished with an error, and no error messages will appear.

The main purpose of this command is to abort the interactive simulation (sweep) initialization macro and avoid the simulation. For more details about interactive simulation initialization macros, please refer to "Custom and Toolkit Interactive Simulations" in Chapter 7 of the VPIphotonics Design Suite™ Simulation Guide.

```
cancelmacro
```

```
cancelmacro()
```

Note: This command is not available in web access API.

categoryorder

Returns a number between `1` and `5` representing the position of a given `categoryname` in the list of known categories (`Global`, `Physical`, `Numerical`, `General`, `Enhanced`) or `6`, if it is unknown.

```
categoryorder categoryname
```

```
categoryorder(categoryname)
```

```
categoryorder?categoryname=<categoryname>
```

`categoryname`: name of the category.

closeproject

Closes the current project. It may display the Save or Save As dialogs, asking the user to save the changes in open project schematics. Raises an error when there is no open project. You can use the `curproject` command to check whether there is an open project.

```
closeproject ?-savedialog on|off?
```

```
closeproject([savedialog=None|True|False])
```

```
closeproject[?savedialog=true|false]
```

`-savedialog`: whether to show the Save dialog or permit closing the schematic and discarding all changes without user intervention. When omitted and there are unsaved changes, an error will be raised and project closing will be stopped.

closeschematic

Closes the current schematic. It may display the **Save** or **Save As** dialog, asking the user to save the changes in the schematic.

```
closeschematic ?-savedialog on|off?
```

```
closeschematic([savedialog=True|False])
```

```
closeschematic[?savedialog=true|false]
```

`-savedialog`: whether to show the Save dialog or permit closing the schematic and discarding all changes without user intervention (default is `on` (`True`)).

compareschematics

Compares two currently open schematic or galaxies. Returns `1`(`True`) if schematics are identical and `0` (`False`) otherwise. Comparison results can be saved to a report file.

```
compareschematics name1 name2 ?file? ?-options options?
```

```
compareschematics(name1, name2[, file[, options]])
```

```
compareschematics?name1=<name1>&name2=<name2>&file=<file>&options=<op
    tions>
```

`name1`: name or URN of first universe (galaxy) to compare.

`name2`: name or URN of second universe (galaxy) to compare.

`file`: name of the report file. If only a file name is specified (without an extension) or the extension is something other than `.html` or `.xml`, the `.html` extension will be added automatically and the report will be saved in HTML format. If the `.xml` extension is specified, the file will be saved in XML format.

`-options x` where *x* is any combination of `p`, `g`, `m`, `r`, `t` and `c` letters, which specify the schematic properties to include in comparison and report:

p specifies that profile sets will be compared.

g specifies that global parameters will be compared.

m specifies that module IDs will be compared.

r specifies that module parameters and their settings will be compared. (automatically turns *m* option).

t specifies that module ports will be compared (automatically turns *m* option).

c specifies that connections between modules will be compared (automatically turns *m* option).

If no *`-options`* are specified, all properties will be included in comparison.

connect

Creates a connection (wire link or labeled link) between two instances.

connect *inst1* *port1*[#*term1*] *inst2* *port2*[#*term2*] ?*delay*? ?-route *route*? ?-labels *labels*?

connect(*inst1*, *port1*[#*term1*], *inst2*, *port2*[#*term2*][, *delay*[, *route*[, *labels*]]])

connect?inst1=<*inst1*>&port1=<*port1*>[%23<*term1*>]&inst2=<*inst2*>&port2=<*port2*>[%23<*term2*>][&delay=<*delay*>][&route=<*route*>][&labels=<*labels*>]

inst1: instance ID of the first instance

port1: port name of the first instance (output port)

term1: terminal index of the port of the first instance when the port is a multiport

inst2: instance ID of the second instance

port2: port name of the second instance (input port)

term2: terminal index of the port of the second instance when the port is a multiport

delay: delay of the wire (optional, default = `0`)

route: set of points that describes how the visual connection should be routed. Currently, only *uninitialized* delays are supported when using the `connect` command.

labels: link labels, when specified makes the link labeled (for more details about labeled links, please refer to "Labeled Links" in Chapter 3 of the VPIphotonics Design Suite™ Simulation Guide)

Note: Currently, only *uninitialized* delays are supported when using the `connect` command.

Note: The route option format and rules are described in the linkroute command section.

Examples:

- Tcl:

```
connect LaserCW_vtms1 output SignalAnalyzer_vtms1 input
connect LaserCW_vtms1 output SignalAnalyzer_vtms1 input -route {9 4 12 4 12 8}
connect LaserCW_vtms1 output SignalAnalyzer_vtms1 input -labels {a, b, c}
```

- Python:

```
connect("LaserCW_vtms1", "output", "SignalAnalyzer_vtms1", "input")
connect("LaserCW_vtms1", "output", "SignalAnalyzer_vtms1", "input", route=[(9,4),
(12,4), (12,8)])
connect("LaserCW_vtms1", "output", "SignalAnalyzer_vtms1", "input", labels="a, b,
c")
```

- Web API:

```
http://localhost:8475/pde/v1/connect?inst1=LaserCW_vtms1&port1=output&inst2=SignalA
  nalyzer_vtms1&port2=input
http://localhost:8475/pde/v1/connect?inst1=LaserCW_vtms1&port1=output&inst2=SignalA
nalyzer_vtms1&port2=input&route=9,4,12,4,12,8
http://localhost:8475/pde/v1/connect?inst1=LaserCW_vtms1&port1=output&inst2=SignalA
nalyzer_vtms1&port2=input&labels=a,b,c
```

curproject

Gets the path to the root folder of the currently open project. It is the same path that was used to open a project via the `openproject` command. Returns an empty string if there is no open project.

```
curproject
```

```
curproject()
```

```
curproject
```

curuniverse

Sets the schematic that commands are applied to. Resets context to the main schematic topology. If called without an argument, the command just returns the URN of the current schematics.

`curuniverse` *`?name?`*

`curuniverse([`*`name`*`])`

`curuniverse[?name=<`*`name`*`>]`

`name`: name with or without extension or URN of a target schematic or galaxy.

curvirtgal

Sets the virtual galaxy that commands are applied to (designates it as the current selection context). If called without an argument, the command just returns the instance ID of the current virtual galaxy or an empty string if the main schematic is in context.

`curvirtgal ?`*`inst`*`? ?-reset?`

`curvirtgal([`*`inst`*`[, reset=True|False]])`

`curvirtgal?inst=<`*`inst`*`>[&reset=true|false]`

`inst`: the instance ID of the virtual galaxy

`-reset`: if specified (`True`) then makes main schematic current.

delschematic

Deletes a schematic (galaxy) with a given name. Before deletion, the confirmation dialog is displayed to the user. The command returns `0` (`False`) if the schematic is not found and `1` (`True`) otherwise.

```
delschematic path
```

```
delschematic(path)
```

```
delschematic?path=<path>
```

`path`: a path or URN of a schematic or galaxy.

If an already opened schematic is specified, the command closes it before deletion (that is, after the user confirmation).

delstar

Deletes an instance (star, galaxy) or virtual galaxy from the current schematic.

```
delstar inst
```

```
delstar(inst)
```

```
delstar?inst=<inst>
```

`inst`: the instance ID.

deselectall

Deselects all selected components in the current schematic.

```
deselectall
```

```
deselectall()
```

```
deselectall
```

deselectstars

Deselects only the instances (stars, galaxies, or virtual galaxies) that are passed as parameters to the command.

```
deselectstars inst1 inst2 ...
```

```
deselectstars(inst1[, inst2[, ...]])
```

```
deselectstars?instances=<inst1>[%2C<inst2>[%2C ...]]
```

`inst1 ... instN`: instance IDs of the instances or virtual galaxies of the current schematic.

disconnect

Disconnects one end of a wire from an instance. Does nothing if the port is not connected.

```
disconnect inst port[#term]
```

```
disconnect(inst, port[#term])
```

```
disconnect?inst=<inst>&port=<port>[%23<term>]
```

inst: the instance ID
port: the name of the port to be disconnected
term: index of terminal to be disconnected when the port is a multiport.

domain

Sets the simulation domain of a schematic or virtual galaxy. If the type is omitted, it returns the current domain.

```
domain ?type?
```

```
domain([type])
```

```
domain[?type=<domain>]
```

type: is one of "`Auto`", "`SDF`", "`BDF`", "`DDF`", or "`SMATRIX`" (the latter is supported for galaxies and virtual galaxies only).

enablewaitdlg

Enables or disables the wait dialog which is shown during long operations. Returns the current status if parameter is omitted. Sometimes it is desirable to suppress it, for example, when a UI is created using a native library like Tk in Tcl or Tkinter in Python but not the standard wizard.

```
enablewaitdlg ?on|off?
```

```
enablewaitdlg([enable=True|False])
```

```
enablewaitdlg[?enable=true|false]
```

enable: on (`True`) to allow wait dialog to be shown, `off` (`False`) to suppress it.

enablewebservice

Enables or disables the web service for the PDE. Returns the current status of the web access service.

```
enablewebservice ?on|off?
```

```
enablewebservice([enable=True|False])
```

```
enablewebservice[?enable=false]
```

enable: `1` (`True`) to allow web access to the PDE, `0` (`False`) to suppress it.

Note: Using the web access, it is only possible to disable the web service or check its status. After successfully disabling the web service, the client computer is expected to receive an error (status 503).

evaluateparams

Calculates the values of parameters whose names satisfy a specified filter, running the schematic in the Evaluate mode as described in the section "Evaluating Parameter Values" in Chapter 4 of the VPIphotonics Design Suite™ Simulation Guide.

```
evaluateparams filters
evaluateparams(filters)
evaluateparams?filters=<filters>
```

filters: list of filters that specify parameters to be calculated.

Exact (that is, "`TimeWindow`") and loose (that is, "`**`") filters are supported. If a loose filter is specified, then all parameters that suit the filter are calculated.

For Tcl, parameters and their values are returned as a list in the format:

```
{param1 value1 param2 value 2 ...}
```

For Python, results are returned as a dictionary where key is the parameter name and value is the evaluated value.

exportschematic

Exports the current schematic using the specified format.

```
exportschematic format path
exportschematic(format, path)
exportschematic?format=<format>&path=<path>
```

format: resulting format (one of "`vmi`", "`sed`", "`sedraw`", or "`dds`")
path: path to the resulting file or directory.

The following formats are supported:
`vmi` to export to the VPI Module Interchange Format
`sed` to export to the `Simulation Engine Driver` package
`sedraw` to export to the raw simulation package
`dds` to export to the dynamicDataSheet file.

getbidirmode

Gets the mode of a bidirectional port group ("`bidir`", "`in`", "`out`", "`inout`", or "`none`" for non-bidirectional ports).

`getbidirmode` *`inst port`*

`getbidirmode(`*`inst, port`*`)`

`getbidirmode?inst=<`*`inst`*`>&port=<`*`port`*`>`

`inst`: instance ID

`port`: name of one of the ports in the bidirectional port group of the module you wish to investigate.

getbidirportnames

Gets the list of names of all ports which belong to the same bidirectional port group as the port specified in the command arguments. The function returns an empty list for the ports which do not belong to the bidirectional port group.

`getbidirportnames` *`inst port`*

`getbidirportnames(`*`inst, port`*`)`

`getbidirportnames?inst=<`*`inst`*`>&port=<`*`port`*`>`

`inst`: instance ID

`port`: name of one of the ports in the bidirectional port group of the module you wish to investigate.

getblocks

Gets the instance IDs of all instances (stars or galaxies) of the current universe or, if the optional parameter is given, all instance IDs of the instances corresponding to the module specified by the parameter. If a virtual galaxy is in context, only instances of this galaxy are returned. May additionally return the names of all virtual galaxies.

`getblocks ?`*`path`*`? ?-includevg?`

`getblocks([`*`path,`*`] [`*`includevg`*`=True|False])`

`getblocks?[path=<`*`path`*`>&][includevg=true|false]`

`path`: specifies the module path (or the URN) corresponding to the instances to be returned

`-includevg`: if specified, the names of virtual galaxies are also returned.

getbounds

Gets the bounds of a given instance, virtual galaxy, or module as a list of coordinates in the form `{x1 y1 x2 y2}`. The position is returned relative to the left upper corner of the schematic. If an instance belongs to a virtual galaxy and this galaxy is in context, the position is returned relative to this virtual galaxy. For a module, the position is returned relative to the module icon center.

```
getbounds inst
```

```
getbounds(inst)
```

```
getbounds?inst=<inst>
```

inst: instance ID for instances or URN for modules.

getconnectedblocks

Returns a list of all instances connected to the specified port of the specified instance. This can be used to find what is connected to a certain port of a particular instance.

```
getconnectedblocks inst port
```

```
getconnectedblocks(inst, port)
```

```
getconnectedblocks?inst=<inst>&port=<port>
```

inst: instance ID
port: port name of the block you wish to investigate.

getconnection

Gets the other end of a connection and link delay value, which has one end specified in the command. The returned value is a list each item of which is in turn a list of {*inst port*[#*term*] *width delay ?labels?*}. For Python, a list of tuples is returned.

```
getconnection inst port[#term] ?-includelabels?
```

```
getconnection(inst, port[#term][, includelabels=False])
```

```
getconnection?inst=<inst>
      &port=<port>[%23<term>][&includelabels=true|false]
```

inst: instance ID
port: name of the port of a given instance to which the known end is connected
term: index of the terminal of the port of a given instance to which the known end is connected (when port is multiport).
`-includelabels`: is an option that defines, whether labels shall be included or not.

getcontenttype

Returns a string representing the type of the package specified by the *path* ("`package.tmm.vtms`" for stars, "`package.tmm.vtmg`" for galaxies, "`package.tmm.vtmu`" for universes, "`package.tmm.vtmy`" for galaxy ports and "`package.tmm.vtmw`" for module sweeps). If the path is omitted, the type of the current schematic is returned.

`getcontenttype` *`?path?`*

`getcontenttype([`*`path`*`])`

`getcontenttype[?path=<`*`path`*`>]`

`path`: specifies the resource path (or the URN).

getmaster

Returns the generic master module type of the given instance (that is, the module from which the instance was created).

`getmaster ?-p?` *`inst`*

`getmaster(`*`inst`*`[,` *`p`*`=True|False])`

`getmaster?inst=<`*`inst`*`>[&p=true|false]`

`inst`: an instance ID

`-p`: if this option is specified, the return value will be a module path; otherwise a URN is returned.

getportdir

Returns the direction of a given port as a string (either "`in`", "`out`", or "`bidir`").

`getportdir` *`inst port`*

`getportdir(`*`inst`*`,` *`port`*`)`

`getportdir?inst=<`*`inst`*`>&port=<`*`port`*`>`

`inst`: instance ID for instances or URN for modules

`port`: name of the port under investigation.

getports

Gets a list with ports of a given instance or module. The list will contain either all ports, if the optional parameter is omitted, or only the input, output, or bidirectional ports depending on the optional parameter. Only visible ports of a bidirectional port group are returned for an instance, while for modules all ports in a bidirectional port group are always returned.

`getports` *`inst`* `?`*`direction`*`?`

`getports(`*`inst`*`[, `*`direction`*`])`

`getports?inst=<`*`inst`*`>[&direction=<`*`dir`*`>]`

> *`inst`*: instance ID for instances or URN for modules
> *`direction`*: must be `-in`, `-out`, or `-bidir` ("`in`", "`out`", or "`bidir`" for Python or the web service).

getpythonexepath

Gets the full path to the Python executable corresponding to the currently active Python environment. This could later be used to execute pure Python scripts (not macros).

`getpythonexepath`

`getpythonexepath()`

`getpythonexepath`

getsignaltypes

Gets a list of signal types (electrical, optical, anytype, integer, float, etc.) for a given port.

`getsignaltypes` *`inst port`*

`getsignaltypes(`*`inst`*`, `*`port`*`)`

`getsignaltypes?inst=<`*`inst`*`>&port=<`*`port`*`>`

> *`inst`*: instance ID
> *`port`*: the port that is under investigation.

Returns one of the following values: "`anytype`", "`integer`", "`float`", "`complex`", "`fixpoint`", "`optical`", "`electrical`", "`elwave`", or "`unknown`".

getstates

Returns a list with the names of all parameters of a given module instance or virtual galaxy, or a list of all global parameters of the current universe or virtual galaxy if the instance ID is omitted.

`getstates ?`*`inst`*`?`

`getstates([`*`inst`*`])`

`getstates[?inst=<`*`inst`*`>]`

> *`inst`*: instance ID of the instance or virtual galaxy.

getterminals

Gets a list of all terminals of a given port (for modules with multiports such as *BusCreate* and *BusSplit*).

`getterminals inst port`

`getterminals(inst, port)`

`getterminals?inst=<inst>&port=<port>`

inst:	instance ID for instances or URN for modules
port:	name of the port under investigation.

