Simulation and Analysis Using OCEAN

Version 5.1.41

Lecture Manual **October 10, 2005**

cadence

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Cadence Trademarks

Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address above or call 800.862.4522.

1st Silicon Success®
Allegro®
Assura™
BuildGates®
Cadence® (brand and logo)
CeltIC®
ClockStorm®
CoBALT™
Conformal®
Connections®
Design Foundry®
Diva®
Dracula®
Encounter™
Fire & Ice®
First Encounter®
FormalCheck®
HDL-ICE®
Incisive®
IP Gallery™
Nano Encounter™
NanoRoute™
NC-Verilog®
OpenBook® online documentation library
Orcad®
Orcad Capture®
Orcad Layout®
PacifIC™
Palladium™
Pearl®
PowerSuite™
PSpice®
QPlace®
Quest®
SeismIC™
SignalStorm®
Silicon Design Chain™
Silicon Ensemble®
SoC Encounter™
SourceLink® online customer support
Spectre®
TtME®
UltraSim®
Verifault-XL®
Verilog®
Virtuoso®
VoltageStorm®

Other Trademarks

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

All other trademarks are the property of their respective holders.

Confidentiality Notice

Table of Contents
Simulation and Analysis Using OCEAN

Module 1 About This Course

Module 2 Beginning to Use SKILL

Module 5 Data Access and Plotting

Module 6 Developing a SKILL Function

Module 9 Advanced Analyses with OCEAN

Module 10 SpectreRF in OCEAN

Module 11 OCEAN and Mixed-Signal Simulation

Module 12 Waveform Data Objects and Custom Calculator Functions

About This Course

Module 1

October 5, 2005

Course Objectives

In this course you will

- Use OCEAN to control simulations and analyze results
- Master SKILL syntax, loop constructs and conditional statements
- Build and manipulate lists
- Define, develop, and debug SKILL functions and programs
- Read and write data to and from UNIX text files
- Develop custom calculator functions for waveform database objects

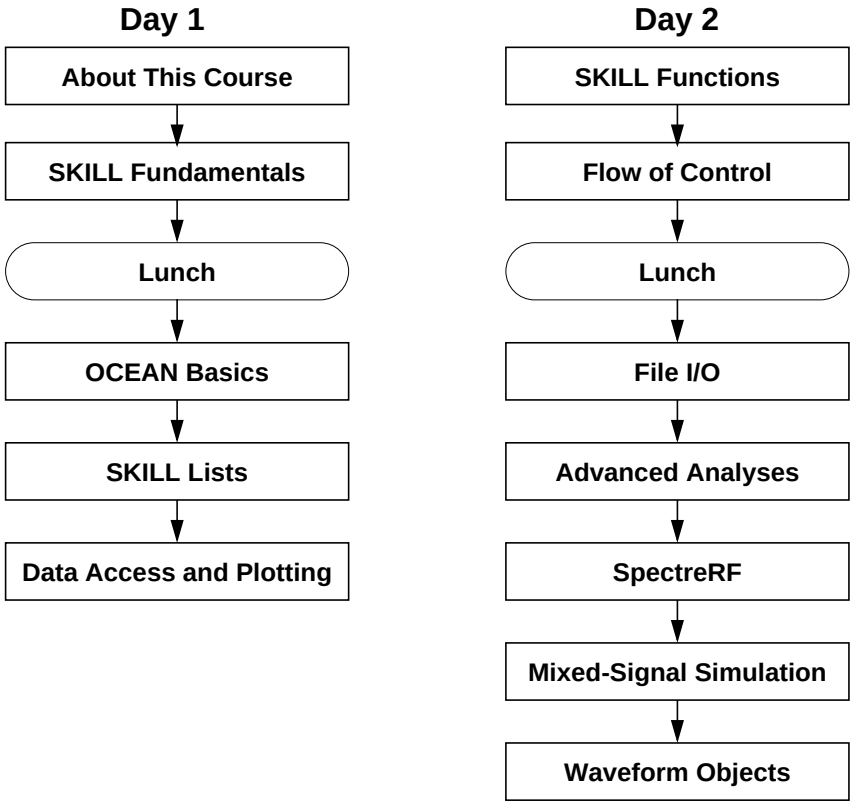
Terms and Definitions

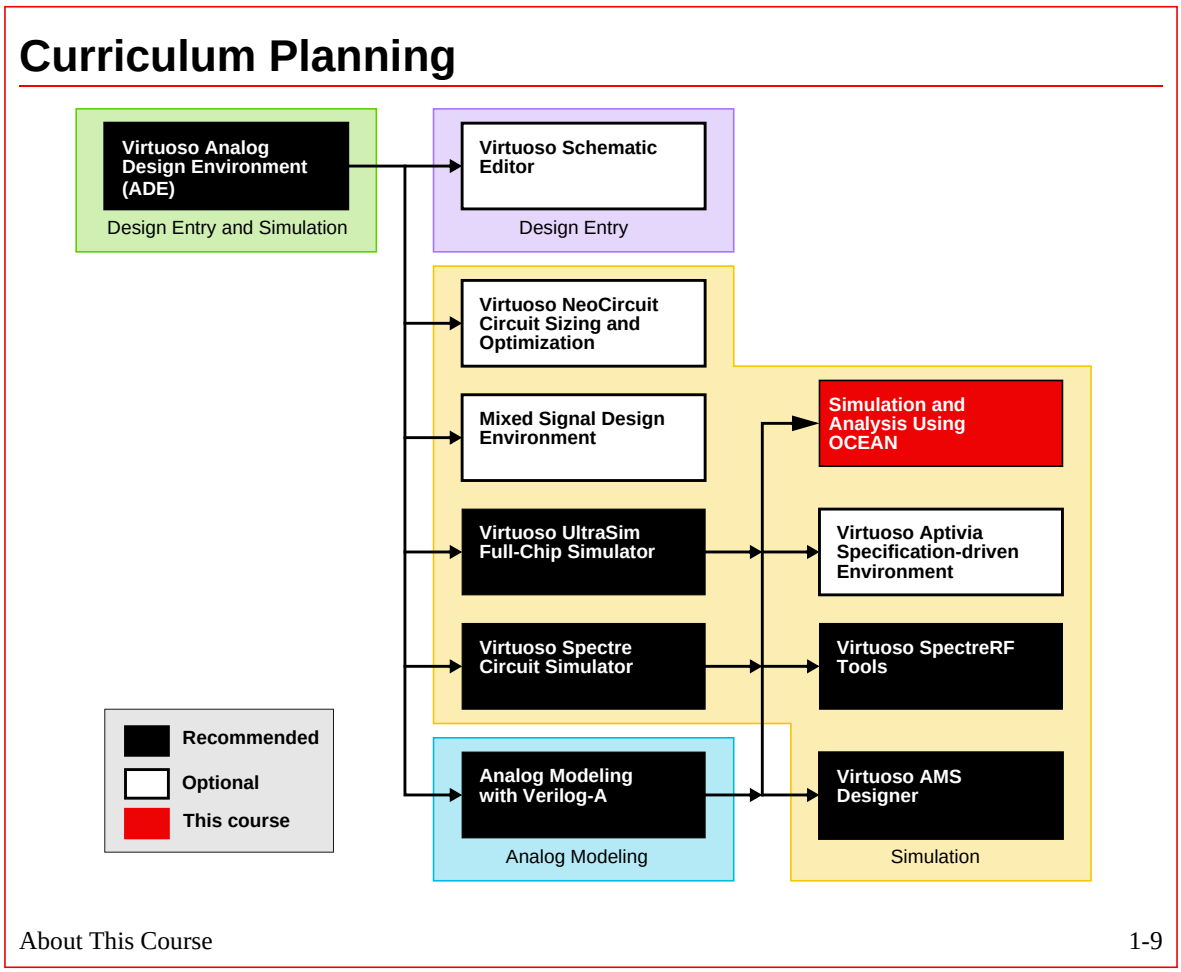
SKILL	SKILL is a LISP-like language with C language IO structures. SKILL extends and enhances the functionality of Cadence tools.
OCEAN	Open Command Environment for ANalysis. API for analog simulations within DFII.

What Is OCEAN?

- Open Command Environment for ANalysis
- SKILL API (Application Programming Interface) for controlling and running Analog/RF/Mixed Signal simulations, and analyzing the results.
- Designed to be easy to use by designers with little knowledge of programming.
- Easily scripted to automate manual tasks in Analog Design Environment.

Course Agenda





Cadence offers courses that are closely related to this one. You can use the course map to plan your future curriculum.

For more information about Cadence courses:

1. Point your web browser to cadence.com.
2. Click **Education**.
3. Click the **Course catalog** link near the top of the center column.
4. Click a Cadence technology platform (such as **Custom IC Design**).
5. Click a course name.

The browser displays a course description and gives you an opportunity to enroll.

Getting Information

There are three ways to get information about SKILL and OCEAN:

- From the CDSDoc online documentation system
- From SourceLink® online customer support
- From the Finder

CDSDoc Online Documentation

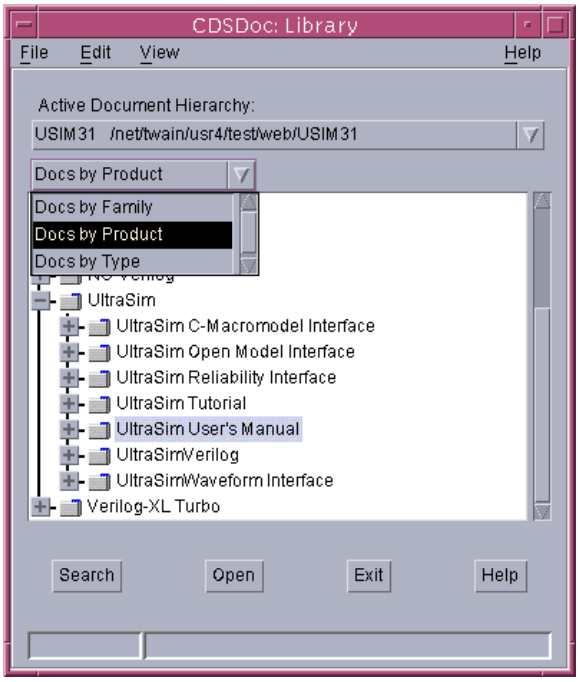
CDSDoc is the Cadence online product documentation system.

Documentation for each product installs automatically when you install the product. Documents are available in both HTML and PDF format.

The Library window lets you access documents by product family, product name, or type of document.

You can access CDSDoc from:

- The graphical user interface, by using the Help menu in windows and the Help button on forms
- The command line
- SourceLink online customer support (if you have a license agreement)



About This Course 1-13

To access CDSDoc from a Cadence software application or the Command Interpreter Window (CIW), click **Help**. The document page is loaded into your browser. After the document page opens, click the **Library** button to open the CDSDoc Library window.

To access CDSDoc from the UNIX command line, enter *cdsdoc &* in a terminal window. When the Library window appears, navigate to the manual you want, then click **Open**. The manual appears in your browser.

To access CDSDoc from the Windows environment, navigate to the *<release>\tools\bin* directory and double-click **cdsdoc.exe**, or select **Start—Programs—Cadence <release>—Online Documentation**. When the Library window appears, navigate to the manual you want, then click **Open**. The manual appears in your browser.

The OCEAN and SKILL Documentation Set

The following manuals cover OCEAN and the SKILL language:

- OCEAN Reference
- SKILL Language Reference Manual
- SKILL User Guide

These manuals cover SKILL application programming interfaces to the Cadence® Design Framework II environment:

- Cadence User Interface SKILL Functions Reference
- Cadence Design Framework II SKILL Functions Reference

This manual covers the SKILL application programming interface to design tools:

- Virtuoso Analog Design Environment SKILL Language Reference

This online resource provides help for OCEAN:

- OCEAN HTML help:

  <install_dir>/tools/dfII/samples/artist/OCEAN/html/index.html

Searching Cadence Documents

You can:

- Search all Cadence documents
- Search selected documents
- Search one document
- Narrow your search with a larger variety of operators
- Search with a combination of operators
- Search for special characters

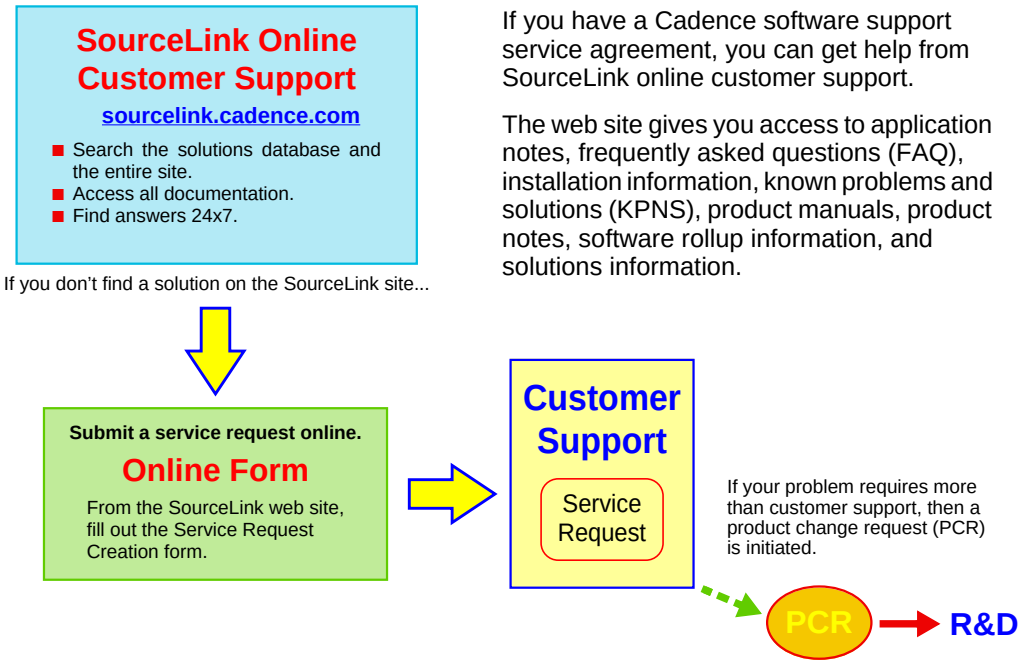
Search Operators

Type this:	Example:	Result:
myword	place	Finds place, placing, places, placed.
all words in a phrase	layer colors	Finds the phrase layer colors. Does not find layer and colors
"myword"	"place"	Finds only place
? for 1 character * for many characters	??Create*	Finds *syCreatePin*, *hiCreateForm*, *leCreateCell*, and other functions that begin with two characters followed by "Create" and another string

Search Operators (continued)

Type this:	Example:	Result:
word AND word	printer AND CalComp	Finds only documents with both printer and CalComp
word OR word	printer OR plotter	Finds either the words printer or plotter
word NOT word	plotter NOT CalComp	Finds all documents with the word plotter that do not have the word CalComp
word <NEAR> word	place <NEAR> route	Finds place, placing, places near route, routing, routes
<CASE> word	<CASE> SUBMIT	Finds SUBMIT but not submit
word "and" word	place "and" route	Finds the phrase place and route
word, word, word	command, block	Finds documents with command and block, ranking those with both words higher than others

SourceLink Online Customer Support



If you have a Cadence software support service agreement, you can get help from SourceLink online customer support.

The web site gives you access to application notes, frequently asked questions (FAQ), installation information, known problems and solutions (KPNS), product manuals, product notes, software rollup information, and solutions information.

To view information in SourceLink:

1. Point your web browser to sourcelink.cadence.com.
2. Log in.
3. Enter search criteria.

   You can search by product, release, document type, or keyword. You can also browse by product, release, or document type.

The Finder

The Finder provides quick online help for the core SKILL language functions. It displays the syntax and a brief description for each SKILL function.

Example description of the *strcat* function:

```
strcat( t_string1 [t_string2 t_string3 ...] )  => t_result
Takes input strings and concatenates them.
```

You can access the Finder in three ways:

- In an xterm window, enter

  ```
  cdsFinder &
  ```

- In the SKILL Development window, click the Finder button.
- In the CIW, enter

  ```
  startFinder()
  ```

The Finder contains the same information as the SKILL Quick Reference Guide. You can save the online documentation to a file.

The Finder database varies according to the products on your system. Each separate product loads its own language information in the Cadence hierarchy that the Finder reads.

You can add your own functions locally for quick reference because the Finder can display any information that is properly formatted and located.

To use the Finder, follow these steps:

1. Specify a name pattern in the Search String pane.
2. Click the Search button.
3. Select a function in the Matches window pane.

The Finder displays the abbreviated documentation for the selected function.

Lab Exercises

There are two types of labs in this course—operational and programming.

- Operational exercises
  - Enter an example OCEAN/SKILL expression into the Command Interpreter and observe the results.
    Modify the example.
  - Examine and run source code.

  Solutions to questions are usually on the next page.
- Programming exercises

Section	Description
Requirements	Describe the functionality of the SKILL function you write.
Recommendations	Outline an approach to solving the problem.
Testing your solution	Run some tests that your SKILL function must pass.
Sample solutions	Study a solution that follows the recommendations.

Lab Overview

There are no labs for this module.

Module Summary

In this module, you

- Reviewed the course objectives and agenda
- Learned what OCEAN is
- Reviewed the list of related courses
- Learned what documentation is available for SKILL and OCEAN
- Learned how to access the documentation through CDSDoc, SourceLink, and the Finder
- Examined the format of the lab exercises

cadence®

Beginning to Use SKILL

Module 2

October 5, 2005

Module Objectives

- Start the Cadence® Design Framework II environment.
- Examine the Command Interpreter Window (CIW).
- Narrate the role of the SKILL Evaluator.
- Examine the *CDS.log* file.
- Summarize SKILL syntax.
- Display data in the CIW output pane.
- Get the most out of SKILL error messages.

Terms and Definitions

CIW	Command Interpreter Window.
SKILL Evaluator	The SKILL Evaluator executes SKILL programs within the Design Framework II environment. It compiles the program's source code before running the program.
Compiler	A compiler translates the source code into the machine language of a target machine. The compiler does not execute the program. The target machine can itself be a virtual machine.
Evaluation	Evaluation is the process whereby the SKILL Evaluator determines the value of a SKILL expression.
SKILL expression	The basic unit of source code. An invocation of a SKILL function, often by means of an operator supplying required parameters.
SKILL function	A SKILL function is a named, parameterizable body of one or more SKILL **expressions**. You can invoke any SKILL function from the CIW by using its name and providing appropriate parameters.
SKILL procedure	This term is used interchangeably with SKILL **function**.

What Is SKILL?

SKILL is a high-level, interactive programming language.

SKILL is the command language of the Design Framework II environment.

Whenever you use forms, menus, and bindkeys, the Design Framework II software calls SKILL functions to complete your task.

You can enter SKILL functions directly into the CIW input pane to bypass the normal user interface.

Beginning to Use SKILL 2-5

SKILL was developed from the language LISP (LISt Processing language). To learn more about the SKILL interpreter core language you can consult literature pertaining to LISP or SCHEME (a newer implementation of language very similar to LISP).

