CppSim Behavioral Simulation Package

Michael H. Perrott
http://www-mtl.mit.edu/~perrott

Copyright © 2002 by Michael H. Perrott
All rights reserved
He had no beauty or majesty to
attract us to him,
nothing in his appearance
that we should desire
him.
But he was pierced for our
transgressions,
he was crushed for our
iniquities;
the punishment that brought us
peace was upon him,
and by his wounds we are
healed.

Isaiah 53:2,5
## Contents

1 Foreword 9

2 Introduction 11
   2.1 Comparison to Other Simulation Packages 11
   2.2 Object Oriented Simulation Code 14
   2.3 The Issue of Ordering 16
   2.4 Outline of Book 17

3 Setup and Use (Windows version) 19
   3.1 Installation 19
   3.2 CppSim Directory Contents 20
   3.3 Running CppSim 21

4 Overview 23
   4.1 Schematic Views 23
   4.2 Netlist Format (netlist) 25
   4.3 Module Description File (modules.par) 27
   4.4 Simulation Description File (test.par) 30
   4.5 CppSim Commands 31
   4.6 Viewing Results 31

5 Specifying Simulation Parameters (test.par) 33
   5.1 Parsing Rules 33
   5.2 num_sim_steps: 34
   5.3 Ts: 34
   5.4 output: 34
   5.5 probe: 35
CONTENTS

5.6 global_nodes: ................................... 36
5.7 global_param: ................................... 36
5.8 top_param: ..................................... 36
5.9 alter: ........................................ 37
5.10 inp_timing: ................................... 38
5.11 inp_dig: ..................................... 38

6 Writing Code for Primitives (modules.par) 41
  6.1 Parsing Rules ................................... 41
  6.2 module: ...................................... 43
  6.3 description: .................................. 43
  6.4 parameters: ................................... 43
  6.5 inputs: ...................................... 43
  6.6 outputs: ...................................... 44
  6.7 classes: ..................................... 44
  6.8 static_variables: .............................. 44
  6.9 init: ........................................ 45
  6.10 code: ........................................ 46
  6.11 sim_order: ................................... 47

7 General Purpose CppSim Classes 49
  7.1 List ......................................... 50
  7.2 Clist ........................................ 53
  7.3 Probe ....................................... 57
  7.4 Filter ....................................... 59
  7.5 Amp ......................................... 63
  7.6 EdgeDetect .................................. 65
  7.7 SdMbitMod ................................... 66
  7.8 Rand ......................................... 68
  7.9 Quantizer .................................... 70

8 CppSim Classes for PLL/DLL Simulation 75
  8.1 SigGen ....................................... 76
  8.2 Vco ......................................... 79
  8.3 Delay ....................................... 81
## CONTENTS

8.4 Divider .................................................. 83
8.5 Latch ..................................................... 84
8.6 Reg ....................................................... 86
8.7 Xor ....................................................... 88
8.8 And ....................................................... 90
8.9 Or ......................................................... 92
8.10 EdgeMeasure ........................................... 94

**A Example Simulation Code (Not Auto-Generated)**

A.1 Classical Synthesizer .................................... 98
A.2 Σ-Δ Synthesizer ........................................ 101
A.3 Linear CDR ............................................. 103
A.4 Bang-bang CDR ........................................ 107

**B Hspice Toolbox for Matlab**

B.1 Setup .................................................. 111
B.2 List of Functions ....................................... 112
B.3 Examples .............................................. 113
Chapter 1

Foreword

As an IC designer, I often found myself frustrated by existing behavioral simulation tools, and would typically go down the road of writing my own C or C++ code to examine architectural issues. Now that I’ve entered the academic realm, I find myself wanting to pass on the ‘tricks of the trade’ I have learned over the years, and thereby speed up the progress of my students. Also, I have observed a general need for system simulation tools that are fast and flexible and also integrated within current CAD tool frameworks for IC design.

The CppSim simulation package is my response to those needs. My hope is that it will allow my students, and others, to quickly assess architectural ideas and then seamlessly move on to circuit design within the same CAD framework, and to leverage each others system designs through the existence of a common framework for behavioral simulation.

C++ was chosen as the simulator language do its powerful features and fast execution speed. It turns out that C++ is a fantastic language for representing high level systems due to its support for object oriented descriptions. Indeed, systems can be described in a hierarchical manner, object code can be executed in a multi-rate manner, and signals can be stored in binary format compatible with other simulators.

A significant problem with C++ is that most circuit designers do not like to program, and the learning curve for C++ is perceived as formidable. Also, complex system descriptions quickly become unrecognizable in the form of text, and are much better specified in a graphical manner to allow the designer to ‘see’ signal paths and topological structures such as feedback loops.

The CppSim package makes two contributions to the behavioral simulation of systems. First, it provides a netlist to C++ translator that allows the C++ simulation code to be automatically written based on a SPICE-compatible netlist produced by a graphical schematic
capture program. In doing so, the designer can quickly piece together a system in a graphical manner based on a library of system primitives with corresponding code descriptions, and benefit from the power and speed of running compiled C++ code. Second, the CppSim package provides a set of C++ classes that allow fast and convenient implementation of system primitives. Common system blocks such as filters, VCO’s, nonlinear amplifiers, and signal generators are easily realized using these classes, so that the creation of new system primitives is typically fast and straightforward. Also, special blocks, which are based on the area-conservation approach described in the paper referenced below, are included which allow fast and accurate behavioral simulation of phase locked loop and delay locked loop systems.

Currently, the CppSim package is license-free software that may be used for either academic or commercial use. The source code for the C++ classes is provided, and binary files for implementing the netlist to C++ translator are included for the SUN Solaris operating system. It is possible that future versions of the CppSim package will be commercialized, but that is not a path that is being pursued at the time of this writing. If you benefit from the use of this package, it would be appreciated if you would tell others, and also include a reference to the package in any papers you publish for which the software proved useful. For general simulation of systems, an appropriate reference would be:

Perrott, M.H., “CppSim Behavioral Simulator Package,”
http://www-mtl.mit.edu/~perrott

If you apply the package to the simulation of phase locked loop or delay locked loop systems, it would be appreciated if you would also include the reference:

Perrott, M.H., “Fast and Accurate Behavioral Simulation of Fractional-N Frequency Synthesizers and other PLL/DLL Circuits,”
Design Automation Conference, June, 2002

Michael Perrott
Massachusetts Institute of Technology
August 2002
Chapter 2

Introduction

This chapter introduces the CppSim package in a broad manner so that the reader can develop a sense of how it fits in with other simulation packages, understand its overall framework, and be aware of the assumptions it makes. We begin by comparing CppSim to other simulation packages, discussing its object oriented framework and the issue of execution order, and then providing a summary of the rest of the book.

2.1 Comparison to Other Simulation Packages

This section compares the SPICE, Simulink, and Verilog AMS simulation packages to the CppSim package. This information will hopefully allow the user to understand the strengths and weaknesses of CppSim, and to see how it fits in with other simulators used in the current IC design flow.

SPICE

The SPICE simulation environment determines the solution to a set of simultaneous equations that are specified through a netlist describing the interaction between the system nodes. This ‘fine-grain’ simulator is required when attempting to estimate the performance of analog circuits implemented with transistors and passive elements. However, the solution of simultaneous equations is a slow process, and the resulting simulation times are too long to allow characterization of the behavior of medium to large systems.
Simulink

Many systems are designed in a block diagram manner in which there is little interaction between elements contained in different blocks. In this case, there is no need to solve simultaneous equations, but, rather, the system can be viewed as a set of expressions relating the outputs of each block to its inputs and internal state. The overall outputs of the system can be calculated in terms of its inputs by properly assembling the individual block relationships into a large computation structure.

Simulink assumes such a block level view of the system, and assembles the individual block relationships into one giant state-space description of the overall system. The response of the overall system outputs to variations of its inputs is then computed using standard matrix computations. Since the level of detail of the system is much lower than would be encountered in SPICE, and there is no need to solve simultaneous equations, the computation runs much faster than SPICE and the behavior of medium size systems can be explored. Also, the graphical interface of Simulink allows beginners to quickly come up to speed in doing simulations of a desired system.

Unfortunately, there are a number of disadvantages to the Simulink approach. First, to facilitate the creation of the overall state-space model of the system, Simulink requires that individual blocks be specified in terms of a state-space description. Although this is a general way to describe the block relationships, it usually proves rather cumbersome for the user when attempting to specify the behavior of new blocks. Second, Simulink operates much more quickly on vectorized expressions than with expressions that are invoked within conditional loops — this fact also proves cumbersome to the user since conditional loops provide a much more general computation structure than vectorized expressions. Third, although Simulink simulations run faster than SPICE for a given system, they are still quite slow compared to custom C/C++ programs (in fact, uncompiled, they are well over an order of magnitude slower than their custom C/C++ counterparts). Fourth, the Matlab language is rather limited compared to C++, so that advanced users can feel stifled in terms of their ability to efficiently describe the functional behavior of their system blocks. Finally, the graphical framework of Simulink is disjoint from other CAD tools used in integrated circuit (IC) design, which creates a significant disconnect between architectural exploration and circuit design investigation.
2.1. COMPARISON TO OTHER SIMULATION PACKAGES

Verilog AMS

Verilog AMS is one of the most promising simulation environments to appear on the IC CAD scene for some time. This simulator combines SPICE and Verilog simulators into a common simulator core, and therefore allows analog blocks to be described in terms of coupling relationships between nodes, and digital blocks to be described in terms of Verilog code. Therefore, analog and digital circuits can be co-simulated, and the overall behavior, and possibly even performance, of the system can be investigated.

Unfortunately, Verilog AMS currently has some deficiencies when trying to investigate systems at an architectural level. Specifically, it lacks a set of fast, high level macromodels to describe analog blocks at a behavioral level. The approach of using SPICE representations to represent such blocks has two major drawbacks — the resulting simulation times are too long, and the level of detail that needs to be supplied by the user is too great. A follow on project to CppSim is to develop fast, high level macromodels for analog blocks within the Verilog language. The resulting simulation times are expected to be orders of magnitude faster than their SPICE counterparts, so that the overall behavior of a large system with analog and digital sections can be investigated in a timely manner.

CppSim

The C++ language offers the flexibility of computing system behavior in any manner desired — it can based on the solution of simultaneous equations as assumed in SPICE or on the solution of input/state/output relationships as assumed in Simulink. It is indeed a powerful language, and allows you to quickly perform low level computation while also offering high level structural constructs such as classes. The ability to represent systems in an object oriented manner allows an elegant framework for their simulation. These facts make C++ the language of choice for designers that want the maximum freedom in developing simulation code for an investigated system.

Unfortunately, C++ has the drawbacks that it requires a large amount of effort to develop simulation code, and that the resulting text description of the system is much less intuitive than a graphical representation. The CppSim package removes these issues by supplying classes that allow easy representation of system building blocks such as filters, amplifiers, VCO’s, etc., and by supplying a netlist to C++ conversion utility that enables automatic code generation from a graphical description using any mainstream schematic editor package. The resulting environment provides both beginners and advanced users a powerful tool for
simulating large systems, and also enables the tool to be completely integrated within all mainstream IC CAD tools that support SPICE netlisting.

The simulation approach taken with CppSim is to represent blocks in the system based on input/state/output relationships as done with Simulink. However, unlike Simulink, these relationships do not need to be placed in state-space form, and conditional loops are supported rather than vectorized expressions. The blocks are represented in an object oriented manner, and the simulation code calculates the overall system behavior by computing the output of each block one at a time for each sample point in the simulation. This approach carries the advantage of allowing straightforward description of blocks, fast computation, and the ability to easily support multi-rate operation of different blocks in the system.

The primary disadvantages to CppSim are that it is limited to architectural investigation since Verilog and SPICE descriptions of blocks are not supported, and that the user needs to pay attention to the computation order of blocks in some cases. Note that the latter issue will be discussed in more detail below. All things considered, however, I believe that CppSim will offer designers an extremely fast and powerful framework for investigating the behavior of large systems.

### 2.2 Object Oriented Simulation Code

The underlying philosophy of the CppSim simulator is to represent the various blocks in a system as objects that update their outputs one sample at a time based on inputs that are specified one sample at a time. The influence of the inputs on the outputs of each block are determined by their specified behavior, which is set at the beginning of a simulation run. The block behavior can be a function of state information as well as the block inputs — the state information is preserved inside its respective block so that the overall simulator need not keep track of it.

An example is in order to illustrate the important concepts of the object oriented approach. Figure 2.1 displays an example system to be simulated which consists of 5 blocks that are connected in a feedback system. The pseudo-code for simulating this system in the CppSim framework is listed below:

```
///////// Declaration Statements //////////
ClassA a(behavior settings);
ClassB b(behavior settings);
⋯ other declarations ⋯
```
In the above code, we see that the simulation consists of the following structure:

1. Declaration statements
   - Set behavior of objects

2. Main simulation loop
   - Compute object outputs one sample at a time
   - Save selected signals to a file

As stated above, the behavior of objects is specified in the declaration section — examples of declaration statements for various classes are given in Chapters 7 and 8. A main simulation loop executes the ‘inp’ routine for all simulation blocks according to the order they are placed within the loop. The ‘inp’ routine updates the current outputs of the object based on inputs entering the routine. If there are no inputs, the output value is updated solely on its current state and declared behavior. As revealed in the above code, blocks can be placed within
conditional statements so that their output value is updated only when the statement is true. By doing so, blocks can be executed in multi-rate fashion according to, for instance, clock signals generated by other blocks in the system. The outputs of each block, as well as important state variables, can be easily accessed for each block by using the notation

\[
\text{block}_\text{name}.\text{signal}
\]

The code above illustrates this point for the output signals of various blocks. Finally, signals associated with different blocks are saved to a file using the ‘probe’ function.