The individual terminals of a multiport can be accessed with the commands such as `connect`, `isconnected`, or `getconnection` using the `port_name#terminal_index` notation.

gettoolkitpath

Gets the path of a specified toolkit, project, or module library variable (key).

`gettoolkitpath key`

`gettoolkitpath(key)`

`gettoolkitpath?key=<key>`

key:	variable name

You can get the root folder path of:

- toolkit via the `<ToolkitAbbreviation>_ROOT` key
- project via the `PROJECT_ROOT` key
- module library via the `LIB_<LibraryNameUppercase>_ROOT` key

For toolkits you may also get the path of some special folders:

- toolkit user content folder via the `<ToolkitAbbreviation>_USER_DIR` key
- toolkit private content folder via the `<ToolkitAbbreviation>_PRIVATE_DIR` key

Where `<ToolkitAbbreviation>` is a short name of the toolkit visible in VPIdesignSuite title or in the title of the current schematic. For example, for standard products an abbreviation is a short name of the product like OS for VPItransmissionMaker Optical Systems. `<LibraryNameUppercase>` is a name of the library with all uppercase letters and all non-letter and non-digit characters replaced with underscores (_). For example, for a library name MyLib@1 variable will be `LIB_MYLIB_1_ROOT`.

Examples:

- Tcl:

```
gettoolkitpath PROJECT_ROOT
gettoolkitpath LIB_MYLIB_1_ROOT
gettoolkitpath TK_USER_DIR
```

- Python:

```
gettoolkitpath("PROJECT_ROOT")
gettoolkitpath("LIB_MYLIB_1_ROOT")
gettoolkitpath("TK_USER_DIR")
```

- Web API:

```
http://localhost:8475/pde/v1/gettoolkitpath?key=PROJECT_ROOT
http://localhost:8475/pde/v1/gettoolkitpath?key=LIB_MYLIB_1_ROOT
http://localhost:8475/pde/v1/gettoolkitpath?key=TK_USER_DIR
```

getvariable

Gets the value of variable that is bound to a text field, combo box, or checkbox of a wizard or set via the `modify` command. This command is available only for Python.

`getvariable(`*`variable`*`)`

`variable`: the name of the variable.

getvirtgal

Gets the instance ID of a virtual galaxy to which the specified instance belongs or returns an empty string if the instance resides on the main schematic topology.

`getvirtgal `*`inst`*

`getvirtgal(`*`inst`*`)`

`getvirtgal?inst=<`*`inst`*`>`

`inst`: instance ID.

getxpos

Gets the x position of the module instance or virtual galaxy. If the instance belongs to a virtual galaxy and this galaxy is in context, the position is returned relative to this virtual galaxy.

`getxpos `*`inst`*

`getxpos(`*`inst`*`)`

`getxpos?inst=<`*`inst`*`>`

`inst`: instance ID.

getypos

Gets the y position of the module instance or virtual galaxy. If the instance belongs to a virtual galaxy and this galaxy is in context, the position is returned relative to this virtual galaxy.

`getypos` *`inst`*

`getypos(`*`inst`*`)`

`getypos?inst=<`*`inst`*`>`

`inst`: instance ID.

isconnected

Checks if the given port (or terminal) is currently connected. It will return `1` (`True`) if the port is connected and `0` (`False`) otherwise.

`isconnected` *`inst port`*`[#`*`term`*`]`

`isconnected(`*`inst, port`*`[#`*`term`*`])`

`isconnected?inst=<`*`inst`*`>&port=<`*`port`*`>[%23<`*`term`*`>]`

`inst`: instance ID
`port`: name of the port under investigation
`term`: index of the terminal name of the port under investigation when the port is a multiport.

isportvisible

Checks if the given port is currently visible. It will return `1` (`True`) if the port is visible and `0` (`False`) otherwise. This command only makes sense for ports in a bidirectional port group. For all other ports, it always returns `1` (`True`).

`isportvisible` *`inst port`*

`isportvisible(`*`inst, port`*`)`

`isportvisible?inst=<`*`inst`*`>&port=<`*`port`*`>`

`inst`: instance ID
`port`: name of the port under investigation.

isschematicmodified

Checks if the schematic is modified. It will return `1` (`True`) if the schematic is modified and `0` (`False`) otherwise. If called without an argument, the command just returns the state for the current schematic.

`isschematicmodified ?`*`name`*`?`

`isschematicmodified([`*`name`*`])`

`isschematicmodified[?name=<`*`name`*`>]`

`name`: name with or without extension or URN of a target schematic or galaxy.

isvirtgal

Checks if the given instance ID corresponds to a virtual galaxy. It will return `1` (`True`) if it is a virtual galaxy or `0` (`False`) if it is a module instance.

`isvirtgal` *`inst`*

`isvirtgal(`*`inst`*`)`

`isvirtgal?inst=<`*`inst`*`>`

`inst`: instance ID.

linkroute

Gets or sets a visual link path between the specified ports.

`linkroute` *`inst1 port1`*`[#`*`term1`*`]` *`inst2 port2`*`[#`*`term2`*`] ?`*`route`*`|-reset?`

`linkroute(`*`inst1, port1`*`[#`*`term1`*`],` *`inst2, port2`*`[#`*`term2`*`][,` *`route`*`][,` *`reset`*`=True|False])`

`linkroute?inst1=<`*`inst1`*`>&port1=<`*`port1`*`>[%23<`*`term1`*`>]&inst2=<`*`inst2`*`>&port2=<`*`port2`*`>[%23<`*`term1`*`>][&route=<`*`route`*`>][&reset=true|false]`

`inst1`:	instance ID of the first instance
`port1`:	port name of the first instance
`term1`:	terminal index of the port of the first instance when the port is a multiport
`inst2`:	instance ID of the second instance
`port2`:	port name of the second instance
`term2`:	terminal index of the port of the second instance when the port is a multiport
`route`:	set of points that describes how the visual link should be routed
`reset`:	if specified (`True`), resets the link path to an automatically generated one.

The link route is specified as a list of x and y coordinates in Tcl and the web API and as a list, tuple, or any other iterable collection of (x, y) pairs in Python.

The returned link path may not be equal to the path specified by the `route` option due to the applied corrections (e.g., inclined segments in the path are replaced with horizontal and vertical segments). In addition, the coordinates of the source and target ports are included automatically in case they are omitted.

Examples:

- Tcl:

```
linkroute LaserCW_vtms1 output SignalAnalyzer_vtms1 input
linkroute LaserCW_vtms1 output SignalAnalyzer_vtms1 input {9 4 12 4 12 8}
linkroute LaserCW_vtms1 output SignalAnalyzer_vtms1 input -reset
```

- Python:

```
linkroute("LaserCW_vtms1", "output", "SignalAnalyzer_vtms1", "input")
linkroute("LaserCW_vtms1", "output", "SignalAnalyzer_vtms1", "input", [(9,4),
(12,4), (12,8)])
linkroute("LaserCW_vtms1", "output", "SignalAnalyzer_vtms1", "input", reset=True)
```

- Web API:

```
http://localhost:8475/pde/v1/linkroute?inst1=LaserCW_vtms1&port1=output&inst2=Signa
  lAnalyzer_vtms1&port2=input
http://localhost:8475/pde/v1/linkroute?inst1=LaserCW_vtms1&port1=output&inst2=Signa
lAnalyzer_vtms1&port2=input&route=9,4,12,4,12,8
http://localhost:8475/pde/v1/linkroute?inst1=LaserCW_vtms1&port1=output&inst2=Signa
lAnalyzer_vtms1&port2=input&reset=true
```

messagedialog

Displays a message window with an optional title and message type. Returns a (lowercase) string depending on the button pressed by the user (either "`ok`", "`cancel`", "`yes`", or "`no`"). This command is not available for the web service.

`messagedialog` *`message ?title? ?messagetype? ?dialogbuttons?`*

`messagedialog(`*`message`*`[, `*`title`*`[, `*`messagetype`*`[, `*`dialogbuttons`*`]]])`

`message`: the message to be displayed

`title`: optional title of the window

`messagetype`: optional type, defining the icon of message window, selected from "`error`", "`information`", "`warning`", "`question`", or "`plain`" (default)

`dialogbuttons`: (optional) specifies which button or buttons to display, selected from "`ok`" (default), "`okcancel`", "`yesno`", or "`yesnocancel`".

mirrorhoriz

Horizontally inverts the module instance (with respect to the **vertical** axis, similar to the Mirror Y button in the PDE).

`mirrorhoriz` *`inst`*

`mirrorhoriz(`*`inst`*`)`

`mirrorhoriz?inst=<`*`inst`*`>`

`inst`: instance ID.

The instance is inverted with respect to its original (not rotated) orientation. So if it is rotated 90 or 270 degrees, then it is inverted vertically relative to the schematic axes.

mirrorvert

Vertically inverts the module instance (with respect to the **horizontal** axis, similar to the Mirror X button in the PDE).

`mirrorvert` *`inst`*

```
mirrorvert(inst)
```

```
mirrorvert?inst=<inst>
```

inst: instance ID.

The instance is inverted with respect to its original (not rotated) orientation. So if it is rotated 90 or 270 degrees, then it is inverted horizontally relative to the schematic axes.

modify

Modifies a variable that is bound to a text field, combo box, or checkbox of a wizard. This command works in the same way as the `set` command, but it also modifies the graphical component in the wizard bound to the variable. That is, the user sees the variable change immediately. For this command to take effect, it should be called after the variable is bound by the `wizard entry`/`wizard combo`/, etc. This command is not available for the web service.

```
modify variable value
```

```
modify(variable, value)
```

variable: name of a variable to be modified

value: new value for the above variable.

In Python, no real variable is used and the value can be accessed using the `getvariable` command.

Note: See the description of the `-variable` option in the `<pagename> entry` and `<pagename> button` commands (or `wizardentry()` and `wizardbutton()` in Python).

newmodulesweep

Creates a new *module sweep*. A module sweep is similar to a galaxy, but has no links, galaxy ports and junction nodes. It is used to compare the performance of several devices in a system by substituting them into a main schematic. For adding and removing stars, changing parameter settings, and saving the schematic, use the same commands that are used for universes and galaxies.

```
newmodulesweep ?name?
```

```
newmodulesweep([name])
```

```
newmodulesweep[?name=<name>]
```

name: the name of the module sweep. A temporary unique name, such as UNTITLED-1, is used if the parameter left out.

newstate

Creates a new global parameter of the current schematic (or galaxy) or a parameter of the current virtual galaxy, or modifies an existing parameter if the name already exists.

```
newstate name type value ?category? ?-func on|off? ?-exprtype
     standard|python|file? ?-show on|off? ?-enum {item1|item2| ...}?
     ?-context context_exp? ?-description desc? ?-unit unit? ?-range
     range?
```

```
newstate(name, type, value[, category[, func=True|False[,
     exprtype="standard"|"python"|"file"[, show=True|False[, enum[,
     context[, description[, unit[, range]]]]]]]]])
```

```
newstate?name=<name>&type=<type>&value=<value>[&category=<category>
     [&func=true|false][&exprtype=standard|python|file]
     [&show=true|false][&enum=<enum>][&context=<context>]
     [&description=<description>][&unit=<unit>][&range=<range>]]
```

name: existing or new name of the global variable parameter

type: the type of parameter

value: the value of the parameter

category: the category the parameter belongs to. Set to `Custom` if omitted.

`-func`: defines if the parameter is function (`on`/`True`) or regular (`off`/`False`). The `-func` keyword can be omitted. In this case, for existing parameters the "Function" flag remains unaltered and for new parameters it is treated as unset.

`-exprtype`: defines an expression type. This keyword can be omitted. In this case, for existing parameters, the expression type remains unaltered and for new parameters it is treated as `"standard"`.

`-show`: defines whether to show (`on`/`True`) the specified parameter under the icon of module/galaxy or not (`off`/`False`). The `-show` keyword can be omitted. In this case the parameter property "show" remains unaltered.

`-enum`: specifies the list of possible values of enumeration parameters. The list can be delimited by commas or vertical bars (|).

`-context`: specifies the context string, which must be a valid Tcl boolean expression, with references to other parameters, like
`($PRBS_Type == "PRBS_N") || ($PRBS_Type == "DB_KN")`.
Note, dollar signs must be escaped with a backslash character '\' if the context string is literally specified in macro code. An empty context string removes context dependency of an existing parameter.

`-description`: specifies the parameter description string.

`-range`: specifies the acceptable range of the parameter values, in the format `"<min>|<max>|closed|ignore"`.

`-unit`: specifies the parameter unit, see the list of supported units below.

The following parameter types are available: "string", "enumeration", "`int`", "`float`", "`complex`", "`inputfile`", "`outputfile`", "`inputdirectory`", "`outputdirectory`", "`stringarray`", "`intarray`", "`floatarray`", "`complexarray`", "`inputfilearray`", and "`outputfilearray`".

> Note: Only the parameter units supported in the **Edit Parameter Properties** dialog can be set using this command. Attempts to use an unsupported unit will cause an error.
> The command can not change the *type* and *category* of an existing schematic parameter.

newuniverse

Creates a new universe with a given name and domain.

```
newuniverse ?name? ?domain? ?-prodset set?
```

```
newuniverse([name[, domain[, prodset]]])
```

```
newuniverse?[name=<name>][&domain=<domain>][&prodset=<prodset>]
```

`name`: name with or without extension, a path or URN of a resulting schematic or galaxy. If *name* is omitted the new universe's name is generated automatically (such as “`UNTITLED-1`”)

`domain`: simulation domain (“`Auto`”, “`SDF`”, “`BDF`”, “`DDF`”, or “`SMATRIX`”). If domain is omitted it defaults to “`Auto`”.

`-prodset`: allows you to specify a desired product set for a new universe. The product set needs to be specified as a comma-separated list (without spaces) of product abbreviations (for example, "`OS,FO`").

If the command runs successfully, it returns the URN of the created schematic.

> Note: Here and below, for the web request, commas should be percent-encoded, such as "`OS%2CFO`".

openproject

Opens the project at a specified location. A previously opened project will be automatically closed.

```
openproject path
```

```
openproject(path)
```

```
openproject?path=<path>
```

path: a path to project folder.

openschematic

Opens the schematic with a given name.

`openschematic` *`path`* `?-prodset` *`set`*`?`

`openschematic(`*`path`*`[,` *`prodset`*`])`

`openschematic?path=<`*`path`*`>[&prodset=<`*`prodset`*`>]`

path: a path or URN of a universe or galaxy.

`-prodset`: the product set that will be set to the schematic on opening it. It needs to be specified as a comma-separated list (without spaces) of product abbreviations (for example, "`OS,FO`"). The schematic will not open if the specified product set is not suitable for it.

When a schematic is located inside the project folder structure, this project will be automatically opened while opening the schematic. A previously opened project will be automatically closed.

packagefolder

Returns the absolute path of the folder containing the "internal" folders (`Resources`, `Inputs`, `Outputs`, `Attachments`, and `Reports`) of the schematic or galaxy.

`packagefolder` *`path`*

`packagefolder(`*`path`*`)`

`packagefolder?path=<`*`path`*`>`

path: a path or URN of a schematic or galaxy.

parseschematic

Traces a path along a linear string of interconnected instances. Used to find the interconnectivity of a system.

`parseschematic ?`*`universe`*`?` *`first last`*

`parseschematic([`*`universe`*`,]` *`first`*`,` *`last`*`)`

`parseschematic?args=[<`*`universe`*`>%2C]<`*`first`*`>%2C<`*`last`*`>`

universe: a universe to be searched (the current universe if omitted)
first: the instance at the first (input) side of the string
last: the instance at the last (output) side of the string.

Searches a route from instance `first` to instance `last` in the given schematic. If the *`universe`* parameter is omitted the search is in the current schematic. If a way was found, this command returns a list of connections to the form

```
{  {{ outInst outPort outTerm { otermx otermy } }
     { inInst inPort inTerm { itermx itermy }}} ... }
```

outInst:	the star/galaxy instance where a wire begins
outPort:	the name of an output port of the above block
outTerm:	the index of its output terminal
otermx:	the x position of the above terminal
otermy:	the y position of the above terminal
inInst:	the star/galaxy instance where a wire ends
inPort:	the name of an input port of the above block
inTerm:	the index of its output terminal
itermx:	the x position of the above terminal
itermy:	the y position of the above terminal.

If *first* and *last* specify the same node, this command returns an empty list. If there is no direct path between *first* and *last*, an error message appears.

pathtourn

Converts an absolute path into a URN.

```
pathtourn path
pathtourn(path)
pathtourn?path=<path>
```

path:	a path of any file or directory.