The IO used in SKILL is that of C. Those of you familiar with Fortran will also see a strong resemblance.

What Can SKILL Functions Do?

The SKILL programming language acts as an extension to the Design Framework II environment.

Some SKILL functions control the Design Framework II environment or perform tasks in design tools. For example, SKILL functions can:

- Set up various analyses.
- Set a design variable.
- Run a sequence of simulations.

Other SKILL functions compute or retrieve data from the Design Framework II environment or from designs. For example, SKILL functions can:

- Retrieve the list of analyses run in a simulation.
- Calculate an expression based on the results of simulation of several nodes in the design.

The Return Value of a SKILL Function

All SKILL functions compute a data value known as the return value of the function. You can

- Assign the return value to a SKILL variable.
- Pass the return value to another SKILL function.

Any SKILL data can become a return value.

Starting the Design Framework II Environment

You can choose from two types of sessions:

- Graphic
- Nongraphic

You can replay a Design Framework II session.

Session	UNIX command line	Notes
Graphic	<*cds*> &	Use an ampersand (&)
Nongraphic	<*cds*> -nograph	Do not use an ampersand (&)
Replay a session	<*cds*> -replay *~/OldCDS.log &*	

Note: In the table above <*cds*> represents the name of your executable.

Graphic Sessions

In a graphic session, the Command Interpreter Window (CIW) is the first window you see.

Nongraphic Sessions

A nongraphic session is useful when you are using an ASCII terminal or modem or do not require graphics. For example, without graphics you still can open designs into virtual memory to query and update them.

After you launch a nongraphic session in an xterm window, the xterm window expects you to enter SKILL expressions.

During a nongraphic session, the Cadence Design Framework II environment suppresses any graphic output. It does not create any windows.

Replaying Sessions

When replaying a session, the Design Framework II environment evaluates all the SKILL expressions contained in the *-replay* transcript file.

Initializing the Design Framework II Environment

During startup, the Design Framework II environment searches the following directories for a *.cdsinit* file:

- <install_dir>/tools/dfII/*local* 3
- The current directory "." 1
- The home directory 2

When the Design Framework II environment finds a *.cdsinit* file, it stops searching and loads the *.cdsinit* file.

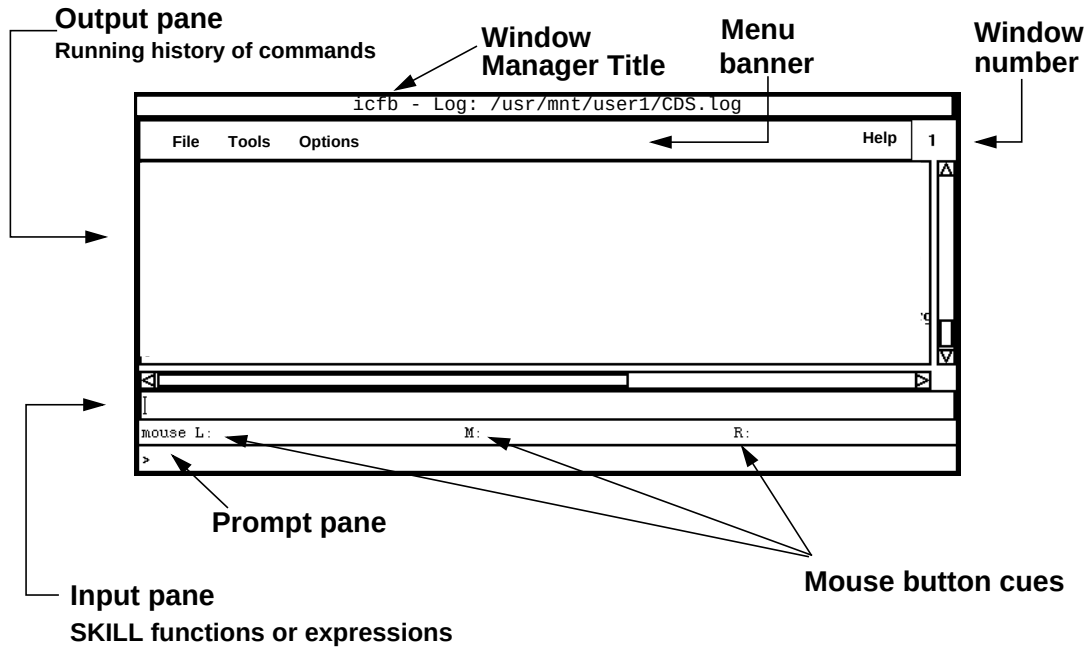
Typically, you use the *.cdsinit* file to define application bindkeys and load customer-specific SKILL utilities.

The site administrator has three ways of controlling the user customization.

Policy	Customization Strategy
The site administrator does all customization.	The <install_dir>/tools/dfII/*local*/*.cdsinit* contains all customization commands. There are no *./.cdsinit* or *~/.cdsinit* files involved.
The administrator does the site customization. The user can add further customization.	The <install_dir>/tools/dfII/*local*/*.cdsinit* file contains a command to load the *./.cdsinit* or *~/.cdsinit* files.
The user does all the customization.	The <install_dir>/tools/dfII/*local*/*.cdsinit* file does not exist. All customization is handled by either the *./.cdsinit* or *~/.cdsinit* files.

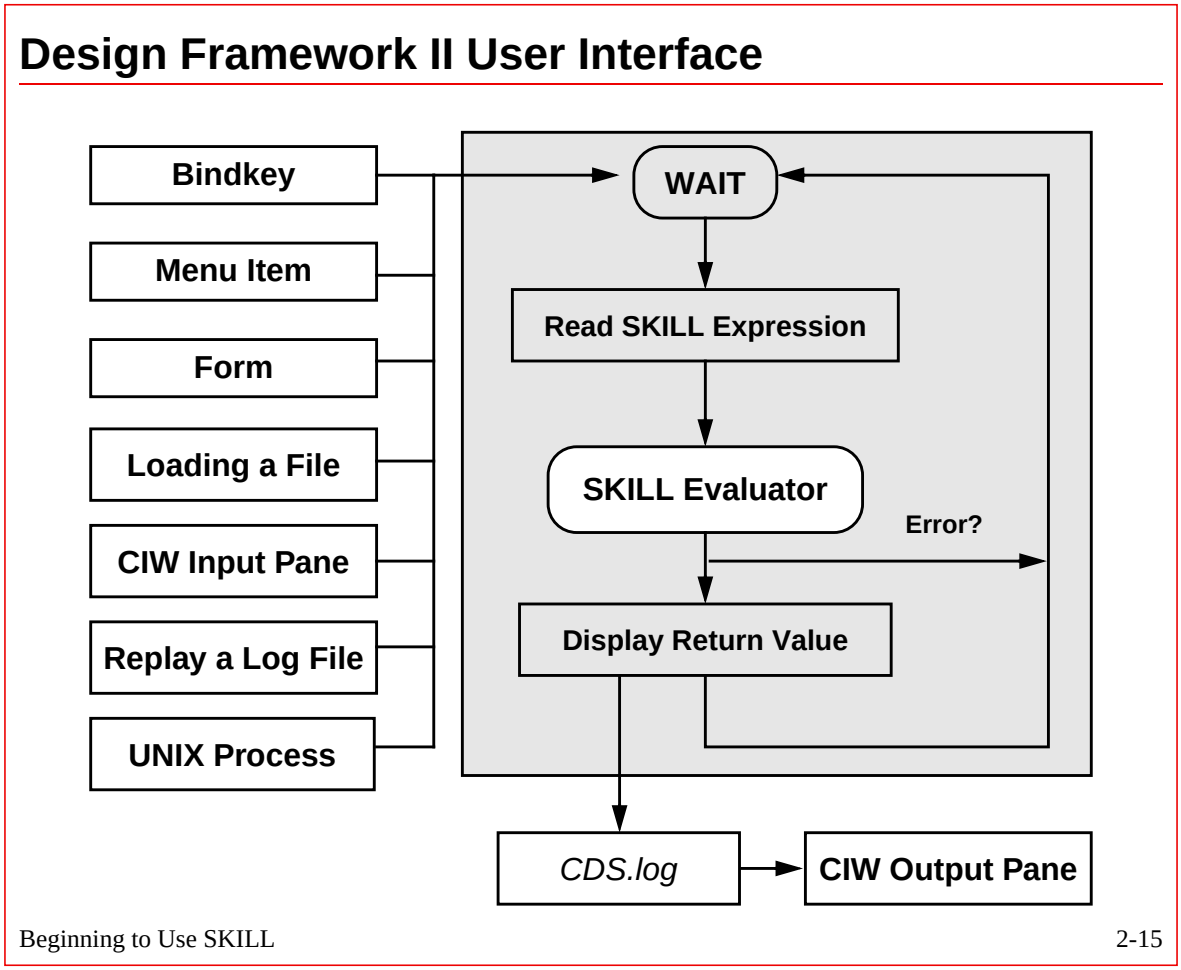
Consult the <install_dir>/tools/dfII/*cdsuser*/*.cdsinit* file for sample user customizations.

Command Interpreter Window



In a graphic session, the Command Interpreter Window (CIW) is the first window you see.

CIW Area	Description
Window manager title	The title indicates the name of the transcript log file
Window menu banner	The window banner contains several pull-down menus and a HELP button.
Output pane	This pane displays information from the session log file. You can control the kind of information the output pane displays.
Input pane	You can enter one or more SKILL expressions on this single line. When you type a carriage return in this pane, your input line is sent to the SKILL Evaluator.
Mouse button cues	When you enter data, this pane indicates the action available to you through the three mouse buttons. Other keys can also have a special meaning at this time.
Prompt pane	The pane displays the command prompt. When you enter data the prompt indicates what you need to do next.



Parsing and Compiling

For each expression, the SKILL Evaluator parses it, compiles it, and then executes the compiled code. SKILL makes safety checks during each phase.

Bindkeys and Menu Items

Whenever you press a bindkey, choose a menu item, or click OK/Apply on a form, the Design Framework II environment activates a SKILL function call to complete your task.

Loading SKILL Source Code

You can store SKILL code in a text file. The *load* or *loadi* function evaluates each SKILL expression in the file.

Replaying a Session File

You can replay a session file. The Design Framework II environment successively sends each SKILL expression in the session file to the SKILL Evaluator. Only the input lines (prefixed by "\i") and the accelerated input lines (prefixed by "\a") are processed.

Sending a SKILL Expression from a UNIX Process

Using SKILL code, you can spawn a UNIX® process that can send a SKILL expression to the SKILL Evaluator.

The *CDS.log* File

The Design Framework II software transcribes the session in a file called *~/CDS.log*.

The log file adds a two-character tag that identifies the line.

The following text illustrates an example transcript file:

```
\p 1>
\i x = 0
\t 0
\p 1>
\i TrBump()
\o Old value: 0 New Value: 1
\t 1
\p 1>
\a TrBump()
\o Old value: 1 New Value: 2
\r 2
\i TrBump()
\o Old value: 2 New Value: 3
\t 3
```

```
prompt
user type-in SKILL expression
SKILL expression's return value
prompt
user type-in SKILL expression
SKILL expression output
SKILL expression's return value
Prompt
Bindkey SKILL expression
SKILL expression output
SKILL expression's return value
user type-in SKILL expression
SKILL expression output
SKILL expression's return value
```


The definition of the *TrBump* function is:

```
procedure( TrBump( )
  printf( "Old value: %d New Value: %d\n" x ++x ) x )
```

The following SKILL expression defines the *<Key>F7* bindkey for the CIW:

```
hiSetBindKey( "Command Interpreter" "<Key>F7" "TrBump()" )
```

By default mouse drag events are not logged. You can turn on logging by entering in the CIW:

```
hiLogDragEvents( t )
```

The *CDS.log* File Code

Tag	Description
\p	The prompt displayed in the CIW. This identifies the boundary between two user-level commands.
\i	A SKILL expression that the user typed into the CIW.
\o	The output, if any, that the SKILL expression generates.
\w	The warnings, if any, that the SKILL expression generates.
\e	The error, if any, that the SKILL expression caused.
\t	The return value for a SKILL expression that the user typed into the CIW.
\a	The SKILL expression activated by a bindkey or menu item.
\r	The return value for a SKILL expression activated by a bindkey or menu item.

When you replay a log file the replay function interprets each of these log file codes and passes those that represent input to the SKILL interpreter.

Setting the Log Filter

You can control the kinds of log file data that the CIW output pane displays.

You can set the seven toggle options of the log filter in several ways.

- Use the *hiSetFilterOptions* function in your *.cdsinit* file. For example, the following line sets up the most unrestrictive filter.

```
hiSetFilterOptions( t t t t t t t )
```

  hiSetFilterOptions argument positions: (1) inputMenuCommands, (2) inputPrompts, (3) outputProgramResults, (4) outputMenuCommands, (5) outputUser, (6) messageErrors, (7) messageWarnings

- From the CIW, use the **Options—Log Filter** command to display the Set Log File Display Filter form.

Log File Category	Toggle Options		
Show Input	menu commands	prompts	
Show Output	user	menu commands	program results
Show Messages	errors	warning	

The arguments to the *hiSetFilterOptions* function correspond to the form as shown. Note the wording on the form.

Argument	Category	Toggle Option	Specific Meaning
a	Show input	Menu commands	SKILL expressions from menu commands, bindkeys, and your form interactions
p	Show input	Prompts	
o	Show output	Program results	*printf* and *println* output
r	Show output	Menu commands	Return results from bindkeys and menu commands
c	Show output	User	Return results from user type-in
e	Show messages	Errors	
w	Show messages	Warnings	

SKILL Syntax Summary

Syntax Category		Example
Comments		; remainder of the line /* several lines */
Data	integer floating point text string list boolean	5 5.3 "this is text" (1 "two" 3 4) t ;;; true nil ;;; false
Variables		line_Count1
	assignment retrieval	x = 5 x
Function call		strcat("Good" " day") (strcat "Good" " day")
Operators		4 + 5 * 6 plus(4 times(5 6))

Syntax Category	Notes
Comments	Do not use a semicolon (;) to separate expressions on a line. You will comment out the remainder of the line! Use /* .... */ to comment out one or more lines or to make a comment within a line. For example, *1+/*add*/2*.
Data	Includes integer, floating-point, text string, list, and boolean data.
Variables	SKILL variables are case sensitive. SKILL creates a variable automatically when it first encounters it during a session. When SKILL creates a variable, it gives the variable a special value to indicate you need to initialize the variable. SKILL generates an error if you access an uninitialized variable.
Function	SKILL function names are case sensitive. SKILL allows you to specify a function call in two ways. You can put multiple function calls on a single line. Conversely, you can span a function call across multiple lines. Separate arguments with whitespace. => designates the return value of the SKILL function call.
Operators	SKILL parser rewrites operator expressions as function calls. Using operators does not affect execution speed.

Data

Each SKILL data type has an input syntax and a print representation.

- You use the input syntax to specify data as an argument or to assign a value to a variable.
- SKILL uses the print representation as the default format when displaying data, unless you specify another format.

An argument to a SKILL function usually must be a specific type. SKILL documentation designates the expected type with a single or double character prefix preceding the variable name. The letter *g* designates an unrestricted type.

The following table summarizes several common data types.

Data Type	Input Syntax	Print Representation	Type Character	Example Variable
integer	*5*	*5*	*x*	*x_count*
floating point	*5.3*	*5.3*	*f*	*f_width*
text string	*"this is text"*	*"this is text"*	*t*	*t_msg*
list	*'(1 "two" 3 4)*	*(1 "two" 3 4)*	*l*	*l_items*

When using the SKILL documentation to look up function details you see the type character used as the start of the arguments. This tells you the variable type that the argument expects.

To determine the *type* of a variable you use the type function.

The *type* function categorizes the data type of its single argument. The return value designates the data type of the argument.

Examples:

```
type( 4 ) => fixnum /* an integer */
type( 5.3 ) => flonum /* a floating point number */
type( "mary had a little lamb" ) => string /* a string */
```

Variables

You do not need to declare variables in SKILL. The SKILL Evaluator creates a variable the first time you use it.

Variable names can contain

- Alphanumeric characters
- Underscores (_)
- Question marks

The first character of a variable cannot be a digit or a question mark.

Use the assignment operator to store a value in a variable. Enter the variable name to retrieve its value.

This example uses the *type* function to verify the data type of the current value of the variable.

```
lineCount = 4           => 4
lineCount               => 4
type( lineCount )       => fixnum
lineCount = "abc"       => "abc"
lineCount               => "abc"
type( lineCount )       => string
```


Variables

SKILL allows both global and local variables. In Module 6, see *Grouping Expressions with Local Variables*.

SKILL Symbols

SKILL uses a data type called *symbol* to represent both variables and functions. A SKILL symbol is a composite data structure that can simultaneously and independently hold the following:

- Data value. For example *x = 4* stores the value *4* in the symbol *x*.
- Function definition. For example, *procedure(x(a b) a+b)* associates a function definition with the symbol *x*. The function takes two arguments and returns their sum.
- Property list. For example, *x.raiseTime = .5* stores the name-value pair *raiseTime .5* on the property list for the symbol *x*.

You can use symbols as tags to represent one of several values. For example, *strength = 'weak* assigns the symbol as a value to the variable *strength*.

Function Calls

Function names are case sensitive.

SKILL syntax accepts function calls in three ways:

- State the function name first, followed by the arguments in a pair of matching parentheses. No spaces are allowed between the function name and the left parenthesis.

```
strcat( "mary" " had" " a" " little" " lamb" )
  => "mary had a little lamb"
```

- Alternatively, you can place the left parenthesis to the left of the function name.

```
( strcat "mary" " had" " a" " little" " lamb" )
  => "mary had a little lamb"
```

- For SKILL function calls that are not subexpressions, you can omit the outermost levels of parentheses.

```
strcat "mary" " had" " a" " little" " lamb"
  => "mary had a little lamb"
```

Use white space to separate function arguments.

You can use all three syntax forms together. However, it is best to use one syntax form consistently, as this will lead to improved readability of the code, and fewer errors of white space.

Multiple Lines

A literal text string cannot span multiple lines.

Function calls

- You can span multiple lines in either the CIW or a source code file.

```
strcat(
  "mary" " had" " a"
  " little" " lamb" ) => "mary had a little lamb"
```

- Several function calls can be on a single line. Use spaces to separate them.

```
gd = strcat("Good" " day" ) println( gd )
```

SKILL implicitly combines several SKILL expressions on the same line into a single SKILL expression.

- The composite SKILL expression returns the return value of the last SKILL expression.
- All preceding return values are ignored.

You can span multiple lines with a single command. You need to be careful with this ability. When you send a segment of your command to the SKILL compiler and it can be interpreted as a statement, the compiler treats it as one.

Example:

```
a = 2
a + 2 * (3 + a) => 12
```

however,

```
a + 2
* (3 + 2) =>
4
* error * - wrong number of arguments: mult expects 2 arguments
```

A text string can span multiple lines by including a \ before the return

Example:

```
myString = "this string spans \
two lines using a backslash at the end of the first line"

"this string spans two lines using a backslash at the end of the first line"
```

Understanding Function Arguments

Study the online documentation or the Cadence Finder to determine the specifics about the arguments for SKILL functions.