### 2.3 The Issue of Ordering

The execution order of the various blocks in a system has an impact on the effective delay seen between the blocks. This point is illustrated in Figure 2.2, which shows the impact of choosing different order arrangements. In case (1), the chosen order arrangement leads to zero delay between all blocks except between the output of D and the other blocks. The reason for this delay is that D is the last block in the simulation loop, and its value does not impact the other blocks until the next simulation time step. Note that the delay value corresponds to one time step of the simulator, and has negligible impact on most analog feedback systems since the simulator sample rate is typically much higher than the bandwidth of the feedback system. For digital circuit networks, much care must be taken to insure that simulation induced delays do not compromise the true behavior of the system. Cases (2) through (4) display alternate ordering arrangements, and illustrate the corresponding delays between blocks that they induce.

When directly creating C++ code with the CppSim classes, as shown in Appendix A, the order of execution is directly controlled by the user by the relative placement of each block in the code. When creating C++ code from a netlist, the CppSim package attempts to order the blocks to achieve the minimal number of induced delays through the efforts of an auto-ordering algorithm. When there are no feedback loops embedded within other feedback loops, the algorithm generally does a good job. However, when multiple-embedded feedback loops are present, the user may want to bypass the auto-ordering algorithm and directly specify the desired order by using the ‘sim_order’ command described in Chapter 6.

It is important to understand that CppSim orders blocks on a cell-by-cell basis. In other words, when it encounters a given cell in the system hierarchy, it executes all the blocks in that cell before moving back to a higher level of hierarchy. Once you encapsulate blocks into a given cell, the ordering of those blocks will remain consistent with respect to each
other regardless of the higher level portion of the system. Therefore, it is a good strategy to encapsulate blocks that are order-sensitive with respect to each other into the same cell so that their order remains constant regardless of changes you make to the rest of the system.

2.4 Outline of Book

An outline of the rest of this book is as follows. Chapter 3 covers basic setup issues involved in the installation of CppSim on your computer. Chapter 4 provides an overview of the CppSim framework in going from schematic description to viewing the results of the simulation. Chapter 5 provides detail with respect to the setting of simulator parameters (such as the number of time steps and simulator sample period), and Chapter 6 provides detail with respect to representing blocks in the schematic with corresponding C++ code. Chapters 7 and 8 provide documentation of the CppSim classes. Finally, Appendix A provides C++
code examples using the CppSim classes, and Appendix B provides documentation for the Hspice Toolbox for Matlab, which is useful for viewing and postprocessing simulation results.
Chapter 3

Setup and Use (Windows version)

This chapter outlines basic instructions for installing and running CppSim.

3.1 Installation

It is assumed that you are currently in possession of a file called setup_cppsime.exe. This self-extracting file supports operation in Windows XP/2000, and assumes that the Sue2 schematic editor will be used for design entry. Other installations of CppSim support use of Cadence for design entry — see the CAD tools link at http://www-mtl.mit.edu/~perrott for details.

Before installing CppSim in Windows, you must install the Cygwin package available at http://www.cygwin.com. Be sure to follow the directions on the web site for proper installation, and also make sure that you select “install” rather than “default” on the “all” item of the Cygwin setup wizard.

You should also install the Sue2 package available through the CAD tools link at http://www-mtl.mit.edu/~perrott. Directions for installation are included in that package.

Given Cygwin and Sue2 have been installed, install CppSim by simply running setup_cppsime.exe in Windows. When prompted for a location for the main CppSim directory, it is highly recommended that you place it in your home directory in the Cygwin environment. Click through the remaining items to complete the file installation.

Before running CppSim, you must include its executables in the Windows path variable. To do so in Windows XP, click into the System icon contained in the Control Panel (which is displayed by clicking on Start->Control Panel). Click the Advanced tab of the
System Properties menu that comes up, and then select the Environment Variables button. Now Edit the Path variable contained in the list of System variables shown in the Environment Variables window. Assuming you have placed Cygwin in \texttt{c:\cygwin} and CppSim in \texttt{c:\cygwin\home\username}, add the following at the beginning of the Path list: \texttt{c:\cygwin\bin;c:\cygwin\home\username\cppsim\bin; ...}

Finally, be sure to modify thecppsim script contained in CppSim/bin according to the instructions in that file.

3.2 CppSim Directory Contents

The CppSim directory should contain the following directories:

- **CommonCode**
  - Contains the CppSim classes, an example my\_classes\_and\_functions.h and .cpp file, and an example modules.par file that matches the CppModules Sue2 library.

- **Doc**
  - Contains this document and an expanded DAC paper describing techniques to achieve fast and accurate PLL simulations. These techniques are implemented in the CppSim classes provided for PLL/DLL simulations.

- **HspiceToolbox**
  - Contains the Hspice Toolbox for Matlab, which allows a convenient and powerful waveform viewer and postprocessor for simulated signals from the Hspice and CppSim simulators.

- **MatlabCode**
  - Contains Matlab code useful for plotting the simulated phase noise of the synthesizer examples and the simulated phase error of the clock and data recovery examples.

- **NonNetlistExamples**
  - Contains example C++ code based on the CppSim classes for frequency synthesizers and clock and data recovery circuits.
3.3 Running CppSim

To run CppSim, open a bash shell in Cygwin and go to an appropriate directory under 
CppSim/SimRuns corresponding to a given cell to simulate. Note that all cells to be sim-
ulated must be placed under the SimRuns directory within a directory corresponding to 
the cell name that is, in turn, contained within a directory corresponding to the Library 
name. For example, go to the directory CppSim/SimRuns/CppExamples/sd_synth_fast, 
which corresponds to a Sue2 cell called sd_synth_fast that is located in the Sue2 library 
CppExamples.

Inside the given cell directory must be placed a file called test.par, which contains direc-
tives to CppSim such as the number of simulation points and the time step value. You will 
find such a file in the CppSim/SimRuns/CppExamples/sd_synth_fast library, along with a 
netlist.

To run CppSim, simply type ccppsim with the bash shell while in the appropriate cell 
directory (i.e., CppSim/SimRuns/CppExamples/sd_synth_fast). Given you have properly 
set up your Windows path variable and ccppsim script parameters, this command will run 
the ccppsim script contained in CppSim/bin and automatically netlist the cell, convert it to 
C++ code, compile it, then run the simulation. Results will be stored in a file called test.tr0, 
which can be read into Matlab using the Hspice Toolbox.
Chapter 4

Overview

This section provides an overview of the steps involved in running simulations within the CppSim framework. The intention is to provide the reader with a feeling of the issues involved — more detailed explanations will be covered in the following chapters. As such, we will look at an example simulation system that is provided with the CppSim package, namely the sd synth cell included in Sue2 library CppExamples. We will examine schematic views associated with this cell, the corresponding netlist and modules.par file, the main simulator description file test.par used to set the key simulator parameters, a description of the UNIX commands required to perform the simulation, and a brief overview of how to view the results.

4.1 Schematic Views

An example schematic that was drawn in the Sue2 schematic capture program, which corresponds to the sd synth cell, is shown in Figure 4.1. The circuit corresponds to a Σ-Δ fractional-N frequency synthesizer, and consists of a phase/frequency detector (PFD), charge pump, lead/lag loop filter, voltage controlled oscillator (VCO), divider, and a Σ-Δ modulator that dithers the instantaneous divide value. In addition, a reference frequency is generated using a VCO module with a lower frequency, and a step input is fed into the Σ-Δ modulator to observe the settling dynamics of the overall synthesizer.

A key observation in the above schematic is that symbols can have associated parameters that specify aspects of their behavior. For instance, the lead/lag filter has parameters $f_p$, $f_z$, and $A$ that specify its associated pole, zero, and gain values. In the case of the lead/lag filter, its parameters are “hard-wired” to fixed values. However, parameters values can
also be specified in terms of expressions that include higher level parameters (see the next paragraph) or global variables. For instance, the step_in symbol in Figure 4.1 contains expressions involving global parameters. Here we have added the suffix ‘gl’ to alert us to the fact that the parameter is global, but this notation is not necessary.

TheCppSim simulator allows schematics to be hierarchical in nature, so that symbols at any level can be represented by schematics consisting of other symbols. As an example, in Figure 4.1, the PFD symbol has an associated schematic that is shown in Figure 4.2. In turn, the nand2 symbol within the PFD schematic also has an associated schematic, which is not shown here for the sake of brevity. It is important to note that parameters can be passed between levels of hierarchy, so that symbols contained in the schematic view of a higher level symbol can inherit parameters values from the higher level symbol.

At the lowest level in the schematic hierarchy, symbols must correspond to C++ code that implements the function associated with the symbol. These symbols are referred to as primitives, and have an associated schematic that typically does not contain other symbols. Such a schematic will often look like the one shown in Figure 4.3, which corresponds to an XOR gate that has inputs $a$ and $b$, and an output $y$. However, the schematic view of primitives can also contain other instances, transistors, resistors, etc. — it simply ignores everything within it when code is specified for it in the ‘modules.par’ file. Therefore, the module definitions within the ‘modules.par’ file control the level of hierarchy that is descended to in any cell. For instance, if you desired to go deeper into the hierarchy of a cell that you already defined code for in ‘modules.par’, simply comment out the module.
Netlist Format (netlist)

The CppSim program operates on a SPICE-compatible netlist, which is typically produced from a schematic capture program, to produce the corresponding C++ simulation code. This netlist must follow the standard format for SPICE simulations in which all modules are declared using a .subckt command, and all instances within modules begin with the letter ‘x’ and include parameter expressions after the associated module name is specified. A portion of the netlist produced by the schematics described in the previous section is shown below:

```
***** Hspice Netlist for Cell 'sd_synth' *****
```
******* Module noise ***********
.subckt noise out
.ends noise

******* Module leadlagfilter ***********
.subckt leadlagfilter out in
.ends leadlagfilter

******* Module sd_modulator ***********
.subckt sd_modulator out clk in
.ends sd_modulator

******* Module constant ***********
.subckt constant out
.ends constant

******* Module step_in ***********
.subckt step_in step
.ends step_in

******* Module vco ***********
.subckt vco sineout squareout vctrl
.ends vco

***** Other blocks not included for brevity *****

******* Module xorpfld ***********
.subckt xorpfld out div ref
xi4 net17 net13 div xor_out net18 diffset
xi3 xor_out net27 net33 xor2
xi8 out net11 net8 nand2
xi7 net8 xor_out net17 nand2
xi5 net18 net11 ref xor_out net13 diffreset
xi1 net27 net24 ref net24 diff2
xi2 net33 net30 div net30 diff2
.ends xorpfld

******* Module sd_synth ***********
xi21 net015 pfdout gated_noise varneg=1.2e−24 varpos=1.85e−25
xi20 net011 noise var=3.25e−16
xi19 net017 chout net015 add2
xi18 vin net019 net011 add2
xi17 net019 net017 leadlagfilter fp=127.2e3 gain=1/(30e−12) fz=11.6e3
xi13 chout pfdout ch_pump ival=1.5e−6
xi12 pfdout div ref xorpfld
xi9 div_val div sd_in sd_modulator order=2
xi8 net01 constant consval=0
4.3 Module Description File (modules.par)

All primitive symbols in the netlist must be associated with corresponding C++ code that describes their function. The code definitions for all of the primitives are contained within one file that we will refer to as ‘modules.par’. A portion of the modules.par file associated with the netlist above is listed below:

```cpp
module: gain
description: gain element
parameters: double gain
inputs: double a
outputs: double y
classes:
static variables:
init:
code:
y=a+gain;
```

```
module: constant
description: constant for input to other blocks
parameters: double consval
```

A few more words are in order with respect to parameters associated with instances in the netlist. The netlist above shows parameters associated with instances but not their corresponding module definitions. In practice, one may also specify parameters in the module definition to specify default values for parameters. For instance, the module definition for the lead/lag filter could be specified as:

```
*************** Module leadlagfilter ***************
.subckt leadlagfilter out in fp=100e3 gain=1/100 fz=10e3
.ends leadlagfilter
```

In this case, if any instances of the lead/lag filter declared in other modules omit any of the parameters declared in the above lead/lag filter module definition, the value of those omitted parameters will be set to the default value specified in the above module definition.
inputs:
outputs: double out
static_variables:
classes:
init:
code:
out = consval;

module: noise
description: Gaussian noise source
parameters: double var
inputs:
outputs: double out
static_variables:
classes: Rand randg("gauss")
init:
code:
out = sqrt(var/Ts)*randg.inp();

module: step_in
description: step input
parameters: double vend double vstart double tstep
inputs:
outputs: double step
classes:
static_variables: double i
init: i=0.0;
code:
step = i*Ts > tstep ? vend : vstart;
i++;
As seen in the above file, each module is described by a list of items that includes its inputs, outputs, and parameters along with a specification of their respective types (i.e., int, double, etc.). The input, output, and parameter names must match those in the corresponding module definition in the netlist (i.e., module ‘gain’ must have nodes a and y in its module
definition along with parameter \textit{gain} in either the module definition or the corresponding instance calls). It should be noted that the netlist is converted to lowercase text, so that the input, output, and parameter names must all be lowercase in any module definition in the module.par file.

Each module definition must also specify all classes that are used for its code implementation, along with initialization and main code descriptions. Initialization code is run only once at the beginning of a simulation, while the main code is run each time step of the simulation. Note that any variables declared in the main code section will lose their value each time the time step is incremented in the simulation. If variables are required which must retain their values between time steps, they should be declared as static variables using the \texttt{\`static\_variables:} command.

Please see Chapter 6 for more information on writing module description files, which includes issues related to syntax and information on the various commands that are available.