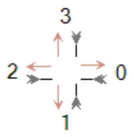
portplace

Gets or sets the placement information for a given port or terminal in the {`x y orientation`} form. If the instance belongs to a virtual galaxy and this galaxy is in context, the position will be relative to this virtual galaxy.

```
portplace inst port[#term] ?place?
portplace(inst, port[#term][, place])
portplace?inst=<inst>&port=<port>[%23<term>][&place=<placement>]
```

inst:	instance ID for an instance or URN for a module
port:	port name of a given instance or module
term:	terminal index of the port of a given instance or module when the port is a multiport

place: placement information (instances only).



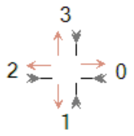
The `orientation` determines the side on which the port is located, taking into account the rotation and mirror of an instance. For example, a port that points to the right side of the schematic will have `orientation` equal to `0`, the down side equal to `- 1`, and so on.

If a desired placement is specified, the whole instance is moved so that the specified port is found in the specified position.

portposition

Note: This command is deprecated and should be replaced with the portplace command.

Command to get the position and rotation of a given port as a list of the form {`x y rotation mirrorHoriz mirrorVert`}. If the instance belongs to a virtual galaxy and this galaxy is in context, the position is returned relative to this virtual galaxy.



```
portposition inst port ?term?
portposition(inst, port[, term])
portposition?inst=<inst>&port=<port>
```

inst:	instance ID
port:	port of the given instance
term:	terminal of the port (defaults to `1`)

The command can be used to query the rotation (including mirroring) of a module. The input ports are reported with 180° rotation.

prodset

Gets or sets the product set of the current schematic.

```
prodset ?prodset?
prodset([prodset])
prodset?[prodset=<prodset>]
```

prodset: allows you to specify a desired product set for the current schematic. The product set needs to be specified as a comma-separated list (without spaces) of product abbreviations (for example, "`OS,FO`").

If the command run successfully, it returns the current product set of the schematic.

querydialog

Displays a dialog window to ask for user input. The command returns a value entered by the user. This command is not available for the web service.

```
querydialog type ?question? ?title? ?required? ?default?
```

```
querydialog(type[, question[, title[, required[, default]]]])
```

type: determines the expected datatype ("`integer`", "`double`", "`string`" or "`boolean`")
question: the text to be displayed
title: window title
required: if "`true`"/`True` it accepts no empty input
default: default value in the dialog.

For `boolean` type strings, "true" is considered as `True` all other strings are considered as `False`.

When the **Cancel** button is clicked, the string "`CANCEL`" is returned.

redraw

Repaints the schematic, in case there are unwanted glitches in the graphics.

```
redraw
```

```
redraw()
```

```
redraw
```

reset

Erases all contents of the current schematic.

```
reset
```

```
reset()
```

```
reset
```

rotate

Rotates the icon of the given instance clockwise by 90, 180, or 270 degrees.

```
rotate inst ?rot?
```

```
rotate(inst[, rot])
```

```
rotate?inst=<inst>[&rot=<rot>]
```

inst: instance ID

rot: number between 1–3, to rotate 90, 180, or 270 degrees (defaults to 1).

run

Submit the schematic for simulation.

```
run ?runs? ?-server server? ?-animation flag? ?-sweep sweepname?
    ?-logfile path? ?-logfilter filter?
```

```
run([runs[, server[, animation=True|False[, sweep=sweepname[,
    logfile=path[, logfilter=filter]]]]]])
```

```
run?runs=<runs>&server=<server>&animation=true|false&
    sweep=<sweepname>&logfile=<path>&logfilter=<filter>&
```

runs: number of runs (default 1)

server: name of the server (host or user@host)

flag: specifies if to simulate animation (on, true, 1 or off, false, 0)

sweepname: the name of a sweep to run (only the sweeps created by the `sweep` command are supported).

path: the path to file to write the simulation messages (such as status, warnings, and errors).

filter: logs only messages of specified type(s). Different message types can be combined via the “|” character. Supported types are: `all`, `error`, `warn`, `status`, `info`, `progr`, `anim`, and `trace`.

Returns a string which uniquely identifies the simulation. This ID can be used to wait until a simulation is finished by passing it to the `waitrun` command.

saveschematic

Saves the current schematic. If a name is given, the current schematic is saved under that name.

```
saveschematic ?name? ?-latestformat?
```

```
saveschematic([name, [latestformat=True|False]])
```

```
saveschematic?name=<path>[&latestformat=true|false]
```

name: name with extension, a path or URN of a resulting schematic or galaxy

`-latestformat`: whether to silently update schematic to the latest format during the save or ask confirmation (False by default).

selectedstars

Returns a list with the instance IDs of all currently selected instances. May also return the names of virtual galaxies.

```
selectedstars ?-includevg?
```

```
selectedstars(includevg=True|False)
```

```
selectedstars[?includevg=true|false]
```

`-includevg`: if this option is specified (`True`), the names of virtual galaxies are also returned.

selectfiledialog

Displays the standard dialog window enabling to select a file for either opening or saving. This command is not available for the web service.

```
selectfiledialog ?-folder dir? ?-filename name? ?-title title?
    ?-filefilter filter? ?-dialogtype type?
```

```
selectfiledialog([folder[, filename[, title[, filefilter[,
    dialogtype="0"|"1"]]]]])
```

`-folder` *`dir`*: path to an initial folder, for example, '`-folder "D:\\My Projects"`' (default is the current user's home directory)

`-filename` *`name`*: name of a file that would be initially selected in the dialog, for example, '`-filename "my data file.dat"`'

`-title` *`title`*: caption of the dialog window

`-filefilter` *`filter`*: list of accepted file types (extensions), delimited by pipe symbols "|", for example: '`-filefilter "Text files (*.txt)|Data files (*.dat)"`' (by default the 'All files' filter is set).

`-dialogtype` *`type`*: type of the dialog. Use `"0"` (default) for "Open" and `"1"` for "Save".

If the user clicks OK, the command returns an absolute path to a selected file; otherwise an empty string is returned.

selectstars

This command selects all instances or virtual galaxies of the schematic given as arguments.

```
selectstars inst1 ?inst2? ...
```

```
selectstars(inst1[, inst2[, ...]])
```

```
selectstars?args=<inst1>[%2C<inst2>[%2C ...]]
```

`instN`: instance ID of instance or virtual galaxy of the current schematic.

setbidirmode

Changes the mode of the bidirectional port group.

`setbidirmode` *`inst port mode`*

`setbidirmode(`*`inst, port, mode`*`)`

`setbidirmode?inst=<`*`inst`*`>&port=<`*`port`*`>&mode=<`*`mode`*`>`

`inst`: instance ID

`port`: name of one of the ports in the bidirectional port group of the instance you wish to alter

`mode`: one of "`bidir`", "`in`", "`out`", or "`inout`".

If the port is not bidirectional or the port type is "`elwave`", an error is issued.

setbitmap

Changes the icon of the current schematic to the bitmap image in the supplied file. The schematic cannot be read-only. To “change” an icon of a read-only module, it is necessary to ‘wrap’ it into a galaxy.

The supported image formats are `gif` and `jpeg`.

`setbitmap` *`file`*

`setbitmap(`*`file`*`)`

`setbitmap?file=<`*`path`*`>`

`file`: is the filename of the bitmap.

setbounds

Places the virtual galaxy at the specified position and changes its size.

`setbounds` *`inst left top right bottom`*

`setbounds(`*`inst, left, top, right, bottom`*`)`

`setbounds?inst=<`*`inst`*`>&left=<`*`left`*`>&top=<`*`top`*`>&right=<`*`right`*`>`
`&bottom=<`*`bottom`*`>`

`inst`: instance ID of the virtual galaxy

`left`: x-coordinate of the left, top corner

`top`: y-coordinate of the left, top corner

`right`: x-coordinate of the right, bottom corner

`bottom`: y-coordinate of the right, bottom corner.

setpos

Places the instance or virtual galaxy at the specified position (xpos, ypos). If the instance belongs to a virtual galaxy and this galaxy is in context, the position is set relative to this virtual galaxy.

```
setpos inst xpos ypos
```

```
setpos(inst, xpos, ypos)
```

```
setpos?inst=<inst>&xpos=<x>&ypos=<y>
```

inst: the instance ID

xpos: is the x position in half-grid units, measured rightward from the top left (0,0)

ypos: is the y position in half-grid units, measured downward from the top left (0,0).

setstate

Sets the values of instance or virtual galaxy parameters.

```
setstate inst param value ?-exprtype standard|python|file? ?-show
     on|off?
```

```
setstate(inst, param, value[, exprtype="standard"|"python"|"file"[,
     show=True|False]])
```

```
setstate?inst=<inst>&param=<param>&value=<value>
     [&exprtype=standard|python|file][&show=true|false]
```

inst: the instance ID

param: the parameter name

value: the value of the parameter

`-exprtype`: the expression type

`-show`: defines whether to show (`on` (`True`)) the specified parameter under the icon of module/galaxy or not (`off` (`False`)). The `-show` keyword can be omitted. In this case the parameter property "show" remains unaltered.

Note: To set the value of a **global** parameter of the current schematic (universe or galaxy), use the newstate command described above.

setview

Sets the size and position of the visible portion of the schematic (viewport) to the given coordinates.

```
setview xmin ymin xmax ymax
```

```
setview(xmin, ymin, xmax, ymax)
```

```
setview?xmin=<xmin>&ymin=<ymin>&xmax=<xmax>&ymax=<ymax>
```

xmin:	the leftmost position
ymin:	the topmost position
xmax:	the rightmost position
ymax:	the bottommost position.

setvirtgal

Assigns an instance to a virtual galaxy or the main schematic topology.

```
setvirtgal inst virtgal
```

```
setvirtgal(inst, virtgal)
```

```
setvirtgal?inst=<inst>&virtgal=<virtgal>
```

inst:	the instance ID
virtgal:	ID of the virtual galaxy

To assign an instance to the main schematic topology, specify an empty string for the virtual galaxy ID.

simexcluded

Excludes instances from simulation. Returns a list of IDs of all the instances excluded from a schematic.

```
simexcluded ?instances excluded?
```

```
simexcluded([instances, excluded])
```

```
simexcluded?[instances=<instance1>[,<instance2>]&excluded=True|False]
```

instances:	list of instance IDs
excluded:	whether to exclude specified instances (True or False)

If “instances” and “excluded” are omitted, returns the list of IDs of excluded instances of the current schematic.

star

Creates a new instance of the star or galaxy from a generic module in the main schematic topology or current virtual galaxy. Returns the instance ID of a newly created instance.

```
star inst module ?-place placement?
```

```
star(inst, module[, place])
```

```
star?inst=<inst>&module=<path>[&place=<placement>]
```

inst: an instance ID of your choice for the instance
module: the module path or URN
place: placement information.

To generate an instance ID automatically, use "." (dot) as an instance ID.

The placement information should be provided in the same form as for the starplace command. All parts located after the position may be omitted. For example, you may wish to specify only the position and rotation, or the position, rotation, and horizontal mirror.

Examples:

- Tcl:

```
star LinkAnalyzer_vtms1 {URN:VPI_LIB::TC Modules\Analyzers\LinkAnalyzer.vtms:}
star "." {URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:} -place {4 8}
star . {URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:} -place [list 6 10 2
1]
```

- Python:

```
star("LinkAnalyzer_vtms1", r"URN:VPI_LIB::TC Modules\Analyzers\LinkAnalyzer.vtms:")
star(".", r"URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:", (4, 8))
star(".", r"URN:VPI_LIB::TC Modules\Optical Sources\LaserCW.vtms:", [6, 10, 2,
True])
```

- Web API:

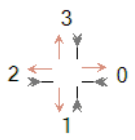
```
http://localhost:8475/pde/v1/star?inst=LinkAnalyzer_vtms1&module=URN%3AVPI_LIB%3A%3
  ATC+Modules%5CAnalyzers%5CLinkAnalyzer.vtms%3A
http://localhost:8475/pde/v1/star?inst=.&module=URN%3AVPI_LIB%3A%3ATC+Modules%5COpt
ical+Sources%5CLaserCW.vtms%3A&place=4,8
http://localhost:8475/pde/v1/star?inst=.&module=URN%3AVPI_LIB%3A%3ATC+Modules%5COpt
ical+Sources%5CLaserCW.vtms%3A&place=6,10,2,true
```

starplace

Gets or sets instance placement information in the {`x y rotation mirrorHoriz mirrorVert`} form. The position (x and y) is relative to the left upper corner of the schematic or currently active virtual galaxy.

Use the following numbers to set the rotation: "`0`" means no rotation, "`1`" is equal to 90° clockwise, "`2`" stands for 180° clockwise, and "`3`" specifies 270° clockwise.

The mirrors are Boolean flags.



`starplace` *`inst ?place?`*

`starplace(`*`inst`*`[, `*`place`*`])`

`starplace?inst=<`*`inst`*`>[?place=<`*`placement`*`>]`

inst: an instance ID
place: placement information.

statevalue

Command to find the values of parameters of module instances and virtual galaxies in the current schematic, or of global states of the current schematic or virtual galaxy if the *inst* specification is omitted.

```
statevalue ?inst? state ?option?
```

```
statevalue([inst,] state[, option])
```

```
statevalue?[inst=<inst>&]name=<name>[&option=<option>]
```

inst: an instance ID (can be omitted to refer to the parameters of the current schematic or current virtual galaxy)

state: the name of a parameter

option: decides which information of the parameter has to be returned.

If *option* is omitted, the current setting of the parameter will be returned. Otherwise it must have one of the following values (for Python and the web service, the hyphen should be omitted):

`-default`, to get the default value

`-type`, to get the datatype (see the description of the `newstate` command for a list of available types)

`-func`, to determine if parameter is a function (`on`/`True`) or regular (`off`/`False`)

`-category`, to get the category

`-unit`, to get the unit, for example, 's' for seconds

`-exprtype`, to get expression type ("`standard`", "`python`" or "`file`")

`-description`, to get the description

`-range`, to get the range

`-enum`, to get the list of enumeration values

`-context`, to get the context condition

`-show`, to determine if parameter is shown on schematic (`on`/`True`) or not (`off`/`False`).

The command fails for the parameters of encrypted galaxies and instances of those galaxies.

stopsimulation

Terminates the specified simulation. Can be used when the `waitrun` command was finished by time-out.

```
stopsimulation id
```

```
stopsimulation(id)
```

```
stopsimulation?id=<id>
```

id: is the simulation job ID as returned by the `run` command.

The command returns either `1` (`True`) if the specified job was successfully finished, or `0` (`False`) if the specified job ID was not found.

sweep

Defines a new sweep.

`sweep` *`sweepname`* `?`*`options`*`?`

`sweep(`*`sweepname, *options`*`)`

`sweep?sweepname=<`*`sweepname`*`>[&options=<`*`option`*`>%2C<`*`value`*`>[%2C]]`

`sweepname`: name of the sweep
`options`: options of the sweep.

After a sweep is created via the `sweep` command, it can be accessed by its name. The following options are available:

`-addrule` *`rulename`*: adds the sweeprule (Interactive Simulation control) given by name to the set of rules.

`-delrule` *`rulename`*: removes the given sweeprule (control) from the set of rules.

`-assign` *`parameter rulename`*: assigns a rule (control) to a parameter. The parameter must have the form `inst.Parametername` for parameters belonging to stars and `Parametername` for global variable parameters.

`-remove` *`parameter`*: removes the assignment of a parameter from the sweep.

`-setoption` *`name value`*: sets different options of sweep. Can be repeated several times.

`-open`: opens the sweep editor with the sweep.

`-load` *`path`*: loads a sweep from an existing `.vsw` file.

In Python and the web service, instead of options, the following set of commands exists:

`sweepaddrule(`*`sweepname, rulename`*`)`

`sweepaddrule?sweepname=<`*`sweepname`*`>&rulename=<`*`rulename`*`>`

`sweepdelrule(`*`sweepname, rulename`*`)`

`sweepdelrule?sweepname=<`*`sweepname`*`>&rulename=<`*`rulename`*`>`

`sweepassign(`*`sweepname, parameter, rulename`*`)`

`sweepassign?sweepname=<`*`sweepname`*`>&parameter=<`*`parameter`*`>`
    `&rulename=<`*`rulename`*`>`

`sweepremove(`*`sweepname, parameter`*`)`

`sweepremove?sweepname=<`*`sweepname`*`>&parameter=<`*`parameter`*`>`

`sweepsetoption(`*`sweepname, option, value`*`)`

`sweepsetoption?sweepname=<`*`sweepname`*`>&option=<`*`option`*`>&value=<`*`value`*`>`

`sweepopen(`*`sweepname`*`)`

`sweepopen?sweepname=<`*`sweepname`*`>`

`sweepload(`*`sweepname, path`*`)`

`sweepload?sweepname=<`*`sweepname`*`>&path=<`*`path`*`>`

The following sweep options can be set:

`-setoption mode sweep|tuning|optimization|yield`
: Sets mode of sweep.