The documentation for each argument tells you

- The expected data type of the argument
- Whether the argument is required, optional, or a keyword argument

A single SKILL function can have all three kinds of arguments. But the majority of SKILL functions have the following type of arguments:

- Required arguments with no optional arguments
- Keyword arguments with no required and no optional arguments

SKILL displays an error message when you pass arguments incorrectly to a function.

To see the list of arguments for a given function use the *arglist* function.

```
arglist( 'printf )
(t_string \@optional g_general "tg")
```


Required Arguments

You must provide each required argument in the prescribed order when you call the function.

Optional Arguments

You do not have to provide the optional arguments. Each optional argument has a default value. If you provide an optional argument, you must provide all the preceding optional arguments in order.

```
view( t_file [g_boxSpec][g_title][g_autoUpdate ][l_iconPosition] )
```

Keyword Arguments

When you provide a keyword argument you must preface it with the name of the formal argument. You can provide keyword arguments in any order.

```
geOpen( ?window w_windowId ?lib t_lib ?cell t_cell
?view t_view ?viewType t_viewType ?mode t_mode )
=> w_windowId/ nil
```

Operators

SKILL provides operators that simplify writing expressions. Compare the following two equivalent SKILL expressions.

```
( 3**2 + 4**2 ) **.5 => 5.0
expt( plus( expt( 3 2 ) expt( 4 2 )) .5 ) => 5.0
```

Use a single pair of parentheses to control the order of evaluation as this nongraphic session transcript shows.

```
> 3+4*5
23
> (3+4)*5
35
>  x=5*6
30
> x
30
> (x=5)*6
30
> x
5
```

However, when you use extra parentheses, they cause an error.

```
((3+4))*5
*Error* eval: not a function - (3 + 4)
```


Operator Precedence

In general, evaluation proceeds left to right. Operators are ranked according to their relative precedence. The precedence of the operators in a SKILL expression determine the order of evaluation of the subexpressions.

Each operator corresponds to a SKILL function.

Operator	Function	Use
++a a++	preincrement postincrement	Arithmetic
*a**b*	expt	Arithmetic
*a*b a/b*	times quotient	Arithmetic
a+b a-b	plus difference	Arithmetic
a==b a!=b	equal nequal	Tests for equality and inequality.
a=b	setq	Assignment

For more information, check the online documentation and search for preincrement.

Tracing Operator Evaluation

To observe evaluation, turn on SKILL tracing before executing an expression to observe evaluation. The arrow (-->) indicates return value.

Notice that the trace output refers to the function of the operator.

```
> tracef(t)
t
> (3+4)*5
|(3 + 4)
|plus --> 7
|(7 * 5)
|times --> 35
35
>
```

To turn off tracing you use:

```
untrace()
```

Beginning to Use SKILL 2-37

Tracing Operator Evaluation

The trace function shows the order and intermediate results of operator evaluation.

SKILL Output	Explanation
`>tracef(t`	The user executes the trace function to turn on tracing.
`t`	The SKILL evaluator acknowledges successful completion of the function.
`(3+4)*5`	The user enters an expression for evaluation.
`|(3 + 4)`	The SKILL evaluator begins evaluation starting from left to right.
`|plus --> 7`	The SKILL evaluator executes the "plus" function resulting in 7.
`|(7 * 5)`	The 7 is returned to the original expression replacing (3 + 4)
`|times --> 35`	The "times" function is executed resulting in 35.
`35`	The result of the expression evaluation is returned.
`>`	The SKILL evaluator is listening for the next command.

Summary of Special Characters in SKILL

Common SKILL syntax characters used in OCEAN scripts:

- Parentheses [()]

```
path( "~/mymodels" )
```

No space

- Double quotes [""]

```
path( "~/mymodels" )
```

Literal strings are surrounded by double quotes.

Parentheses

Parentheses surround the arguments to the command. The command name is followed immediately by the left parenthesis, with no intervening space as in this example:

```
path( "~/mymodels" "~/mymodels_extra" )
```

This is correct. There is no space between the word path and the first parenthesis.

```
path  ( "~/mymodels" "~/mymodels_extra" )
```

This example is **incorrect**. The space after the command name causes a syntax error.

Double Quotes

Double quotes are used to surround string values. A string value is a sequence of characters, such as "abc". In the following example, the directory names provided to the path command are strings, which must be surrounded by double quotes:

```
path("~/mymodels" "~/mymodels_extra")
```

In *OCEAN*, a SKILL convention indicates when an argument must be a string. When you see the prefix *t_*, you must substitute a string value (surrounded by double quotes) for the argument as in this example:

```
desVar( t_desVar1 g_value1 t_desVar2 g_value2)
```

This example includes two string values that must be supplied: **t_desVar**

Single Quote and Question Mark

- Single Quotes

  Example:

  ```
  analysis( 'tran .... )
  ```

  The single quote specifies a predefined symbol for the command.

  If the single quote is left out, OCEAN treats *tran* as a SKILL variable.

  A single quote prevents evaluation of the item that follows it.

- Question Mark

  Example:

  ```
  analysis( 'tran ?stop 1u )
  ```

  The question mark specifies an optional argument for the command, passed by name.

Single Quote

A single quote indicates that an item is a symbol. Symbols in SKILL correspond to constant enumerated values in the *C* language. In the context of OCEAN, there are predefined symbols that you must use to avoid errors.

In example below, *tran* is a predefined symbol and must be preceded by a single quote. You can determine the valid symbols for a command by checking the valid values for the command's arguments.

```
analysis( 'tran .... )
```

Another example is the predefined *save* command. The *'v* symbol indicates that the item to save is the voltage on a net.

```
save( 'v "net1" )
```

Question Mark

A question mark indicates an optional keyword argument, which is the first part of a keyword parameter. For a keyword parameter, the first component is the keyword, which has a question mark in front of it. The second component is the value being passed, and immediately follows the keyword.

In the example, *analysis('tran ?stop 1u)*, all the keyword/value pair arguments to the analysis command are optional, except the '**tran** keyword.

Lab Overview

Lab 2-1 Starting the Software

Lab 2-2 Using the Command Interpreter Window

Lab 2-3 Exploring SKILL Numeric Data Types

Lab 2-4 Exploring SKILL Variables

Displaying Data in the CIW

Every SKILL data type has a default display format that is called the print representation.

Data Type	Print Representation
integer	5
floating point	1.3
text string	"mary learned SKILL"
list	(1 2 3)

SKILL displays a return value with its print representation.

SKILL functions often display data before they return a value.

Both the *print* and *println* functions use the print representation to display data in the CIW output pane. The *println* function sends a newline character.

The *print* and *println* Functions

Both the *print* and *println* functions return *nil*.

This nongraphic session transcript illustrates *println*.

```
> x = 8
8
> println( x )
8
nil
>
```

This nongraphic session transcript shows an attempt to use the *println* function to print out an intermediate value *3+4* during the evaluation of *(3+4)*5*. The *println*'s return value of *nil* causes the error.

```
> println(3+4)*5
7
*Error* times: can't handle (nil * 5)
>
```

Displaying Data with Format Control

The *printf* functions writes formatted output to the CIW. This example displays a line in a report.

```
printf(
  "%-15s %-15s %-10d %-10d %-10d %-10d\n"
  layerName purpose
  rectCount labelCount lineCount miscCount
  )
```

The first argument is a conversion control string containing directives.

```
%[-][width][.precision]conversion_code
  [-] = left justify
  [width] = minimum number of character positions
  [.precision] = number of characters after the decimal
  conversion_code
    d - digit(integer)
    f - floating point
    s - string or symbol
    c - character
    n - numeric
    L - default format
```

The *%L* directive specifies the default format. Use the print representation for each type to display the value.

The *printf* Function

If the conversion control directive is inappropriate for the data item, *printf* gives you an error message.

```
> printf( "%d %d" 5 6.3 )
*Error* fprintf/sprintf: format spec. incompatible with data - 6.3
>
```

The %L Directive

The *%L* directive specifies the print representation. This directive is a very convenient way to intersperse application specific formats with default formats. Remember that *printf*
returns *t*.

```
>  aList = '(1 2 3)
(1 2 3)
>  printf( "This is a list: %L\n" aList )
This is a list: (1 2 3)
t
>
```

Solving Common Problems

These are common problems you might encounter:

- The CIW does not respond.
- The CIW displays inexplicable error messages.
- You pass arguments incorrectly to a function.

What If the CIW Doesn't Respond?

Situation:

- You typed in a SKILL function.
- You pressed Return.
- Nothing happens.

You have one of these problems:

- Unbalanced parentheses
- Unbalanced string quotes

Solution:

The following steps trigger a system response in most cases.

- You might have entered more left parentheses than right parentheses.
- Enter a **]** character (a closing right square bracket). This character closes all outstanding right parentheses.
- If still nothing happens, enter the " character followed by the **]** character.

White Space Sometimes Causes Errors

White space can cause error messages.

Do **not** put any white space between the function name and the left parenthesis.

The error messages:

- Do not identify the white space as the cause of the problem.
- Vary depending on the surrounding context.

Examples

A SKILL function to concatenate several strings

```
strcat ( "mary" " had" " a" " little" " lamb")
*** Error in routine eval:
Message: *Error* eval: illegal function mary
```

An assignment to a variable

```
greeting = strcat ( "happy" " birthday" )
*** Error in routine eval:
Message: *Error* eval: unbound variable strcat
```

Passing Incorrect Arguments to a Function

All built-in SKILL functions validate the arguments you pass. You must pass the appropriate number of arguments in the correct sequence. Each argument must have the correct data type.

If there is a mismatch between what the caller of the built-in function provides and what the built-in function expects, then SKILL displays an error message.

Examples

The *strcat* function does not accept numeric data.

```
strcat( "mary had a" 5 )
Message: *Error* strcat: argument #2 should be either
a string or a symbol (type template = "S") - 5
```

The *type template* mentioned in the error message encodes the expected argument types.

The *strlen* function expects at least one argument.

```
strlen()
*Error* strlen: too few arguments (1 expected, 0 given) - nil
```


Use the Cadence Finder to verify the following information for a SKILL function:

- The number of arguments that the function expects.
- The expected data type of each argument.

The following table summarizes some of the characters in the type template which indicate the expected data type of the arguments. The Cadence Finder and the Cadence online SKILL documentation follow the same convention.

Character in Type Template	Expected Data Type
x	integer
f	floating point
s	variable
t	text string
S	variable or text string
g	general

Lab Overview

Lab 2-5 Displaying Data in the CIW

Lab 2-6 Solving Common Input Errors

Module Summary

This module

- Introduced SKILL, the command language for the Design Framework II environment.
- Defined what user interface action sends SKILL expressions to the SKILL Evaluator.
- Explored SKILL data, function calls, variables, and operators.
- Showed you ways to solve common problems.

OCEAN Basics

Module 3

October 5, 2005

Module Objectives

- OCEAN overview
- Command types
- Sample OCEAN scripts

Terms and Definitions

CMRR	Common-mode rejection ratio
PSF	Parameter storage format (waveform storage)
PSRR	Power supply rejection ratio
PWLF	Piece-wise linear file (input stimulus)

OCEAN Overview

- OCEAN is a capability included with the Virtuoso® Analog Design Environment.
- OCEAN checks out an Analog Design Environment license (feature 34510) if used to run simulations.
- Use OCEAN for the following tasks:
  - To create scripts to run batch mode simulations.
  - To run parametric, corners, Monte Carlo, and optimization analyses.
  - To run long simulations without starting the Analog Design Environment graphical user interface.
  - To run simulations from a non-graphical, remote terminal.
- OCEAN is based on the SKILL programming language.

  You can create scripts automatically within the Analog Design Environment.
- After the design has been debugged in the Analog Design Environment, use OCEAN to test your circuit under a variety of conditions.

OCEAN Basics 3-5

The Open Command Environment for ANalysis (OCEAN) lets you set up, simulate, and analyze circuit data. OCEAN is a text-based process that you can run from a UNIX shell (by starting the *ocean* executable) or from the Command Interpreter Window (CIW). You can type OCEAN commands in an interactive session, or you can create scripts containing your commands, then load those scripts into OCEAN. You can use OCEAN with any simulator integrated into the Analog Design Environment.

Typically, you use the Analog Design environment when creating your circuit (in the Virtuoso Schematic Editor software) and when interactively debugging the circuit. After the circuit has the performance you want, you can use OCEAN to run your scripts and test the circuit under a variety of conditions. After making changes to your circuit, you can easily run your scripts again. OCEAN lets you

- Create scripts that you can run repeatedly to verify circuit performance
- Run longer analyses such as parametric, Monte Carlo, optimization, and corners Analyses more effectively
- Run long simulations in OCEAN without the ADE graphical user interface
- Run simulations from a nongraphic, remote terminal

OCEAN is based on the SKILL programming language and uses SKILL syntax. You can use all the SKILL language commands in OCEAN. These commands include *if* statements, *case* statements, *for* loops, *while* loops, *read* commands, *print* commands, and so on. The most frequently used SKILL commands relevant to OCEAN are listed in the *OCEAN Reference Manual*.

Running OCEAN Interactively

You can run OCEAN from a UNIX prompt or from the CIW.

- In UNIX:
  - Enter *ocean* from the UNIX prompt to start.

    This loads *awd.exe* and loads and reads the *.oceanrc* startup file. The *.oceanrc* file contains OCEAN commands that include alias definitions and user-defined procedures and characterization scripts.

    An *ocean>* prompt appears in the UNIX window. Enter commands at the prompt.

    The *.cdsinit* file is not loaded.

    Type ***exit*** to quit OCEAN.
- In the CIW, through the Analog Design Environment:
  - Enter OCEAN commands at any time.
  - The *.oceanrc* is not loaded automatically. If it exists, you must load it by entering:

    ```
    load ".oceanrc"
    ```

OCEAN Basics 3-7

You can run OCEAN from a UNIX prompt or from the CIW. The primary method is to run it from a UNIX shell.

When you enter *ocean* at a UNIX prompt, a series of messages is printed. This includes a note that the *awd.exe* code is loaded. This code drives the Analog Waveform Display tool that drives the Waveform Window. Other messages indicate that context files are loaded to activate the Simulation Environment.

The *.oceanrc* file contains OCEAN commands that include alias definitions, and user-defined procedures and characterization scripts. For example, an *.oceanrc* file might contain:

```
alias h ocnHelp
alias d ocnDisplay
alias sr selectResult
alias op openResults
alias p ocnPrint
installDebugger()
```

Once this file is loaded, you can enter:

```
h( 'commandName)
```

instead of typing

```
ocnHelp( 'commandName)
```

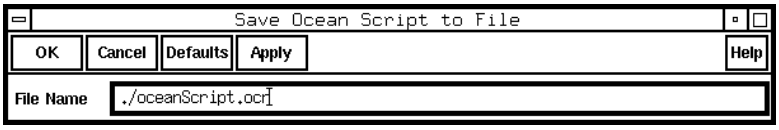
Using OCEAN in the Analog Design Environment is straightforward, because you can enter commands directly into the CIW.

Creating OCEAN Scripts in ADE

You can easily create OCEAN scripts in the Analog Design Environment.

- OCEAN commands of tasks performed in ADE are automatically saved in the *../simulation/design/simulator/schematic/netlist/simulatorX.ocn* files.

- In the Simulation Environment form, select **Session—Save Ocean Script**.



  - Specify the file name of the script to create.
  - Only enabled analyses are saved.
  - The command saves the design, library model file, design variables, data directory, simulator options, plot set, etc.

You can automatically create an OCEAN script within the Analog Design Environment. The following example illustrates the difference between the script that is automatically created in the *../simulation/design/simulator/schematic/netlist/simulatorX.ocn* location, and the one created by the **Session—Save Ocean Script** command.

For example, walk through the following design flow:

1. Start the Analog Design Environment.
2. Specify DC analysis.
3. Select nets on schematic to save.
4. Run simulation.
5. Turn off DC analysis.
6. Select transient analysis.
7. Run simulation.
8. Save OCEAN script.

The *simulatorX.ocn* file in the *netlist* directory will contain the entire design session, including the DC analysis. The *oceanScript.ocn* (default name) will not have the DC analysis, because it was turned off before the script was created.

During a nongraphic session, the Cadence® Design Framework II environment suppresses any graphic output. It does not create any windows.

Loading OCEAN Scripts

You can load OCEAN scripts from a UNIX window or from the CIW.

- In the *ocean* executable or the CIW:

  ```
  ocean> load("script_name.ocn")
  ```

  or

  ```
  ocean> load "script_name.ocn"
  ```

  This will load and run the OCEAN script.

- In UNIX, to have the script load and then have OCEAN exit:

  ```
  ocean < script_name.ocn
  ```

- In UNIX, to have the script load and then go interactive:

  ```
  ocean -restore script_name.ocn
  ```

OCEAN Basics 3-11

Once an OCEAN script exists, it is easy to load it and run it through the CIW or in a UNIX window.

The netlist that you specify with the *design* command needs to have already been created. It is recommended that you use the netlist generated from the Analog Design Environment. You can use netlists from other sources as long as they are in the same format as those created by the Analog Design Environment.

Any models, include files, stimulus files, or PWLF files must be in the locations set by the *path* command.

The scripts that are created automatically in the *../simulation/design/simulator/schematic/netlist/simulatorX.ocn* location or with the **Session—Save Ocean Script** command have the proper syntax to run in OCEAN.

OCEAN Help

Online help is available for all the OCEAN commands. In an OCEAN session, type the following:

```
ocnHelp( 'commandName )
```

For example, enter:

```
ocnHelp( 'analysis )
```

An explanation of the command and examples of use are returned.