\section{Simulation Description File (test.par)}

The overall systems parameters, such as number of time steps and the simulation step size are specified in a simulation description file that we will refer to as \texttt{test.par}'. An example test.par file is given below:

\begin{verbatim}
num_sim_steps: 2e6
Ts: 1/10e9

** save most signals every time step
output: signals
probe: out ref vin pfdout sd_in xi12.xor_out

** save sd_modulator output only on rising edges of divider output
output: sd_out trigger=div
probe: div_val

global_param: in_gl=92.31793713 delta_gl=4 step_time_gl=.7e6*Ts
global_nodes: vdd=1.0 gnd=-1.0
* top_param: x=in_gl+2.0
* alter: delta_gl=1:0.25:4
* inp_timing: 1e-9 5e-9 1/2e9 0 1
* inp_dig: node1 [1 0 1 (0 3) (1 4) 0]
* inp_dig: node2 [1 0 0 (1 3) (0 5) 1]
\end{verbatim}
4.5 CppSim Commands

The recommended method of running CppSim is to use the cppsim script provided in the CppSim package, so that minimal effort is required of the user to run simulations. See the Setup section (Chapter 3) for details. Note that, upon completion of the simulation, you can view the results using Awaves or Matlab (using the Hspice Toolbox included with this package).

4.6 Viewing Results

The output of the CppSim simulation run is in Hspice-compatible binary format, and is stored for this example in the files ‘signals.tr0’ and ‘sd_out.tr0’ as directed by the ‘output:’ commands in the above ‘test.par’ file. To view signals, one can either use an Hspice output viewer such as Awaves, or the Hspice Toolbox for Matlab that is included with the CppSim package. Documentation for the Hspice Toolbox is included as an appendix at the end of this document.

Using the Hspice Toolbox, the signals sd_in and vin were plotted and are illustrated in Figure 4.4.
Figure 4.4  Synthesizer response to a step input — top plot: input to sd_modulator, bottom plot: closed loop response of VCO input voltage.
Chapter 5

Specifying Simulation Parameters
(test.par)

This chapter discusses the various commands available for use in the simulation description file, which we refer to as the ‘test.par’ file. We begin by covering general parsing issues, including the notation for commenting out lines, and will then discuss the various commands in detail.

5.1 Parsing Rules

The CppSim environment is designed to be very forgiving with respect to parsing rules so that one does not need to remember adding commas or semicolons at the right place. In general, spaces are used to separate commands from their arguments, as well as arguments from each other, and commas and semicolons are ignored. As an example, the statement

    global_param: a_g=3.3 b_g=-5 c_g=.7e6*Ts

can also be written as

    global_param: a_g=3.3 b_g=-5 c_g=.7e6*Ts

or as

    global_param: a_g=3.3, b_g=-5, c_g=.7e6*Ts;

or as

    global_param:
    a_g=3.3
    b_g=-5
    c_g=.7e6*Ts
As a rule of thumb, one should never separate an expression into multiple lines. As an example, you should NOT write

```
  global_param: a_gl=
  3.3
```

Either '*' or '%' or '//’ characters can be used to comment out lines provided that they are the first characters encountered on a line. As an example

```
* global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts
*** global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts
% global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts
%%% global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts
// global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts
///// global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts
```

are all valid ways of commenting out a line.

Descriptions of the individual commands used in a ‘test.par’ file are presented below:

### 5.2 `num_sim_steps:`

The number of simulation steps is specified with this statement. An example of the syntax of this command is:

```
num_sim_steps: 2e6
```

### 5.3 `Ts:`

The value of the simulation period, Ts, is specified with this statement. Note that Ts becomes a global variable in the C++ simulation code, and is often used in module parameter statements as well as module code. An example of the syntax of this command is:

```
Ts: 1/10e9
```

### 5.4 `output:`

The name of the Hspice-compatible binary output file for signals specified in the following ‘probe:’ statement. Nominally, the specified signals are stored every time step of the simulator. An optional parameter allows one to store the signal values only on the rising edges of a trigger signal.

Example: save signals every time step in the binary file ‘signals.tr0’
output: signals
probe: a b y xi12.net12

Example: save signals only when the signal ‘xi1.clk’ has a rising edge in the binary file ‘signals2.tr0’
output: signals2 trigger=xi1.clk
probe: a2 b2 xi3.xi1.sd_out

Example: save signals only when the signal ‘clk’ has a falling edge in the binary file ‘signals3.tr0’
output: signals3 trigger=-xi1.clk
probe: a2 b2 xi3.xi1.sd_out

Example: do all of the above
output: signals
probe: a b y xi12.net12
output: signals2 trigger=xi1.clk
probe: a2 b2 xi3.xi1.sd_out
output: signals3 trigger=-xi1.clk
probe: a2 b2 xi3.xi1.sd_out

Note that in all cases, the trigger signal must be a square wave that alternates between either -1 and 1, 0 and 1, or 0 and -1 (see the description of the EdgeDetect class).

5.5 probe:

The signals specified with this statement are saved in an Hspice-compatible binary file whose name is specified by the last ‘output:’ command encountered before this statement. Signals contained in the top level of the schematic are specified by their name, and signals at lower levels of hierarchy are specified using standard SPICE notation by their name and by the chain of instances that lead to the cell view that the signal is contained in. An example of the syntax of this command is:

probe: vin pfdout sd_in div_val xi12.xor_out xi12.xi7.y

where xi12.xor_out corresponds to signal xor_out contained within instance xi12 in the top level of hierarchy, and xi12.xi7.y corresponds to signal y contained within instance xi7 that is within instance xi12. Note that the number of levels of hierarchy within the system can be as large as the user desires.
5.6 global_nodes:

A global node is assumed to have a constant signal value across all levels of hierarchy. The signal value of such nodes are specified with this statement. For example, nodes that are named vdd and gnd can be specified to have signal levels 1.0 and -1.0, respectively, using the statement:

    global_nodes: vdd=1.0 gnd=-1.0

In practice, it is inappropriate for global nodes to correspond to the output node of any instance — they should always correspond to input nodes. No checking is done to insure this is the case.

5.7 global_param:

Global parameters are defined at all levels of hierarchy in the system. These parameters can be used within parameter expressions, and can also be used within module code (although that is not generally recommended). It is suggested that the user label these parameters in a manner that reflects the fact that they are global, such as by adding the suffix ‘.gl’ to their names. Note that the variable Ts is automatically supplied as a global parameter, and its value is set according to the ‘Ts:’ statement described above. An example of the syntax of this command is:

    global_param: a_gl=3.3 b_gl=-5 c_gl=.7e6*Ts

5.8 top_param:

Top level parameters are defined only in the top level of the system hierarchy. Therefore, this command would be used instead of the ‘global_param:’ command if one wanted to constrain the scope of a parameter to the top level as opposed to having it pervade all levels of hierarchy. These parameters cannot be altered using the ‘alter:’ command described below, but can be defined in terms of global parameters which can be altered. An example of the syntax of this command is:

    top_param: yval=1/4e9 xval=a_gl+2.0
5.9 alter:

You can alter global parameters several ways using the ‘alter:’ statement, which will now be explained through a series of examples. In all of the examples, the ‘alter:’ statement(s) are assumed to be placed after the ‘global_param:’ statement that defines the global parameters being altered.

Example: do simulations over all combinations of \(a_{gl} = 15,18\) and \(b_{gl} = 1e3,2e3\)

\[
\text{alter: } a_{gl} = 15 \ 18 \\
\text{alter: } b_{gl} = 1e3 \ 2e3
\]

Example: do simulations where \(a_{gl}\) and \(b_{gl}\) are altered together, i.e., \(a_{gl},b_{gl} = 15,1e3\) and \(a_{gl},b_{gl} = 18,2e3\)

\[
\text{alter: } a_{gl} = 15 \ 18 \ b_{gl} = 1e3 \ 2e3
\]

Example: do combinations where \(a_{gl}\) and \(b_{gl}\) are altered together in combination with values of \(c_{gl} = 1,2,3,4,5\)

\[
\text{alter: } a_{gl} = 15 \ 18 \ b_{gl} = 1e3 \ 2e3 \\
\text{alter: } c_{gl} = 1 \ 2 \ 3 \ 4 \ 5
\]

Example: an easier way to do the above is to use Matlab notation:

\[
\text{alter: } a_{gl} = 15 \ 18 \ b_{gl} = 1e3 \ 2e3 \\
\text{alter: } c_{gl} = 1:5
\]

Example: suppose you want to increment \(c_{gl}\) by 0.1 instead of 1

\[
\text{alter: } c_{gl} = 1:0.1:5
\]

Example: combine Matlab notation with individual specifications

\[
\text{alter: } c_{gl} = 1e3 \ 5e3:1e3:10e3 \ 20e3 \ 50e3 \ 100e3:100e3:1e6 \ 2e6
\]

The resulting output of performing such ‘alter:’ operations is to produce a separate output file for each global parameter combination. As an example, if

\text{output: signals}

is specified in the test.par file, then a a series of files

\text{signals.tr0 signals.tr1 signals.tr2 \ldots}

will be produced. If you have multiple ‘output:’ statements, then a series of files will be produced for each of those output files.

To see how the global variable combinations match up to their corresponding transient runs in this case, you can load testGlobals.tr0 in Matlab. Note that the prefix ‘test’ was determined by the name of the simulation description file, ‘test.par’. If you instead, as an example, named this file ‘test2.par’, you would load test2Globals.tr0. Each altered global variable will be a signal in that file whose value for each simulation run is specified.
5.10 inp_timing:

Input signals into the simulated system should generally be created within the graphical environment of the schematic capture program. However, there are cases where it is easier to specify signals directly in the test.par file. Specifically, digital signals are easier to specify in an ASCII editor as a vector sequence than by going through the graphical environment. In the future, other types of signals may also be supported.

The ‘inp_timing:’ command is used to specify the timing parameters associated with input signals that follow it. The parameters of this command are as follows:

inp_timing: delay rise/fall_time period vlow vhigh

In the above statement, ‘delay’ corresponds to the initial delay of the waveform, ‘rise/fall_time’ corresponds to its rise and fall times, ‘period’ is its period in seconds, and ‘vlow’ and ‘vhigh’ are its minimum and maximum signal values. Currently, digital inputs ignore all of the parameters except for ‘delay’ and ‘period’, but all of the parameters must still be specified. An example of the syntax of this command is:

inp_timing: 1e-9 .5e-9 1/2e9 0 1

Note that the inp_timing command can NOT span over multiple lines.

5.11 inpDig:

Digital inputs are specified in vector form by the ‘inpDig:’ command, and take on the timing specifications of the last ‘inp_timing:’ statement encountered. Each input alternates between 1.0 (corresponding to bit value 1) and -1.0 (corresponding to bit value 0). Transition values between these two levels will take on values between -1.0 and 1.0 depending on the location of the edge within the simulator time sample period — these signals therefore conform to the area-conserving transition technique discussed in the paper included in the CppSim package. The signals will also repeat if the time span over which they are defined is exceeded. Note that the inpDig command can NOT span over multiple lines. The command is best explained through a few examples.

Example: create a digital clock signal that drives node1

inpDig: node1 [1 0]

The input into node1 therefore consists of a square wave signal that alternates between 1.0 and -1.0 according to the area-conserving transition technique.

Example: create a digital signal with the pattern (-1,1,1,1,-1,-1,1) that is continually repeated
Example: another way of specifying the above sequence is

```plaintext
inp_dig: node2 [0 1 1 0 0 1]
```

Example: create two clock signals at different frequencies along with accompanying signals

```plaintext
inp_timing: 0 0 1/1e9 0 1
inp_dig: clk_slow [1 0]
inp_dig: data_slow [1 0 1 1 0]
inp_timing: 0 0 1/2e9 0 1
inp_dig: clk_fast [1 0]
inp_dig: data_fast [0 1 1 1 0 0 1]
```
Chapter 6

Writing Code for Primitives
(modules.par)

This chapter discusses the various commands available for use in the module code description file, which we refer to as the ‘modules.par’ file. We begin by covering general parsing issues, including the notation for commenting out lines, and will then discuss the various commands in detail.

6.1 Parsing Rules

As in the case of the ‘test.par’ file, the CppSim environment is designed to be very forgiving with respect to parsing rules for module code definitions so that one does not need to remember adding commas or semicolons at the right place. In general, spaces are used to separate commands from their arguments, as well as arguments from each other, and commas and semicolons are ignored. As an example, the statement

\[
\text{inputs: double a double b}
\]

can also be written as

\[
\text{inputs: double a, double b;}
\]

or as

\[
\text{inputs:}
\text{\hspace{1em}}
\text{double a}
\text{\hspace{1em}}
\text{double b}
\]
As a rule of thumb, one should never separate a specific description into multiple lines. As an example, you should NOT write

```plaintext
inputs: double

a
```

Either the ‘*’ or ‘%’ or ‘//’ characters can be used to comment out lines provided that they are the first characters encountered on a line.