`-setoption interactive on|off`
: Enables or disables an interactive sweep. Valid only for ***sweep*** mode.

`-setoption parallel on|off`
: Enables or disables parallel calculations in sweeps. Valid only for ***sweep*** mode.

`-setoption tickslider on|off`
: Switch on or off the "Stick slider to ticks" option.

Note: For options that expect a Boolean value in Tcl, like `interactive`, `parallel`, and `tickslider`, use `True` and `False` in Python and Web API.

Note: `-setoption` affects the options of a sweep only until the sweep is opened via `-open` option. So all the `-setoption` options will be placed before the `-open` option.

An example in Tcl:

```
sweep sweep1 -setoption interactive off -setoption parallel on
sweep1 -addrule rule1
sweep1 -open
```

An example in Python:

```
sweep("sweep1")
sweepsetoption("sweep1", "interactive", False)
sweepsetoption("sweep1", "parallel", True)
sweepaddrule("sweep1", "rule1")
sweepopen("sweep1")
```

The example above defines a sweep with the name ***sweep1***. The interactive mode is switched off and the parallel calculation of sweep points is switched on. Next ***rule1*** is added to the sweep ***sweep1***, and after that the sweep editor is opened for the newly created sweep.

sweeprule

Defines a new sweep rule (Interactive Simulation control).

`sweeprule `*`ruletype rulename ?options?`*

`sweeprule(`*`ruletype, rulename`*`[, `*`options`*`])` `sweeprule?ruletype=<`*`ruletype`*`>&rulename=<`*`rulename`*`>[&<option>=<`*`value`*`>]`

`ruletype`: type of the sweep rule ("`random`", "`continuous`" or "`list`")

`rulename`: name of the rule

`options`: options of the sweep rule.

To modify an existing sweep rule, use the following commands:

`rulename ?options?`

`sweeprulemodify(`*`rulename, options`*`)`

`sweeprulemodify?rulename=<`*`rulename`*`>&<option1>=<`*`value1`*`>[&<option2>=<`*`value2`*`>[...]]`

`rulename`: name of the rule

`options`: options of the sweep rule.

Options for Random Sweeps

The options of random sweeps depend on the type of the rule, and are as follows:

`-depth` *`sweepdepth`*: determines the order the sweeprules are applied

`-datatype` *`type`*: sets the type of the sweep values to *`type`*, must be "`int`" or "`float`"

`-distribution` *`distribution`*: sets the type of distribution to *`distribution`*, must be "`gaussian`" or "`uniform`"

`-iterations` *`number`*: sets the number of iterations to *`number`*

`-lowerlimit` *`limit`*: sets the lower limit for uniform distribution to *`limit`*

`-upperlimit` *`limit`*: sets the upper limit for uniform distribution to *`limit`*

`-mean` *`mean`*: sets the mean for Gaussian distribution to *`mean`*

`-variance` *`variance`*: sets the variance for Gaussian distribution to *`variance`*.

An example in Tcl:

```
sweeprule random rule1
rule1 -depth 1
rule1 -datatype float
rule1 -iterations 10
rule1 -distribution gaussian
rule1 -mean 3.0
rule1 -variance 1.0
```

An example in Python:

```
sweeprule("random", "rule1")
sweeprulemodify("rule1", depth=1)
sweeprulemodify("rule1", datatype="float")
sweeprulemodify("rule1", iterations=10
sweeprulemodify("rule1", distribution="gaussian")
sweeprulemodify("rule1", mean=3.0)
sweeprulemodify("rule1", variance=1.0)
```

The example above defines a random sweep rule with a Gaussian distribution. The mean is set to the initial value of the parameter assigned to that rule.

Options for Continuous Sweeps

The options of continuous sweeps depend on the rule type, and are as follows:

`-depth` *`sweepdepth`*: determines the order the sweep rules are applied, so that a multidimensional plot can be created.

`-datatype` *`datatype`*: type of the sweep values: must be "`int`" or "`float`"

`-startvalue` *`value`*: sets the first sweep value to *`value`*

`-endvalue` *`value`*: sets the last sweep value to *`value`*

`-stepsize` *`size`*: sets the size between two sweep values to *`size`*

`-iterations` *`iterations`*: sets the number of iterations between the start-value and end-value to *`iterations`* (*Note*: Either `stepsize` or `iterations` must be defined, not both.)

`-expression` *`expr`*: the expression to be applied to the current sweep value.

An example in Tcl:

```
sweeprule continuous rule2
rule2 -depth 1
rule2 -datatype float
rule2 -startvalue 1.0
rule2 -endvalue 2.0
rule2 -stepsize 0.1
```

An example in Python:

```
sweeprule("continuous", "rule2")
sweeprulemodify("rule2", depth=1)
sweeprulemodify("rule2", datatype="float")
sweeprulemodify("rule2", startvalue=1.0)
sweeprulemodify("rule2", endvalue=2.0)
sweeprulemodify("rule2", stepsize=0.1)
```

This example defines a continuous sweep rule with start value `1.0`. The assigned parameter(s) will be incremented by `0.1` each iteration until the end value is reached.

Options for List Sweeps

The options for list sweeps are as follows:

`-depth` *`depth`*: determines the order the sweep rules are applied

`-datatype` *`type`*: type of the sweep values, valid types are "`int`", "`float`", "`complex`", "`string`", "`floatarray`", "`intarray`", "`complexarray`", "`stringarray`", and "`file`"

`-values` *`val1 val2`* `...`: adds the remaining arguments to the list of sweep values

`-expression` *`expr`*: expression to be applied to the current sweep value.

All options may be abbreviated unless they become ambiguous.

Note: Sweep rules with a different number of iterations cannot have the same depth.

Examples in Tcl:

```
sweeprule list rule1
rule1 -depth 1
rule1 -datatype float
rule1 -values 8e-2 1.6e-1

sweeprule list rule2
rule2 -depth 1
rule2 -datatype intarray
rule2 -values {1 2} {3 4}

sweeprule list rule3
rule3 -datatype complex
rule3 -values (1.5,3) (4.1,5.5)

sweeprule list rule4
rule4 -datatype complexarray
rule4 -values {(1,2) (3,4)}
rule4 -values {(5,6) (7,8)}
```

Examples in Python:

```
sweeprule("list", "rule1")
sweeprulemodify("rule1", depth=1)
sweeprulemodify("rule1", datatype="float")
for v in [8e-2, 1.6e-1]:
      sweeprulemodify("rule1", values=v)

sweeprule("list", "rule2")
sweeprulemodify("rule2", datatype="intarray")
for v in ["1 2", "3 4"]:
      sweeprulemodify("rule2", values=v)

sweeprule("list", "rule3")
sweeprulemodify("rule3", datatype="complex")
for v in ["(1.5,3)", "(4.1,5.5)"]:
      sweeprulemodify("rule3", values=v)

sweeprule("list", "rule4")
sweeprulemodify("rule4", datatype="complexarray")
for v in ["(1,2) (3,4)", "(5,6) (7,8)"]:
      sweeprulemodify("rule4", values=v)
```

uniquename

Returns a name in the form of "UNTITLED-1,2,3 etc.", which is unique to the current session.

```
uniquename
```

```
uniquename()
```

```
uniquename
```

urntopath

Converts a URN into a an absolute path.

urntopath *urn*

urntopath(*urn*)

urntopath?urn=<*urn*>

usesimscript

Controls whether a user-defined simulation script attached to the current schematic will be used during the simulation. Returns the current value of the use simulation script flag.

```
usesimscript ?on|off?
```

```
usesimscript([use=True|False])
```

```
usesimscript[?use=true|false]
```

use: on (True) to enable user defined simulation script, off (False) to disable it.

virtgal

Creates a new virtual galaxy in the main schematic topology. If the position and/or size are omitted, the virtual galaxy is created at the default position and with the default size. If the position and size are not specified, the virtual galaxy creates around the instances specified by the *children* option. Returns the instance ID of a newly created virtual galaxy.

```
virtgal inst ?x y ?width height?? ?-children children?
```

```
virtgal(inst[, x, y[, width, height]][, children])
```

```
virtgal?inst=<inst>[&x=<x>&y=<y>[&width=<width>&height=<height>]][&ch
    ildren=<children>]
```

inst: instance ID of the virtual galaxy
x: x-coordinate of the left top corner
y: y-coordinate of the left top corner
width: width of the virtual galaxy
height: height of the virtual galaxy.
children: IDs of instances to assign to the newly created virtual galaxy.

To generate an instance ID automatically, use "." (dot) as an instance ID.

A newly created virtual galaxy is made current.

waitrun

Waits until all simulations with the specified identifiers are finished.

```
waitrun id1 ?id2? ... ?idn? ?-timeout timeout?
```

```
waitrun(id1[, id2[, ... idn[, timeout=timeout]]])
```

```
waitrun?jobids=<id1>[%2C<id2>[%2C ...]][&timeout=<timeout>]
```

id1 ... idn: the IDs of simulations returned by the run command *timeout*: timeout interval in milliseconds. The function returns if the interval elapses, even if some simulations aren't finished.

The command returns one of the following results:

`1`: the simulation was successfully finished

`0`: a timeout was encountered (the simulation is likely still running)

`-1`: the simulation finished with an error.

writemessage

Adds a message to the Message Log.

```
writemessage msg ?-type progress|info|warning|error?
```

```
writemessage(msg[, type="progress"|"info"|"warning"|"error"])
```

```
writemessage?msg=<msg>[&type=progress|info|warning|error]
```

msg: the message to be added

-type: the optional parameter that defines whether the message will be displayed as an informational (default), warning, error, or progress message.

zoomall

Sizes the zoom of the schematic so that all blocks are visible within the schematic.

```
zoomall
```

```
zoomall()
```

```
zoomall
```

Wizard Command

The wizard command displays an interactive wizard (a dialog), on which you can place descriptions, questions, and from which you can receiver user input.

The wizard command has several options defined. All options can be written in the same line. However, for readability it is recommended to define only one option per line.

Note: This functionality is not available for the web service.

wizard clear

Resets the current wizard state (size, title, pages, etc.). Should be called if several different wizards should be created in one macro.

```
wizard clear
```

```
wizardclear()
```

wizard width

Sets the width of the wizard page area to ***width***.

```
wizard width width
```

```
wizardwidth(width)
```

wizard height

Sets the height of the wizard page area to ***height***.

```
wizard height height
```

```
wizardheight(height)
```

wizard size

Sets the width and height of the of the wizard page area to ***width*** × ***height***.

```
wizard size width height
```

```
wizardsize(width, height)
```

If one of the options above is set, the wizard window will not be resizable.

wizard title

Sets the title of the wizard window.

```
wizard title title
```

```
wizardtitle(title)
```

wizard open

This call displays the wizard window. The command argument must contain a *Tcl* script (or a Python function that takes no arguments). The script will be executed once the **Finish** button has been clicked.

```
wizard open command
```

```
wizardopen(command)
```

wizard gotopage

This option immediately opens the page given as parameter.

```
wizard gotopage pagename
```

```
wizardgotopage(pagename)
```

wizard addpage

This option will add a new page to the wizard window. The string *pagename* must not contain spaces or other special characters, because in Tcl it is created as a new command. In Python, it is used as a key to identify a page in page option commands.

```
wizard addpage pagename
```

```
wizardaddpage(pagename)
```

Wizards with multiple pages are created by using multiple `wizard addpage` commands. In Tcl, multiple wizard commands can be combined on a single line so long as the syntax is unambiguous. For example, to create a Wizard with four pages, the following line can be used:

```
wizard addpage page1 addpage page2 addpage page3 addpage page4
```

Page options

After a page has been created by the wizard, it can be filled by a number of different items and controls. If a page with name `pagename` has been created (with `wizard addpage` *pagename*), the following options for *pagename* are available.

<pagename> label

This option will add a text with alignment to the page.

```
pagename label label ?-tooltip tooltip?
```

```
wizardlabel(pagename, label[, tooltip])
```

The `-tooltip` option provides a pop-up tooltip displaying the string *`tooltip`*.

<pagename> text

This option does the same as the `label` option, but without alignment.

```
pagename text text
```

```
wizardtext(pagename, text)
```

<pagename> newline

If used between entry commands (below), it ensures that the second entry has a line of its own, otherwise the entries will appear on the same line. This is similar to a newline command for printers.

```
pagename newline
wizardnewline(pagename)
```

<pagename> separator

Creates a thin visible line to separate several lines of text.

```
pagename separator
wizardseparator(pagename)
```

<pagename> checkbox

Displays a checkbox.

```
pagename checkbox title vname ?-command procedure? ?-tooltip tooltip?
wizardcheckbox(pagename, title, vname[, command[, tooltip]])
```

The parameter *`title`* specifies a descriptive text that will appear to the right of the checkbox.

The parameter *`vname`* must be the name of a global *Tcl* variable. If that variable does not exist, it will be created. It contains the status of the checkbox (either `0` (`False`) or `1` (`True`)). No variable is used for Python. Instead the `getvariable` command will be used to access this value.

The optional parameter `-command` requires as argument a *Tcl* script (or Python function which takes no arguments). That script is executed when the checkbox state is changed.

The `-tooltip` option can be used to add a tooltip with a message *`tooltip`*.

<pagename> combo

Adds a combo box to the page.

```
pagename combo vname enumeration ?-width width? ?-editable true|false?
     ?-validate validate_proc? ?-focuslost focuslost_proc? ?-command
     command? ?-tooltip tooltip?

wizardcombo(pagename, vname, enumeration[, width[,
     editable=True|False[, validate[, focuslost[, command[,
     tooltip]]]]]])
```

The parameter ***vname*** must be a global *Tcl* variable. If the variable does not exist, it will be created. The variable contains the current choice visible in the text field. No variable is used for Python. Instead the `getvariable` command can be used to access this value.

The parameter `enumeration` must be a list of strings separated by '|' symbols (a standard list in Python). It is an enumeration of strings to be displayed, allowing the user to choose one from the list.

The optional parameter `-width` sets the width of the combo box.

If the parameter `-editable` is set to `true`, the user is able to enter values into the text field which do not occur in the enumeration. It defaults to `false`.

The parameter `-validate` requires a *Tcl* script (or a Python function that takes no arguments). That script must return either `0` (`False`) or `1` (`True`). Whenever the user changes the contents of the combo box, the script is executed. It can be used to prevent the macro from being continued with invalid data. If the user enters a value out of range, for example, the script should return `0` (`False`). This will disable the **Next** or **Finished** button. So the user is expected to correct the value before continuing or finishing the wizard.

The parameter `-focuslost` expects also a *Tcl* script (or a Python function that takes no arguments). This script is executed whenever the combo box acquires or loses the focus, or a value is selected in the popup.

The parameter `-command` expects a *Tcl* script (or a Python function that takes no arguments) which is called when the value is selected from the dropdown list.

The `-tooltip` option can be used to add a tooltip.

<pagename> entry

This produces an entry field for text and numbers. It includes features to make the contents of one field depend on the last entry of another field. This can be used to specify parameters in multiple units or forms, such as in wavelength or frequency terms.

```
pagename entry vname ?-width width? ?-editable true|false? ?-validate
    procedure? ?-variable variable? ?-focuslost procedure? ?-tooltip
    tooltip?
```

```
wizardentry(pagename, vname[, width[, editable=True|False[, validate[,
    variable[, focuslost[, tooltip]]]]]])
```

The parameter ***vname*** must be a global *Tcl* variable. If the variable does not exist, it will be created. The variable contains the current choice visible in the text field. No variable is used for Python. Instead `getvariable` command can be used to access this value.

The optional parameter `-width` sets the width of the entry.

If the parameter `-editable` is set to `true`, the user is able to enter values. It defaults to `true`.

The parameter `-validate` has the same meaning as for combo.

The parameter `-focuslost` expects a *Tcl* script (or a Python function that takes no arguments). The script is executed whenever the entry acquires or loses the focus. The script may return some value.

The `-variable` option specifies the name of the variable that will be changed when the user moves the input focus (cursor) out of the entry field. The specified variable will then have a value returned by the procedure specified by the `-focuslost` option, and the wizard elements that use the variable will be updated (as with the `modify` command).

The `-tooltip` option can be used to add a tooltip.

Note: For the entry fields, if the `-validate` option is not set, the input is regarded as valid when the entry is **not empty** (when the user entered at least one symbol). Use a procedure that always returns 1 (true) to enable empty strings.

<pagename> button

This command adds a new button with given title to the page

```
pagename button title ?-width width? ?-variable variable? ?-command
     procedure? ?-tooltip tooltip?
```

```
wizardbutton(pagename, title[, width[, variable[, command[,
     tooltip]]]])
```

The option `-width` sets the button width.