For a list of all types of OCEAN commands enter:

```
ocnHelp()
```

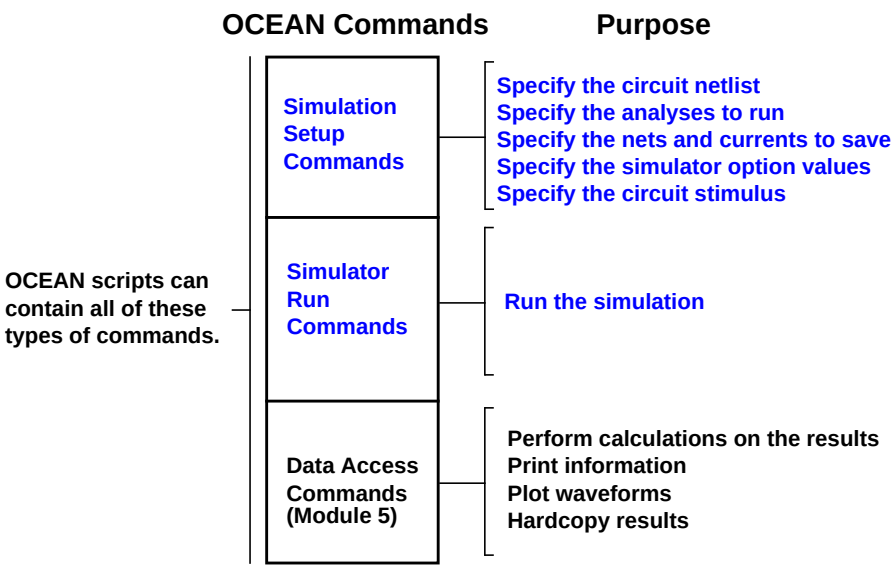
Additional help and samples of OCEAN scripts are located at:

```
<install_dir>/tools/dfII/samples/artist/OCEAN
```


Online help is available for OCEAN commands. Enter **ocnHelp('analysis)** and note:

```
PROTOTYPE      analysis( s_analysisType
             [?<analysisOption1> g_analysisOptionValue1] ...
             [?<analysisOptionN> g_analysisOptionValueN]) => undefined/nil

DESCRIPTION    Specifies the analysis to be simulated. You can
               include as many analysis options as you want. Analysis
               options vary, depending on the simulator you are using. To
               include an analysis option, replace <analysisOption1> with
               the name of the desired analysis option and include another
               argument to specify the value for the option. If you have an
               ac analysis, the first option/value pair might be [?from 0].
               Note: Some simplified commands are available for basic SPICE
               analyses, See the ac, dc, tran, and noise commands. You can
               use ocnHelp( 'analysis ) for more information on the
               analysis types for the simulator you choose.

EXAMPLE(S)
               analysis( 'ac ?start 1 ?stop 10000 ?lin 100 )

               For the Spectre simulator, specifies that an ac analysis is
               to be performed.

               analysis( 'tran ?start 0 ?stop 1u ?step 10n )
               Specifies that a transient analysis is to be performed.
```

Types of OCEAN Commands



You can create OCEAN scripts to accomplish the full suite of simulation and data access tasks that you can perform in ADE. An OCEAN script can contain three types of commands:

- Simulation setup commands
- Simulator run commands
- Data access commands

All the parameter storage format (PSF) information created by the simulator is accessible through the OCEAN data access commands. (The data access commands include all of the ADE calculator functions.)

Initializing the Session

- Specifying the simulator:

  ```
  simulator( 'spectre )
  ```

  - Starts a new OCEAN session using the specified simulator.
  - Works with any simulator that Analog Design Environment supports.
  - Previous session is closed.

Choosing the Design

- Specifying the netlist:

  ```
  design("~/simulation/ampTest/spectre/schematic/netlist/netlist")
  ```

  Netlist often created by the Analog Design Environment.

- Specifying a design by cellView:

  ```
  design( "solutions" "ampTest" "schematic" )
  ```

  - Uses the default netlist in the path of the design.
  - Creates a new netlist if needed.
  - Works only within an *icms*, *icfb*, or *msfb* session.
    The *ocean* executable does not have ability to netlist.

When you specify the Spectre® netlist using:

```
design("~/simulation/ampTest/spectre/schematic/netlist/netlist")
```

you should make sure the following files are in the netlist directory:

```
netlistHeader
netlistFooter
```

These files are required by OCEAN and contain additional statements needed to put together the complete information for Spectre to run the simulation.

These files are automatically created when you netlist from ADE.

If you type in your netlist by hand the additional files can be empty. You can create empty files in UNIX by typing:

```
touch netlistHeader
touch netlistFooter
```

If you specify your design to as a specific cellview in a library you can create the netlist using:

```
createNetlist()
```

This only works if you are running OCEAN within *icms*, *icfb*, and *msfb*.

Specifying the Results Directory

- Specifying the results directory:

```
resultsDir( "~/simulation/ampTest/spectre/schematic" )
```

  If a director is not specified, results are placed in *../psf* relative to the netlist directory.

Specifying Device Models

- Specifying model files when using Spectre® interface:

```
modelFile( "~/Models/myModels.scs" )
```

or

```
path( "~/Models" )
modelFile( "myModels.scs" )
```

- Specifying model files when they contains library sections:

```
modelFile( '("myNPNs.scs" "typ") '("myRes.scs" "slow") )
```

- Specifying paths when using socket simulators like SpectreS:

```
path( "~/models" "/tmp/models" )
```

Design Variables and Simulator Options

Design Variables

- Specify design variables:

```
desVar( "CAP" .5p "myResistor" 1k )
```

- Use SKILL variables to set a design variable:

```
myvar = 1e-12
desVar( "CAP" myvar )
```

Simulator Options

- Specify temperature:

```
temp( 27 )
```

- Specify simulator specific options:

```
option( 'reltol 1e-6 )
option( 'temp 27 )
option( 'sensfile "~/mysensfile" )
```

Choosing What to Save

You can choose what signals will be saved.

- Save all node voltages and branch currents

```
save( 'all )
```

- Save all node voltages (default)

```
save( 'allv )
```

- Save all branch currents

```
save( 'alli )
```

- Save specific node voltages

```
save( 'v "out" "net6" )
```

- Save specific branch currents

```
save( 'i "V0:p" "M1:d" )
```

If you need to list the current save settings:

```
ocnDisplay( 'save )
```

Analysis Functions

Specify only one analysis per analysis command.

- To run a Spectre transient analysis to 50us you specify:

```
analysis( 'tran ?stop 50u )
```

- To also set *errpreset* for that analysis, you can specify:

```
analysis( 'tran ?errpreset "conservative" )
```

- You could also specify the entire statement as one argument:

```
analysis( 'tran ?stop 50u ?errpreset "conservative" )
```

- Using simplified commands to set up analyses

```
tran( 0 50u 100n )
ac( 1 10000 "Linear" 100 )
dc( "v1" 0 5 1 )
noise( "n1" "v1" )
```

- List all optional arguments and their current settings for the analysis

```
ocnDisplay( 'analysis )
```

Analysis Functions (continued)

To delete commands and analyses use the *delete* command.

```
delete( 'tran )
delete( 'desVar "CAP" )
delete( 'ic )
delete( 'forcenode )
```

Sequential Execution

OCEAN runs all commands sequentially.

The following example will only run the **last** analysis:

```
analysis( 'tran ?stop "10e-9" )
analysis( 'tran ?stop "20e-9" )
run()
```

To run two separate analysis and keep the data you need to run the simulation in separate results directories:

```
resultsDir( "~/simulation/tran10ns" )
analysis( 'tran ?stop "10e-9" )
run()
resultsDir( "~/simulation/tran20ns" )
analysis( 'tran ?stop "20e-9" )
run()
```

Sample OCEAN Script

```
simulator( 'spectre )
design("~/simulation/ampTest/spectre/schematic/netlist/netlist")
resultsDir( "~/simulation/ampTest/spectre/schematic" )
modelFile( '( "~/Models/myModels.scs" "") )
desVar( "CAP" .5p )
temp( 27 )
analysis('ac ?start "100"  ?stop "150M"  ?dec "20"  )
run()
selectResult( 'ac )
plot( getData("/out") )
```

Note: You can specify the *resultsDir(...)* to set the directory where simulation data is saved. In the example above, *resultsDir(...)* points to the default location. When using the default, the *resultsDir(...)* expression is not needed.

You can annotate the sample OCEAN script as follows:

```
simulator( 'spectre ) ->Choose simulator.
design("~/simulation/ampTest/spectre/schematic/netlist/netlist")
->Specify schematic to be simulated. The netlist was created in ADE.
resultsDir( "~/simulation/ampTest/spectre/schematic" )
->Specify location of psf directory that holds simulation results. In this
example, the default is specified as an argument to resultsDir(), so the
command is not necessary.
modelFile( '( "~/Models/myModels.scs" "")) ->Set Library Model File in
Simulation Environment.
analysis('ac ?start "100"  ?stop "150M"  ?dec "20"  ) ->Specify AC Analysis.
desVar(   "CAP" .5p ) ->Set the value of the design variable, CAP.
temp( 27 ) ->Set the simulation temperature.
run() ->Run the simulation.
selectResult( 'ac ) ->Select the AC analysis results.
plot(getData("/out") ) -> Generic command to plot the data. Try using
plot(vm("/out")) to plot the magnitude of the voltage at node "/out".
```

Note: There are “easy” commands available for common analyses. For example, you can enter *tran(0 100n 1n)* for a transient analysis, *ac(1 10000 “linear” 100)* for an AC analysis, and *dc(“r1” “r” 0 5 1)* for a DC analysis. See the *OCEAN Reference Manual* for more examples and the general form of these commands.

Lab Overview

Lab 3-1 Using an OCEAN Script in ADE to Run a Simple Simulation

Lab 3-2 Measuring PSRR and CMRR with OCEAN

PSRR stands for power supply rejection ratio. CMRR stands for common mode rejection ratio.

Module Summary

This module

- Gave an overview of OCEAN
- Illustrated the command types
- Showed some sample OCEAN scripts

cadence®

SKILL Lists

Module 4

October 5, 2005

Module Objectives

- Build lists.
- Retrieve list elements.
- Use lists to represent points and bounding boxes.

What Is a SKILL List?

A SKILL list is an ordered collection of SKILL data objects.

The elements of a list can be of any data type, including variables and other lists.

The special data item *nil* represents the empty list.

SKILL functions commonly return lists you can display or process.

OCEAN list examples:

- Results available
- Outputs from an analysis
- Swept values

User interface list examples:

- Design Framework II windows currently open
- Pull-down menus in a window
- Menu items in a menu

SKILL Lists 4-5

You can use a list to represent many different types of objects. The arbitrary meaning of a list is inherent in the programs that manipulate it.

How SKILL Displays a List

To display a list, SKILL surrounds the elements of the list with parentheses.

```
( "rect" "polygon" "rect" "line" )
( 1 2 3 )
( 1 ( 2 3 ) 4 5 ( 6 7 ))
( ( "one" 1 ) ( "two" 2 ) )
( "one" one )
```

To display a list as a return value, SKILL splits the list across multiple lines when the list:

- Contains sublists
- Has more than *_itemsperline* number of items

Use the *printf* or *println* functions to display a list. SKILL displays the output on a single line.

Consider the examples shown below based on these assignments. The output is taken from a nongraphical session.

```
aList = '( 1 2 3 )
aLongList = '( 1 2 3 4 5 6 7 8 )
aNestedList = '( 1 ( 2 3 ) 4 5 ( 6 7 ))
```


The examples above are displayed in the CIW as follows:

```
> aList = '( 1 2 3 )
(1 2 3)
> aLongList = '( 1 2 3 4 5 6 7 8 )
(1 2 3 4 5
    6 7 8
)
> aNestedList = '( 1 ( 2 3 ) 4 5 ( 6 7 ))
(1
    (2 3) 4 5
    (6 7)
)
>
```

Creating New Lists

There are several ways to build a list of elements. Two straightforward ways are to do the following:

- Specify all the elements literally. Apply the ’ operator to the list.

Expressions	Return Result
’(1 2 3)	*(1 2 3)*
’("one" 1)	*("one" 1)*
’(("one" 1) ("two" 2))	*(("one" 1) ("two" 2))*

- Make a list by computing each element from an expression. Pass the expressions to the *list* function.

Expressions	Return Result
a = 1	*1*
b = 2	*2*
list(a b 3)	*(1 2 3)*
list(a**2+b**2 a**2-b**2)	(5 -3)

SKILL Lists 4-9

Store the new list in a variable. Otherwise, you cannot refer to the list again.

The ’ Operator

Follow these guidelines when using the ’ operator to build a list:

- Include sublists as elements with a single set of parentheses.
- Do not use the ’ operator in front of the sublists.
- Separate successive sublists with white space.

The *list* Function

SKILL normally evaluates all the arguments to a function before invoking the function. The function receives the evaluated arguments. The *list* function allocates a list in virtual memory from its evaluated arguments.

Adding Elements to an Existing List

Here are two ways to add one or more elements to an existing list:

- Use the *cons* function to add an element to an existing list.

Expressions	Return Result
result = '(2 3)	*(2 3)*
result = cons(1 result)	*(1 2 3)*

- Use the *append* function to merge two lists together.

Expressions	Return Result
oneList = '(4 5 6)	*(4 5 6)*
aList = '(1 2 3)	*(1 2 3)*
bList = append(oneList aList)	*(4 5 6 1 2 3)*

SKILL Lists 4-11

The *cons* Function

The construct (*cons*) function adds an element to the beginning of an existing list. This function takes two arguments. The first is the new element to be added. The second is the list to add the element to. The result of this function's execution is a list containing one more element than the input list.

Store the return result from *cons* in a variable. Otherwise, you cannot refer to the list subsequently. It is common to store the result back into the variable containing the target list.

The *append* Function

The *append* function builds a new list from two existing lists. The function takes two arguments. The first argument is a list of the elements to begin the new list. The second argument is a list of the elements to complete the new list.

Store the return result from *append* in a variable. Otherwise, you cannot refer to the list subsequently.

Points of Confusion

People often think that *nil*, *cons*, and the *append* functions violate common sense. Here are some frequently asked questions.

Question	Answer
What is the difference between *nil* and *'(nil)*?	*nil* is a list containing nothing. Its length is *0*. *'(nil)* builds a list containing the single element *nil*. The length is *1*.
How can I add an element to the end of a list?	Use the *append* and *list* functions. *aList = '(1 2)* *aList = append(aList list(3)) => (1 2 3)*
Can I reverse the order of the arguments to the *cons* function? Will the results be the same?	You either get different results or an error. *cons('(1 2) '(3 4)) => ((1 2) 3 4)* *cons('(3 4) '(1 2)) => ((3 4) 1 2)* *cons(3 '(1 2)) => (3 1 2)* *cons('(1 2) 3) =>* *** Error in routine cons *Error* cons: argument #2 should be a list
What is the difference between *cons* and *append*?	*cons('(1 2) '(3 4)) => ((1 2) 3 4)* *append('(1 2) '(3 4)) => (1 2 3 4)*

Question	Answer
What is the difference between *nil* and *'(nil)*?	*nil* is a list containing nothing. Its length is *0*. *'(nil)* builds a list containing a single element. The length is 1.
How can I add an element to the end of a list?	Use the *list* function to build a list containing the individual elements. Use the append function to merge it to the first list. There are more efficient ways to add an element to the end of a list. They are beyond the scope of this course.
Can I reverse the order of the arguments to the *cons* function? Will the results be the same?	Common sense suggests that simply reversing the elements to the *cons* function will put the element on the end of the list. This is not the case.
What is the difference between *cons* and *append*?	The *cons* function requires only that its second argument be a list. The length of the resulting list is one more than the length of the original list. The *append* function requires that both its arguments be lists. The length of resulting list is the sum of the lengths of the two argument lists.

Working with Existing Lists

Task	Function	Example	Return Result
Retrieve the first element of a list	*car*	*numbers = '(1 2 3)* *car(numbers)*	*(1 2 3)* *1*
Retrieve the tail of the list	*cdr*	*cdr(numbers)*	*(2 3)*
Retrieve an element given an index	*nth*	*nth(1 numbers)*	*2*
Tell if a given data object is in a list	*member*	*member(4 numbers)* *member(2 numbers)*	*nil* *(2 3)*
Count the elements in a list	*length*	*length(numbers)*	*3*
Apply a filter to a list	*setof*	*setof(* *x* *'(1 2 3 4)* *oddp(x))*	*(1 3)*

The *nth* Function

Lists in SKILL are numbered from 0. The 0 element of a list is the first element, the 1 element of a list is the second element and so on.

The *member* Function

The member function returns the tail of the list starting at the element sought or nil, if the element is not found. Remember, if it is not nil - it is true.

The *setof* Function

The *setof* function makes a new list by copying only those top-level elements in a list that pass a test. You must write the test in terms of a single variable. The first parameter to the *setof* function identifies the variable you are using in the test.

- The first argument is the variable that stands for an element of the list.
- The second argument is the list.
- The third argument is one or more expressions that you write in terms of the variable. The final expression is the test. It determines if the element is included in the new list.

Frequently Asked Questions

Students often ask these questions:

- Why are such critical functions as *car* and *cdr* called such weird names?
- What is the purpose of the *car* and *cdr* functions?
- Can the *member* function search all levels in a hierarchical list?
- How does the *setof* function work? What is the variable *x* for?

Questions	Answers
Why are such critical functions as *car* and *cdr* called such weird names?	*car* and *cdr* were machine language instructions on the first machine to run Lisp. *car* stands for *contents of the address register* and *cdr* stands for *contents of the decrement register*.
What is the purpose of the *car* and *cdr* functions?	Lists are stored internally as a series of doublets. The first element is the list entry, the second element of the doublet is a pointer to the rest of the list. The *car* function returns the first element of the doublet, the *cdr* function returns the second. For any list *L* it is true that *cons(car(L) cdr(L))* builds a list equal to *L*. This relates the three functions *cons*, *car*, and *cdr*.
Can the *member* function search all levels in a hierarchical list?	No. It only looks at the top-level elements. Internally the *member* function follows right branches until it locates a branch point whose left branch dead ends in the element.
How does the *setof* function work? What is the variable *x* for?	The *setof* function makes a new list by copying only those top-level elements in a list that pass a test. The test must be written in terms of a single variable. The first parameter to the *setof* function identifies the variable you are using in the test.

Two-Dimensional Points

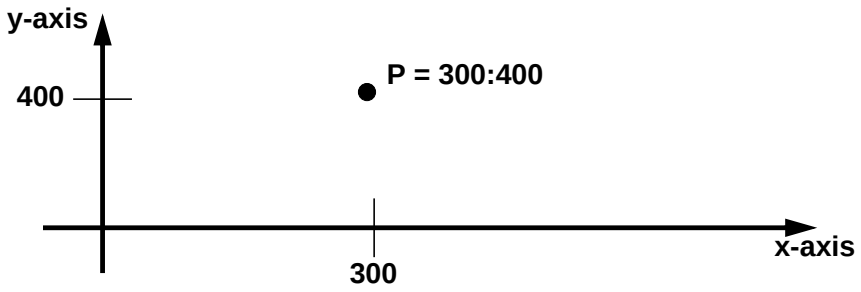
SKILL represents a two-dimensional point as a two-element list.