Example: comment out specific lines in a module description:

```plaintext
module: xor2
description: two-input xor gate
// parameters: double w
inputs: double a, double b
outputs: double y
classes: Xor xor1()
// init:
y = -1.0;
code:
xor1.inp(a,b);
y = xor1.out;
```

Example: comment out entire module description:

```plaintext
// module: xor2
description: two-input xor gate
parameters: double w
inputs: double a, double b
outputs: double y
classes: Xor xor1()
init:
y = -1.0;
code:
xor1.inp(a,b);
y = xor1.out;
```

Descriptions of the individual commands used to describe module code are presented below:
6.2 module:

The name of the module specified here must match the associated module it represents in the netlist. An example of the syntax of this command is:

```plaintext
module: and2
```

6.3 description:

A text description of the module function. This is ignored by CppSim, but is useful for sharing the code with others. An example of the syntax of this command is:

```plaintext
description:
  implements a two input ‘and’ function whose signals conform to the area conservation protocol for representing transitions
```

6.4 parameters:

The type and name of all parameters associated with the module are specified with this command. The parameter names must match those in the associated module in the netlist — since CppSim converts the netlist text to lowercase, lowercase parameter names must be specified here. An example of the syntax of this command is:

```plaintext
parameters: double gain double fc int order
```

One should note that expressions for parameter values given in the netlist will be converted to lowercase, so that it is good practice to define lowercase variables in the test.par file. A related issue is that the global variable 'Ts' has an uppercase letter — if you desire to use it in expressions for parameter values within the netlist, then you should define a lowercase version in the test.par file using

```plaintext
global_param: ts=Ts
```

6.5 inputs:

The type and name of all inputs associated with the module are specified with this command. The input names must match those in the associated module in the netlist — since CppSim converts the netlist text to lowercase, lowercase input names must be specified here. An example of the syntax of this command is:

```plaintext
inputs: double a double b int c
```
6.6 outputs:

The type and name of all outputs associated with the module are specified with this command. The output names must match those in the associated module in the netlist — since CppSim converts the netlist text to lowercase, lowercase output names must be specified here. An example of the syntax of this command is:

outputs: double y double yb int yd

6.7 classes:

The declaration statement of all classes used in the module code must be placed here. Any number of classes can be specified, and parameters specified in the ‘parameters:’ statement can be passed to class declaration expressions. A few examples are in order.

Example: declare two classes for use in the module code

classes: Reg reg1() Xor xor1()

or

classes:
Reg reg1()
Xor xor1()

Example: declare a class using parameters ‘freq’ and ‘kvco’ that have been specified with the ‘parameters:’ command, and the global parameter ‘Ts’

classes: Vco vco("fc + Kv*x","fc,Kv,Ts",freq,kvco,Ts)

Note that a few words are in order with regards to syntax. First, all class objects must have an associated set of parenthesis, as seen above with the class objects reg1() and xor1(). Second, class declarations cannot be broken up between lines, i.e., do NOT write:

classes: Vco vco("fc + Kv*x","fc,Kv,Ts",freq,kvco,Ts)

6.8 static variables:

Since CppSim enters and exits the module code during each simulation time sample, variables that are declared in the ‘code:’ section will go in and out of scope during each time sample, and therefore will not retain their values from one simulation time step to the next. In the
case that you want to have a variable that retains its value from one simulation time step to the next, you can declare it as a static variable using the ‘static_variables:’ command. In addition, static variables can be probed using the ‘probe:’ statement in the test.par file, and therefore offer a means of probing signals embedded in the module code. An example of the syntax of this command is:

```
static_variables: double var1 int SigNum char Name[10]
```

Note that static variable names can freely use uppercase letters.

### 6.9 init:

The ‘init:’ command allows you to run initialization code that is executed only once at the beginning of a simulation run (note that if you use the ‘alter:’ statement in the test.par file, the initialization code will be run once for each global parameter combination). Therefore, you can use this command to initialize variables and also to redefine class behavior based on module parameters specified with the ‘parameters:’ command. For instance, the code below redefines the noise shaping transfer function of a Σ-∆ modulator object based on the module parameter ‘order’, and initializes the variable ‘out’ to the value of 2:

```cpp
init:
    if (order == 1)
        sd_mod.set("1 - z^-1");
    else if (order == 2)
        sd_mod.set("1 -2z^-1 + z^-2");
    else
        sd_mod.set("1 -3z^-1 + 3z^-2 - z^-3");
    out = 2.0;
```

You can see the full module definition corresponding to the above ‘init:’ command in Chapter 4, section 3.

An important issue in writing code in the ‘init:’ section is that it is not parsed by CppSim, but rather is passed straight to the C++ simulation code. Therefore, the syntax of the code must conform to the C++ language — if it does not, the C++ compiler will generate error messages.
6.10 code:

The ‘code:’ section contains the C++ code that implements the desired function of the module. This code is run every time the simulator time step is incremented, and can include variable declarations, functions, class objects, and all the standard C++ directives such as for loops and if-else condition statements. One thing to be aware of is that if the user wants to use their own user defined classes or functions, the class/function declarations must be placed in the my_classes_and_functions.h file and the class/function code must be placed in the my_classes_and_functions.cpp file, both of which must be located in the same directory as cppsim_classes.h and cppsim_classes.cpp. The need for user defined classes/functions should be minimal given the flexibility of the CppSim classes. Another issue is that variables declared in the ‘code:’ section will lose their value from time step to time step, so that the ‘static_variables:’ command should be used in some cases as discussed above. Finally, you should be aware of the fact that the code specified in the ‘code:’ section is not parsed by CppSim, but rather is passed straight to the C++ simulation code. Therefore, the syntax of the code must conform to the C++ language — if it does not, the C++ compiler will generate error messages.

Some examples are in order — each of the ones to follow are taken from the modules.par file that is shown in Chapter 4, section 3.

Example: implement a ‘gain’ module by setting its output, ‘y’, equal to its input, ‘a’, times a gain parameter ‘gain’:

```cpp
code:
y=a*gain;
```

Example: implement a ‘noise’ module using a class object ‘randg’ (which implements a Gaussian random sequence with variance one) whose output is scaled by the variance parameter ‘var’ and global variable ‘Ts’:

```cpp
code:
out = sqrt(var/Ts)*randg.inp();
```

The reader is invited to look at more examples of ‘code:’ sections listed in Chapter 4, Section 3, as well as the module descriptions contained in the example modules.par file that is included in the CppSim package.
6.11 sim_order:

As discussed in the Introduction chapter, CppSim executes each instance contained in a non-primitive module (i.e. a module whose functionality is defined by the netlist rather than by corresponding code in the modules.par file) one at a time until all have been executed, updates the simulator time step, and then repeats the instance by instance execution. The instance order in this execution sequence is nominally set by an algorithm internal to the CppSim simulator. Generally, if the system does not contain feedback loops embedded within other feedback loops, the algorithm does the right thing. However, if you do have a feedback loop embedded within another feedback loop, you may want to bypass the algorithm and directly specify the order of instance execution.

The ‘sim_order:’ command allows you to specify the order of execution of the instances in a non-primitive module. As an example, consider ordering the instances contained in the ‘xorpdf’ module specified in the netlist file in Chapter 4, Section 2. The module definition within the modules.par file that would accomplish this task is given as:

module: xorpdf
sim_order: xi1 xi2 xi3 xi5 xi4 xi7 xi8

Note that if you want to specify the order of instance execution in the top module in the netlist (i.e., the highest cellview in the hierarchy), but your netlist does not include the top module within a .subckt definition, then simply use the same description technique but specify the module name as ‘top’, i.e.,

module: top
sim_order: xi8 xi6 ...

Note that the default behavior for the Sue2 setup is to include the top module within a .subckt definition, in which case you refer to the top module by its schematic cell name rather than by ‘top’.

CppSim will check that all instances are included in your list, and also alert you to any instance names not being present in the associated netlist description of the module.

Note that there is an alternative method of specifying the order of execution of the instances within a cell that does not involve changes to the modules.par file. Specifically, CppSim reserves the parameter ‘sim_order’ to specify the order of execution of instances within a cell, so that the user may add this parameter to each instance and set the execution order in their schematic editor. If this method is used, all instances in the cell must have ‘sim_order’ as a parameter, and its value for a given instance should be set lower than the
values of other instances that should be executed after it. In other words, CppSim orders the instances from lowest to highest ‘sim_order’ value. One can use decimal values for the ‘sim_order’ value, so that an instance with ‘sim_order=1.5’ executes after an instance with ‘sim_order=1’ and before an instance with ‘sim_order=2’.
Chapter 7

General Purpose CppSim Classes

This chapter provides reference information on the general purpose classes available in the CppSim package. Most of these classes update the output one sample at a time as new input values are fed in. Exceptions to this rule are the List and Clist classes which offer the ability to perform operations on sequences. These classes prove very useful when simulating the behavior of communication systems that require algorithms to be performed on a sequence of data values. For instance, in an OFDM communication system, FFT and inverse FFT operations must be performed on sequences of data in order to implement the modulation and demodulation functions. The Clist class offers support of such operations, as well as a variety of other processing functions.

The description of each class is partitioned into several different sections:

- Declaration: specification of the object behavior,
- Redefinition: redefinition of the object behavior,
- Variables: accessible output and state variables,
- Functions: supported operations on the object,
- Example of Usage: example code.

In each section, an attempt has been made to use a large number of examples to convey the fundamental concepts associated with the respective class.
7.1 List

A linked list of double values that provides storage for sequences of numbers.

Declaration

```c
List list1;
```

Variables

```c
double out; // current entry value
int length; // number of entries in list
int notdone; // notdone = 0 or 1 depending if current entry is last value or not last value, respectively
```

Functions

```c
double read(); // returns current element value and increments pointer
    // recycles back to first element after last is read
void reset(); // reset read pointer to first element
void flush(); // delete all elements from list
int inp(double in); // add a new element of value 'in' to the end of the list (returns list length)
double mean(); // returns mean of list element values
double var(); // returns variance of list element values
void add(const List &other); // adds, element by element, other list values to list values
void add(double in); // adds constant 'in' to all element values in list
void mul(const List &other); // multiplies, element by element, other list values to list values
void mul(double in); // multiplies all element values in list by constant 'in'
void conv(const List &other); // convolves list with other list
void cat(const List &other); // concatenates other list to list
int load(char *filename); // loads values from file into list (previous elements destroyed, returns list length)
void save(char *filename); // saves element values to file
void copy(const List &other); // copy other list into list (previous elements destroyed)
void print(char *name); // print list to stdout using 'name' as label
```

Example of Usage

```c
#include "com_blocks.h"

main()
{
    // declarations
    List list1,list2,list3,list4;
    double val;
    int i;
```
// incrementally load values into list1
list1.inp(1.0);
list1.inp(-1.0);
list1.inp(3.0);
list1.inp(5.0);

// incrementally load values into list2
list2.inp(3.0);
list2.inp(-4.0);
list2.inp(7.0);
list2.inp(5.0);

// load values into list3 from file
list3.load("list3.dat");

// copy all entries of list3 into list4 (all previous list4 entries deleted)
// (list3 unchanged)
list4.copy(list3);

// concatenate list2 and list4, store result in list4 (list2 unchanged)
list4.cat(list2);

// illustrate read() by printing out entries of list4 ‘by hand’
list4.reset(); // resets read pointer to first entry
i = 1;
while(list4.notdone)
{
    val = list4.read(); // read current entry, increment read pointer
    printf("list4[%d] = %5.3f\n", i++, val);
}

// also illustrate that you can refer to list4.out after read() instead of recording its return value
list4.reset(); // resets read pointer to first entry
for (i = 1; i <= 30; i++) // will recycle through elements of list4 many times
{
    list4.read(); // read current entry, increment pointer
    printf("list4[%d] = %5.3f\n", i, list4.out);
}

// print out entries of list4 using print function
list4.print("list4"); // character string serves as label for printout
// save entries of list4 to file 'list4.dat'
list4.save("list4.dat");

// purge all entries from list4
list4.flush();

// add list1 and list2 elements, store result in list1 (list2 unchanged)
list1.add(list2);
// add the constant 5.0 to all elements in list2
list2.add(5.0);

// multiply list1 and list2 elements, store result in list2 (list1 unchanged)
list2.mul(list1);
// multiply list1 by the constant -3.0
list1.mul(-3.0);

// convolve list1 and list3 elements, store result in list3 (list1 unchanged)
list3.conv(list1);

// calculate mean of list3
val = list3.mean();

// calculate variance of list3
val = list3.var();
}
7.2 Clist

A grouping of two linked lists of double values that form a complex sequence.

Declaration

```c
Clist clist1;
```

Variables

```c
List real; // sequence of real element values
List imag; // sequence of imag element values
double outr; // current real entry value
double outi; // current imag entry value
int length; // number of entries in clist
int notdone; // notdone = 0 or 1 depending if current complex entry is last value or not last value, respectively
```

Functions

```c
void read(); // read current element value into outr and outi and increment pointer
    // recycles back to first complex element after last is read
void reset(); // reset read pointer to first complex element
void flush(); // delete all elements from clist
int inp(double rin, double iin); // add a new element to the end of the list (returns list length)
    // real part of new element is 'rin', and imag part of new element is 'iin'
int inp(const List &rin, const List &iin); // create a complex sequence using list 'rin' as
    // real part and list 'iin' for imag part (all previous entries destroyed, returns list length)
void cat(const Clist &other); // concatenates other clist to clist
void add(const Clist &other); // adds, element by element, other clist values to clist values
void add(double rin, double iin); // adds constant 'rin + j*iin' to all element values in clist
void mul(const Clist &other); // multiplies, element by element, other clist values to clist values
void mul(double rin, double iin); // multiplies all element values in clist by constant 'rin + j*iin'
void conv(const Clist &other); // convolves clist with other clist
void conv(const List &other); // convolves clist with other list (other list assumed to be real)
void copy(const Clist &other); // copy other list into list (previous elements destroyed)
void fft(const List &data); // calculate fft of list 'data' ('data' assumed to be real)
void fft(const Clist &data); // calculate fft of clist 'data' (data is complex)
void fft(const List &data_real, const List &data_imag); // calculate fft of sequence with real part
    // 'data_real' and imag part 'data_imag'
void ifft(const Clist &fft_in); // calculate inverse fft of clist 'fft_in'
void ifft(const List &fft_real, const List &fft_imag); // calculate inverse fft of sequence with real part
    // 'fft_real' and imag part 'fft_imag'
void print(char *name); // print clist to stdout using 'name' as label
```
Example of Usage

```c
#include "com_blocks.h"

main()
{
    // declarations
    List list1, list2;
    Clist clist1, clist2;
    int i;

    // incrementally load values into list1
    list1.inp(1.0);
    list1.inp(-1.0);
    list1.inp(3.0);
    list1.inp(5.0);

    // incrementally load values into list2
    list2.inp(3.0);
    list2.inp(-4.0);
    list2.inp(7.0);
    list2.inp(5.0);

    // create a complex list with list1 as real part, list2 as imag part
    clist1.inp(list1, list2);

    // copy clist1 to clist2
    clist2.copy(clist1);

    // change clist1 to its complex conjugate
    clist1.imag.mul(-1.0);

    // multiply elements of clist1 and clist2 and store in clist1 (clist2 unchanged)
    clist1.mul(clist2);

    // add elements of clist1 and clist2 and store in clist2 (clist1 unchanged)
    clist2.add(clist1);

    // multiply elements of clist2 by 1+j3
    clist2.mul(1, 3);

    // add 2-j7 to elements of clist2
```
clist2.add(2, -7);