The parameter `-command` expects a *Tcl* script (or a Python function that takes no arguments) which is called when the button is clicked. The script may return some value.

The `-variable` option specifies the name of the variable that will be changed to the value returned by the script specified via `-command` when the button is clicked, and the wizard elements that use this variable will be updated (as with the `modify` command).

The `-tooltip` option can be used to add a tooltip.

<pagename> nextpage

The `nextpage` command makes it possible to change the order of the pages, without moving chunks of the script. This command *chains* pages together. Note that every page must be defined in the `wizard addpage` command.

```
pagename nextpage ?otherpage?
```

```
wizardnextpage(pagename[, otherpage])
```

`otherpage:` is the name of another wizard page (or empty)

The command defines the page to be displayed once the **Next** button is pressed. If `otherpage` is an empty string (for example, *`pagename`* `nextpage`), *`pagename`* will be the last page.

<pagename> prevpage

The `prevpage` command is similar to the `nextpage` command. It defines the page to be displayed when the **Back** button is pressed.

```
pagename prevpage ?otherpage?
```

```
wizardprevpage(pagename[, otherpage])
```

`otherpage` is the name of another wizard page (or empty)

COM Object Commands

The COM object commands allows communication with an external server using the late binding approach. Macros can serve as automation controllers for external applications that support COM Automation (such as Microsoft Office). All of the necessary properties and methods for working in macros with the commands listed below are described in the Microsoft Office Help under **Microsoft (Word/Excel/PowerPoint, etc.) Visual Basic Reference**.

These commands are not available in Python and the web service. In Python, the same functionality may be reproduced using `pywin32`, a third-party library distributed together with VPI Design Suite.

createobject

Creates external ActiveX object instance based on progID.

`createobject` *`progId`*

`progId`: the COM ProgID - string which uniquely identifies object type, for example "Excel.Application"

The command returns the *`objectInstance`* value—the instance that will be used in subsequent calls of `releaseobject`, `putproperty`, `getproperty` and `invoke` commands.

Example:

```
set excel [createobject "Excel.Application"]
```

releaseobject

This command releases ActiveX object instance previously created with `createobject` command.

`releaseobject` *`objectInstance`*

Example:

```
set excel [createobject "Excel.Application"]
releaseobject $excel
```

putproperty

Modifies a property of an ActiveX object.

`putproperty` *`objectInstance property_name property_value`*

`property_name`: the name of the property of an object

`property_value`: the value of the specified property.

Example:

```
set excel [createobject "Excel.Application"]
putproperty $excel "Visible" true
```

getproperty

Returns a property value of an ActiveX object.

`getproperty` *`objectInstance property_name property_value`*

Example:

```
set excel [createobject "Excel.Application"]
set isVisible [getproperty $excel “Visible“ true]
```

If the property implies returning an object reference, this reference can be used in subsequent calls of the `getproperty`, `putproperty` and `invoke` methods.

invoke

Invokes a method on an object instance with a set of arguments. Returns a value (object ID, data, etc.) specific for the corresponding ActiveX application.

`invoke` *`objectInstance "method_name" arguments_list`*

`method_name`: the name of the method running on an object

`arguments_list`: the list of available arguments.

Example:

```
set excel [createobject "Excel.Application"]
set wbooks [getproperty $excel "Workbooks"]
set wbook [invoke $wbooks "Open" $path]
```

If invoking of particular method implies returning of object reference then this reference can be used in subsequent calls of `getproperty`, `putproperty` and `invoke` methods.

File Transfer Commands

The following commands enable a limited API to transfer files using web access.

As a security measure, only the files from the standard 'internal' folders of the current schematic are available.

Delete

Deletes a specified file or folder on the target computer (server).

`file/delete?folder=`*`<folder>`*`&path=`*`<path>`*

`folder`: the folder in the opened schematic, such as `Resources`, `Inputs`, `Outputs`, `Attachments`, or `Reports`.

`path`: the path to file or subfolder within the specified folder.

Download

Downloads a file from a server computer to a client computer. The payload of the server response is the requested file in chunked transfer encoding. The HTTP header `Content-Type: application/octet-stream` is used.

```
file/download?folder=<folder>&file=<file>
```

`folder`: the folder in the opened schematic, such as `Resources`, `Inputs`, `Outputs`, `Attachments`, or `Reports`.

`file`: the name of file within the specified folder.

Upload

Uploads a file to a server using a POST HTTP request. The payload of this request should be a stream containing the local file in the `application/x-www-form-urlencoded` format (chunked transfer encoding is used).

```
file/upload?folder=<folder>&file=<file>&overwrite=true|false
```

`folder`: the folder in the opened schematic, such as `Resources`, `Inputs`, `Outputs`, `Attachments`, or `Reports`.

`file`: the target name of file within the specified folder.

`overwrite`: the boolean flag to overwrite an existing file of the server. If set to `false` and the file already exists, an error will be returned.

Appendix B

Build-In Macros

As mentioned in Chapter 1 “Macros”, VPIphotonics Design Suite™ allows you to combine a sequence of commands in a script that can be called from a menu or within a schematic to automate a task or series of tasks. These scripts are called “macros” and written in VPI’s *PDE Scripting Language*. Moreover, VPIphotonics Design Suite™ offers built-in standard macros available from the Macros button menu on the Home tab of the ribbon. Table B-1 below provides a complete list of built-in macros along with a brief description of their functionality.

Table B-1 *VPIcomponentMaker™ Macro Description*

Macro Name	Description	Notes
Set Global Parameter	allows you to automatically choose the single signal bit/symbol rate, sample rates, and the signal duration according to your needs, but still ensure high efficiency and accuracy of the simulations. The macro will check that the values you chose satisfy the requirements. If not, the macro will suggest alternative values that are closest to your choices.	
Set Global Parameters (Multi-Symbol Rate)	allows you to automatically choose the complex signal, involving signals with different symbol rates bit/symbol rates, sample rates, and the signal duration according to your needs, but still ensure high efficiency and accuracy of the simulations. The macro will check that the values you chose satisfy the requirements. If not, the macro will suggest alternative values that are closest to your choices. See the demonstration *Optical Systems Demos >Simulation Techniques > General > Multiple Symbol Rate Simulation.*	
Search Global Parameters	allows you to recursively find all the modules and module parameters where the specified global schematic parameter is used. The parameters that depend on specified global parameters and on each other are also sought.	

Macro Name	Description	Notes
Set Bidir Port Mode	switches the default view of all the bidirectional ports on a schematic. This is useful, for example, when you wish to convert all the bidirectional PIC Elements to unidirectional or vice versa. Importantly, all the links between the module ports are preserved whenever possible.	
Link Components	allows you to automatically link neighboring ports of the modules placed on schematics.	
Add NullSource Module(s)	allows you to add instances of the *NullSource* modules nearby and link them to all the unconnected input ports of modules placed on schematics. These macros should usually be used after the Link Components macro.	
Add Ground Terminations	allows you to add instances of *Ground* modules nearby and link them to all the unconnected output ports of modules placed on schematics. These macros should usually be used after the Link Components macro.	
Add NullSource and Ground Module(s)	allows you to add instances of *Ground* and *NullSource* modules nearby and link them to all the unconnected output or input ports of modules placed on schematics. These macros should usually be used after the Link Components macro.	
Edit Multiple Modules	lets you easily set identical values for the desired common parameters in a group of selected modules, stars, and/or galaxies.	
Compare Parameters	allows you to compare and set parameters with identical names in two selected modules. These modules can even be of different types (for example, two different lasers or receivers).	
Parts List	allows you to create a list of all modules present on a schematic, calculating also the number of instances for each of these modules.	
Create Custom Module	lets you create a galaxy using as a template some other module or galaxy with a suitable set of ports and parameters (which you only need to slightly adjust for your needs).	
Set Frequency Domain Simulations	helps to configure your schematic for frequency-domain simulations of passive photonic circuits. The macro switches all the optical and electrical sources to periodic Block mode, creates S-matrix domain virtual galaxies around the passive subcircuits, and lets you adjust the global simulation control and user-defined parameters.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Set Time Domain Simulations	serves to configure your schematic for time-domain simulations of active and hybrid photonic circuits. The macro switches all the optical and electrical sources to aperiodic Sample mode, creates S-matrix domain virtual galaxies around the passive subcircuits, and lets you adjust the global simulation control and user-defined parameters.	The macro is only available for VPIcomponentMaker Photonic Circuits.

Macro Name	Description	Notes
Set Accurate PhotonicsTLM Simulations	helps to set the global parameter **`SampleModeBandwidth`** for most accurate simulations of *PhotonicsTLM* and derived modules, and allows to monitor the accuracy of the used TLM discretization for each particular device section.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Add PIC_CoSimInterface Galaxy Parameters	helps to automatically create all the standard (for custom cosimulated PIC Elements) galaxy parameters and configure accordingly the parameters of the *PIC_CoSimInterface* module.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Add SMATRIX-Domain Galaxy Parameters	helps to automatically create all the standard (for custom compound PIC Elements) SMATRIX-domain galaxy parameters with all the required parameter descriptions, context dependencies, allowed value ranges, and default values.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Create Measured Passive Component	allows you to automatically create a custom measured passive PIC Element (as an SMATRIX-domain galaxy) whose S-matrix is described by the given S-matrix file. The required S-matrix file could be generated based on simulations or measurements using external scripts. The file should comply with the file format described in the "Measured Models of PIC Elements" section of Photonic Circuits User's Manual.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Characterize Passive Circuit	lets you automatically create a new test setup which replicates the selected passive subcircuit and all the global schematic parameters, and adds *FuncImpulseOpt* and *SignalAnalyzer* modules connected to the selected subcircuit ports to calculate and visualize the device transfer functions.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Characterize Laser	allows you to automatically create a new test setup which replicates the selected laser (or an equivalent subcircuit with one input electrical port for the injected current and one output optical port for the emitted light) and all the global schematic parameters, and adds the *TestSetLaser* module connected to the selected device ports to calculate and visualize the required laser characteristics.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Characterize SOA	lets you automatically create a new test setup which replicates the selected laser (or an equivalent subcircuit with one input electrical port for the injected current, one input optical port for the launched signal, and one output optical port for the amplified signal) and all the global schematic parameters, and adds the *TestSetSOA* module connected to the selected device ports to calculate and visualize the required amplifier characteristics.	The macro is only available for VPIcomponentMaker Photonic Circuits.
Characterize Modulator	allows you to automatically create a new test setup which replicates the selected optical modulator (or an equivalent subcircuit with one input electrical port for the applied modulation voltage, one input optical port for the launched signal, and one output optical port for the modulated signal) and all the global schematic parameters, and adds the *TestSetModulator* module connected to the selected device ports to calculate and visualize the required modulator characteristics.	The macro is only available for VPIcomponentMaker Photonic Circuits.

Macro Name	Description	Notes
Set Fiber Type	adjusts the parameters of the selected single-mode fiber module (an instance of *UniversalFiber*, *UniversalFiberFwd*, *FiberNLS*, *FiberNLS_PMD*, *TimeDomainFiber*, *or FiberRamanDynamic*) so that it will simulate one of the commercially available optical fibers.	The macro is only available for VPItransmissionMaker Optical Systems and VPI-componentMaker Fiber Optics.
Set Multimode Fiber Type	adjusts the parameters of the selected multimode fiber module (an instance of *SolverFiberMM* or *SolverMeasuredFiberMM*) so that it will simulate one of the commercially available optical fibers including OM3, OM4, and Wide Band Multimode Fiber.	The macro is only available for VPItransmissionMaker Optical Systems and VPI-componentMaker Fiber Optics.
Calculate Message Length for BCH Code	allows you to calculate a message length for a specified codeword length (and an error-correction capability) for BCH codes. The calculated value can be used for the parameter **`CodewordLength`** of the *FECencoder* module.	The macro is only available for VPItransmissionMaker Optical Systems and VPIlab-Expert.
Synthesize WDM Link	allows you to set up a multispan DWDM system.	The macro is only available for VPItransmissionMaker Optical Systems and VPIcomponentMaker Fiber Optics.
Verify WDM Link	lets you verify a multispan DWDM system.	The macro is only available for VPItransmissionMaker Optical Systems and VPI-componentMaker Fiber Optics.
Set ITU Frequency	allows you to easily set the frequency of some selected optical source (lasers and/or transmitters) according to the ITU frequency classification (and to remind you such a classification, if needed).	The macro is only available for VPItransmissionMaker Optical Systems and VPI-componentMaker Fiber Optics.
Convert Data File	lets you convert measured data file formats to the formats compatible with VPI Design Suite. Namely, the Convert Data File macro provides four different Python based converters: • (Time,Frequency,Power) To (Time,Phase, Amplitude) • High Resolution Spectrometer to Power Spectrum • CITIfile To FilterMeasuredEl • Touchstone To FilterMeasuredEl.	The macro is only available for VPItransmissionMaker Optical Systems and VPI-componentMaker Fiber Optics.

Appendix C

Simulation Engine Driver — API Reference

As mentioned in Chapter 4, “Simulation Engine Driver”, the VPI Design Suite Simulation Engine Driver is a .NET component which provides an Application Programming Interface (API) for external clients. As a pure .NET component, the Simulation Engine Driver has a native .NET API which is accessible from any client written for the .NET platform.

Additionally it provides a COM Automation API which is accessible in any third-party language supporting COM, such as Python, MATLAB, C++, VisualBasic, VisualBasic for Applications (from Microsoft Office tools), etc.

This chapter provides a detailed description of all API objects and their members (methods and properties).

Comprehensive examples of usage are given with the function definitions, and the examples show enough context to demonstrate how the API can be used to perform real tasks.

Object Diagram

The hierarchy shown in Figure C-1 provides an overview of the objects and collections that can be accessed via the API. As shown in the figure, the API is very compact.

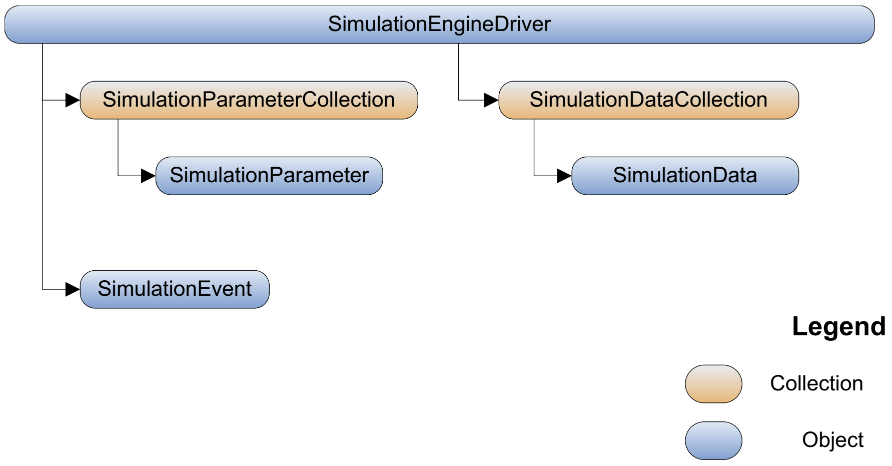


Figure C-1 *Simulation Engine Driver objects diagram*

Objects

SimulationData

Description

Represents named runtime data which can be received from the simulation engine during simulation and sent back.

Currently, scalar and matrix numerical values are supported.

Access

Can be acquired from the collections of runtime data which are accessible through either `InputRuntimeData`, `OutputRuntimeData` (scalar), or `OutputData` (scalar or matrix) properties of `SimulationEngineDriver`:

- MATLAB:

```
inData = driver.InputRuntimeData.Add('A1', 10.5);
inData = driver.InputRuntimeData.Item('A1');
outData = driver.OutputRuntimeData.Item('A1');
% Getting matrices works the same way
mxOutData = driver.OutputData.Item('M1');
```

- Python:

  ```
  inData = driver.InputRuntimeData.Add("A1", 10.5)
  inData = driver.InputRuntimeData.Item("A1")
  outData = driver.OutputRuntimeData.Item("A1")
  # Getting matrices works the same way
  mxOutData = driver.OutputData.Item("M1")
  ```

Properties

Key

Description

Gets the key assigned to the data object.

Examples

- MATLAB:

  ```
  fprintf('Key = %s\n', data.Key);
  ```

- Python:

  ```
  print("Key = " + data.Key)
  ```

Value

Description

Gets or sets the value of the data object.

Currently, numerical and matrix values are supported.

Examples

- MATLAB:

  ```
  fprintf('Value = %g\n', data.Value);
  ```

- Python:

  ```
  print("Value = " + str(data.Value))
  ```

SimulationDataCollection

Description

Represents a collection of `SimulationData` objects received from the simulation engine or prepared for sending back.