The binary operator (:) builds a point from an x-value and a y-value.

```
xValue = 300
yValue = 400
P = xValue:yValue => ( 300 400 )
```

The *xCoord* and *yCoord* functions access the x-coordinate and the y-coordinate.

```
xCoord( P ) => 300
yCoord( P ) => 400
```



The : Operator

You can use the ’ operator or list function to build a coordinate.

```
P = '( 3.0 5.0 )
P = list( xValue yValue )
```

The : operator expects both of its arguments to be numeric. The *range* function implements the : operator.

```
> "hello":3
*Error* range: argument #1 should be a number
(type template = "n") - "hello"
```

The *xCoord* and *yCoord* Functions

Alternatively, you can use the *car* function to access the x-coordinate and *car(cdr (...))* to access the y-coordinate.

```
xValue = car( P )
yValue = car( cdr( P ) )
```

return a list ()

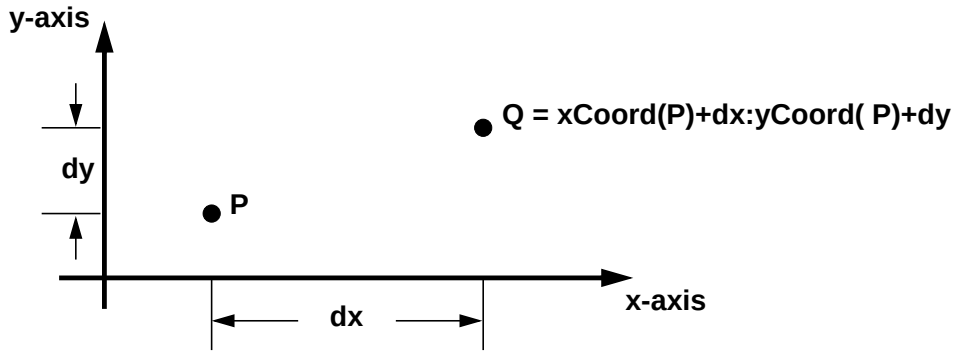
Computing Points

The : operator combines naturally with arithmetic operators. It has a lower precedence than the + or the * operator.

```
3+4*5:4+7*8 => (23 60)
```

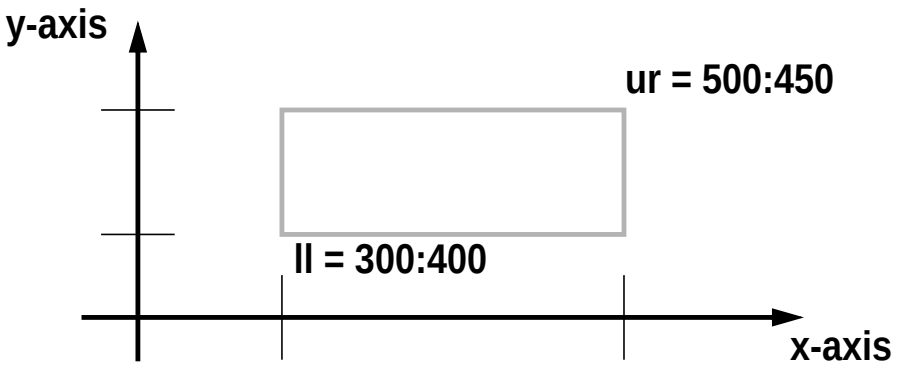
Computing a point from another point is easy.

For example, given a point *P*, apply an offset *dx* and *dy* in both directions.



Bounding Boxes

SKILL represents a bounding box as a two-element list. The first element is the lower-left corner and the second element is the upper-right corner.



This is returned to you by the system as: ((300 400) (500 450))

Remember that : is a point operator.

’(300:400 500:450) does not create a list of two lists since the : operator is not evaluated.

Use list(300:400 500:450) which will evaluate the point and thus create a list containing two lists and so, in this case, also a bounding box.

Creating a Bounding Box

Use the *list* function to build a bounding box. Specify the points by variables or by using the : operator.

```
ll = 300:400 ur = 500:450
bBox = list( ll ur ) =>
       (( 300 400 ) ( 500 450 ))

bBox = list( 300:400 500:450 ) =>
     (( 300 400 ) ( 500 450 ))
```

SKILL Lists 4-25

When you create a bounding box, put the points in the correct order. When SKILL prompts the user to digitize a bounding box, it returns the bounding box with the lower-left and upper-right corner points correctly ordered, *even though the user may have digitized the upper-left and lower-right corners!*

You may use the ’ operator to build the bounding box ONLY if you specify the coordinates as literal lists.

```
bBox = '(( 300 400 ) ( 500 450 ))
  => (( 300 400 ) ( 500 450 ))
```

Retrieving Elements from Bounding Boxes

Use the *lowerLeft* and *upperRight* functions to retrieve the lower-left corner and the upper-right corner points of a bounding box.

```
lowerLeft( bBox ) => ( 300 400 )
upperRight( bBox ) => ( 500 450 )
```

These functions assume that the order of the elements is correct.

Use the *xCoord* and *yCoord* functions to retrieve the coordinates of these corners.

```
xCoord( lowerLeft( bBox )) => 300
yCoord( upperRight( bBox )) => 450
```

Combinations *car* and *cdr* Functions

SKILL provides a family of functions that combine *car* and *cdr* operations. You can use these functions on any list.

Bounding boxes provide a good example of working with the *car* and *cdr* functions.

Functions	Combination	Bounding box examples	Expression
car	car(...)	lower left corner	ll = car(bBox)
cadr	car(cdr(...))	upper right corner	ur = cadr(bBox)
caar	car(car(...))	x-coord of lower left corner	llx = caar(bBox)
cadar	car(cdr(car(...))	y-coord of lower left corner	lly = cadar(bBox)
caadr	car(car(cdr(...)))	x-coord of upper right corner	urx = caadr(bBox)
cadadr	car(cdr(car(cdr(...]	y-coord of upper right corner	ury = cadadr(bBox)

Using the *xCoord*, *yCoord*, *lowerLeft* and *upperRight* functions is preferable in practice to access coordinates, bounding boxes, and paths.

The *cadr, caar, cadar* Functions

The functions *cadr*, *caar*, and so forth are built in for your convenience.
Any combination of four a’s or d’s.

Lab Overview

Lab 4-1 Creating New Lists

Lab 4-2 Extracting Items from Lists

Module Summary

In this module we covered:

- SKILL lists can contain any type of SKILL data. *nil* is the empty list.
- Using the *'* operator and the *list* function to build lists.
- Using the *cons* and *append* functions to build lists from existing lists.
- Using the *length* function to count the number of elements in a list.
- Using the *member* function to find an element in an existing list.
- Using the *setof* function to filter a list according to a condition.
- How two-dimensional points are represented by two element lists.
- How bounding boxes are also represented by two element lists.

Data Access and Plotting

Module 5

October 5, 2005

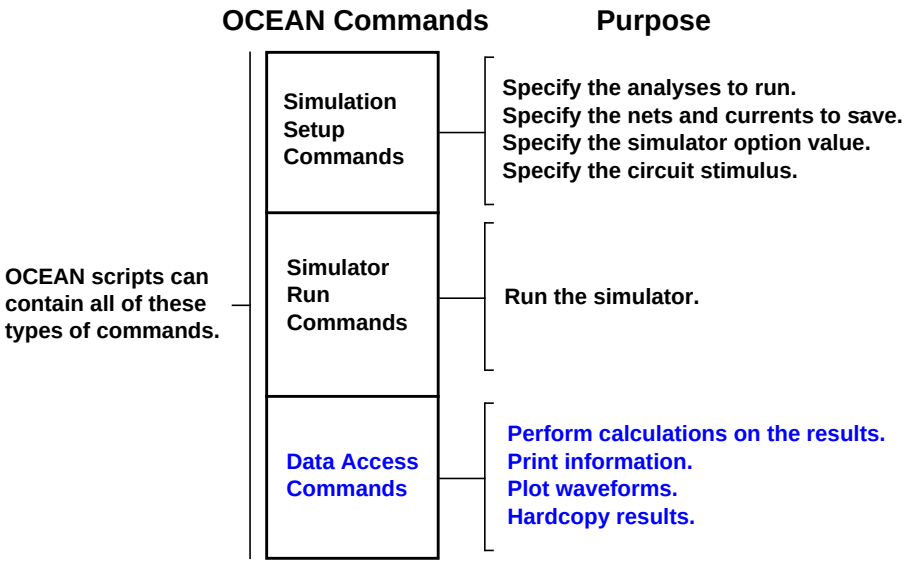
Module Objectives

- Accessing data results
- Post-processing
- Plotting
- Waveform functions
- Hardcopy

Terms and Definitions

AWD	Analog Waveform Display tool
Wavescan	A waveform display tool with mixed-signal capability

Types of OCEAN Commands



You can create OCEAN scripts to accomplish the full suite of simulation and data access tasks that you can perform in the Virtuoso® Analog Design Environment (ADE). An OCEAN script can contain three types of commands:

- Simulation setup commands
- Simulator run commands
- Data access commands

All the parameter storage format (PSF) information created by the simulator is accessible through the OCEAN data access commands. (The data access commands include all of the ADE calculator functions.)

Data Access Commands

You can open simulation results and analyze data.

- Data access does not require a completed simulation in the current session.
- Examples of data access commands include:

```
openResults( "./simulation/opamp/spectre/schematic/psf" )
results()
selectResult( 'tran )
ocnPrint( v( "/net56") )
i( "/R1" )
plot( v("/out") )
pv( "/Q19" "ib" )
```

- List of commands includes:

```
dataTypes, getData, i, noiseSummary, ocnPrint, openResults,
outputParams, outputs, pv, report, results, selectResult,
sweepNames, sweepValues, v
```

- Common arithmetic operators and all ADE calculator functions are available.

There are special commands available in OCEAN that allow you to access data after a simulation is run. These include *dataTypes*, *getData*, *i*, *noiseSummary*, *ocnPrint*, *openResults*, *outputParams*, *outputs*, *pv*, *report*, *results*, *selectResult*, *sweepNames*, *sweepValues*, and *v*. A full list of commands and their usage is found in the *OCEAN Reference Manual*. Some of these commands function as follows:

`openResults( "./simulation/opamp/spectreS/schematic/psf" )`

Opens simulation results stored in PSF format in the *psf* directory.

`results()`

Returns a list of the type of results that can be selected.

`selectResult( 'tran )`

Selects Transient results.

`ocnPrint( v( "/net56" ) )`

Prints the text data of the waveform net56.

`i( "/R1" )`

Returns the current through R1.

`plot( v("/out") )`

Plots node “out" in a waveform window.

`pv( "Q19" "ib" )`

Returns the value of the *ib* parameter for the *Q19* component.

Opening an Existing Results Database

You can open an existing results database with the *openResults* command:

```
openResults( "~/simulation/myTest/results/psf" )
```

This gives the same results as

```
openResults( "~/simulation/myTest/results" )
```

if you have a *psf* results directory in the above directory which is the default for OCEAN and the Virtuoso® Analog Design Environment.

You do not need to specify the *openResults* statement if you just ran a simulation in the same session. OCEAN will open the latest results for you as soon as the *run()* command has finished.

The command *openResults* can also be used directly on results created in any ADE session.

Selecting the Analysis Result

To be able to plot and post-process the simulation results you need to select the result specific for each analysis:

```
selectResult( 'tran )
```

This will select the transient analysis results for plotting and post-processing.

If you are analyzing several analyses you need to specify *selectResult(...)* again.

If you are unsure about what analysis names to use you can view the existing results by using the results command:

```
results()
=>   (output tran instance tranOp variables
       model )
```

Having selected a result, you can find out what is available for that result by using the *outputs()* function:

```
outputs()
=>   ("I3:sink" "I4:sink" "I5:sink" "M1:d" "M1:g" "M1:s" "M1:b"
     "M3:d" "M3:g" "M3:s" "M3:b" "M4:d" "M4:g" "M4:s" "M4:b" "M5:d"
     "M5:g" "M5:s" "M5:b" "M6:d" "M6:g" "M6:s" "M6:b" "net24" "net25"
     "net51" "V0:p" "vdd!")
```

Data Access Functions

Voltages can be accessed via the *v()* function:

```
v("/out")
```

Currents can be accessed via the *i()* function:

```
i("/R1/PLUS")
```

General data can be accessed via the *getData()* function:

```
getData("NF")
```

Access functions also allow the *?result* and *?resultsDir* arguments to override the default results (as selected with *selectResult()*) and results directory (as selected with *openResults()*):

```
v("/out" ?result 'tran ?resultsDir "./resultData/psf")
```

This can be useful when combining results from several simulations.

Schematic Names versus Netlist Names

Names passed to the data access functions can either use schematic names or netlist names.

- Schematic names

  Schematic names always begin with “/” and use the names that appear in the schematic (or rather design) hierarchy. If these are ultimately mapped during netlisting, then that will be taken into account. For example:

  ```
  v("/I1/I2/net5")
  ```

  For schematic names to work, the *map* and *amap* directories must exist alongside the netlist specified with the *design()* function, or the *design()* with library, cell and view names must be used.

- Netlist (or simulator) names

  Names that appear in the netlist can always be used. These are the names that the simulator will write into the results database. These can be used if a non-ADE-generated netlist was simulated and the mapping information is not present. For example:

  ```
  v("I1.I2.net5")
  ```

  The hierarchy separator for the Virtuoso Spectre® Circuit Simulator is “.”

How does OCEAN decide whether a name is a schematic or netlist name?

If the name begins with “/” then it is a schematic name; otherwise it is a netlist (or simulator) name.

Accessing Parameters

Some results (such as dcOp, instance, or model) produce parameters for instances in the design. For example:

```
selectResults( 'dcOp )
outputs()
=>    ("other_source" "my_source" "I5" "I4" "I3" "M1" "M3" "M5" "M4"
      "M7" "M6" "V0")
```

outputs() tells you the instances for which there is operating point data. To find out what types of data are available, use *dataTypes()*:

```
dataTypes() => ("isource" "mos3" "vsource")
```

Then you can find out the available parameters for each type by using *outputParams()*:

```
outputParams("mos3")
=>    ("cbd" "betaeff" "he_vdsat" "ids" "vgs" "vds" "vbs" "vth"
      "vdsat" "gm" "gds" "gmbs" "gameff" "cbs" "cgs" ...)
```

Then to find a specific parameter, you can use *pv()*:

```
pv("M3" "gm") => 9.812257e-05
```

Available OCEAN Data Aliases

Aliases are available in OCEAN to simplify your scripts.

Alias	Syntax	Description
vm	mag(v(t_net))	Magnitude of voltage on the net
vdb	db20(v (t_net)) dB20(v (t_net))	20*log(*magnitude of voltage on the net*)
vp	phase(v(t_net))	Phase of voltage on net
vr	real(v(t_net))	Real part of complex voltage
vim	imag(v(t_net))	Imaginary part of complex voltage
im	mag(i(t_component))	Magnitude of AC current
ip	phase(i(t_component))	Phase of AC current
ir	real(i(t_component))	Real part of complex number representing AC current
iim	imag(i(t_component))	Imaginary part of complex number representing AC current

OCEAN Calculator Functions

These functions correspond to the functions in the ADE calculator:

```
 abs              acos             add1             asin
atan             average          b1f              bandwidth
cPwrContour      cReflContour     clip             complex
complexp         compression      compressionVRI   compressionVRICurves
conjugate        convolve         cos              cross
db10             db20             dbm              delay
deriv            dft              exp              flip
fourEval         frequency        ga               gac
gainBwProd       gainMargin       gmax             gmin
gmsg             gmux             gp               gpc
groupDelay       gt               harmonic         harmonicFreqList
harmonicList     iinteg           imag             integ
ipn              ipnVRI           ipnVRICurves     kf
linRg            ln               log              log10
logRg            lsb              lshift           mag
max              min              mod              nc
overshoot        peakToPeak       phase            phaseDeg
phaseDegUnwrapped                 phaseMargin      phaseRad
phaseRadUnwrapped                 pow              psd
psdbb            random           real             riseTime
rms              rmsNoise         root             round
rshift           sample           settlingTime     sin
slewRate         spectralPower    sqrt             srandom
ssb              sub1             tan              tangent
thd              value            xmax             xmin
xval             ymax             ymin
```


More help on these functions can be found in the OCEAN reference manual and by using *ocnHelp()*.

OCEAN Plotting Commands

- Waveform plot command:

```
plot( v("/in") v("/out") )
plot( v("/out")/v("/in") )
```

  The plot function will accept multiple arguments and will plot all the signals specified.

- Specify the *?expr* argument if you want to change the waveform name:

```
plot( getData( "NF" ) ?expr list("Noise Figure") )
```

  The argument needs to be a list with as many entries as signals being plotted, with the name of each signal.

OCEAN Printing Commands

- Printing tables and scalar values to terminal or file:

```
ocnPrint( v("/out") )
ocnPrint( ?output "myFile" v("/out") )
```

- Precision and formatting can be controlled:

```
ocnPrint( ?precision 10 ?numberNotation 'suffix
     v("/out") v("/in")
)
```

- You can get more information with *ocnHelp('ocnPrint)*.

ocnPrint

```
PROTOTYPE        ocnPrint( [?output t_filename | p_port]
                 [?precision x_precision] [?numberNotation s_numberNotation]
                [?numSpaces x_numSpaces] [?width x_width] o_waveform1
                [o_waveform2 ...] ) => t/nil


DESCRIPTION      Prints the text data of the waveforms specified
                 in the list of waveforms.

                 If you provide a filename as the ?output argument, the
                 ocnPrint command opens the file and writes the information
                 to it. If you provide a port (the return value of the SKILL
                 outfile command), the ocnPrint command appends the
                 information to the file that is represented by the port.

                 Valid values for s_numberNotation are: 'suffix,
                 'engineering, 'scientific, 'none.
```

OCEAN Waveform Tool

- In IC5141 there is a choice of two waveform tools:
  - Wavescan (executable name *wavescan*)—the default, new waveform tool
  - Analog Waveform Display (executable name *awd*)—the older waveform tool
- The *ocnWaveformTool()* function may be used to select the waveform tool for the session:

```
ocnWaveformTool( 'wavescan )
ocnWaveformTool( 'awd )
```

Waveform Window Functions

- Creating new window and clearing existing window:
  - newWindow()
  - clearAll()
- Sub window manipulation:
  - addSubwindow()—adds new subwindow
  - clearSubwindow()—clears current subwindow
  - deleteSubwindow()—deletes current subwindow
- Getting/setting current window:
  - currentWindow([windowId])—gets or sets current window
  - currentSubwindow([subWindowNum])—gets or sets current subwin
- Deleting waveforms:
  - deleteWaveform({index|"all"})—deletes a numbered waveform or all
- Plotting modes and styles:
  - displayMode({"composite" | "strip" | "smith"})
  - plotStyle({'auto | 'scatterPlot | 'bar | 'joined})

Waveform Window Functions (continued)

- Axis limits
  - xLimit(list(f_min f_max))—sets range for X axis
  - xLimit(nil)—fits X axis range
  - yLimit(list(f_min f_max))—sets range for Y axis
  - yLimit(nil)—fits Y axis range
- Graph titles
  - addTitle("waveform title")—sets title for waveform window
  - addSubwindowTitle("subwin title")—sets title for subwindow
- Waveform labels (see documentation or *ocnHelp* for details)
  - addWaveLabel
  - addWindowLabel
  - removeLabel

Graphics Performance Optimization

Each time a plot is done, the waveform window is redrawn. If you are performing several separate *plot()* calls, the performance can be improved by disabling the graphics, doing the plots, and then turning the graphics back on again.

- graphicsOff()—disables graphics, but queues any graphics updates.
- graphicsOn()—reenables graphics, drawing any pending graphics updates.

SKILL Waveform Window Functions

There are also the SKILL *awv* functions, which can be used for plotting waveforms.