// concatenate clist1 to clist2
clist2.cat(clist1);

// convolve clist1 and clist2 and place in clist2 (clist1 unchanged)
clist2.conv(clist1);

// compute the fft of clist2 and place in clist1 (clist2 unchanged)
clist1.fft(clist2);

// compute the ifft of clist1 and place back into clist1
clist1.ifft(clist1);

// compute the fft of list1 (assumed real) and place in clist2
clist2.fft(list1);

// delete all entries in clist2
clist2.flush();

// enter in new values
clist2.inp(1.0, 2.0); // add element of value 1.0 + j2.0
clist2.inp(-1.0, 3.0); // add element of value -1.0 + j3.0
clist2.inp(7.0, 4.0); // add element of value 7.0 + j4.0
clist2.inp(5.0, 1.0); // add element of value 5.0 + j

// illustrate read() by printing out entries of clist2 ‘by hand’
clist2.reset(); // resets read pointer to first complex entry
i = 1;
while (clist2.notdone)
{
    clist2.read(); // read current complex entry, increment pointer
    printf("clist2[%d] = %5.3f + j%5.3f\n", i++, clist2.outr, clist2.outi);
}

// illustrate recycling property of read() by reading beyond length of list
clist2.reset(); // resets read pointer to first complex entry
for (i = 1; i <= 20; i++) // will recycle through complex elements of clist2 many times
{
    clist2.read(); // read current complex entry, increment pointer
    printf("clist2[%d] = %5.3f + j%5.3f\n", i, clist2.outr, clist2.outi);
}

// print out entries of clist2 using print function
clist2.print("clist2"); // character string serves as label for printout

// save values of clist2 to files
clist2.real.save("real.dat");
clist2.imag.save("imag.dat");
}
7.3 Probe

Save data to binary file which can be read with Hspice Toolbox in Matlab.

Declaration

```c
Probe probe1("test.tr0"); // save probed data in file 'test.tr0', sample period = 1
Probe probe2("test2.tr0",1e-6); // save probed data in file 'test2.tr0', sample period = 1e-6
Probe probe3("test3.tr0",1e-6,10); // save probed data in file 'test3.tr0', sample period = 1e-6
    // subsample data by a factor of 10 before saving to file
```

Variables

None

Functions

```c
void inp(double node_value, char *node_name); // send double value to probe file with label 'node_name'
void inp(int int_node_value, char *node_name); // send integer value to probe file with label 'node_name'
```

Example of Usage

```c
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double sig1,sig2;

    Ts = 1e-6;
    Probe probe1("test.tr0");
    Probe probe2("test2.tr0",Ts);
    Probe probe3("test3.tr0",Ts,10);
    Probe probe4("test4.tr0",Ts);

    for (i = 0; i < 1000; i++)
    {
        sig1 = sin((2*PI/200)*i);
        sig2 = cos((2*PI/200)*i);
        // save signals to 'test.tr0' with TIME signal in file having period = 1
        probe1.inp(sig1,"sine");
        probe1.inp(sig2,"cosine");
        // save signals to 'test2.tr0' with TIME signal in file having period = Ts
```
probe2.inp(sig1,"sine");
probe2.inp(sig2,"cosine");
// save signals to 'test3.tr0' with TIME signal in file having period = Ts
// and being subsampled by a factor of 10 (i.e. only 100 samples saved per signal in this case)
probe3.inp(sig1,"sine");
probe3.inp(sig2,"cosine");
}

// Recommendation: don't re-use probe statements in two different loops
for (i = 0; i < 1000; i++)
{
    sig1 = sin((2*PI/250)*i);
    sig2 = cos((2*PI/250)*i);
    // The following probe statements will produce an error
    // probe1.inp(sig1,"sine2"); // don't do this!
    // probe1.inp(sig2,"cosine2"); // don't do this!

    // Probe statements used in previous loops must follow same order of signal names probed
    probe2.inp(sig1,"sine"); // acceptable, but not recommended
    probe2.inp(sig2,"cosine"); // acceptable, but not recommended
    // probe2.inp(sig1+sig2,"sum"); // will produce an error since 'sum' not probed in loop above

    // The following probe statements are fine since probe4 was not used above
    probe4.inp(sig1,"sine"); // recommended approach - new probe statement for this loop
    probe4.inp(sig2,"cosine"); // ditto - this is fine
    probe4.inp(sig1+sig2,"sum"); // ditto - this is fine
}
}
7.4 Filter

Continuous-time and discrete-time filters.

Declaration

```plaintext
Filter difference("1 - z^-1","1"); // discrete-time first difference
Filter accum("1","1 - z^-1"); // discrete-time accumulator
Filter zfilt("1 - a*z^-1","1 - b*z^-1","a","b",8,9); // general discrete-time
    // filter consisting of one pole and one zero
Filter accum_lim("1","1 - z^-1","Max","Min",2.5,0.0); // discrete-time accumulator
    // with max and min limits set on its output
Filter delay("z^-no","1","no",10); // delay of 10 samples (must use integer sample delay)
Filter diff("K*s","1","Ts,K",1e-6,2.0); // continuous-time differentiator K*s
    // (Ts is simulation sample period)
Filter integ("K","s","K,Ts",1e-6); // continuous-time integrator K/s
Filter RC_fil("K","1 + 1/(2*pi*fo)*s","K,fo,Ts",1.0,1e3,1e-6); // continuous-time filter
    // corresponding to an RC network with transfer function K/(1 + s/(2*pi*fo))
Filter LC_fil("K","1 + 1/(wo*Q)*s + 1/(wo^2)*s^2","Ts,K,wo,Q",1e-6.1,0.1e3,2*pi,1.2); // continuous-time filter corresponding to an LC network with
    // transfer function K/(1 + 1/(wo*Q)*s + (s/wo)^2)
List list1;
list1.load("list1.dat"); // load data from file into list
Filter(list1,"1"); // create z-domain FIR filter whose numerator z-polynomial is created from
    // list1 elements and whose denominator is 1 (see Usage example below)
Filter("1",list1); // create z-domain IIR filter whose denominator z-polynomial is created from
    // list1 elements and whose numerator is 1
List list2;
list2.load("list2.dat");
Filter(list1,list2); // create z-domain IIR filter whose denominator z-polynomial is created from
    // list1 elements and whose numerator z-polynomial is created from list2 elements
```

Redefinition (this is rarely required)

```
// Declaration
Filter filt1("1 - z^-1","1");

// Redefine
filt1.set("1","1 - z^-1");
// Redefine again
filt1.set("K","1 + 1/(2*pi*fo)*s","K,fo,Ts",1.0,1e3,1e-6);
```
CHAPTER 7. GENERAL PURPOSE CPPSIM CLASSES

Variables

```cpp
double out; // output of filter
char ivar; // independent variable of transfer function - either 'z' or 's'
int num_a_coeff; // number of denominator coefficients
int num_b_coeff; // number of numerator coefficients
```

Functions

```cpp
double inp(double in); // input new sample to filter, resulting output is returned
void reset(double value); // set output and all state information of filter to value
```

Example of Usage

```cpp
#include "com_blocks.h"

int main()
{
    List list1;
    Probe probel("test.tr0");
    int i;
    double in;

    // incrementally load values into list1 to later set z-polynomial of a0 + a1*z^-1 + a2*z^-2
    list1.inp(1.0); // set a0
    list1.inp(-2.0); // set a1
    list1.inp(1.0); // set a2

    // discrete-time filter implementation
    Filter double_accum("1",list1);
    Filter first_diff("1 - z^-1","1");

    in = 1.0;
    for (i = 0; i < 1000; i++)
    {
        if (i == 200)
            in = -1.0;
        else if (i == 400)
            in = 2.0;
        else if (i == 700) // zero out filters at the 700th sample
            {
                double_accum.reset(0.0);
                first_diff.reset(0.0);
```
7.4. FILTER

} // cascade the filters
double_accum.inp(in);
first_diff.inp(double_accum.out);

// save the signals to file 'test.tr0' using probe1
probe1.inp(in,"in");
probe1.inp(double_accum.out,"accum2");
probe1.inp(first_diff.out,"accum1");
}

/////////////////////////////////////////////////////////////////////// continuous-time filter implementation////////////////////

double sample_per;
sample_per = 1e-6; // choose 1 us sample period

Probe probe2("test2.tr0",sample_per); // need a new probe statement for new iteration loop

// create two filters (cascade of which is an integrator)
Filter double_integ("K","s^2","K,Ts",1e5,sample_per);
Filter diff("s","1","Ts",sample_per);
in = 1.0;
for (i = 0; i < 1000; i++)
{
    if (i == 200)
        in = -1.0;
    else if (i == 400)
        in = 2.0;
    else if (i == 700) // zero out filters at the 700th sample
        {
            double_integ.reset(0.0);
            diff.reset(0.0);
        }

    // cascade the filters
    double_integ.inp(in);
    diff.inp(double_integ.out);

    // save the signals to file 'test2.tr0' using probe2
    probe2.inp(in,"in");
probe2.inp(double_integ.out,"int2");
probe2.inp(diff.out,"int1");
}
7.5 Amp

General amplifier with nonlinear characteristic specified by a polynomial and saturating characteristic specified by Min, Max values.

Declaration

```plaintext
Amp amp("off + A*x","off,A",1.0,5.0); // amp offset of 1 V, gain of 5
Amp amp2("off + A*x + A1*x^2 + A2*x^(1/2)","off,A,A1,A2",5.10.0,1.1); // nonlinear characteristic described by a polynomial
Amp amp("off + A*x","off,A,Min,Max",1.0,5.0,0.5,2.0); // Min output is 0.5, Max output is 2.0
```

Redefinition (this is rarely required)

```plaintext
// Declaration
Amp amp("off + A*x","off,A",1.0,5.0); // amp offset of 1 V, gain of 5

// Redefine
amp.set("off + A*x + A1*x^2 + A2*x^(1/2)","off,A,A1,A2",5.10.0,1.1);
// Redefine again
amp.set("off + A*x","off,A,Min,Max",1.0,5.0,0.5,2.0);
```

Variables

```plaintext
double out; // output of amplifier
```

Functions

```plaintext
double inp(double in); // input voltage to amp, returns resulting value of out
```

Example of Usage

```plaintext
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in;

    Ts = 1e−6;
    Probe probel("test.tr0",Ts);
    // create an amplifier with offset -0.5, gain 3.0, and saturation at 0 (min) and 2.0 (max)
    Amp amp1("off + A*x","off,A,Max,Min",−0.5,3.0,2.0,0.0);
```
for (i = 0; i < 1000; i++)
{
    in = (1.0/1000.0)*i;
    amp1.inp(in); // input ramped from 0 to 1
    probe1.inp(in,"in"); // save input of amp
    probe1.inp(amp1.out,"out"); // save output of amp
}
7.6 EdgeDetect

Output is 0 except at the rising edge of its input, at which point the output is 1. The input must be a square wave that alternates between -1 and 1, 0 and 1, or -1 and 0.

**Declaration**

```plaintext
EdgeDetect edge1;
```

**Variables**

```plaintext
int out; // output is 1 if input is rising edge, 0 otherwise
```

**Functions**

```plaintext
int inp(double in); // returns out, input can alternate between -1 and 1,
// 0 and 1, or -1 and 0
int inp(int in); // same as previous function, but with integer input
```

**Example of Usage**
7.7 SdMbitMod

Implements a multi-bit Σ-Δ modulator with a signal transfer function (STF) of 1.0 and a noise transfer function (NTF) that is specified as a $z$ polynomial.

Declaration

```cpp
SdMbitMod sd_mod1("1 - 2*z^-1 + z^-2"); // sd modulator with second order noise shaping
SdMbitMod sd_mod2("1 - 3z^-1 + 3z^-2 - 1z^-3"); // sd modulator with third order noise shaping
```

Redefinition (this is rarely required)

```cpp
// Declaration
SdMbitMod sd_mod1("1 - 2*z^-1 + z^-2");

// Redefine
sd_mod1.set("1 - 3z^-1 + 3z^-2 - 1z^-3");
// Redefine again
sd_mod1.set("1 - z^-1");
```

Variables

```cpp
double out; // output of sd modulator
```

Functions

```cpp
double inp(double in); // returns sd modulator 'out' given input 'in'
```

Example of Usage

```cpp
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Filter lowpass("1",1/(2*pi*fo)*s","fo,Ts",100e3,Ts);
    SdMbitMod sdmod("1 - 2z^-1 + z^-2");
```
for (i = 0; i < 10000; i++)
{
    in = sin(2*pi*50e3*Tsi); // create a sine wave for input to the sd modulator
    sdmod.inp(in); // input the sine wave into the sd modulator
    lowpass.inp(sdmod.out); // filter the sd modulator output
    probe1.inp(in, "in"); // save input sine wave
    probe1.inp(sdmod.out, "sdmod"); // save output of sd modulator
    probe1.inp(lowpass.out, "out"); // save output of lowpass
}

CHAPTER 7. GENERAL PURPOSE CPPSIM CLASSES

7.8 Rand

Produces a random, white sequence whose sample values are chosen according to three different probability distributions:

- “gauss”: Gaussian distribution with mean 0 and variance 1,
- “uniform”: Uniform distribution between 0 and 1 (mean 1/2, variance 1/12),
- “bernoulli”: Bernoulli distribution — value is 1 with probability $p$ or -1 with probability of $1 - p$.