Access

Can be acquired from the `SimulationEngineDriver` object:

- MATLAB:

```
inputRuntimeData = driver.InputRuntimeData;
outputRuntimeData = driver.OutputRuntimeData;
outputData = driver.OutputData;
```

- Python:

```
inputRuntimeData = driver.InputRuntimeData;
outputRuntimeData = driver.OutputRuntimeData;
outputData = driver.OutputData;
```

Properties

Count

Description

Gets the number of objects in the collection.

Examples

- MATLAB:

```
count = data.Count
```

- Python:

```
count = data.Count
```

Keys

Description

Gets the set of keys for all objects in the collection.

Keys can be useful to iterate all objects in the collection.

Examples

- MATLAB:

```
keys = data.Keys;
for key_idx = 1:length(keys)
      key = keys{key_idx};
      fprintf('%s\n', key);
end
```

- Python:

```
keys = data.Keys
for key in keys:
      print(key)
```

Methods

Add()

Description

Adds a new object to the collection.

Prototype

```
SimulationData Add(string key, object value)
```

Parameters

Parameter	Description
key	Key for the new object.
value	Value of the new object.

Returns

`SimulationData` object which represents the added named value.

Exceptions

Message Text	Possible Problem
An item with the same key has already been added.	Object with this name already exists in the collection.
Value of unsupported type specified.	Attempt to add data of unsupported type.

Examples

- MATLAB:

```
data = driver.InputRuntimeData.Add('A1', 0);
for i = 1:5
      % Call helper function to wait till runtime data requested
      WaitDataRequestedEvent;
      % Change value
      data.Value = i;
      % Send runtime data to the simulation engine
      driver.SendRuntimeData;
end
```

- Python:

```
data = driver.InputRuntimeData.Add("A1", 0)
for i in range(5):
      # Call helper function to wait till runtime data requested
      WaitDataRequestedEvent()
      # Change value
      data.Value = i
      # Send runtime data to the simulation engine
      driver.SendRuntimeData()
```

Clear()

Description

Removes all objects from the collection.

Prototype

```
Clear()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
for i = 1:5
        % Call helper function to wait till runtime data requested
        WaitDataRequestedEvent;
        % Prepare collection for adding new data
        driver.InputRuntimeData.Clear();
        % Add data
        driver.InputRuntimeData.Add('A1', i);
        % Send runtime data to the simulation engine
        driver.SendRuntimeData;
end
```

- Python:

```
for i in range(5):
        # Call helper function to wait till runtime data requested
        WaitDataRequestedEvent
        # Prepare collection for adding new data
        driver.InputRuntimeData.Clear()
        # Add data
        driver.InputRuntimeData.Add("A1", i)
        # Send runtime data to the simulation engine
        driver.SendRuntimeData()
```

ContainsKey()

Description

Checks whether an object with the specified key already exists in the collection.

Prototype

```
bool ContainsKey(string key)
```

Parameters

Parameter	Description
key	Key to check for.

Returns

`True` if object exists in the collection, `false` otherwise.

Exceptions

None.

Examples

- MATLAB:

```
% Check if object already added
if driver.InputRuntimeData.ContainsKey('A1')
      % Set data value
      driver.InputRuntimeData.Item('A1').Value = i;
else
      % Add data
      driver.InputRuntimeData.Add('A1', i);
end
% Send runtime data to the simulation engine
driver.SendRuntimeData;
```

- Python:

```
# Check if object already added
if driver.InputRuntimeData.ContainsKey("A1"):
      # Set data value
      driver.InputRuntimeData("A1").Value = i
else:
      # Add data
      driver.InputRuntimeData.Add("A1", i)
# Send runtime data to the simulation engine
driver.SendRuntimeData()
```

Item()

Description

Returns an item from the collection by its key.

This is the default method of `SimulationDataCollection` and in some languages, its name can be omitted (for example in Python).

Prototype

```
SimulationData Item(string key)
```

Parameters

Parameter	Description
key	Unique key of data object.

Returns

`SimulationData` object with the specified name if it exists in the collection or throws an exception otherwise.

Exceptions

Message Text	Possible Problem
The given key was not present in the dictionary.	Raised if there is no runtime data with this name in the collection.

Examples

- MATLAB:

```
value = driver.InputRuntimeData.Item('A1');
```

- Python:

```
value = driver.InputRuntimeData.Item("A1")
# Contracted form is allowed
value = driver.InputRuntimeData("A1")
```

Remove()

Description

Removes an object with the specified key from the collection.

Prototype

```
Remove(string key)
```

Parameters

Parameter	Description
key	Key for object to remove.

Returns

None.

Exceptions

Message Text	Possible Problem
The given key was not present in the dictionary.	No object with specified key in the collection.

Examples

- MATLAB:

```
driver.InputRuntimeData.Remove('A1');
```

- Python:

```
driver.InputRuntimeData.Remove("A1")
```

SimulationEngineDriver

Description

The `SimulationEngineDriver` object is an entry point for the Simulation Engine Driver component. It provides a means to set up the simulation, access collections of parameters, runtime and output data, run the simulation, send and receive numerical data, receive simulation results, handle events and send commands to the simulation engine.

Access

Create a new instance to begin work with the `SimulationEngineDriver`. The object is addressed by its COM ProgID (a string which uniquely identifies the object in the system):

- MATLAB:
```
progId = 'VPI.TC.SED.SimulationEngineDriver';
driver = actxserver(progId);
```
- Python:
```
import win32com.client
progId = "VPI.TC.SED.SimulationEngineDriver"
driver = win32com.client.Dispatch(progId)
```

Properties

InputRuntimeData

Description

Gets the collection of `SimulationData` objects that represents runtime data, which can be sent to the simulation engine via `SendRuntimeData()` and received by the simulation engine via `SimDataFetch` commands.

For more information on runtime data, see "SimulationData Object" on page 325.

Examples

- MATLAB:
```
data = driver.InputRuntimeData;
```
- Python:
```
data = driver.InputRuntimeData
```

OutputData

Description

Gets the collection of `SimulationData` objects which represents the results of simulation sent from the simulation engine via the `SimOutputReturn` or `SimOutputReturnEx` command.

For more information on outputs, see "SimulationData Object" on page 325.

Examples

- MATLAB:

```
output = driver.OutputData;
```

- Python:

```
output = driver.OutputData
```

OutputRuntimeData

Description

Gets the collection of `SimulationData` objects which represents the runtime data sent from the simulation engine via `SimDataReturn`.

For more information on runtime data, see "SimulationData Object" on page 325.

Examples

- MATLAB:

```
data = driver.OutputRuntimeData;
```

- Python:

```
data = driver.OutputRuntimeData
```

Parameters

Description

Gets the collection of `SimulationParameter` objects which represents the default values for global parameters and parameters of module instances at the schematic level.

Examples

- MATLAB:

```
parameters = driver.Parameters;
```

- Python:

```
parameters = driver.Parameters
```

ProjectPath

Description

Gets the path to the root folder of the currently open project. It's the same path that can be used to open a project via the `OpenProject()` method. Returns an empty string if there is no open project.

Useful for testing is there was an open project prior to calling `CloseProject()`.

Examples

- MATLAB:

  ```
  path = driver.ProjectPath
  ```

- Python:

  ```
  path = driver.ProjectPath
  ```

SchematicInputsDir and SchematicOutputsDir

Description

Point to the `Inputs` and `Outputs` folder of the schematic.

Once a package or schematic (e.g., `*.sed`, `*.vmi`, or `*.vtmu`) is opened using the Simulation Engine Driver, the properties will contain paths to the current `Inputs` and `Outputs` folders.

If a package or schematic is unpacked in `Temp`, the properties will contain paths to temporary locations.

Both the `SchematicInputsDir` and `SchematicOutputsDir` properties are read-only and contain values regardless of the presence of files in the respective folders.

Examples

- MATLAB:

```
% Write data to file in schematic Inputs folder
file_name = fullfile(driver.SchematicInputsDir, 'data.dat');
fid = fopen(file_name, 'w');
% write to file here
fclose(fid);
```

- Python:

```
# Write data to file in schematic Inputs folder
file_name = os.path.join(driver.SchematicInputsDir, "data.dat")
with(open(file_name, "w") as file)
      # Write to file here
```

- MATLAB:

```
% Read data from file in schematic Outputs folder
file_name = fullfile(driver.SchematicOutputsDir, 'results.dat');
fid = fopen(file_name, 'r');
% read from file here
fclose(fid);
```

- Python:

```
# Read data from file in schematic Outputs folder
file_name = os.path.join(driver.SchematicOutputsDir, "results.dat")
with(open(file_name, "r") as file)
      # Read from file here
```

SimulationEngineState

Description

Gets the current state of the simulation engine. It can be one of the states described in "SimulationEngineState" on page 445.

Examples

- MATLAB:

```
% Check if script is still running
if ~strcmp(driver.SimulationEngineState, 'SimulationEngineState_Idle')
      fprintf('Simulation is running\n');
end
```

- Python:

```
from win32com.client import constants
# Check if script is still running
if not driver.SimulationEngineState == constants.SimulationEngineState_Idle:
      print("Simulation is running")
```

SimulationName

Description

Gets or sets the name which will be displayed in Analyzer Manager pane of the VPIphotonicsAnalyzer window.

By default, this property contains the SimulationEngineDriver object creation time.

Examples

- MATLAB:

```
driver.SimulationName = 'Parameter X sweep'
```

- Python:

```
driver.SimulationName = "Parameter X sweep"
```

SimulationScript

Description

Gets or sets the simulation script which should be executed via the `Run` or `BeginRun` methods.

Only one of the properties `SimulationScript` or `SimulationScriptPath` should be specified. Whichever is specified last takes precedence.

Examples

- MATLAB:

```
% Read contents of ptds.pt into SimulationScript property
driver.SimulationScript = fileread('ptds.pt');
```

- Python:

```
# Read contents of ptds.pt into SimulationScript property
f = open("ptds.pt")
driver.SimulationScript = f.read()
```

SimulationScriptPath

Description

Gets or sets the path to the simulation script which should be executed via the `Run` or `BeginRun` methods. Paths may be absolute or relative. For relative paths, the directory specified via the `WorkingDir` property is treated as the base directory.

Only one of the properties `SimulationScript` or `SimulationScriptPath` should be specified. Whichever is specified last takes precedence.

Examples

- MATLAB:

```
% Use ptds.pt from working directory
driver.SimulationScriptPath = 'ptds.pt';
```

- Python:

```
# Use ptds.pt from working directory
driver.SimulationScriptPath = "ptds.pt"
```

IsEventQueueEnabled

Description

Gets or sets the flag indicating that the "pull" event handling mechanism shall be used.

Event handling mechanisms are described in detail under "Events" on page 327.

Examples

- MATLAB:

```
% Switch pull event handling mechanism on
driver.IsEventQueueEnabled = true;
```

- Python:

```
# Switch pull event handling mechanism on
driver.IsEventQueueEnabled = True
```

IsCustomScriptEnabled

Description

Gets or sets the flag indicating whether a custom simulation script (`simulation_script.tcl`) will be used. If set to `false`, then the default simulation run sequence is used (e.g., `"run 1"`).

This property is available for design packages (`.vtmu`) and VPI Module Interchange files (`.vmi`) and is not available for specialized Simulation Engine Driver packages (`.sed`), raw simulation packages, and for cases when the main simulation script is specified using the `SimulationScript` or `SimulationScriptPath` properties.

Examples

- MATLAB:

```
% Enable custom simulation script
driver.IsCustomScriptEnabled = true;
```

- Python:

```
# Enable custom simulation script
driver.IsCustomScriptEnabled = True
```

VpaPath

Description

Path to the VPIphotonicsAnalyzer document (`.vpa`) file which should be associated with consequent simulation executions.

Only full path is supported.

Examples

- MATLAB:

```
% Use *.vpa file from current working directory
driver.VpaPath = strcat(pwd, '\Test.vpa');
```

- Python:

```
import sys
from os.path import *
# Use *.vpa file from directory where script resides
driver.VpaPath = join(sys.path[0], "Test.vpa")
```

WorkingDir

Description

Gets or sets the working directory for the simulation engine. It is also used as the base directory for some other operations (such as simulation script paths).

By default, it is set to the working directory of the process hosting the Simulation Engine Driver component.

This directory can be referenced in simulation scripts using the *$WORK* variable.

Examples

- MATLAB:

```
% Set MATLAB working directory as
% working directory for the simulation engine
driver.WorkingDir = pwd;
```

- Python:

```
import sys
# Set Python script directory as
# working directory for the simulation engine
driver.WorkingDir = sys.path[0]
```

- Simulation Script:

```
# Use file from working directory as input
setstate "ReadFromFile_vtmg1" InputFilename {$WORK/signal.dat}
```

Methods

Abort()

Description

Stops the simulation respectfully, allowing the simulation engine to perform any necessary cleanup and finalize pending communications. With this method, the simulation engine process associated with the running simulation is only force quit if it is not possible to stop the simulation within a specified time:

This method may be needed if a simulation hangs and there are no other possibilities to stop it (for instance by sending the `Stop` command).

Prototype

```
Abort(int timeout = -1)
```

Parameters

Parameter	Description
`timeout`	Amount of time in milliseconds which a function shall wait for the simulation to finish. When this time is elapsed, the simulation engine is forcibly terminated. If -1 is specified, the function waits indefinitely.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
% Run simulation
driver.BeginRun;
% Check if script is still running
if ~strcmp(driver.SimulationEngineState, 'SimulationEngineState_Idle')
      driver.Abort(1000);
end
```

- Python:

```
# Run simulation
driver.BeginRun()
# Check if script is still running
if not driver.SimulationEngineState == constants.SimulationEngineState_Idle:
      driver.Abort(1000)
```

BeginRun()

Description

Runs the simulation engine process, transfers the simulation script to it and begins execution asynchronously, in the background. This is a nonblocking method that exits immediately, allowing client code to perform other tasks while the simulation is running.

Because of its asynchronous nature, errors are not reported immediately. Use the `EndRun()` method somewhere in the code to wait for the end of script execution and to access error information.

Prototype

```
BeginRun()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
% Start script execution in background
driver.BeginRun;
% Wait until script execution is finished
while ~strcmp(driver.SimulationEngineState, 'SimulationEngineState_Idle')
      pause(3);
end
```

- Python:

```
import time
# Start script execution in background
driver.BeginRun()
# Wait until script execution is finished
while driver.SimulationEngineState != constants.SimulationEngineState_Idle:
      time.sleep(3)
```

CheckinLicense()

Description

Releases licenses previously acquired via `CheckoutLicense()`.

This function should be called when a batch simulation is over. It is also called implicitly when the simulation script is changed or the `SimulationEngineDriver` object is destroyed.

Prototype

```
CheckinLicense()
```

Parameters

None.

Returns

None.

Exceptions

Message Text	Possible Problem
No licenses are checked out	CheckoutLicense() method wasn't called.

Examples

See examples of the `CheckoutLicense()` method.

CheckoutLicense()

Description

Acquires licenses necessary to perform simulation.

This method can be useful in scenarios in which simulation is performed multiple times and the license server is running on a remote machine.

It requires that a simulation script is already specified, because the set of licenses is determined from this script.

Prototype

```
CheckoutLicense()
```

Parameters

None.

Returns

None.

Exceptions

Message Text	Possible Problem
Simulation script isn't set	Simulation script wasn't specified.

Message Text	Possible Problem
License server system does not support this feature.	There is no license available to run specified simulation script.

Examples

- MATLAB:

```
% Acquire licenses only once
driver.CheckoutLicense

% Run simulation many times
for i = 1:10
      driver.Run
end

% Release licenses at the end
driver.CheckinLicense
```

- Python:

```
# Acquire licenses only once
driver.CheckoutLicense()

# Run simulation many times
for i in range(10):
      driver.Run()

# Release licenses at the end
driver.CheckinLicense()
```

ClosePackage()

Description

Cleans up intermediate files created by *OpenPackage()* and restores properties set by it to their default state.

`ClosePackage()` is also automatically called when a simulation script is set via `SimulationScript` or the `SimulationScriptPath` or `SimulationEngineDriver` object is destroyed.

Prototype

```
ClosePackage()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

See examples for OpenPackage().

CloseProject()

Description

Closes a currently open project.

If there is no project open, an exception will be raised. An open project may be checked using the `ProjectPath` property.

Prototype

```
CloseProject()
```

Parameters

None.

Returns

None.

Exceptions

Message Text	Possible Problem
There is no open project.	You are trying to close the project when there is no open project.

Examples

- MATLAB:

```
if ~isempty(driver.ProjectPath)
        driver.CloseProject
end
```

- Python:

```
if driver.ProjectPath:
        driver.CloseProject()
```

CopyOutputRuntimeDataToInput()

Description

Copies output runtime data to the input runtime data. This can be useful in some scenarios where runtime data can be modified both by a simulation script and by a client (for instance, in sweep-like simulations) and it is necessary to resend data received after execution of one simulation run back to the simulation engine so that it can be used on the next run.