A selection of functions from the API:

- awvResetWindow(window(2))
- awvLogXAxis()
- awvLogYAxis()
- awvResetAllWindows( )
- awvUpdateWindow( )
- awvUpdateAllWindows( )
- awvPlotWaveform (...)
- awvSetOptionValue(“displayGrids” t)
- awvGetCurrentWindow( )

Hardcopy Functions

- Set up the hardcopy options *before* plotting or opening waveform windows.
- Setting the hardcopy options to print to a printer:

```
hardCopyOptions( ?hcNumCopy 1 ?hcPaperSize "A4"
  ?hcPlotterName "printernal" )
```

- Setting the hard copy options to print to a file:

```
hardCopyOptions( ?hcOutputFile "myOutFile" )
```

  If the waveform tool is *wavescan* and the output file suffix is *.png*, *.bmp*, *.tif*, or *.tiff* then an image will be created (as with the **File—Save As Image** menu command in *wavescan*)

- Sending the plot to a printer or a file:

```
hardCopy()
```


Saving image files was added in the 5.10.41.500.0.4 ISR.

Lab Overview

Lab 5-1 Using OCEAN Plotting Functions

Module Summary

In this module we covered

- Accessing data results
- Post-processing
- Plotting
- Waveform functions
- Hardcopy

cadence®

Developing a SKILL Function

Module 6

October 5, 2005

Module Objectives

- Grouping several SKILL expressions into a single SKILL expression.
- Declaring local variables.
- Declaring a SKILL function.
- Understanding the SKILL software development cycle:
  - Loading your SKILL source code.
  - Redefining a SKILL function.

Terms and Definitions

load	A SKILL procedure that opens a file and reads it one SKILL expression at a time. The *load* function evaluates immediately. Usually, you set up your *.cdsinit* file to load your SKILL source code during the initialization of the Design Framework II environment.

Grouping SKILL Expressions Together

Sometimes it is convenient to group several SKILL statements into a single SKILL statement.

Use the curly braces, { }, to group a collection of SKILL statements into a single SKILL statement.

The return value of the single statement is the return value of the last SKILL statement in the group. You can assign this return value to a variable.

This example computes the height of the bounding box stored in *bBox*.

```
bBox = list( 100:200 350:450 )
bBoxHeight = {
   ll   = lowerLeft( bBox )
   ur   = upperRight( bBox )
   lly  = yCoord( ll )
   ury  = yCoord( ur )
   ury - lly
}
=> 250
```

The variables *ll*, *ur*, *lly*, and *ury* are global variables. The curly braces, { }, do not make them local variables.

Curly Braces

The following statements refer to the example above:

- The *ll* and *ur* variables hold the lower-left and upper-right points of the bounding box respectively.
- The *xCoord* and *yCoord* functions return the *x* and *y* coordinate of a point.
- The *ury*-*lly* expression computes the height. It is the last statement in the group and consequently determines the return value of the group.
- The return value is assigned to the *bBoxHeight* variable.

All of the variables, *ll*, *ur*, *ury*, *lly*, *bBoxHeight* and *bBox* are global variables.

Grouping Expressions with Local Variables

Use the *let* function to group SKILL expressions and declare one or more local variables.

- Include a list of the local variables followed by one or more SKILL expressions.
- These variables will be initialized to *nil*.

The SKILL expressions compose the body of the *let* function. The *let* function returns the value of the last expression computed within its body.

This example computes the height of the bounding box stored in *bBox*.

```
bBox = list( 100:200 350:450 )
bBoxHeight = let( ( ll ur lly ury )
  ll   = lowerLeft( bBox )
  ur   = upperRight( bBox )
  lly = yCoord( ll )
  ury = yCoord( ur )
  ury - lly ) ; let
=> 250
```

- The local variables *ll*, *ur*, *lly*, and *ury* are initialized to *nil*.
- The return value is the *ury-lly*.

The *let* Function

You can freely nest *let* statements.

You can access the value of a variable any time from anywhere in your program. SKILL transparently manages the value of a variable like a stack. Each variable has a stack of values.

- The current value of a variable is simply the top of the stack.
- Assigning a value to a variable changes only the top of the stack.

Whenever the flow of control enters a *let* function, SKILL pushes a temporary value onto the value stack of each variable in the local variable list. The local variables are normally initialized to *nil*.

When the flow of control exits a *let* function, SKILL pops the top value of each variable in the local variable list. If a variable with the same name existed outside of the *let* it will have the same value that it had before the *let* was executed.

Two Common *let* Errors

The two most common *let* errors are:

- Including whitespace after *let*.
  The error message depends on whether the return value of the *let* is assigned to a variable.

```
let ( ( ll ur lly ury )
   ll   = lowerLeft( bBox )
   ur   = upperRight( bBox )
   lly = yCoord( ll )
   ury = yCoord( ur )
   ury - lly
) ; let
*Error* let: too few arguments (at least 2 expected, 1 given)
```

- Omitting the list of local variables.
  The error messages vary and can be obscure and hard to decipher.

```
let(
   ll   = lowerLeft( bBox )
   ur   = upperRight( bBox )
   lly = yCoord( ll )
   ury = yCoord( ur )
   ury - lly
) ; let
```

Defining SKILL Functions

Use the *procedure* function to associate a name with a group of SKILL expressions. The name, a list of arguments, and a group of expressions compose a SKILL function declaration.

- The name is known as the function name.
- The group of statements is the function body.

Example

```
bBox = list( 100:200 350:450 )                    Write global variable

procedure( TrBBoxHeight( )                        Define function
    let( ( ll ur lly ury )                        Local variables
        ll      = lowerLeft( bBox )               Read global variable
        ur      = upperRight( bBox )              Read global variable
        lly     = yCoord( ll )
        ury     = yCoord( ur )
        ury - lly                                 Return value
    ) ; let
) ; procedure

bBoxHeight = TrBBoxHeight()                       Invoke function
```

Developing a SKILL Function 6-11

The *procedure* Function

Local Variables

You can use the *let* syntax function to declare the variables *ll*, *ur*, *ury*, and *lly* to be local variables. The arguments presented in an argument list of a procedure definition are also local variables. All other variables are global variables.

Global Variable

The *bBox* variable is a global variable because it is neither an argument nor a local variable.

Return Value

The function returns the value of the last expression evaluated. In this example, the function returns the value of the *let* expression, which is the value of the *ury-lly* expression.

Three Common *procedure* Errors

The three most common *procedure* errors are:

- Including whitespace after the *procedure* function.

```
procedure ( TrBBoxHeight( )
   ...
) ; procedure
*Error* procedure: too few arguments (at least 2 expected, 1 given)
```

- Including whitespace after your function name.

```
procedure( TrBBoxHeight ( )
   ...
) ; procedure
*Error* procedure: illegal formal list - TrBBoxHeight
```

- Omitting the argument list.

```
procedure( TrBBoxHeight
   ...
) ; procedure
*Error* procedure: illegal formal list - TrBBoxHeight
```

Notice that the error message refers to the *procedure* function. The arguments to *procedure* are the function name, the argument list of the function, and the expression composing the body of the function.

- Including white space after the *procedure* function
- Including white space after the function name
- Omitting the argument list

Defining Required Function Parameters

The fewer global variables you use, the more reusable your code is.

Turn global variables into required parameters. When you invoke your function, you must supply a parameter value for each required parameter.

In our example, make *bBox* be a required parameter.

```
procedure( TrBBoxHeight( bBox )
  let( ( ll ur lly ury )
    ll   = lowerLeft( bBox )
    ur   = upperRight( bBox )
    lly  = yCoord( ll )
    ury  = yCoord( ur )
    ury - lly
  ) ; let
) ; procedure
```

To execute your function, you must provide a value for the *bBox* parameter.

```
bBoxHeight = TrBBoxHeight( list( 50:150 200:300 ) )
  => 150
```


You can use the *let* function to declare the variables *ll*, *ur*, *ury*, and *lly* to be local variables.

In the example, the variable *bBox* occurs both as a global variable and as a formal parameter.

Here's how SKILL keeps track. When you call the *TrBBoxHeight* function, the SKILL Evaluator:

- Saves the current value of *bBox*.
- Evaluates *list(50:150 200:300)* and temporarily assigns it to *bBox*.
- Restores the saved value of *bBox* when the *TrBBoxHeight* function returns.

Defining Optional Function Parameters

Include @**optional** before any optional function parameters.

Use parentheses to designate a default value for an optional parameter.

```
procedure( TrOffsetBBox( bBox @optional (dx 0)(dy 0))
  let( ( llx lly urx ury )
    ...
    list( ;;; return the new bounding box
      llx+dx:lly+dy
      urx+dx:ury+dy
      )
  ) ; let
) ; procedure
```

To call the *TrOffsetBBox* function, specify the required and the optional arguments as follows:

- *TrOffsetBBox(someBBox)*
  dx and *dy* default to *0*.
- *TrOffsetBBox(someBBox* 0.5)
  dy defaults to *0*.
- *TrOffsetBBox(someBBox 0.5 0.3)*.

The procedure above takes as input the original bounding box and the desired change in the *x* and *y* directions. The procedure returns a new bounding box that has been offset.

Unless you provide a default value for an optional parameter in the procedure declaration, the default value will be *nil*. This can lead to an error if *nil* is not appropriate for the operations executed using the parameter. This is the case for the procedure shown here.

You provide a default value for a parameter using a list (<*parameter*> <*default value*>).

Defining Keyword Function Parameters

Include @**key** before keyword function parameters.

Use parentheses to designate a default value for a keyword parameter.

```
procedure( TrOffsetBBox( bBox @key (dx 0)(dy 0))
  let( ( llx lly urx ury )
    ...
    list(
      llx+dx:lly+dy
      urx+dx:ury+dy
      )
  ) ; let
) ; procedure
```

To call the *TrOffsetBBox* function, specify the keyword arguments as follows:

- *TrOffsetBBox(someBBox ?dx 0.5 ?dy 0.4)*
- *TrOffsetBBox(someBBox ?dx 0.5)*
  dy defaults to *0*.
- *TrOffsetBBox*(*someBBox ?dy 0.4)*
  dx defaults to *0*.

Developing a SKILL Function 6-19

Keyword parameters free you from the need for a specific order for your parameters. This can be very valuable when you have a significant number of optional parameters. When you must preserve the order for optional parameters you need to supply values for all parameters preceding the one you actually want to set.

Keyword parameters are always syntactically optional. Care should be taken however that the procedure will give correct results without a parameter specified. If you cannot do this check each parameter for an acceptable value and emit an error message when execution of the function will not yield a sensible answer.

As with all parameters, unless you provide a default value for a keyword parameter the default value will be *nil*.

Collecting Function Parameters into a List

An **@rest** argument allows your procedure to receive all remaining arguments in a list. Use a single **@rest** argument to receive an indeterminate number of arguments.

Example

The *TrCreatePath* function receives all of the arguments you pass in the formal argument *points*.

```
procedure( TrCreatePath( @rest points )
  printf(
    "You passed these %d arguments: %L\n"
    length( points ) points
  )
) ; procedure
```

The *TrCreatePath* function simply prints a message about the list contained in *points*.

To call the *TrCreatePath* function, specify any number of points.

```
TrCreatePath( 3:0 0:4 )
TrCreatePath( 3:0 0:4 0:0 )
```


The *@rest* parameter structure allows you to specify input parameters even when you do not know how many input elements the user will want to operate on. This type of argument specification is perfect for functions like *dbCreatePath* or *dbCreatePolygon*. It is also very convenient for inputting the elements of an arbitrary length list.

As written, the *TrCreatePath* function doesn't really create a path. You can make it create a path in a cellview by passing the list *points* to the *dbCreatePath* function.

Calling a SKILL Function with a list of Arguments

Sometimes you may have a list of arguments, and want to call an existing function with that list of arguments.

For example, you may have a list of numbers that you wish to add up:

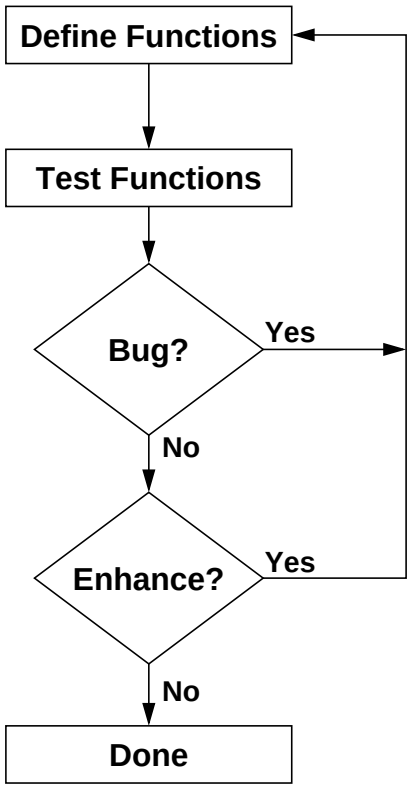
```
nums=list( 1 2 3 4 5 6 )
```

The *plus* function (which corresponds to the + operator) takes a variable number of arguments, but we can't just do *plus(nums)*.

You can use the *apply* function to call a function, with a list of arguments

```
apply( 'plus nums ) => 21
```

SKILL Development Cycle



Developing a SKILL Function 6-25

SKILL allows you to redefine functions without exiting the Design Framework II environment. A rapid edit and run cycle facilitates bottom-up software development.

Defining SKILL Functions

Launching an Editor from the Design Framework II Environment

The *edit* function launches the editor from the Design Framework II environment. When you use a path name, the *edit* function uses the UNIX path variable.

```
edit( "~/SKILL/YourCode.il" )
```

Before you use the *edit* function, set the editor global to a string that contains the UNIX shell command to launch your editor.

```
editor = "xterm -e vi"
```

xterm -e gvim

If you install the SKILL Debugger before you load your code, you can edit the source code for one of your functions by passing the function name to the *edit* function.

```
edit( TrExampleFunction )
```

Testing SKILL Functions

You can initially test your application SKILL functions directly in the CIW. Then, you can create a user interface for your application.

Loading Source Code

The *load* function evaluates each expression in a given SKILL source code file.

You use *load* function to define SKILL functions and execute initialization expressions.

The *load* function returns *t* if all expressions evaluate without errors. Typical errors include:

- Syntax problems with a *procedure* definition.
- Attempts to *load* a file that does not exist.

Any error aborts the *load* function. The *load* function does not evaluate any expression that follows the offending SKILL expression.

When you pass a relative pathname to the *load* function, the *load* function resolves it in terms of a list of directories called the SKILL path.

You usually establish the SKILL path in your *.cdsinit* file by using the *setSkillPath* or *getSkillPath* functions.

The *loadi* Function

The *loadi* function also loads the SKILL code contained in a file. The difference in this function is that it continues despite errors completing all the statements within the file. In fact, no error messages are passed to the user and *loadi* always completes with a return value of *t*. This function is very valuable when the statements within a file are independent, for examples statements that set the bindkeys for a session.

The SKILL Path Functions

- The *getSkillPath* function returns the current list of directories in search order.
- The *setSkillPath* function sets the path to a list of directories.

Pasting Source Code into the CIW

The *load* function accesses the current version of a file. Any unsaved edits are invisible to the *load* function.

Sometimes you want to define a function without saving the source code file.

You can paste source code into the CIW with the following procedure:

1. Use the mouse in your editor to select the source code.
2. Move the cursor over the CIW input pane.
3. Click the middle mouse button.
   Your selection is displayed in the CIW input pane in a single line.
4. Press Return.
   Your entire selection is displayed in the CIW output pane.

If you are using the *vi* editor, make sure line numbering is turned off. Otherwise, when you paste your code into the CIW, the line numbers become SKILL expressions in the virtual memory definition of your functions. To turn off line numbers, enter the following into your *vi* window:

```
:set nonumber
```

Make sure you select all the characters of the SKILL function definition. Be particularly careful to include the final closing parentheses.

If you paste the definition of a single SKILL function into the CIW, then the function name is the last word displayed in the CIW output pane.

Why is that?

Because the SKILL *procedure* function returns the function symbol.

Lab Overview

Lab 6-1 Developing a SKILL Function

- Develop a SKILL function, *TrBBoxArea*, to compute the area of a bounding box. The bounding box is a parameter.
- Model your SKILL function after the example SKILL function *TrBBoxHeight*.

Lab 6-2 Writing a function to postprocess results

Module Summary

In this module, we covered

- Using curly braces, { }, to group SKILL statements together
- Using the *let* function to declare local variables
- Declaring SKILL functions with the procedure function
  - Name
  - Arguments
  - Body
- The SKILL development cycle
- Maintaining SKILL code
  - The *edit* function and *editor* variable
  - The *load* function

Flow of Control

Module 7

October 5, 2005

Module Objectives

- Review relational operators
- Describe logical operators
- Examine branching statements
- Discuss several methods of iteration

Terms and Definitions

Iteration	To repeatedly execute a collection of SKILL expressions.

Relational Operators

Use the following operators to compare data values.

These operators all return *t* or *nil*.

Operator	Arguments	Function	Example	Return Value
<	numeric	lessp	3 < 5 3 < 2	t nil
<=	numeric	leqp	3 <= 4	t
>	numeric	greaterp		
>=	numeric	geqp		
==	numeric string list	equal	3.0 == 3 "abc" == "ABc"	t nil
!=	numeric string list	nequal	 "abc" != "ABc"	 t

Flow of Control 7-5

Use parentheses to control the order of evaluation. This example assigns *3* to *x*, returning *3*, and next compares *3* with *5*, returning *t*.

```
(x=3)<5 => t
```

SKILL generates an error if the data types are inappropriate. Error messages mention the function in addition to the operator.

```
1 > "abc"
*** Error in routine greaterp:
Message: *Error* greaterp: can't handle (1 > "abc")
```

Logical Operators

SKILL considers *nil* as FALSE and any other value as TRUE.

SKILL provides generalized boolean operators.

Operator	Arguments	Function	Example	Return Value
!	general	null	!3 !nil !t	nil t nil
&&	general	and	x = 1 y = 5 x < 3 && y < 4 x < 3 && 1/0 y < 4 && 1/0	5 nil *** Error nil
||	general	or	x < 3 || y < 4 x < 3 || 1/0 y < 4 || 1/0	t t *** Error

The && and || operators only evaluate their second argument if they must to determine the return result.

The *&&* and || operators return the value last computed.

Flow of Control 7-7

The && Operator

Evaluates its first argument. If it is *nil*, then *&&* returns *nil*.
The second argument is not evaluated.

If the first argument evaluates to non-*nil*, then *&&* evaluates the second argument. The && operator returns the value of the second argument.

The || Operator

Evaluates its first argument. If it is non-*nil*, then || returns the value of the first argument. The second argument is not evaluated.

If the first argument evaluates to *nil*, then the second argument is evaluated. The || operator returns the value of the second argument.

Using the *&&* and || Operators to Control Flow

You can use both the *&&* and || operators to avoid cumbersome *if* or *when* expressions.

Example of Using the || Operator

Suppose you have a default name, such as *"noName"* and a variable, such as *userName*. To use the default name if *userName* is *nil*, use this expression:

```
theUser = userName || "noName"
```

Branching

Branching Task	Function
Binary branching	if when unless
Multiway branching	case cond
Arbitrary exit	prog return

The *if* Function

Use the *if* function to selectively evaluate two groups of one or more expressions.