Declaration

```cpp
Rand rand1("gauss"); // Gaussian distribution
Rand rand2("uniform"); // Uniform distribution
Rand rand3("bernoulli"); // Bernoulli distribution with $p = 1/2$
Rand rand4("bernoulli",.25); // Bernoulli distribution with $p = 1/4$
```

Variables

```cpp
double out; // sequence sample chosen according to specified probability distribution
```

Functions

```cpp
void reset(); // resets seed to -1 for random sequence (impacts all random number generators)
void set_seed(int in); // sets seed to value in (impacts all random number generators)
double inp(); // returns sequence output 'out'
```

Example of Usage

```cpp
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in,noise1;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ta",1e6,1e6,Ts);
    Rand rand1("gauss");
```
for (i = 0; i < 10000; i++)
{
    if (i == 5000) // reset random sequence at i = 5000
        rand1.reset();
    if (i == 7000) // change seed value of random sequence at i = 7000
        rand1.set_seed(-2);

    in = (1.0/10000.0)*i; // create a ramp signal
    noise1 = .01*rand1.inp() + 0.2; // gaussian sequence with mean 0.2, variance (.01)^2
    vco.inp(in+noise1); // add noise to input signal to VCO

    probe1.inp(in+noise1,"in"); // save input to VCO
    probe1.inp(noise1,"noise"); // save noise sequence
    probe1.inp(vco.out,"vco"); // save square wave output of VCO
}
}
7.9 Quantizer

Quantizes input according to five parameters: levels, step_size, in_center, out_min, and out_max as shown in Figure 7.1

![Illustration of behavior of Quantizer class in relation to parameter settings.](image)

### Declaration

```
Quantizer quant1(2, .5, 0.0, -1, 1); // levels=2, step_size=0.5, in_center=0, out_min=-1, out_max=1
Quantizer quant2(2, 5.0, .5, -1, 1);
Quantizer quant3(2, 5.0, -0.5, 0, 1);
Quantizer quant4(3, 5.0, 0.0, -1, 1);
Quantizer quant5(5, 5.0, 0.0, -1.1);
Quantizer quant6(5, 5.0, 0.0, 4);
Quantizer quant7(16, 3.0, 0.0, 15);
```

### Variables

```
double out; // output of quantizer
```

### Functions

```
double inp(double in); // returns quantized out for given in
double inp(double in, double clk); // returns quantized out for given in on rising edge of clk
```

### Example of Usage

```
#include "com_blocks.h"
```
main()
{
  double Ts=1;
  Quantizer quant1(2.5,0.0,−1,1);
  Quantizer quant2(2.5,0.5,−1,1);
  Quantizer quant3(2.5,−0.5,0.1);
  Quantizer quant4(3.5,0.0,−1,1);
  Quantizer quant5(5.5,0.0,−1,1);
  Quantizer quant6(5.5,0.0,0.4);
  Quantizer quant7(16.3,0.0,0.15);
  Vco vco("fc + Kv*x","Ts,fc,Kv",Ts,1/150.0,1.0);
  Probe probe1("test.tr0");
  double in;
  int i;

  in = −2.0;
  for (i = 0; i < 4000; i++)
  {
    // generate input signal
    if (i % 2000 < 1000)
      in += .004;
    else
      in −= .004;

    // clockless quantizer examples
    quant1.inp(in);
    quant2.inp(in);
    quant3.inp(in);
    quant4.inp(in);
    quant5.inp(in);
    quant6.inp(in);

    // clocked quantizer examples
    vco.inp(0.0);
    quant7.inp(in,vco.out);

    // save signals to file
    probe1.inp(in,"in");
    probe1.inp(vco.out,"clk");
    probe1.inp(quant1.out,"quant1");
    probe1.inp(quant2.out,"quant2");
  }
probe1.inp(quant3.out,"quant3");
probe1.inp(quant4.out,"quant4");
probe1.inp(quant5.out,"quant5");
probe1.inp(quant6.out,"quant6");
probe1.inp(quant7.out,"quant7");
}
}

Figure 7.2 Plot of quant1, quant2, and quant3 from example simulation.
Figure 7.3  Plot of quant4, quant5, and quant6 from example simulation.

Figure 7.4  Plot of quant7 and associated signals from example simulation.
Chapter 8

CppSim Classes for PLL/DLL Simulation

These classes are used in the same manner as the general purpose ones described in the previous chapter, but are specialized for PLL/DLL simulation in that they implement the area conservation approach for digital signal transitions that is described in the paper

Perrott, M.H., “Fast and Accurate Behavioral Simulation of Fractional-N Frequency Synthesizers and other PLL/DLL Circuits,”
Design Automation Conference, June, 2002

An expanded version of the above paper is included in the CppSim package — the included version more explicitly relates the described techniques to the classes provided in this library.
8.1 SigGen

Produces a waveform of a specified frequency that corresponds to one of four different waveform types:

- “square”: a square wave (alternating between 1 and -1),
- “sine”: a sine wave (amplitude 1, DC offset 0),
- “prbs”: a prbs data sequence (alternating between 1 and -1),
- “impulse”: an impulse data sequence.

Declaration

double Ts=1e-9; // simulation sample period
SigGen siggen1("square",1.0e6,Ts); // square wave of frequency 1 MHz
SigGen siggen2("sine",1.0e6,Ts); // sine wave of frequency 1 MHz
SigGen siggen3("prbs",1.0e6,Ts); // prbs sequence of period 1/(1 MHz), data chosen randomly
List list1;
list1.load("list1.dat"); // load data sequence from file, which must have values of either 1 or -1
SigGen siggen4("prbs",1.0e6,Ts,list1); // prbs sequence of period 1/(1 MHz), data repeats through list1
SigGen siggen4("prbs",1.0e6,Ts,list1,5); // prbs sequence of period 1/(1 MHz),
    // data repeats through list1, start with 5th entry in list
SigGen siggen5("impulse",1.0e6,Ts); // impulse sequence of period 1/(1 MHz), data chosen randomly
SigGen siggen6("impulse",1.0e6,Ts,list1); // impulse sequence of period 1/(1 MHz), data repeats through list1
SigGen siggen6("impulse",1.0e6,Ts,list1,3); // impulse sequence of period 1/(1 MHz),
    // data repeats through list1, start with 3rd entry in list

Redefinition (this is rarely required)

    // Declaration
SigGen siggen1("square",1.0e6,Ts);

    // Redefine
siggen1.set("sine",1.0e6,Ts);
    // Redefine again
siggen1.set("prbs",1.0e6,Ts);

Variables

double out; // waveform output
double phase; // normalized phase of waveform (ramps between 0.0 and 1.0)
double square; // square wave (clk) that output is derived from
8.1. SIGGEN

Functions

```c
double inp(double in); // return waveform output, input specifies phase offset
double reset(); // reset waveform (useful when SigGen initialized with a List)
```

Example of Usage

```c
#include "com_blocks.h"

main()
{
    int i;
    double Ts, freq;

    Ts = 1e-9;
    freq = 1e6;
    Probe probe1("test.tr0", Ts);
    SigGen sine1("sine", freq, Ts);
    SigGen sine2("sine", freq, Ts);
    List list1;
    list1.inp(1);
    list1.inp(-1);
    list1.inp(-1);
    SigGen prbs1("prbs", freq, Ts, list1, 2); // sequence repeats through value 1, -1, -1; starts with entry 2 of list
    SigGen prbs2("prbs", freq, Ts); // sequence randomly takes on values of 1 or -1
    Rand rand1("gauss");

    for (i = 0; i < 10000; i++)
    {
        if (i == 5000) // reset sine1 at i = 5000
            sine1.reset();

        // note: phase shifts specified at input are lowpassed filtered by discrete-time filter
        // freq*Ts/(1 - (1-freq*Ts)*z^-1)
        sine1.inp(0.0); // sine wave with 0 radian phase shift
        sine2.inp(0.25); // sine wave with 1/4*(2*pi) radian phase shift (i.e. cosine wave)
        prbs1.inp(0.0); // 1, -1, -1 sequence with 0 degree phase shift
        prbs2.inp(0.0001*rand1.inp()); // prbs sequence with random phase shift (mean 0, var (.0001*2*pi)^2)

        probe1.inp(sine1.out, "sine1"); // save sine wave
        probe1.inp(sine2.out, "sine2"); // save cosine wave
        probe1.inp(prbs1.out, "wave"); // save 1, -1, -1 waveform
        probe1.inp(prbs2.out, "prbs"); // save prbs waveform
```
}
8.2 VCO

Voltage controlled oscillator. Output is a square wave that alternates between -1 and 1. At edges, value of square wave is between -1 and 1 according to the time occurrence of the edge within the sample period.

Declaration

```c
Vco vco("1.84e9 + 30e6*x","Ts",Ts); // center frequency 1.84 GHz, gain 30 MHz/V,
    // simulation sample period of Ts
Vco vco2("fc + Kv*x","fc,Kv,Ts",1.84e9,30e6,Ts); // center frequency fc, gain of Kv
Vco vco3("fc + Kx*x + Kv2*x^2 + Kx3*x^(1/2)","fc,Kv,Kv2,Kv3,Ts",1.84e9,30e6,5e6,1e6,Ts);
    // Nonlinear VCO characteristic specified by a polynomial
Vco vco4("fc + Kv*x","fc,Kv,Ts,Max,Min",1.84e9,30e6,Ts,2e9,1.7e9); // Max frequency 2e9,
    // Min frequency 1.7e9
```

Redefinition (this is rarely required)

```c
    // Declaration
Vco vco1("1.84e9 + 30e6*x","Ts",Ts);

    // Redefine
vco1.set("fc + Kv*x","fc,Kv,Ts",1.84e9,30e6,Ts);
    // Redefine again
vco1.set("fc + Kx*x + Kv2*x^2 + Kx3*x^(1/2)","fc,Kv,Kv2,Kv3,Ts",1.84e9,30e6,5e6,1e6,Ts);
```

Variables

```c
double out; // square wave alternating between 1 and -1
double phase; // phase wraps so that it varies between 0 and 2pi
```

Functions

```c
double inp(double in); // input voltage to vco, returns resulting value of out
double inp(double in, int divide_val); // vco divided down by divide_value, out returned
double inp(double in, double divide_val); // divide_value can be double or integer
```

Example of Usage

```c
#include "com_blocks.h"

main()
{
    int i;
```
double Ts;
double in,sig1;

Ts = 1e-9;
Probe probe1("test.tr0",Ts);
Vco vco("fc + Kv*x","fc,Kv,Ts",10e6,1e6,Ts);
Vco vco2("fc + Kv*x","fc,Kv,Ts",10e6,1e6,Ts);

for (i = 0; i < 10000; i++)
{
    in = (1.0/10000.0)*i;
    vco.inp(in);  // frequency will gradually increase since in is a ramp
    vco2.inp(in,3);  // divide vco output down in frequency by a factor of 3
    sig1 = sin(vco.phase);  // sine wave output from VCO instead of square wave

    probe1.inp(vco.out,"square");  // save square wave output
    probe1.inp(vco2.out,"square3");  // save square wave output divided by 3
    probe1.inp(sig1,"sine");  // save sine wave output
}
}
8.3  Delay

Variable delay element for inputs that follow the interpolation convention (i.e. they must alternate between -1.0 and 1.0, with transition values taking on a value in the range of -1.0 to 1.0 depending on the location of the transition relative to the sample period). Output alternates between -1.0 and 1.0, with edge values between -1.0 and 1.0 according to their actual time position within the sample period.

**Declaration**

```c
Delay delay1(5.4,11.7); // nominal delay of 5.4 simulator time samples
            // maximum allowable delay of 11.7 simulator time samples
            // minimum allowable delay is always 1.0 simulator time sample

Delay delay2(3.6); // fixed delay of 3.6 simulator time samples
```

**Variables**

```c
double out; // output of delay element (alternates between 1.0 and -1.0)
```

**Functions**

```c
double inp(double in); // input 'in' signal to delay, keep the delay value fixed

double inp(double in, double delay_val); // adjust delay value about
                                             // nominal value according to 'delay_val' signal
```

**Example of Usage**

```c
#include "cppsim_classes.h"

main()
{
    double Ts = 1/20e9;
    Probe probe("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",1e9,30e6,Ts);
    Delay delay2(2.0,10.0); // nominal delay is 2.0 simulation time samples,
                             // allowable delay range is between 1.0 and 10.0 sim. time samples
    Delay delay2(2.5);      // delay will be fixed at 2.5 simulation time samples
    EdgeMeasure vco_period,delay_period;
    EdgeDetect rising_edge;
    double in;
    int i,count;

    count = 0;
```
for (i = 0; i < 700; i++)
{
    // delay will be varied about its nominal value by
    // 1.5 - 1.0 = 0.5 time samples to 1.5 + 1.0 = 2.5 time samples
    if (rising_edge.inp(delay.out))
        in = 1.5 + sin(2.0*pi*(1.0/20.0))*((double) count++);

    // VCO
    vco.inp(0.0);

    // Delay elements
    // vary delay from vco.out to delay.out about its nominal value of 2.0
    // simulation time samples as specified by signal 'in'
    // -> for above 'in', overall delay range spans from 2.5 to 4.5 time samples
    delay.inp(vco.out,in);
    // delay from vco.out to delay2.out is fixed at 2.5 time samples
    delay2.inp(vco.out);

    // Save signals to file
    probe.inp(vco.out,"vco");
    probe.inp(delay.out,"delay");
    probe.inp(delay2.out,"delay2");
    probe.inp(vco_period.inp(vco.out),"vco_period");
    probe.inp(delay_period.inp(delay.out),"delay_period");
    probe.inp(in,"in");
}
}
8.4 Divider

Divides down input square wave (which much alternate between -1 and 1) according to a specified divide value. Output also alternates between -1 and 1.