Prototype

```
CopyOutputRuntimeDataToInput()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
driver.CopyOutputRuntimeDataToInput;
driver.SendRuntimeData;
```

- Python:

```
driver.CopyOutputRuntimeDataToInput()
driver.SendRuntimeData()
```

EndRun()

Description

Waits until script execution is finished and reports errors if any.

This method is called to access error information from the simulation that was previously run via the `BeginRun()` method.

Prototype

```
EndRun(int timeout = -1)
```

Parameters

Parameter	Description
timeout	Amount of time in milliseconds which a function shall wait for the simulation to finish. When this time is elapsed, the function generates an exception. If -1 is specified, the function waits indefinitely.

Returns

None.

Exceptions

Message Text	Possible Problem
A timeout was reached while waiting for the end of simulation.	The specified timeout has expired.

All other exceptions which were thrown during the background execution of a simulation script are also re-thrown.

Examples

- MATLAB:

```
% Run in background
driver.BeginRun;
try
      % Wait until finished and catch exceptions
      driver.EndRun(-1);
catch ex
      fprintf('Exception = %s', ex.message);
end
```

- Python:

```
# Wait until finished and catch exceptions
driver.BeginRun()
try:
      # Catch exceptions
      driver.EndRun()
except Exception as ex:
      print(ex)
```

GetNextEvent()

Description

When the “pull” event handling mechanism is switched on via the `IsEventQueueEnabled` property, this method returns the next event object from the internal event queue. The returned object may have a different set of properties depending on the type of event.

When simulation is finished and the queue is empty, a null object is returned (the actual syntax depends on the target language, for example None in Python).

Event handling mechanisms are described in detail under "Events" on page 327.

Prototype

```
object GetNextEvent(int timeout = -1)
```

Parameters

Parameter	Description
timeout	Amount of time in milliseconds which a function shall wait for the next event. If time is elapsed function generates exception. If -1 is specified, the function waits infinitely.

Returns

Object which contains additional information for the event or null object if simulation is finished and queue is empty.

Exceptions

Message Text	Possible Problem
Event queue is disabled. Set IsEventQueueEnabled to True to enable it.	Pull event handling mechanism is switched off.
A timeout was reached while waiting for the next event.	Timeout is reached.

Examples

- MATLAB:

```
STATE_CHANGED_EVENT = 0;
IDLE_STATE = 0;
while true
      % Get next event from the queue
      event = driver.GetNextEvent(-1);
      % Check if there are any events in queue
      if isempty(event)
            break;
      end
      % Check if it is SimulationEngineStateChanged event
      if event.get('Type') == STATE_CHANGED_EVENT
            % Check if it is final event
            if event.get('CurState') == IDLE_STATE
                  break;
            end
      end
end
```

- Python:

```
from win32com.client import constants
while True:
      # Get next event from the queue
      event = driver.GetNextEvent()
      # Check if there are any events in queue
      if event is None:
            break
      # Check if it is SimulationEngineStateChanged
      trgtEvent =
constants.SimulationEventType_SimulationEngineStateChanged
      if event.Type == trgtEvent:
            # Check if it is final event
            if event.CurState ==
constants.SimulationEngineState_Idle:
                  break
```

Kill()

Description

Immediately terminates the simulation engine process associated with the currently running simulation.

This method may be needed if a simulation hangs and there are no other possibilities to stop it (for instance by sending the `Stop` command).

Prototype

```
Kill()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
% Run simulation
driver.BeginRun;
% Check if script is still running
if ~strcmp(driver.SimulationEngineState, 'SimulationEngineState_Idle')
      driver.Kill;
end
```

- Python:

```
# Run simulation
driver.BeginRun()
# Check if script is still running
if not driver.SimulationEngineState == constants.SimulationEngineState_Idle:
      driver.Kill()
```

OpenPackage()

Description

Opens a design package (.vtmu), VPI Module Interchange file (.vmi), specialized Simulation Engine Driver package (.sed) or raw simulation package.

Writable design packages and raw simulation packages are opened in place. Read-only design packages are copied to a temporary location and VPI Module Interchange files and Simulation Engine Driver packages are unpacked to the temporary location before opening. If a design package is located inside the project folder structure, this project will be automatically opened.

Depending on the type of package, different properties of the `SimulationEngineDriver` object are set after opening the package.

The `PackageDir` property is set to point to the directory where the opened package is located. It will be set to the parent directory for design packages and raw packages or the temporary folder to which the VPI Module Interchange file and specialized package was unpacked.

The `SchematicInputsDir` and `SchematicOutputsDir` properties are always set for design packages and VPI Module Interchange files to point to the `Inputs` and `Outputs` folders of the package. For specialized packages, those properties are set to point to folders which correspond to the `Inputs` and `Outputs` folders on the schematic level.

For all package types, the `VpaPath` property is set to point to the VPIphotonicsAnalyzer file (.vpa) that is present in the package or to the location where it should be placed by default after simulation if it doesn't already exist.

Prototype

```
OpenPackage(string path)
```

Parameters

Parameter	Description
path	Path to schematic, VPI Module Interchange file, specialized package file or raw simulation package folder.

Returns

None.

Exceptions

Message Text	Possible Problem
Cannot open package (<system message>).	File or directory doesn't exist, is inaccessible or package is in wrong format, etc.
Unsupported file format.	The specified file is either not in Simulation Engine Driver format, or the file version is newer than that which is supported by the current version of the Simulation Engine Driver.

Examples

- MATLAB:

```
% Open demonstration
driver.OpenPackage('c:\Program
Files\VPI\VPIdesignSuite 11.5\tooldata\demos\OS\Simulation
Techniques\General\Resampling Options.vtmu');

% Run it
driver.Run

% Close package to cleanup files
driver.ClosePackage
```

- Python:

```
# Open demonstration
driver.OpenPackage(r"c:\Program
Files\VPI\VPIdesignSuite 11.5\tooldata\demos\OS\Simulation
Techniques\General\Resampling Options.vtmu")

# Run it
driver.Run

# Close package to cleanup files
driver.ClosePackage()
```

OpenProject()

Description

Opens a project at a specified location. Already open project will be closed prior to opening a new one.

Prototype

```
OpenProject(string path)
```

Parameters

Parameter	Description
path	Path to project folder.

Returns

None.

Exceptions

Message Text	Possible Problem
Cannot open project. It was created in a newer version.	You are trying to open project of a newer, unsupported, version.

Examples

- MATLAB:

```
% Open project
driver.OpenProject('c:\MyPorjects\DemoProject');


% Open schematic
driver.OpenPackage('c:\MyPorjects\DemoProject\demo1.vtmu');

% Run it
driver.Run

% Close package to cleanup files
driver.ClosePackage

% Close project
driver.CloseProject
```

- Python:

```
# Open project
driver.OpenProject(r"c:\MyPorjects\DemoProject")


# Open schematic
driver.OpenPackage(r"c:\MyPorjects\DemoProject\demo1.vtmu")

# Run it
driver.Run()

# Close package to cleanup files
driver.ClosePackage()

# Close project
driver.CloseProject()
```

Pause()

Description

Sends `Pause` command to the simulation engine. This command can be used to control simulation script execution. To react, a simulation script shall explicitly call the `SimGetCommand` and handle the returned value.

For more information on commands, see "Commands" on page 329.

Prototype

```
Pause()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:
  ```
  % Run simulation
  driver.BeginRun;
  % Command script to pause execution
  driver.Pause;
  ```
- Python:
  ```
  # Run simulation
  driver.BeginRun()
  # Command script to pause execution
  driver.Pause()
  ```
- Simulation script:
  ```
  switch [SimGetCommand] {
  	Exit { break }
  	Pause {
  		SimPaused
  		switch [SimGetCommand] {
  			Exit { break }
  			Resume { SimResumed }
  		}
  	}
  }
  ```

Run()

Description

Creates a simulation package if necessary, runs a simulation engine process, transfers a simulation script to it and begins its execution. This method waits until the script is finished before exiting.

The simulation package is created on the fly for design packages, VPI Module Interchange files or Simulation Engine Driver packages. Raw simulation packages are used as is. The `WorkingDir` property is set before simulation to point to the root of the simulation package and restored after the simulation finishes.

For more information on script execution, see “Running the Simulation” on page 314.

Prototype

```
Run()
```

Parameters

None.

Returns

None.

Exceptions

Message Text	Possible Problem
Simulation script isn't set	Simulation script wasn't specified.
License server system does not support this feature.	There is no license available to run the specified simulation script.
License server doesn't respond. Simulation is aborted.	Connection to the license server was lost and it is not possible to check whether license is still valid.
Connection to the simulation engine lost.	The simulation engine process terminated unexpectedly.

All other exceptions which were thrown during the background execution of a simulation script are also re-thrown.

Examples

- MATLAB:

```
% Run simulation
driver.Run;
fprintf('Script execution is finished');
```

- Python:

```
# Run simulation
driver.Run()
print("Script execution is finished")
```

Resume()

Description

Sends the `Resume` command to the simulation engine. This command can be used to control simulation script execution. To react, a simulation script shall explicitly call the `SimGetCommand` command and handle the return value.

For more information on commands, see “Commands” on page 329.

Prototype

```
Resume()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
% Run simulation
driver.BeginRun;
% Command to pause execution
driver.Pause;
% Command to resume execution
driver.Resume;
```

- Python:

```
# Run simulation
driver.BeginRun()
# Command to pause execution
driver.Pause()
# Command to resume execution
driver.Resume()
```

SendRuntimeData()

Description

Prepares runtime data for sending to the simulation engine. The actual data transfer is initiated when the simulation script calls the `SimDataFetch` command. While the recommended strategy is to send runtime data when the simulation engine is in the `RuntimeDataPending` state, this is not strictly necessary, because the asynchronous behavior of this method allows sending data without synchronizing client actions with calls to `SimDataFetch` in the simulation script.

Prototype

```
SendRuntimeData()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
% Set data value
driver.InputRuntimeData.Add('A1', 10.5);
% Send data
driver.SendRuntimeData;
```

- Python:

```
# Set data value
driver.InputRuntimeData.Add("A1", 10.5)
# Send data
driver.SendRuntimeData()
```

Stop()

Description

Sends the `Exit` command to the simulation engine. This command can be used to control simulation script execution. To react, a simulation script shall explicitly call the `SimGetCommand` command and handle the return value.

For more information on commands, see “Commands” on page 329.

Prototype

```
Stop()
```

Parameters

None.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
% Run simulation
driver.BeginRun;
% Command to stop execution
driver.Stop;
```

- Python:

```
# Run simulation
driver.BeginRun()
# Command to stop execution
driver.Stop()
```

Events

Events are fired when something important happens in the Simulation Engine Driver (for example when a simulation is started or finished). Some events are fired by commands in simulation script, like `SimDataReturn` (for details on these commands, see "Special Scripting Commands for Interactive Simulation" on page 277).

> **Note:** The events described in this section are based on standard COM events architecture and may not be supported by all clients. For that reason, examples are given for Microsoft Excel only. An alternative mechanism, based on the 'pull' approach (or event queue), is described in "SimulationEvent" on page 432 and "SimulationEventType" on page 442.

CommandRequested

Description

Fired when the simulation script calls the `SimGetCommand` and is waiting for client response.

This event is a simplified analog of `SimulationEngineStateChanged` event with *curState* equal to `SimulationEngineState.CommandPending`.

Parameters

None.

Examples

- Simulation script:

```
switch [SimGetCommand]
{
	Exit { break }
	Resume { SimResumed }
}
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_CommandRequested()
	' Command script to exit
	driver.Stop
End Sub
```

RuntimeDataReceived

Description

Fired when runtime data is received by Simulation Engine Driver after it was sent from the simulation engine via the `SimDataReturn` command.

Parameters

None.

Examples

- Simulation script:

  ```
  # Initialize data and send to client
  set _values(Value1) 25.18
  SimDataReturn _values
  ```

- Excel:

  ```
  ' Declare driver object to receive events
  Dim WithEvents driver As SimulationEngineDriver

  Private Sub driver_RuntimeDataReceived()
        ' Store value in named cell
        [Result] = driver.OutputRuntimeData("Value1")
  End Sub
  ```

RuntimeDataRequested

Description

Fired when the simulation engine is waiting for runtime data. This happens when a script calls `SimDataFetch`.

This event is a simplified analog of `SimulationEngineStateChanged` event with *curState* equal to `SimulationEngineState.RuntimeDataPending`.

Parameters

None.

Examples

- Simulation script:

  ```
  # Receive data from client and use it
  SimDataFetch _values
  setstate "Const_vtms1" "level" _values(Value1)
  ```

- Excel:

  ```
  ' Declare driver object to receive events
  Dim WithEvents driver As SimulationEngineDriver

  Private Sub driver_RuntimeDataRequested()
        ' Set runtime data and send it to simulation engine
        driver.InputRuntimeData.Clear
        Set param = driver.InputRuntimeData.Add("Value1", CDbl([Result]))
        driver.SendRuntimeData
  End Sub
  ```

SimulationEngineStateChanged

Description

Fired when the state of the simulation engine changes. Transitions include the beginning or end of script execution, waiting for commands or runtime data. Both the previous and current states are sent as parameters to this event.

This event is fired for all state transitions and can be used in scenarios when generic event handling routines need to be written. Specific events are additionally triggered for some particular state changes (`ScriptStarted`, `ScriptFinished`) and the use of these is recommended in most cases.

Prototype

```
SimulationEngineStateChanged(SimulationEngineState prevState, SimulationEngineState
  curState)
```

Parameters

Parameter	Description
prevState	State which was active before the engine switched to the current state.
curState	Current state of the simulation engine.

Examples

- Simulation script:

```
switch [SimGetCommand] {
       Exit { break }
       Pause {
              SimPaused
              switch [SimGetCommand] {
                     Exit { break }
                     Resume { SimResumed }
              }
       }
}
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_SimulationEngineStateChanged()
       ' Check if new state indicates waiting for command
       trgtState = SimulationEngineState.SimulationEngineState_CommandPending
       If driver.SimulationEngineState = trgtState Then
              ' Command script to exit
              driver.Stop
       End If
End Sub
```

MessageReceived

Description

Fired when a new message arrives from the simulation engine. This message may be sent by a module (e.g., warning messages) or by a simulation script via the `ptclmessage` command.

Prototype

```
MessageReceived(string message)
```

Parameters

Parameter	Description
message	Message which was received from the simulation engine.

Examples

- Simulation script:

```
# Send warning
ptclmessage "Value is out of range"
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_MessageReceived(ByVal message As String)
      Debug.Print message
End Sub
```

MessageReceivedEx

Description

Fired when a new message arrives from the simulation engine. It is similar to the `MessageReceived` event but contains a type of the received message (informational or warning).

Prototype

```
MessageReceived(MessageType messageType, string message)
```

Parameters

Parameter	Description
messageType	Type of the message (see "MessageType" on page 440).
message	Message which was received from the simulation engine.

Examples

- Simulation script:

```
# Send warning
ptclmessage "Value is out of range" Warning
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_MessageReceivedEx(ByVal messageType As MessageType, ByVal message
As String)
      If messageType = MessageType_Info Then
            Debug.Print "Informational message"
      ElseIf messageType = MessageType_Warning Then
            Debug.Print "Warning message"
      End If
End Sub
```

OutputDataReceived

Description

Fired when output data is received by the Simulation Engine Driver. Output data may be sent from the simulation engine via the `SimOutputReturn` (for scalars) or `SimOutputReturnEx` (for matrices) command.

Parameters

None.

Examples

- Simulation script for passing scalar values:

```
# Initialize output and return it to client
set _outputs(Output1) [statevalue PostValue_vtms1 InputValue]
SimOutputReturn _outputs
```

- Simulation script for passing matrix values:

```
# Read matrix data from PostValueMxFlt module and send output value in one line
SimOutputReturnEx [list [list PostValueMxFlt_vtms1]]
# Or for this particular case a shorter syntaxis is also available:
# SimOutputReturnEx PostValueMxFlt_vtms1
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_OutputDataReceived(ByVal instanceID As String)
      [Result] = driver.OutputData("Output1")
End Sub
```

ScriptFinished

Description

Fired when script execution is finished and the simulation engine is terminated.

This event is a simplified analog of the `SimulationEngineStateChanged` event with *`prevState`* equal to `SimulationEngineState.Running` and *`curState`* equal to `SimulationEngineState.Idle`.

Parameters

None.

Examples

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_ScriptFinished()
      Debug.Print "Script execution finished"
End Sub
```

ScriptStarted

Description

Fired when the simulation engine process is started and script execution begins.

This event is a simplified analog of `SimulationEngineStateChanged` event with *`prevState`* equal to `SimulationEngineState.Idle` and *`curState`* equal to `SimulationEngineState.Running`.

Parameters

None.

Examples

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_ScriptStarted()
      Debug.Print "Script execution started"
End Sub
```

SimulationFinished

Description

Fired when a simulation script calls the `SimDone` command. May be used to indicate that the execution of the `run` command has finished (the `run` command performs the actual simulation of the schematic).