The selection is based on whether the condition evaluates to *nil* or non-*nil*.

- Use *if(exp ...)* instead of *if(exp != nil ...)*
- Use *if(!exp ...)* instead of *if(exp == nil ...)*

The return value of the *if* expression is the value last computed.

```
if( shapeType == "rect" then
  println( "Shape is a rectangle" )
  ++rectCount
else
  println( "Shape is not a rectangle" )
  ++miscCount
) ; if rect
```

Two Common *if* Errors

Two common *if* errors are:

- Including whitespace after *if*

```
if ( shapeType == "rect"
   ...
)
*** Error in routine if
Message: *Error* if: too few arguments ...
```

- Including a right parenthesis after the conditional expression

```
if( shapeType == "rect" )
***Error* if: too few arguments (at least 2 expected, 1 given)
```

Flow of Control 7-15

SKILL does most of its error checking during execution. Error messages involving *if* expressions can be obscure.

Remember

- To avoid whitespace immediately after the *if* syntax function.
- To place parentheses as follows:

```
if( ... then ... else ... )
```

Nested *if-then-else* Expressions

Be careful with parentheses. Comment the closing parentheses and indent consistently as in this example.

```
if( shapeType == "rect" then
  ++rectCount
else
  if( shapeType == "line" then
   ++lineCount
  else
   ++miscCount
  ) ; if line
) ; if rect
```

The *when* and *unless* Functions

Use the *when* function whenever you have only *then* expressions.

```
when( shapeType == "rect"
  println( "Shape is a rectangle" )
  ++rectCount
) ; when

when( shapeType == "ellipse"
  println( "Shape is a ellipse" )
  ++ellipseCount
) ; when
```

Use the *unless* function to avoid negating a condition.

```
unless(
  shapeType == "rect" || shapeType == "line"
  println( "Shape is miscellaneous" )
  ++miscCount
) ; unless
```


The *when* and *unless* functions both return the last value evaluated within their body or *nil*.

The *case* Function

The *case* function sequentially compares a candidate value against a series of target values. The target must be a value and cannot be a symbol or expression that requires evaluation.

When it finds a match, it evaluates the associated expressions and returns the value of the last expression evaluated.

```
case( shapeType
  ( "rect"                                    Candidate value
    ++rectCount                               Target
    println( "Shape is a rectangle" )
  )
  ( "line"                                    Target
    ++lineCount
    println( "Shape is a line" )
  )
  ( "label"                                   Target
    ++labelCount
    println( "Shape is a label" )
  )
  ( t                                         Catch all
    ++miscCount
    println( "Shape is miscellaneous" )
  )
) ; case
```


If you have expressions to evaluate when no target value matches the candidate value, include those expressions in an arm at the end. Use *t* for the target value for this last arm. If the flow of control reaches an arm whose target value is the *t* value, then SKILL unconditionally evaluates the expressions in the arm.

When target value of an arm is a list, SKILL searches the list for the candidate value. If the candidate value is found, all the expressions in the arm are evaluated.

```
case( shapeType
  ( "rect"
    ++rectCount
    println( "Shape is a rectangle" )
  )
  ( ( "label" "line" )
    ++labelOrLineCount
    println( "Shape is a line or a label" )
  )
  ( t
    ++miscCount
    println( "Shape is miscellaneous" )
  )
) ; case
```

The *cond* Function

Use the *cond* function when your logic involves multiway branching.

```
cond(
   ( condition1 exp11 exp12 ... )
   ( condition2 exp21 exp22 ... )
   ( condition3 exp31 exp32 ... )
   ( t expN1 expN2 ... )
) ; cond
```

The *cond* function

- Sequentially evaluates the conditions in each arm, until it finds one that is non-*nil*. It then executes all the expressions in the arm and exits.
- Returns the last value computed in the arm it executes.

The *cond* function is equivalent to

```
if       condition1  then exp11exp12 ...
else if  condition2  then exp21exp22 ...
else if  condition3  then exp31exp32 ...
...
else                 expN1expN2 ....
```


This example *TrClassify* function includes the *cond* function and the *numberp* function.

```
procedure( TrClassify( signal )
  cond(
    ( !signal nil )
    ( !numberp( signal ) nil )
    ( signal >= 0 && signal < 3 "weak" )
    ( signal >= 3 && signal < 10 "moderate" )
    ( signal >= 10 "extreme" )
    ( t  "unexpected" )
  ) ; cond
) ; procedure
```

The *numberp, listp, stringp,* and *symbolp* Functions

SKILL provides many functions that recognize the type of their arguments, returning *t* or *nil*. Traditionally, such functions are called *predicates*. Their names end in *"p"*.

Each function takes one argument, returning *t* if the argument is of the type specified, otherwise returning *nil*. See the Cadence online documentation to read more about this function.

Iteration

Iteration task	Function
Numeric range	for
List of values	foreach
While a condition is non-*nil*	while

Flow of Control 7-25

These functions repeat a statement block for a specific number of times, or through each element of a list, or until a certain condition is satisfied.

The *for* Function

This example adds the integers from *1* to *5* using a *for* function.

```
sum = 0
for( i 1 5
   sum = sum + i
   println( sum )
)
```

The *for* function increments the index variable by 1.

The *for* function treats the index variable as a local variable.

- Saves the current value of the index variable before evaluating the loop expressions.
- Restores the index variable to its saved value after exiting the loop.
- Returns the value *t*.

SKILL does most of its error checking during execution. Error messages about *for* expressions can be obscure. Remember

- The placement of the parentheses: *for(...)*.
- To avoid putting whitespace immediately after the *for* syntax function.

The only way to exit a *for* loop early is to call the *return* function. To use the *return* function, you must enclose the *for* loop within a *prog* expression. This example finds the first odd integer less than or equal to *10*.

```
prog( ( )
  for( i 0 10
    when( oddp( i )
      return( i )
    ) ; when
  ) ; for
) ; prog
```

The *foreach* Function

Use the *foreach* function to evaluate one or more expressions for each element in a list of values.

```
rectCount = lineCount = miscCount = 0
shapeTypeList = '( "rect" "polygon" "rect" "line" )

foreach( shapeType shapeTypeList
  case( shapeType
    ( "rect"    ++rectCount )
    ( "line"    ++lineCount )
    ( t         ++miscCount )
  ) ; case
) ; foreach

=> ( "rect" "polygon" "rect" "line" )
```

When evaluating a *foreach* expression, SKILL determines the list of values and repeatedly assigns successive elements to the index variable, evaluating each expression in the *foreach* body.

The *foreach* expression returns the list of values over which it iterates.

In the example,

- The variable *shapeType* is the index variable. Before entering the *foreach* loop, SKILL saves the current value of *shapeType*. SKILL restores the saved value after completing the *foreach* loop.
- The variable *shapeTypeList* contains the list of values. SKILL successively assigns the values in *shapeTypeList* to *shapeType*, evaluating the body of the *foreach* loop once for each separate value.
- The body of the *foreach* loop is a *case* statement.
- The return value of the *foreach* loop is the list contained in the *shapeTypeList* variable.

The *foreach mapcar* function

- Sometimes you want to transform a list into a new list
- This can be done by using a normal *foreach* and then collecting results as you go along, but that may be cumbersome, and you need to be careful about the order of the resulting list:

```
nums='(1 2 3 4 5 6)
squares=nil
foreach(num nums
  squares=cons(num*num squares)
  )
squares=reverse(squares)
```

- More convenient is to use the *foreach mapcar* function which allows a list to be built efficiently as you move along it. The list is created from the return value of the last statement in the body of the foreach:

```
squares=foreach(mapcar num nums
  num*num
)
```

There is no benefit in using *foreach mapcar* unless you use the return value of the function.

The *while* Function

Use the *while* function to execute one or more expressions repeatedly as long as the condition is non-*nil*.

Example of a *while* expression.

```
let( ( inPort nextLine )
  inPort = infile( "~/.cshrc" )
  when( inPort
    while( gets( nextLine inPort )
      println( nextLine )
    ); while
    close( inPort )
    inPort = nil
  ) ; when
) ; let
```

read the line from port and store it in variable

remember to close the file

The *prog* and *return* Functions

If you need to exit a collection of SKILL statements conditionally, use the *prog* function.

```
prog( ( local variables ) your SKILL statements )
```

Use the *return* function to force the *prog* function to immediately return a value. The *prog* function does not execute any more SKILL statements.

If you do not call the *return* function within the *prog* body, *prog* returns *nil*.

For example, the *TrClassify* function returns either *nil*, "*weak*", "*moderate*", "*extreme*", or "*unexpected*", depending on the *signal* argument. This *prog* example does not use any local variables.

```
procedure( TrClassify( signal )
  prog( ()
    unless( signal return( nil ))
    unless( numberp( signal ) return( nil ))
    when( signal >= 0 && signal < 3 return( "weak" ))
    when( signal >= 3 && signal < 10 return( "moderate" ))
    when( signal >= 10 return( "extreme" ))
    return( "unexpected" )
  ) ; prog
) ; procedure
```


Use the *prog* function and the *return* function to exit early from a *for* loop. This example finds the first odd integer less than or equal to *10*.

```
prog( ( )
  for( i 0 10
    when( oddp( i )
      return( i )
    ) ; when
  ) ; for
) ; prog
```

A *prog* function can also establish temporary values for local variables. All local variables receive temporary values initialized to *nil*.

The current value of a variable is accessible at any time from anywhere.

The SKILL Evaluator transparently manages a value slot of a variable as if it were a stack.

- The current value of a variable is simply the top of the stack.
- Assigning a value to a variable changes only the top of the stack.

Whenever your program invokes the *prog* function, the SKILL Evaluator pushes a temporary value onto the value stack of each variable in the local variable list.

When the flow of control exits, the system pops the temporary value off the value stack, restoring the previous value.

Lab Overview

Lab 7-1 Exploring Flow of Control

You write a SKILL function to validate 2-dimensional points.

Lab 7-2 More Flow of Control

You write a SKILL function to compare 2-dimensional points.

Lab 7-3 Controlling Complex Flow

You write a SKILL function to validate a bounding box.

Lab 7-4 Writing a function to plot a list of nets

You write a SKILL function to plot a variable length list of signals.

Module Summary

In this module, we covered

Category	Function
Relational Operators	< <= > >= == !=
Logical Operators	! && ||
Branching	if when unless case cond
Iteration	for foreach while
Miscellaneous	prog return

File I/O

Module 8

October 5, 2005

Module Objectives

- Write UNIX text files.
- Read UNIX text files.
- Open a text window.

Writing Data to a File

Instead of displaying data in the CIW, you can write the data to a file.

Both *print* and *println* accept a second, optional argument that must be an output port associated with the target file.

Use the *outfile* function to obtain an output port for a file. Once you are finished writing data to the file, use the *close* function to release the port.

This example uses the *for* function to execute the *println* function iteratively. The *i* variable is successively set to the values 1,2,3,4, and 5.

```
myPort = outfile( "/tmp/myFile" ) => port:"/tmp/myFile"
for( i 1 5
  println( list( "Number:" i) myPort )
  ) => t
close( myPort ) => t
myPort = nil
```

After you execute the *close* function, */tmp/myFile* contains the following lines:

```
("Number:" 1)
("Number:" 2)
("Number:" 3)
("Number:" 4)
("Number:" 5)
```


The *print* and *println* Functions

Notice how SKILL displays a port in the CIW.

```
myPort = outfile( "/tmp/myfile" )
port:"/tmp/myfile"
```

Use a full pathname with the *outfile* function. Keep in mind that *outfile* returns *nil* if you do not have write access to the file, or if you cannot create the file in the directory you specified in the pathname.

The *print* and *println* functions raise an error if the port argument is *nil*. Observe that the type template uses a *p* character to indicate a port is expected.

```
println( "Hello" nil )
*** Error in routine println:
Message: *Error* println: argument #2 should be an I/O port
  (type template = "gp")
```

The *close* Function

The *close* function does not update your port variable after it closes the file. To facilitate robust program logic, set your port variable to *nil* after calling the *close* function.

Writing Data to a File (continued)

Unlike the *print* and *println* functions, the *printf* function does not accept an optional port argument.

Use the *fprintf* function to write formatted data to a file. The first argument must be an output port associated with the file.

Example

```
myPort = outfile( "/tmp/myFile" )
for( i 1 5
  fprintf( myPort "Number: %d\n" i )
) ; for
close( myPort )
```

The example above writes the following data to */tmp/myFile*:

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
```

Reading Data from a File

Use the *infile* function to obtain an input port on a file.

The *gets* function reads the next line from the file.

This example prints every line in *~/.cshrc* to the CIW.

```
let( ( inPort nextLine )
  inPort = infile( "~/.cshrc" )
  when( inPort
    while( gets( nextLine inPort )
      println( nextLine )
    ); while
    close( inPort )
  ) ; when
) ; let
```

File I/O 8-9

The *infile* Function

The *gets* Function

The *gets* function reads the next line from the file. The arguments of the *gets* function are the:

- Variable that receives the next line.
- Input port.

The *gets* function returns the text string or returns *nil* when the end of file is reached.

The *when* Function

The first expression within a *when* expression is a condition. SKILL evaluates the condition. If it evaluates to a non-*nil* value, then SKILL evaluates all the other expressions in the *when* body. The *when* function returns either *nil* or the value of the last expression in the body.

The *while* Function

SKILL repeatedly evaluates all the expressions in the *while* body as long as the condition evaluates to a non-*nil* value. The *while* function returns *nil*.

The *fscanf* Function

The *fscanf* function reads data from a file according to conversion control directives.

It's useful for reading csv file or numbers into variables in .ocn file

The arguments of *fscanf* are

- The input port
- The conversion control string
- The variable(s) that receive(s) the matching data values.

The *fscanf* function returns the number of data items matched.

This example prints every word in *~/.cshrc* to the CIW.

```
let( ( inPort word )
  inPort = infile( "~/.cshrc" )
  when( inPort
    while( fscanf( inPort "%s" word )
      println( word )
    ); while
    close( inPort )
  ) ; when
) ; let
```

fscanf(port "%f %f" s d)

The format directives commonly found include the ones in this table.

Format Specification	Data Type	Scans Input Port
%d	integer	for next integer
%f	floating point	for next floating point
%s	text string	for next text string

The following is an example of output from the *~/CDS.log*.

```
\o "#.cshrc"
\o "for"
\o "Solaris"
\o "#Cadence"
\o "Training"
\o "Database"
\o "setup"
\o "for"
\o "the"
\o "97A"
\o "release"
\o "#"
```

Opening a Text Window

Use the *view* function to display a text file in a read-only window.

```
view( "~/SKILL/.cdsinit" ) => window:5
```

The *view* function is very useful for displaying a report file to the user.

The *view* function resolves a relative pathname in terms of the SKILL path. This is a list of directories you can establish in your *.cdsinit*.

See Module 6, *Developing a SKILL Function*, for specifics on the SKILL path.

To select text from your *CDS.log* file, try the following:

```
view(
  "~/CDS.log" ;;; pathname to CDS.log
  nil           ;;; default location
  "Log File"    ;;; window title
  t             ;;; auto update
) => window:6
```

File I/O 8-13

The *view* Function

The *view* function takes several optional arguments.

Argument	Status	Type	Meaning
file	required	text	Pathname
winSpec	optional	bounding box/*nil*	Bounding box of the window. If you pass *nil*, the default position is used.
title	optional	text	The title of the window. The default is the value of the file parameter.
autoUpdate	optional	*t/nil*	If *t*, then the window updates for each write to the file. The default is *nil*.
appName	optional	text	The Application Type for this window. The default is *"Show File"*.
help	optional	text	Text string for online help. The default means no help is available.

Lab Overview

Lab 8-1 Writing Data to a File

Lab 8-2 Reading Data from a Text File

Lab 8-3 Writing Output to a File

Write a function *TrPrintNets* based on your *TrPlotNets* function from the last module.

Module Summary

In this module, we covered

- Writing text data to a file by using
  - The *outfile* function to obtain an output port on a file.
  - An optional output port parameter to the *print* and *println* functions.
  - A required port parameter to the *fprintf* function.
  - The *close* function to close the output port.
- Reading a text file by using
  - The *infile* function to obtain an input port.
  - The *gets* function to read the file a line at a time.
  - The *fscanf* function to convert text fields upon input.
  - The *close* function to close the input port.

cadence®

Advanced Analyses with OCEAN

Module 9

October 5, 2005

Module Objectives

- Parametric analysis in OCEAN
- Monte Carlo analysis
- Corners analysis
- ADE Optimizer
- Distributed processing

Terms and Definitions

Blocking non-blocking semi-blocking	Simulation modes related to distributed processing. *Blocking mode*—OCEAN waits for a simulation job to complete before proceeding. *Non-blocking mode*—OCEAN submits the simulation job but does not wait for it to complete before continuing the OCEAN script. *Semi-blocking mode*—The jobs are submitted as with non-blocking mode, but the OCEAN script contains code requiring it to wait for a set of specified jobs to complete.
DCF	Design customization file
Distributed processing	A system for running parallel simulations distributed among many machines
LBS	Load Balancing Software, a simple load-balancing tool built into the Cadence environment
Optimizer	A tool within the Virtuoso® Analog Design Environment that lets you adjust circuit parameters to achieve desired simulation goals
PCF	Process customization file

Parametric Analysis in OCEAN

You can run parametric analysis in OCEAN.

- The parametric analysis commands can be saved to an OCEAN script when you select **Tool—Save Script** in the parametric analysis tool.
  - The *paramAnalysis()* and *paramRun()* commands are added to the script.
  - The *run()* command will not run a parametric analysis, just a single run.

  Example:

```
paramAnalysis("CAP" ?start 0.1p ?stop 0.5p ?step 0.1p)
paramRun()
```

  Runs a parametric analysis on the CAP variable.

- You can nest parametric analyses for multiple variables.

  Example:

```
paramAnalysis( "rl" ?start 200 ?stop 600 ?step 200
     paramAnalysis( "rs" ?start 300 ?stop 700 ?step 200 ))
paramRun()
```

  Runs a parametric analysis on **rs** for each value of **rl**, yielding a nested parametric analysis.

For example, if you add the following to your OCEAN script:

```
paramAnalysis("CAP" ?start 0.1p ?stop 0.5p ?step 0.1p)
paramRun()
```

This example will run a parametric analysis on the *CAP* variable.

You can set up multiple parametric analyses and only simulate the variables desired, as in this example:

```
paramAnalysis( "rs" ?start 200 ?stop 1000 ?step 200 )
paramAnalysis( "rl" ?start 200 ?stop 600 ?step 200 )
paramRun( 'rs)
```

This example runs a parametric analysis on the *rs* design variable only.

You can run nested parametric analyses with multiple variables by setting up an OCEAN script with syntax similar to the following example:

```
paramAnalysis( "rl" ?start 200 ?stop 600 ?step 200
paramAnalysis( "rs" ?start 300 ?stop 700 ?step 200 ))
paramRun()
```

This example runs a parametric analysis on *rl* for each value of *rs*, yielding a nested parametric analysis.

Parametric Analysis in OCEAN (continued)

You can also use normal SKILL loops to run parametric analyses.