Declaration

```c
Divider divider1;
```

Variables

```c
double out; // output square wave alternating between 1 and -1
```

Functions

```c
double inp(double in, int divide_value); // returns output given specified input and divide value
double inp(double in, double divide_value); // divide value can be double or integer valued
```

Example of Usage

```c
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in,sig1;

    Ts = 1e−9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",10e6,1e6,Ts);
    Divider divider;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out,10.0); // divide down VCO frequency by a factor of 10

        probe1.inp(vco.out,"vco"); // save square wave output of VCO
        probe1.inp(divider.out,"divide"); // save square wave output of divider
    }
}
```
8.5 Latch

Performs latch function with input, clock, set and reset that alternate between -1 and 1.

Declaration

```c
Latch latch1;
```

Variables

```c
double out; // output is 1 or -1 depending on latch state
```

Functions

```c
double inp(double in, double clk); // returns out based on in and clk
double inp(double in, double clk, double set, double reset); // includes set and reset
void init(double in); // initializes latch to value in (must be -1 or 1)
```

Example of Usage

```c
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in,sig1;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x"," fc,Kv,Ts",10e6,1e6,Ts);
    Divider divider;
    Latch latch1, latch2, latch3, latch4;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out,10.0); // divide down VCO frequency by a factor of 10

        // implement a register with divider output as its input, VCO output as its clock
        latch1.inp(divider.out,vco.out); // first latch of register
        latch2.inp(latch1.out,-vco.out); // second latch of register; note clk is inverted
    }
}
```
8.5. LATCH

// implement a register with divider output as its input, VCO output as its clock
// reset the register according to above register output
latch3.inp(divider.out,vco.out,−1.0,latch2.out); // first latch of register
latch4.inp(latch3.out,−vco.out,−1.0,latch2.out); // second latch of register

probe1.inp(vco.out,"vco"); // save square wave output of VCO
probe1.inp(divider.out,"divide"); // save square wave output of divider
probe1.inp(latch2.out,"reg1"); // save output of register 1
probe1.inp(latch4.out,"reg2"); // save output of register 2
8.6 Reg

Performs register function with input, clock, set and reset that alternate between -1 and 1.

Declaration

```cpp
Reg reg1;
```

Variables

```cpp
double out; // output is 1 or -1 depending on register state
Latch lat1; // first latch in register
Latch lat2; // second latch in register
```

Functions

```cpp
// for all functions, in, clock, set, reset must be -1 or 1, values between -1 and 1 correspond to transitions
double inp(double in, double clk); // returns out based on in and clk
double inp(double in, double clk, double set, double reset); // includes set and reset
void init(double in); // initializes both register latches to value in (must be -1 or 1)
```

Example of Usage

```cpp
#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in,sig1;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",1e6,1e6,Ts);
    Divider divider;
    Reg reg1,reg2;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out,10.0); // divide down VCO frequency by a factor of 10

        // implement a register with divider output as its input, VCO output as its clock
```
8.6. **REG**

```
reg1.inp(divider.out,vco.out);

// implement a register with divider output as its input, VCO output as its clock
// reset the register according to above register output
reg2.inp(divider.out,vco.out,−1.0,reg1.out);

probe1.inp(vco.out,"vco"); // save square wave output of VCO
probe1.inp(divider.out,"divide"); // save square wave output of divider
probe1.inp(reg1.out,"reg1"); // save output of register 1
probe1.inp(reg2.out,"reg2"); // save output of register 2
probe1.inp(reg1.lat1.out,"lat1"); // save output of lat1 of register 1
probe1.inp(reg2.lat1.out,"lat2"); // save output of lat1 of register 2
```
8.7 Xor

Performs ‘xor’ function with 2 inputs that alternate between -1 and 1.

Declaration

Xor xor1;

Variables

double out; // output is 1 or -1 depending on xor of inputs

Functions

// inputs must be either -1 or 1
// values between -1 and 1 correspond to transitions
double inp(double in0, double in1); // returns out = -in0*in1; (the xor function)

Example of Usage

#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in,sig1;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",10e6,1e6,Ts);
    Divider divider;
    Reg reg1;
    Xor xor1, xor2, xor3;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out,10.0); // divide down VCO frequency by a factor of 10
        reg1.inp(divider.out,vco.out); // register divider output with VCO output as clk
        xor1.inp(divider.out,reg1.out); // xor divider out and reg1 out
    }
}
xor2.inp(divider.out, reg1.out); // xor not(divider out) and reg1 out
xor3.inp(divider.out, reg1.out); // xor divider out and not(reg1 out)

probe1.inp(vco.out, "vco"); // save square wave output of VCO
probe1.inp(divider.out, "divide"); // save square wave output of divider
probe1.inp(reg1.out, "reg1"); // save output of register 1
probe1.inp(xor1.out, "xor1"); // save output of xor1
probe1.inp(~xor2.out, "xor2_not"); // save output of not(xor2)
probe1.inp(xor3.out, "xor3"); // save output of xor3
8.8 And

Performs ‘and’ function with 2 to 5 inputs that alternate between -1 and 1.

Declaration

And and1;

Variables

double out; // output is 1 or -1 depending on ‘and’ of inputs

Functions

// inputs must be either -1 or 1, values between -1 and 1 correspond to transitions
double inp(double in0, double in1); // returns out = 1 if and(in0,in1)=1, -1 if and(in0,in1)=0
double inp(double in0, double in1, double in2); // three inputs
double inp(double in0, double in1, double in2, double in3); // four inputs
double inp(double in0, double in1, double in2, double in3, double in4); // five inputs

Example of Usage

#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in,sig1;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",10e6,1e6,Ts);
    Divider divider;
    Reg reg1;
    And and1, and2, and3;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out,10.0); // divide down VCO frequency by a factor of 10
        reg1.inp(divider.out,vco.out); // register divider output with VCO output as clk
    }
}
and1.inp(divider.out, reg1.out); // 'and' divider out, reg1 out
and2.inp(−divider.out, reg1.out); // 'and' not(divider out), reg1 out
and3.inp(vco.out, −reg1.out, and1.out, and2.out); // 'and' vco out, not(reg1 out), and1 out, and2 out

probe1.inp(vco.out, "vco"); // save square wave output of VCO
probe1.inp(divider.out, "divide"); // save square wave output of divider
probe1.inp(reg1.out, "reg1"); // save output of register 1
probe1.inp(and1.out, "and1"); // save output of and1
probe1.inp(−and2.out, "and2_not"); // save output of not(and2)
probe1.inp(and3.out, "and3"); // save output of and3
}
8.9 Or

Performs ‘or’ function with 2 to 5 inputs that alternate between -1 and 1.

Declaration

Or or1;

Variables

double out; // output is 1 or -1 depending on ‘or’ of inputs

Functions

// inputs must be either -1 or 1, values between -1 and 1 correspond to transitions
double inp(double in0, double in1); // returns out = 1 if or(in0,in1)=1, -1 if or(in0,in1)=0
double inp(double in0, double in1, double in2); // three inputs
double inp(double in0, double in1, double in2, double in3); // four inputs
double inp(double in0, double in1, double in2, double in3, double in4); // five inputs

Example of Usage

#include "com_blocks.h"

main()
{
    int i;
    double Ts,
    double in,sig1;

    Ts = 1e-9;
    Probe probe1("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",10e6,1e6,Ts);
    Divider divider;
    Reg reg1;
    Or or1, or2, or3;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out,10.0); // divide down VCO frequency by a factor of 10
        reg1.inp(divider.out,vco.out); // register divider output with VCO output as clk
or1.inp(divider.out, reg1.out); // 'or' divider out, reg1 out
or2.inp(¬ divider.out, reg1.out); // 'or' not(divider out), reg1 out
or3.inp(vco.out, ¬ reg1.out, or1.out, or2.out); // 'or' vco out, not(reg1 out), or1 out, or2 out

probe1.inp(vco.out, "vco"); // save square wave output of VCO
probe1.inp(divider.out, "divide"); // save square wave output of divider
probe1.inp(reg1.out, "reg1"); // save output of register 1
probe1.inp(or1.out, "or1"); // save output of or1
probe1.inp(¬ or2.out, "or2_not"); // save output of not(or2)
probe1.inp(or3.out, "or3"); // save output of or3
}
8.10 EdgeMeasure

Measures time between rising edges of its input. Normalized to a sample time equal to one. The output is zero except at the location of edges.

Declaration

EdgeMeasure edge_time1;

Variables

doUBLE out; // output is time since last edge if input is rising edge, 0 otherwise

Functions

doUBLE inp(double in); // returns out, input must be a square wave alternating between -1 and 1

Example of Usage

#include "com_blocks.h"

main()
{
    int i;
    double Ts;
    double in;
    Ts = 1e−9;
    Probe probe1("test.tr0", Ts);
    Vco vco("fc + Kv*x", "fc,Kv,Ts", 10e6, 1e6, Ts);
    Divider divider;
    EdgeMeasure edge_time1, edge_time2;

    for (i = 0; i < 10000; i++)
    {
        in = (1.0/10000.0)*i;
        vco.inp(in); // frequency will gradually increase since in is a ramp
        divider.inp(vco.out, 10.0); // divide down VCO frequency by a factor of 10
        probe1.inp(vco.out, "vco"); // save square wave output of VCO
        probe1.inp(divider.out, "divide"); // save square wave output of divider
        probe1.inp(edge_time1.inp(vco.out), "vco_time");
        // note: a given EdgeMeasure module cannot be used twice in one loop
probe1.inp(edge_time2.inp(divider.out), "divide_time");
}
Appendix A

Example Simulation Code (Not Auto-Generated)

This appendix provides four different examples illustrating the ability of the CppSim classes to quickly and accurately simulate the behavior of PLL systems ranging from frequency synthesizers to clock and data recovery circuits. These examples are NOT generated from netlists, but rather are directly implemented in C++ code using the CppSim classes. Although the netlist driven method should be used whenever possible, these examples provide the user with a straightforward presentation of the structure and issues associated with doing C++ simulations with the provided classes.
A.1 Classical Synthesizer

Figure A.1 illustrates a classical synthesizer, which consists of a phase/frequency detector (PFD), loop filter, VCO, and frequency divider. The frequency divider value is nominally held to a constant integer value labeled $N_{\text{nom}}$, but is stepped in value when a new output frequency is desired.

A key decision when designing the synthesizer is the choice of PFD structure. Two main styles are available — the Tristate PFD and the XOR-based PFD. The Tristate PFD is the most popular of the two, and allows charge pump noise to be minimized due to the small pulse widths it achieves. The XOR-based PFD has advantages for $\Sigma$-$\Delta$ frequency synthesizers since it avoids small pulses, and thereby achieves better linearity. For this example, we will assume the Tristate PFD shown in Figure A.2 is used.

Example C++ code to achieve simulation of the above system is shown below.

```cpp
#include "com_blocks.h"

main()
{
    double Ts = 1/200e6;
    Probe probe("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",1.84e9,30e6,Ts);
    SigGen ref_clk("square",20e6,Ts);
    Reg reg1,reg2;
    And and1;
    Filter rc_filt("1.0","1 + 1/(2*pi*fp)*s","fp,Ts",127.2e3,Ts);
    Filter int_filt("2*pi*fp/10","s","fp,Ts",127.2e3,Ts);
    double chp_out,vco_in;
}
```
A.1. CLASSICAL SYNTHESIZER

![Diagram of a classical synthesizer](image)

Figure A.2 Tristate PFD.

```c
int i, N;

N = 90;
for (i = 0; i < 200000; i++)
{
    // step desired VCO frequency by 1.0*Kv at sample 160000
    if (i == 160000)
        N += 1;

    // reference oscillator
    ref_clk.inp(0.0);

    // PFD
    reg1.inp(1.0, vco.out, -1.0, and1.out);
    reg2.inp(1.0, ref_clk.out, -1.0, and1.out);
    and1.inp(reg1.out, reg2.out);

    // Charge Pump
    chp_out = (reg2.out - reg1.out) * 1989 * PI;

    // Loop Filter
    rc_filt.inp(chp_out);
    int_filt.inp(chp_out);
```

```
// VCO and divider
vco_in = rc_filt.out+int_filt.out;
vco.inp(vco_in,N);

probe.inp(N,"N");
probe.inp(vco_in,"vco");
}

Inspection of the above code reveals that is quite straightforward to represent the system using the CppSim classes.

Simulated results are shown in Figure A.3. The initial part of the response corresponds to the PLL cycle slipping before it becomes locked in frequency. The right portion of the plot illustrates the step response of the PLL for the case where it remains frequency locked.

![Settling Response of Classical Frequency Synthesizer (Tristate PFD)](image)

**Figure A.3** Simulation plot for classical synthesizer (Tristate PFD).
A.2 Σ-∆ Synthesizer

Figure A.4 illustrates a Σ-∆ frequency synthesizer. In this case, rather than remaining constant, the divide value is dithered according to the output of a Σ-∆ modulator.

Example C++ code to achieve simulation of the Σ-∆ synthesizer is shown below. As in the case of the classical synthesizer, the resulting code is compact and straightforward to implement. Simulated results are shown in Figure A.5.