Parameters

None.

Examples

- Simulation script:

```
SimRunning
run 1
SimDone
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngineDriver

Private Sub driver_SimulationFinished()
    Debug.Print "Simulation Finished"
End Sub
```

SimulationStarted

Description

Fired when a simulation script calls the `SimRunning` command. May be used to indicate that the execution of the `run` command has started (the `run` command performs the actual simulation of the schematic).

Parameters

None.

Examples

- Simulation script:

```
SimRunning
run 1
SimDone
```

- Excel:

```
' Declare driver object to receive events
Dim WithEvents driver As SimulationEngine

Private Sub driver_SimulationStarted()
    Debug.Print "Simulation Started"
End Sub
```

SimulationEvent

Description

The `SimulationEvent` object stores event data. It differs from other objects because it is dynamic in nature. The set of properties that it provides depends on the type of event it is returned for. Also, due to its dynamic nature, there are some peculiarities in accessing properties in some languages (for instance in MATLAB).

Only one property is provided for all events: event type. It can be used to distinguish what additional information is available.

Access

`SimulationEvent` object is returned by `GetNextEvent()` method of `SimulationEngineDriver` object.

- MATLAB:

  ```
  event = driver.GetNextEvent(-1);
  ```

- Python:

  ```
  event = driver.GetNextEvent()
  ```

Properties (all events)

Type

Description

Gets type of the event described by `SimulationEventType` enumeration (see "SimulationEventType" on page 442). Type can be used to distinguish which event happened and whether it has any additional parameters.

Properties (SimulationEngineStateChanged event)

CurState

Description

Gets the state of the simulation engine after the transition.

Examples

- MATLAB:

  ```
  STATE_CHANGED_EVENT = 0;
  IDLE_STATE = 0;
  RUNNING_STATE = 1;
  % Get next event from the queue
  event = driver.GetNextEvent(-1);
  % Exit loop when simulation is finished
  if ~isempty(event) && event.get('Type') == STATE_CHANGED_EVENT
        prevState = event.get('PrevState');
        curState = event.get('CurState');
        if prevState == RUNNING_STATE && curState == IDLE_STATE
              fprintf('Simulation Finished');
        end
  end
  ```

- Python:

  ```
  # Get next event from the queue
  event = driver.GetNextEvent()
  # Exit loop when simulation is finished
  if not event is None and event.Type ==
  constants.SimulationEventType_SimulationEngineStateChanged:
        if event.CurState == constants.SimulationEngineState_Idle and event.PrevState
  == constants.SimulationEngineState_Running:
              print("Simulation Finished")
  ```

PrevState

Description

Gets the state of the simulation engine before the transition.

Properties (MessageReceived event)

Message

Description

Gets a message received from the simulation engine.

MessageType

Description

Gets the type of the message received from the simulation engine (see “MessageType” on page 440).

Examples

- MATLAB:

```
MESSAGE_RECEIVED_EVENT = 3;
MESSAGE_INFO_TYPE = 1;
MESSAGE_WARNING_TYPE = 2;
% Get next event from the queue
event = driver.GetNextEvent(-1);
% Check if it is MessageReceived event
if event.get('Type') == MESSAGE_RECEIVED_EVENT
      msg = event.get('Message');
      msgType = event.get('MessageType');
      if msgType == MESSAGE_INFO_TYPE
            fprintf('Informational message: %s\n', msg);
      elseif msgType == MESSAGE_WARNING_TYPE
            fprintf('Warning message: %s\n', msg);
      else
            fprintf('Message: %s\n', msg);
      end
end
```

- Python:

```
# Get next event from the queue
event = driver.GetNextEvent()
# Check if it is MessageReceived event
if event.Type == constants.SimulationEventType_MessageReceived:
      messageType = event.MessageType
      if messageType == win32com.client.constants.MessageType_Info:
            print("Informational message:")
      elif messageType == win32com.client.constants.MessageType_Warning:
            print("Warning message:")
      print("%s" % event.Message)
```

SimulationParameter

Description

This object represents initial values for schematic parameters and parameters of module instances at the schematic level. It can read and alter values of parameters, but any changes will only affect the simulation if they are performed before the script execution is started.

Access

Can be acquired from the collection of parameters which is accessible through `SimulationEngineDriver`.

- MATLAB:

```
% Get level parameter of instance Const_vtms1
parameter = driver.Parameters.Item('Const_vtms1', 'level')
```

- Python:

```
# Get level parameter of instance Const_vtms1
parameter = driver.Parameters("Const_vtms1", "level")
```

Properties

ExprType

Description

Gets expression type specified for the parameter. It is an enumeration of type `ParameterExpressionType` (see “ParameterExpressionType” on page 441).

InstanceId

Description

Gets the ID of the instance to which the parameter belongs (or an empty string for global parameters).

Examples

- MATLAB:

```
fprintf('%s.%s = %g (%s)\n', param.InstanceId, param.Name, param.Value,
  param.ExprType);
```

- Python:

```
print("%s.%s = %g (%s)" % (param.InstanceId, param.Name, param.Value,
  param.ExprType))
```

Name

Description

Gets the name of the parameter.

Value

Description

Gets the value of the parameter.

SimulationParameterCollection

Description

Collection of `SimulationParameter` objects representing the parameters acquired while parsing the simulation script with the `SimulationEngineDriver` object.

Access

Can be acquired through `SimulationEngineDriver` property `Parameters`.

- MATLAB:
  ```
  parameters = driver.Parameters;
  ```
- Python:
  ```
  parameters = driver.Parameters
  ```

Properties

Count

Description

Gets number of objects in the collection.

Examples

- MATLAB:
  ```
  count = parameters.Count
  ```
- Python:
  ```
  count = parameters.Count
  ```

InstanceIds

Description

Gets a set of instance IDs for all parameters in the collection.

This set can be useful to iterate all objects in the collection.

Examples

- MATLAB:

```
ids = data.InstanceIds;
for id_idx = 1:length(ids)
      id = ids{id_idx};
      fprintf('%s\n', id);
end
```

- Python:

```
ids = data.InstanceIds
for id in ids:
      print(id)
```

Methods

Contains()

Description

Checks whether a global parameter or parameter of the specified instance exists.

Prototype

```
bool Contains(string instanceId, string name)
```

Parameters

Parameter	Description
`instanceId`	ID of instance. Should be empty string for global parameters.
name	Name of the parameter to check.

Returns

`True` if object exists in the collection, `False` otherwise.

Exceptions

None.

Examples

- MATLAB:

```
% Check if specific parameter exists
if driver.Parameters.Contains('Const_vtms1', 'level')
      % Set parameter value
      driver.Parameters.Item('Const_vtms1', 'level').Value = 10;
end
```

- Python:

```
# Check if specific parameter exists
if driver.Parameters.Contains("Const_vtms1", "level"):
      # Set data value
      driver.Parameters("Const_vtms1", "level").Value = 10
```

GetNames()

Description

Returns the names of all global parameters or the parameters of the specified instance.

Prototype

```
GetNames(string instanceId)
```

Parameters

Parameter	Description
instanceId	Instance ID or empty string for global parameters.

Returns

None.

Exceptions

None.

Examples

- MATLAB:

```
ids = driver.Parameters.InstanceIds;
for id_idx = 1:length(ids)
    id = ids{id_idx};
    names = driver.Parameters.GetNames(id);
    for name_idx = 1:length(names)
        name = names{name_idx};
        fprintf('%s.%s\n', id, name);
    end
end
```

- Python:

```
for id in driver.Parameters.InstanceIds:
      for name in driver.Parameters.GetNames(id):
            print("%s.%s" % (id, name))
```

Item()

Description

Gets an item from the collection by its name.

This is the default method of `SimulationParameterCollection` and in some languages its name can be omitted (for example in Python).

Prototype

```
SimulationParameter Item(string instanceId = "", string name)
```

Parameters

Parameter	Description
`instanceId`	ID of some instance or empty string for global parameters.
`name`	Name of the parameter.

Returns

`SimulationParameter` object with the specified name if it exists in the collection or throws an exception otherwise.

Exceptions

Message Text	Possible Problem
The given key was not present in the dictionary.	Raised if there is no global parameter with this name or no such parameter in the specified instance.

Examples

- MATLAB:

```
% Get global parameter
param1 = driver.Parameters.Item('', 'A1');
% Get parameter of module with instance ID Const_vtms1
param2 = driver.Parameters.Item('Const_vtms1', 'level');
```

- Python:

```
% Get global parameter
param1 = driver.Parameters("", "A1")
% Get parameter of module with instance ID Const_vtms1
param2 = driver.Parameters("Const_vtms1", "level")
```

Constants

Constants represent predefined named numerical values. In different languages they may be represented differently, either as the original numerical value or as named values (Python, Excel) or strings (MATLAB). In MATLAB, same constants may be represented with either numerical or string values, depending on how they are accessed.

MessageType

The enumeration defines type of the message received from the simulation engine by `MessageRecieved` event via “pull” or `MessageReceivedEx` event via “push” event handling mechanisms.

Info

Description

Informational message.

Value

- Numerical value:
  `1`
- MATLAB
  `'MessageType_Info'`
- Python:
  `win32com.client.constants.MessageType_Info`

Warning

Description

Warning message.

Value

- Numerical value:
  `2`
- MATLAB
  `'MessageType_Warning'`
- Python:
  `win32com.client.constants.MessageType_Warning`

ParameterExpressionType

This enumeration defines the type of expression specified for the parameter.

Standard

Description

Standard expression.

Value

- Numerical value:

  `1`

- MATLAB

  `'ParameterExpressionType_Standard'`

- Python:

  `win32com.client.constants.ParameterExpressionType_Standard`

Python

Description

Python expression.

Value

- Numerical value:

  `2`

- MATLAB

  `'ParameterExpressionType_Python'`

- Python:

  `win32com.client.constants.ParameterExpressionType_Python`

File

Description

Filename expression.

Value

- Numerical value:

  `3`

- MATLAB

  `'ParameterExpressionType_File'`

- Python:

  `win32com.client.constants.ParameterExpressionType_File`

SimulationEventType

This enumeration defines types of events which are supported via the “pull” event handling mechanism (see “Events” on page 327 for more information).

Some events may have additional parameters which are explicitly mentioned. Those parameters can then be accessed via the dynamic properties of the `SimulationEvent` object.

MessageReceived

Description

An event of this type is fired when a message or warning is received from the simulation engine.

Parameters

Index	Description
Message	Gets a message received from the simulation engine.
MessageType	Type of the message (see “MessageType” on page 440).

Value

- Numerical value:

  3

- Python:

  `win32com.client.constants.SimulationEventType_MessageReceived`

Examples

- MATLAB:

```
MESSAGE_RECEIVED_EVENT = 3;
while true
      % Get next event from the queue
      event = driver.GetNextEvent(-1);
      % Check if it is MessageReceived event
      if event.get('Type') == MESSAGE_RECEIVED_EVENT
            msg = event.get('Message');
            msgType = event.get('MessageType');
            if msgType == MESSAGE_INFO_TYPE
                  fprintf('Informational message: %s\n', msg);
            elseif msgType == MESSAGE_WARNING_TYPE
                  fprintf('Warning message: %s\n', msg);
            else
                  fprintf('Message: %s\n', msg);
            end
```

```
        end
end
```

- Python:

```
while True:
        # Get next event from the queue
        event = driver.GetNextEvent()
        # Check if it is MessageReceived event
        if event.Type==constants.SimulationEventType_MessageReceived:
                messageType = event.MessageType
                if messageType == win32com.client.constants.MessageType_Info:
                        print("Informational message:")
                elif messageType == win32com.client.constants.MessageType_Warning:
                        print("Warning message:")
                print("%s" % event.Message)
```

RuntimeDataReceived

Description

An event of this type is fired when runtime data is received by the Simulation Engine Driver after it was sent from the simulation engine via the `SimDataReturn` command.

Parameters

None.

Value

- Numerical value:

  `1`

- Python:

  `win32com.client.constants.SimulationEventType_RuntimeDataReceived`

Examples

- Simulation script:

```
# Initialize data and send to client
set _values(Value1) 25.18
SimDataReturn _values
```

- MATLAB:

```
RUNTIME_DATA_RECEIVED_EVENT = 1;
% Get next event from the queue
event = driver.GetNextEvent(-1);
% Check if it is RuntimeDataReceived event
if event.get('Type') == RUNTIME_DATA_RECEIVED_EVENT
        % Get run-time value
        value = driver.OutputRuntimeData.Item('Value1');
end
```

- Python

```
# Get next event from the queue
event = driver.GetNextEvent()
# Check if it is RuntimeDataReceived event
trgtEvent = constants.SimulationEventType_RuntimeDataReceived
if event.Type == trgtEvent:
        # Get run-time value
```

```
value = driver.OutputRuntimeData.Item("Value1")
```

SimulationEngineStateChanged

Description

An event of this type is fired when the simulation engine state changes.

Parameters

Index	Description
PrevState	State which was active before the simulation engine switched to the current state.
CurState	Current state of the simulation engine at the moment of the event.

Value

- Numerical value:

  `0`

- Python:

  `win32com.client.constants.SimulationEventType_SimulationEngineStateChanged`

Examples

- MATLAB:

```
STATE_CHANGED_EVENT = 0;
IDLE_STATE = 0;
while true
      % Get next event from the queue
      event = driver.GetNextEvent(-1);
      % Check if it is SimulationEngineStateChanged event
      if event.get('Type') == STATE_CHANGED_EVENT
            % Check if it is final event
            if event.get('CurState') == IDLE_STATE
                  break;
            end
      end
end
```

- Python:

```
STATE_CHANGED_EVENT = constants.SimulationEventType_SimulationEngineStateChanged
IDLE_STATE = constants.SimulationEngineState_Idle
while True:
      # Get next event from the queue
      event = driver.GetNextEvent()
      # Check if it is SimulationEngineStateChanged event
      if event.Type == STATE_CHANGED_EVENT:
            # Check if it is final event
            if event.CurState == IDLE_STATE:
                  break
```

OutputDataReceived

Description

An event of this type is fired when output data is received by the Simulation Engine Driver. Output data may be sent from the simulation engine via the `SimOutputReturn` or `SimOutputReturnEx` command.

Parameters

None.

Value

- Numerical value:

  ```
  2
  ```

- Python:

  ```
  win32com.client.constants.SimulationEventType_OutputDataReceived
  ```

Examples

- Simulation script:

  ```
  # Initialize data and send to client
  set _output(Value1) 25.18
  SimOutputReturn _output
  ```

- MATLAB:

  ```
  OUTPUT_RECEIVED_EVENT = 2;
  % Get next event from the queue
  event = driver.GetNextEvent(-1);
  % Check if it is OutputsReceived event
        if event.get('Type') == OUTPUT_RECEIVED_EVENT
              % Get output value
              value = driver.OutputData.Item('Output1');
        end
  end
  ```

- Python

  ```
  OUTPUT_RECEIVED_EVENT = constants.SimulationEventType_OutputDataReceived
  # Get next event from the queue
  event = driver.GetNextEvent()
  if event.Type == OUTPUT_RECEIVED_EVENT:
        # Get output value
        value = driver.OutputData("Output1")
  ```

SimulationEngineState

During its life cycle, the simulation engine may be in different states. Those states are represented by the `SimulationEngineState` enumeration.

The current state of the simulation engine can be accessed via the `SimulationEngineState` property of the `SimulationEngineDriver` object.

Also the `SimulationEngineStateChanged` event is sent when the simulation engine transitions from one state to another. This event has additional parameters representing the previous and current states.

CommandPending

Description

The simulation engine requested a command via `SimGetCommand` and is waiting for it.

Value

- Numerical value:

  `2`

- MATLAB

  `'SimulationEngineState_CommandPending'`

- Python:

  `constants.SimulationEngineState_CommandPending`

RuntimeDataPending

Description

The simulation engine requested a command via `SimDataFetch` and is waiting for it.

Value

- Numerical value:

  `3`

- MATLAB

  `'SimulationEngineState_RuntimeDataPending'`

- Python:

  `constants.SimulationEngineState_RuntimeDataPending`

Idle

Description

Simulation engine isn't running (it is either wasn't started yet or already finished execution).

Value

- Numerical value:

  `0`

- MATLAB

`'SimulationEngineState_Idle'`

- Python:

  `constants.SimulationEngineState_Idle`

Running

Description

Simulation engine is running.

Value

- Numerical value:

  `1`

- MATLAB

  `'SimulationEngineState_Running'`

- Python:

  `constants.SimulationEngineState_Running`

Index