```c++
#include "com_blocks.h"

main()
{
  double Ts = 1/200e6;
  SdMbitMod sd_mod("1 - 3z^-1 + 3z^-2 - 1z^-3");
  Probe probe("test.tr0",Ts);
  Vco vco("fc + Kv*x","fc,Kv,Ts",1.84e9,30e6,Ts);
  SigGen ref_clk("square",20e6,Ts);
  Reg reg1,reg2;
  And and1;
  Filter rc_filt("1.0","1 + 1/(2*pi*fp)*s","fp,Ts",127.2e3,Ts);
  Filter int_filt("2*pi*fp/10","s","fp,Ts",127.2e3,Ts);
  Edge vco_edge;
  double chp_out,vco_in,in;
  int i;

  in = 90.3;
  for (i = 0; i < 100000; i++)
  {
  
  
  
```
// step desired VCO frequency by .1*Kv at sample 80000
if (i == 80000)
in += .1;

// SD modulator
if (vco_edge.inp(vco.out))
sd_mod.inp(in);

// reference oscillator
ref_clk.inp(0.0);

// PFD
reg1.inp(1.0,vco.out,−1.0,and1.out);
reg2.inp(1.0,ref_clk.out,−1.0,and1.out);
and1.inp(reg1.out,reg2.out);

// Charge Pump
chp_out = (reg2.out−reg1.out)*1.989*PI;

// Loop Filter
rc_filt.inp(chp_out);
int_filt.inp(chp_out);

// VCO and divider
vco_in = rc_filt.out+int_filt.out;
vco.inp(vco_in,sd_mod.out);

probe.inp(sd_mod.out,"sd");
probe.inp(int_filt.out,"int");
probe.inp(chp_out,"chp");
probe.inp(vco_in,"vco");
probe.inp(reg1.out,"reg1");
probe.inp(reg2.out,"reg2");
probe.inp(vco.out,"out");
probe.inp(vco.phase,"phase");
probe.inp(ref_clk.out,"ref");
}
A.3. LINEAR CDR

The CppSim classes also allow straightforward simulation of clock and data recovery (CDR) circuits. Figure A.6 illustrates a general CDR architecture that uses a phase-locked loop to lock the phase and frequency of a VCO to that of the input data. As with frequency synthesizers, a critical component in the design is the phase detector structure that is chosen. Common choices for these detectors are the Hogge topology, which leads to linear CDR dynamics, or the Bang-bang topology, which leads to nonlinear behavior. The Hogge topology is often chosen for systems that have stringent requirements on the transfer function response of the CDR to input jitter from the data sequence. The Bang-bang topology offers superior phase acquisition speed at the expense of having nonlinear dependence on the input jitter.

Figure A.6 A PLL-based clock and data recovery circuit.
In this section, we will examine the simulation of a CDR that has linear dynamics by virtue of using a Hogge phase detector. The Hogge structure is illustrated in Figure A.7.

**Figure A.7** A linear phase detector (Hogge topology).

C++ simulation code for a linear CDR system is given below. As with the synthesizer examples, one can see that the CppSim classes allow straightforward implementation of this system in code.

```cpp
#include "com_blocks.h"

main()
{
    double Ts = 1/15e9;
    Probe probe("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",2.5e9,30e6,Ts);
    SigGen prbs_data("prbs",2.502e9,Ts);
    Reg reg1;
    Latch latch1;
    Xor xor1,xor2;
    Filter int_filt("1","C*s","C,Ts",2e-9,Ts);
    Filter rc_filt("R","1 + 1/(2*pi*fp)*s","R,fp,Ts",1.07e3,40e6,Ts);
    EdgeMeasure vco_period,in_period;
    Rand randg("gauss");
    double chp_out,vco_in,pd_out,in;
    double N_dbC,f_off,noise_var,Kv;
    int i;

    /* VCO noise */
```
\[ N_{dBc} = -100; \quad // -100 \, dBc/Hz \, at \, 1 \, MHz \, offset \]
\[ f_{off} = 1e6; \]
\[ Kv = 30e6; \]
\[ \text{noise\_var} = \text{pow}(10, N_{dBc}/10.0) * \text{pow}(f_{off}/Kv, 2); \]

```plaintext
for \( i = 0; i < 500000; i++ \) 
{
    // Input PRBS data - set jitter to zero
    in = prbs\_data.inp(0.0);

    // Hogge phase detector
    reg1.inp\( \text{in, vco.out} \);
    latch1.inp\( \text{reg1.out, vco.out} \);
    xor1.inp\( \text{in, reg1.out} \);
    xor2.inp\( \text{reg1.out, latch1.out} \);
    pd\_out = xor1.out - xor2.out;

    // Charge Pump
    chp\_out = 150e-6 * pd\_out;

    // Loop filter
    vco\_in = int\_filt.inp(chp\_out) + rc\_filt.inp(chp\_out);
    vco\_in += sqrt(noise\_var/Ts) * randg.inp(); // add VCO noise

    // VCO
    vco.inp(vco\_in);

    // Save signals to file
    probe.inp\( \text{vco\_period.inp(vco.out), "vco\_period"} \);
    probe.inp\( \text{in\_period.inp(prbs\_data.square), "in\_period"} \);
    probe.inp\( \text{vco\_in, "vco"} \);
    probe.inp\( \text{int\_filt.out, "integ"} \);
    probe.inp\( \text{pd\_out, "pd\_out"} \);
}
```

Simulated results generated by the above simulation code are shown in Figure A.8. The plot reveals an exponential decay of the phase error over time (or, equivalently, over VCO cycle number). For reference, the Matlab code used to generate this plot is given below (this code is contained in the CppSim/MatlabCode directory).
x = loadsig('test.tr0');

raw_period_vco = evalsig(x,'vco_period');
raw_period_in = evalsig(x,'in_period');

%phase = extract_phase(raw_period_vco);
[phase,avg_period] = extract_phase(raw_period_vco,raw_period_in);

%phase = phase-1/avg_period;
phase = phase + .5;

% phase = phase(30000:length(phase));
plot(phase,'-k');
grid on;
xlabel('VCO rising edge number');
ylabel('Instantaneous Jitter (U.I.)');
title_str = sprintf('Instantaneous Jitter of VCO in CDR: \n Steady-state RMS jitter = %5.4f mUI',1e3*std(phase(30000:length(phase))));
title(title_str);
axis([-1e3 5e4 -.2 .05])
A.4 Bang-bang CDR

This section investigates the simulation of a CDR that uses a Bang-bang detector rather than a Hogge detector as illustrated in Figure A.9. The code corresponding to this system is shown below, and the simulated phase error produced by the code is shown in Figure A.10. As with the previous systems, the simulation code is seen to be straightforward to implement. The simulated phase error plot is seen to quickly settle to zero phase error in a non-linear manner, which is in contrast to exponential response of the linear CDR.

```
#include "com_blocks.h"

main()
{
    double Ts = 1/15e9;
    Probe probe("test.tr0",Ts);
    Vco vco("fc + Kv*x","fc,Kv,Ts",2.5e9,50e6,Ts);
    SigGen prbs_data("prbs",2.502e9,Ts);
    Reg reg1,reg2,reg3;
    Latch latch1;
```
Xor xor1,xor2;
Filter int_filt("2*pi*40e9","s","Ts",Ts);
EdgeMeasure vco_period,in_period;
Rand randg("gauss");
double chp_out,vco_in,pd_out,in;
double N_dBc,f_off,noise_var,Kv;
int i;

/* VCO noise */
N_dBc = -90; // -90 dBc/Hz at 1 MHz offset
f_off = 1e6;
Kv = 50e6;
noise_var = pow(10,N_dBc/10.0)*pow(f_off/Kv,2);

for (i = 0; i < 300000; i++)
{
    // Input PRBS data
    in = prbs_data.inp(0.0); // set jitter to zero

    // Bang-bang phase detector
    reg1.inp(in,vco.out);
    reg2.inp(reg1.out,vco.out);
    reg3.inp(in,-vco.out);
    latch1.inp(reg3.out,-vco.out);
    xor1.inp(reg1.out,latch1.out);
    xor2.inp(reg2.out,latch1.out);
    pd_out = xor1.out-xor2.out;

    // Charge Pump
    chp_out = 1e-6*pd_out;

    // Loop filter
    vco_in = int_filt.inp(chp_out) + chp_out*125.0e3;
    vco_in += sqrt(noise_var/Ts)*randg.inp(); // add VCO noise

    // VCO
    vco.inp(vco_in);

    // Save signals to file
    probe.inp(vco_period.inp(vco.out),"vco_period");
A.4. BANG-BANG CDR

```c
probe.inp(in_period.inp(prbs_data.square),"in_period");
probe.inp(vco_in,"vco");
probe.inp(int_filt.out,"integ");
probe.inp(pd_out,"pd_out");
}
```
Instantaneous Jitter of VCO in CDR:
Steady-state RMS jitter = 3.4598 mUI

Figure A.10 Simulation plot for CDR (bang-bang phase detector).
Appendix B

Hspice Toolbox for Matlab

Documentation and code written by Michael H. Perrott
Copyright © 1999 by Silicon Laboratories, Inc.

The Hspice toolbox for Matlab is a collection of Matlab routines that allow you to manipulate and view signals generated by Hspice simulations. The primary routine is a mex program called loadsig.mexsol that reads binary output files generated by Hspice transient, DC, and AC sweeps into Matlab. The remaining routines are used to extract particular signals and view them.

We will begin this document by explaining how to include the Hspice toolbox in your Matlab session. A list of each of the current functions will then be presented. Finally, we will provide examples of using these routines to view and postprocess signals from Hspice output files.

B.1 Setup

To use the Hspice toolbox, simply place the included files into a directory of your choice, and then add that directory to your Matlab path. For example, inclusion of the path '/home/username/CppSim/HspiceToolbox' in Matlab can be done by adding the line

```
addpath('/home/username/Cppsim/HspiceToolbox')
```

to the file startup.m located in your home directory. In addition, you can specify the plot background to be black (similar to the look of Awaves) by adding another line to startup.m:

```
colordef none;
```

111
Once you’ve made the above changes to `startup.m`, start Matlab as you normally would. Matlab will automatically read `startup.m` from your home directory and execute its commands.

## B.2 List of Functions

The following functions are currently included in the Hspice toolbox:

- **x = loadsig('hspice_output_filename');**
  
  - Returns a Matlab structure into variable `x` that includes all of the signals that are present in the Hspice binary output file, `hspice_output_filename`.

- **lssig(x)**
  
  - Lists all of the Hspice signal names present in the structure `x`.

- **y = evalsig(x,'nodename');**
  
  - Pulls out the signal `nodename` from the structure `x` and places into variable `y`. The string `nodename` can be an expression involving several Hspice signals. If you only performed one sweep in the simulation (as is common), then `y` will contain one column. If you performed several sweeps, `y` will contain several columns that correspond to the data for each sweep. If you have set the global Matlab variable `sweep` to a nonzero number, however, then `y` will contain only one column corresponding to the value of `sweep`. If `sweep` equals zero, all the sweep columns are included in `y`.

- **plotsig(x,'plot_expression','optional_plotspec')**
  
  - Plots signals from the structure `x` according to the expression given in `plot_expression`. The string `optional_plotspec` is used to create logscale plots; it can be specified as `logx`, `logy`, or `logxy`. The string `plot_expression` specifies the nodenames, and corresponding mathematical operations, that you would like to view. In this expression, commas delimit curves to be overlayed and semicolons delimit separate subplots on the same figure. All numeric node names should be prepended by `@` to distinguish them from constants. Some examples of using `plotsig` are:
B.3. EXAMPLES

* `plotsig(x,'v1,v2;v3')`: overlays v1 and v2 on the same subplot, and plots v3 on a separate subplot.
* `plotsig(x,'(v1+v2)^2; log(abs(v3))')`: plots the listed expressions on separate subplots.
* `plotsig(x,'db(v1); ph(v1)', 'logx')`: plots the magnitude (in dB) and phase (in degrees) of v1 on a semilogx axis.
* `plotsig(x,'v1+@2+3')`: plots the addition of node v1, node 2, and the constant 3.
* `plotsig(x,'integ(TIME,v1); avg(TIME,v2)')`: plots the integral of v1 and average of v2 on separate subplots.

- **xlima**
  - Sets the x-limits of all subplots in a figure. Three options are possible:
    * `xlima`: sets all subplots to the same x-axis as the last subplot that was zoomed into,
    * `xlima([xs xe])`: sets all subplots to the x-axis limits specified,
    * `xlima('auto')`: resets all subplots back to autoscaling.
  - Note: `ylima` and `xylim` functions are also provided. See comments in `ylima.m` and `xylim` for proper usage.

- **eyesig(x,period,start_off,'nodename')**
  - Creates an eye diagram for nodename contained in x with the specified period. All data samples prior to `start_off` are ignored when creating the diagram (useful for removing the influence of transient effects from the eye diagram). The string `nodename` can be an expression involving several variables.

B.3 Examples

**Viewing Signals**

Use the Matlab command `cd` to go to a directory containing a binary transient, DC, or AC sweep file generated from Hspice. We will assume a filename of `test.tr0`, and now list a series of Matlab commands that will be used to display nodes q and qB in that file.
Doing Postprocessing in Matlab

Use the Matlab command `cd` to go to a directory containing a binary transient, DC, or AC sweep file generated from Hspice. We will assume a filename of `test.tr0`, and now list a series of Matlab commands that will be used to postprocess nodes q and qb in that file.

- `x = loadsig('test.tr0');`  \(\%\) loads Hspice signals into x
- `lssig(x)`  \(\%\) verify that nodes q and qb are present
- `t = evalsig(x,'TIME');`  \(\%\) loads time samples into Matlab variable t
- `q = evalsig(x,'q');`  \(\%\) loads signal q into Matlab variable q
- `qb = evalsig(x,'qb');`  \(\%\) loads signal qb into Matlab variable qb
- `qdiff = q-qb;`  \(\%\) perform expressions in Matlab
- `plot(t,q,t,qb,t,qdiff)`  \(\%\) plot variables using Matlab plot command