```
mytemp = list( 0.0 50.0 100.0)
ac(1 10M 20)
  foreach(val mytemp
    desVar("temp" val)
      resultsDir( sprintf(nil "./psf/temp=%g" val))
      run()
      )
  )
```


You can run customized parametric analysis by using SKILL loop functions like *foreach* or *while.* Using SKILL functions to set up more complicated analyses gives you the ability to run simulations of parameter sets or customized corners analysis.

The SKILL command *sprintf* formats its arguments, resulting in a string rather than printing to the screen or a file.

Accessing Results from Spectre Nested Sweeps

If you have run the Virtuoso® Spectre® Circuit Simulator in standalone mode, using Spectre's *sweep* statement as follows:

```
parameters p1=1 p2=2 p3=3 p4=4
r1 1 2 resistor r=p1
r2 2 3 resistor r=p2
r3 3 4 resistor r=p3
r4 4 0 resistor r=p4
v1 1 0 vsource dc=1
sweep1 sweep param=p1 start=1 stop=2 step=1 {
    sweep2 sweep param=p2 start=1 stop=2 step=1 {
       sweep3 sweep param=p3 start=1 stop=2 step=1 {
          sweep4 sweep param=p4 start=1 stop=2 step=1 {
             dcOp dc
          }
       }
    }
}
```

then you may then want to select results from this run.

You may find out the parameters being swept with *sweepNames()*. You can peel off the outer layer of the sweep with the second argument to *selectResult().*

```
selectResult( 'dcOp 2.0 )
```

Or you could use the *value* function to access an inner sweep:

```
value( value(v("2") 'p1 2.0) 'p2 1.0 )
```

Advanced Analyses with OCEAN 9-9

sweepNames() in the above example would return *("p1" "p2" "p3" "p4").*

Monte Carlo Analysis

Spectre includes a built-in Monte Carlo analysis as part of the simulator.

- Simplified methodology for specifying statistics, correlation, and mismatch.
- All runs from a single netlist.
- Online help available with the following command in a UNIX window:

```
spectre -h montecarlo
```

OCEAN supports specific Monte Carlo commands.

Monte Carlo in OCEAN

You can run Monte Carlo analysis in OCEAN.

- The Monte Carlo analysis commands are saved to a separate OCEAN script when you select **Session—Save Ocean Script** in the ADE Monte Carlo window.
- OCEAN Monte Carlo analysis works with Spectre and socket simulators.

Example

```
monteCarlo( ?numIters "3" ?startIter "1"
    ?analysisVariation "Process Only" ?sweptParam "None"
    ?sweptParamVals "27" ?saveData t
    ?nomRun nil ?append nil)
monteExpr( "bw" "bandwidth(VF(\"OUT\") 3 'low')" )
monteExpr( "phase" "value(phase(VF(\"OUT\")) 100000)" )
monteExpr( "db20" "value(dB20(VF(\"OUT\")) 100000)" )
monteRun()
monteResults()
```

Monte Carlo OCEAN Commands

- Set up Monte Carlo run, saving family plots (Spectre only):

```
monteCarlo( ?numIters "3" ?startIter "1"
     ?analysisVariation "Process Only" ?sweptParam "None"
     ?sweptParamVals "27" ?saveData t
     ?nomRun nil ?append nil)
```

- Print to the screen the currently defined Monte Carlo analysis, including all expressions that are defined:

```
monteDisplay()
```

- Set up the Monte Carlo scalar expressions that will be used to create the histogram file:

```
monteExpr( "bw" "bandwidth(VF(\"OUT\") 3 \"low\")" )
```


If you want to use custom calculator functions in your *monteExpr()* functions, access SourceLink® online customer support and look up solution 11007118.

Monte Carlo OCEAN Commands (continued)

- Run all the simulations specified by the *monteCarlo* statement:

  ```
  monteRun()
  ```

- Initialize Monte Carlo data analysis tools:

  ```
  monteResults()
  ```

  - A statistical analysis menu is added to the waveform window (if AWD).
  - The menu items are equivalent to those found on the Monte Carlo Results menu in ADE.

- Use *selectResult* to plot or post-process family curves saved by Monte Carlo:

  ```
  selectResult( 'analysis )
  ```

Monte Carlo Plotting Functions

- Plot a histogram of a statistical measurement:

```
histogram(t_monteExprName ?type s_type ?numBins
     x_numBins ?density b_density)
```

  This command plots to an individual subwindow. The value of the *s_type* argument determines the style of the line. Setting *b_density* to *t* causes the histogram command to plot a smooth distribution curve for the data.

  Valid values for *s_type* are: *'standard*, *'passFail*, *'cumulativeLine*, *'cumulativeBox*.

- Plot a scatter plot of a pair of measurements against each other:

```
scatterPlot(t_monteExprName_X t_monteExprName_Y
  ?bestFit b_bestFit )
```

  Tightly correlated parameters show linear relationships. If *?bestFit* is passed with *t* as the argument, a best fit line is drawn.

Other Monte Carlo results functions:

yield—displays yield of Monte Carlo expressions

iterVsValue—shows the value of each Monte Carlo expression for each iteration

correlationTable—shows how each Monte Carlo expression is correlated with each of the others

Corners Analysis

In real life, manufacturing processes are subject to tolerances. The process variables can fluctuate randomly around their ideal values.

The combined random variation for all the components results in an uncertain yield for the circuit as a whole.

Corners analysis

- Looks at the performance outcomes generated from the most extreme variations expected in the
  - Process
  - Design variables
  - Temperature
- Allows you to determine whether circuit performance specifications will be met, even when the random process variations are combined in their most unfavorable patterns.
- Monte Carlo will give a better representation of the spread. Corners tends to be a more pessimistic approach. However, statistical models are more complex and are not always available.

Setting Up Corners Analysis

Files associated with setting up a corners analysis are:

- Process customization file (PCF)—ends in *.pcf*
  - Adds the name of a new process to the corners tool GUI
  - Defines the basic set of corners
- Design customization file (DCF)—ends in *.dcf*
  - Adds design specific variables and measurements to the Corners tool GUI

You can load the files explicitly via the GUI or with the *.cdsinit* file:

```
loadDcf( "./CORNERS/multipleModelLib.dcf" )
loadPcf( "./CORNERS/singleModelLib.pcf" )
```


Files associated with setting up a corners analysis are:

- **Process customization files (PCFs)**: Define processes, groups, variants, and corners that are shared by an entire organization. PCFs are usually created by a process engineer or process group.
- **Design customization files (DCFs)**: Contain definitions that are used for a particular design or for several designs within a design group. DCFs are usually created by designers, who use the DCFs to add design specific information to the general information provided in PCFs.

You can use any of the corners SKILL API commands in either the PCFs or DCFs.

- Commands used to define the process, the corners, and the corner variables are customarily placed in the PCF.
- Commands used to specify design variables and measurements, because they are design specific, are usually placed in the DCF.

To debug PCFs and DCFs, consider using OCEAN. The feedback that the corners tool provides is limited, but OCEAN provides more detailed feedback that makes it easier to find and correct errors.

Example PCF and DCF Files

```
corAddProcess( "mySingle" "./cornersDir" "Single Model Library" )
corSetModelFile( "mySingle" "mySingle.scs" )
corAddProcessVar( "mySingle" "CAP" )

corAddCorner( "mySingle" "fastslow" )

corAddCorner( "mySingle" "typtyp" )

corAddCorner( "mySingle" "slowfast" )

corAddCorner( "mySingle" "fastfast" )

corAddCorner( "mySingle" "slowslow" )

corAddMeas( "PhaseMargin" )
corSetMeasExpression( "PhaseMargin" "phaseMargin(VF('/out'))" )
corSetMeasTarget( "PhaseMargin" 28 )
corSetMeasEnabled( "PhaseMargin" t )
corSetMeasGraphicalOn( "PhaseMargin" t )
corSetMeasTextualOn( "PhaseMargin" t )

corAddMeas( "Max DC Gain" )
corSetMeasExpression( "Max DC Gain" "ymax(mag(VF('/out')))" )
corSetMeasTarget( "Max DC Gain" 2.8 )
corSetMeasEnabled( "Max DC Gain" t )
corSetMeasGraphicalOn( "Max DC Gain" t )
corSetMeasTextualOn( "Max DC Gain" nil )

corAddMeas( "Output" )
corSetMeasExpression( "Output" "mag(VF('/out'))" )
corSetMeasEnabled( "Output" t )
corSetMeasGraphicalOn( "Output" t )
corSetMeasTextualOn( "Output" nil )

corAddMeas( "Phase" )
corSetMeasExpression( "Phase" "phase(VF('/out'))" )
corSetMeasEnabled( "Phase" t )
corSetMeasGraphicalOn( "Phase" t )
corSetMeasTextualOn( "Phase" nil )
```

mySingle.pcf

```
corAddProcess( "mySingle" "./cornersDir" "Single Model Library" )
corSetModelFile( "mySingle" "mySingle.scs" )
corAddProcessVar( "mySingle" "CAP" )

corAddCorner( "mySingle" "fastslow" )
corSetCornerRunTempVal( "mySingle" "fastslow" 27 )
corSetCornerVarVal( "mySingle" "fastslow" "CAP" "600f" )

corAddCorner( "mySingle" "typtyp" )
corSetCornerRunTempVal( "mySingle" "typtyp" 27 )
corSetCornerVarVal( "mySingle" "typtyp" "CAP" "800f" )

corAddCorner( "mySingle" "slowfast" )
corSetCornerRunTempVal( "mySingle" "slowfast" 27 )
corSetCornerVarVal( "mySingle" "slowfast" "CAP" "400f" )

corAddCorner( "mySingle" "fastfast" )
corSetCornerRunTempVal( "mySingle" "fastfast" -55 )
corSetCornerVarVal( "mySingle" "fastfast" "CAP" "250p" )

corAddCorner( "mySingle" "slowslow" )
corSetCornerRunTempVal( "mySingle" "slowslow" 125 )
corSetCornerVarVal( "mySingle" "slowslow" "CAP" "950f" )

corAddMeas( "PhaseMargin" )
corSetMeasExpression( "PhaseMargin" "phaseMargin(VF('/out'))" )
corSetMeasTarget( "PhaseMargin" 28 )
corSetMeasEnabled( "PhaseMargin" t )
corSetMeasGraphicalOn( "PhaseMargin" t )
corSetMeasTextualOn( "PhaseMargin" t )

corAddMeas( "Max DC Gain" )
corSetMeasExpression( "Max DC Gain" "ymax(mag(VF('/out')))" )
corSetMeasTarget( "Max DC Gain" 2.8 )
corSetMeasEnabled( "Max DC Gain" t )
corSetMeasGraphicalOn( "Max DC Gain" t )
corSetMeasTextualOn( "Max DC Gain" nil )

corAddMeas( "Output" )
corSetMeasExpression( "Output" "mag(VF('/out'))" )
corSetMeasEnabled( "Output" t )
corSetMeasGraphicalOn( "Output" t )
corSetMeasTextualOn( "Output" nil )

corAddMeas( "Phase" )
corSetMeasExpression( "Phase" "phase(VF('/out'))" )
corSetMeasEnabled( "Phase" t )
corSetMeasGraphicalOn( "Phase" t )
corSetMeasTextualOn( "Phase" nil )
```

mySingle.dcf

Corners Analysis OCEAN Commands

- Select one of the predefined processes:

```
selectProcess( "myProcess" )
```

- Set the design variable value for the specified corner:

```
cornerDesVar( "fast" "CAP" "1p" )
```

- Set the analysis temperature (in degrees Celsius) to be used for that corner:

```
cornerRunTemp( "slow" "75" )
```

- Run all corners simulation defined in the *.pcf* or *.dcf* files and selected by the *selectProcess* command:

```
cornerRun()
```

- Plot or print the predefined measurements:

```
cornerMeas()
```

Using the ADE Optimizer from OCEAN

- The Optimizer is a tool within ADE that lets you adjust parameters to achieve simulation goals.
- The Optimizer has a menu selection under **Session—Save OCEAN Script** to allow OCEAN scripts to be written for batch optimization jobs.
- The following OCEAN commands have been added to allow for optimization simulations:

```
optimizeAlgoControl
optimizeGoal
optimizeRun
optimizeVar
```

Optimizer OCEAN Commands

- Allow user to change the internal algorithm controls:

```
optimizeAlgoControl( ?relDelta .05 )
```

- Set up the goals for optimization:

```
optimizeGoal("bandwidth" 'bandwidth(v("\out") 3 "low")
'le 18M 15M )
```

- Run the Optimizer given the goals specified with the *optimizeGoal* command:

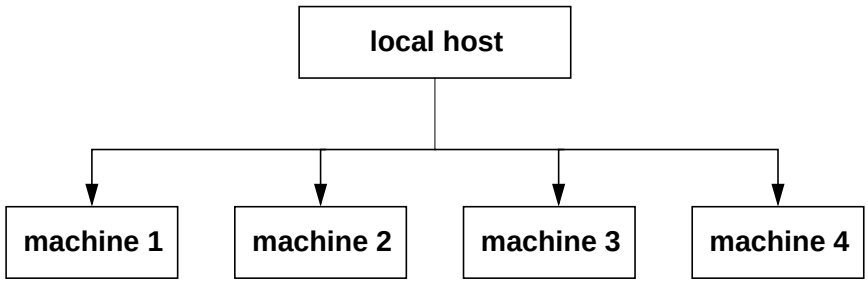
```
optimizeRun(?goals '("bandwidth" "slewrate" ) ?vars '("r" "w")
?iter 5)
```

- Specify the design variables to be used with optimization:

```
optimizeVar( "r" 1k 500 2k )
```

Distributed Processing

- Distributes multiple simulations across several systems on the network
- Efficient use of existing resources
- Big productivity gain for large jobs



Advanced Analyses with OCEAN 9-33

Distributed processing takes multiple simulation jobs like corner analysis and Monte Carlo or single simulation jobs and distributes them across many different machines. Distributed processing can be a big time saver for multiple long simulations. Users can line up a number of jobs in a queue for multiple simulation, or choose to send the simulations to a list of specific machines. Users can also specify when to start the simulation runs.

Cadence provides a basic system for job distribution management. This system also interfaces with the Platform LSF load-sharing facility from Platform Computing, Inc.

After the simulations have ended it is necessary to select results of the job you want to plot. They are identified by the job numbers. This is necessary because there may be several jobs running simultaneously. You need to select the results that corresponds to the respective job number.

Load Balancing

- Cadence provides a simple default load-balancing tool called Load Balancing Software (LBS) that manages which job goes where and when.
- Cadence also provides interfaces to other load-balancing tools:
  - Platform LSF load-sharing facility from Platform Computing, Inc. (http://www.platform.com)
  - Support for the Sun Grid Engine in distributed processing command-line mode

The Cadence distributed processing capability comes with a default load balancing tool (Load Balancing Software, or LBS). This is a simple tool. It manages which job goes where and when. It handles the load balancing for the queues by sending jobs to the first host in the queue until the maximum number of jobs for that host is reached. Then it sends the jobs to the next host in the queue and so on. It also sends e-mail notifications to users when a job is terminated.

LBS also interfaces with Platform LSF, which does more sophisticated load balancing, sending jobs to hosts dependent upon their loads.

For a workaround using Sun Grid Engine, see SourceLink solution 11015590.

Distributed Processing Commands

The following commands are used to run distributed processing in OCEAN.

- Enable distributed processing:

  ```
  hostMode( "distributed" )
  ```

- Disable distributed processing:

  ```
  hostMode( "local" )
  ```

- Submit a job:

  ```
  run(?jobPrefix "jobName")
  ```

  The *run* function returns the name of the submit job, which can be used in some of the functions on the following slides.

- To access results:

  ```
  openResults( jobName )
  ```

Note: This is equivalent to calling *openResults* with a results directory path.

Distributed Processing Commands (continued)

- Enable distributed processing on the current host:

```
hostMode( "distributed" )
```

- Display the name, host, and queue for all pending tasks:

```
monitor( ?taskMode t )
```

- Halt execution of a job or series of jobs:

```
suspendJob( 'job001 )
```

- Resume execution of a previously suspended job or series of jobs:

```
resumeJob( 'job001 )
```

- Abort processing of a job or series of jobs:

```
killJob( 'job001 )
```

- Remove a job or series of jobs from queue:

```
deleteJob( 'job001 )
```

- This command will block execution of a script until the specified jobs complete:

```
wait( 'job001 )
```


Job Identifier for Distributed Processing Functions

Several distributed processing functions (e.g. *suspendJob()*, *resumeJob()*, *killJob()*, *deleteJob()*, and *wait()*) expect a job identifier.

The various functions that submit jobs (e.g. *run()*, *paramRun()*, *monteRun()*, and *cornerRun()*) all return the job identifier for the submitted job.

These functions are not normally blocking when run in distributed mode. To make the run functions blocking, you need to pass *?block t* to the appropriate run function.

Blocking and Non-Blocking Modes

- Blocking mode
  - Simulation runs without using distributed processing.
  - All commands are processed sequentially.
  - Distributed runs can also be made blocking by passing *?block t* to the *run*, *monteRun*, *paramRun*, or *cornersRun* function.
- Non-blocking mode
  - Uses distributed processing.
  - Several jobs can be submitted that have no interdependencies.
- Semi-blocking mode
  - Submit simulations using distributed processing.
  - Wait for the simulations to finish before invoking post-processing

Distributed Processing Example

```
simulator( 'spectre )
...
ac ( 1 1000 "linear" 100 )
desVar( "rs" 1k )
job1=run( ?jobPrefix "myAc1" )
desVar( "rs" 2k )
job2=run( ?jobPrefix "myAc2" )
tran( 0 1u 1n )
delete( 'analysis 'ac )
job3=run( ?jobPrefix "myTran1" )
wait( job1 job2 job3 )
selectResult( job1 )
plot( vm("out") )
```

Note: Variables are used to store the job identifiers, so that you know the names of the submitted jobs for waiting on, and for selecting the results later. In the above case, the three jobs may be run in parallel (depending on available resources in the queue), but the results post-processing waits until all three have finished before proceeding.

Lab Overview

Lab 9-1 Running Parametric Analysis

Lab 9-2 Running Monte Carlo Analysis

Lab 9-3 Running Corners Analysis

Module Summary

In this module, we discussed

- Parametric analysis in OCEAN
- Monte Carlo analysis
- Corners analysis
- ADE Optimizer
- Distributed processing

SpectreRF in OCEAN

Module 10

October 5, 2005

Module Objectives

- Calculator functions for the Virtuoso® Spectre® RF Simulation Option
- Plotting functions for Spectre RF

SpectreRF Calculator Functions

Examples:

- Returns the waveform for a given harmonic index:

```
harmonic( v("/out" ?result "pac") 1 )
```

- Returns a list of lists containing a harmonic index and the minimum and maximum frequency values that the particular harmonic ranges between:

```
harmonicFreqList( ?result "pac" )
```

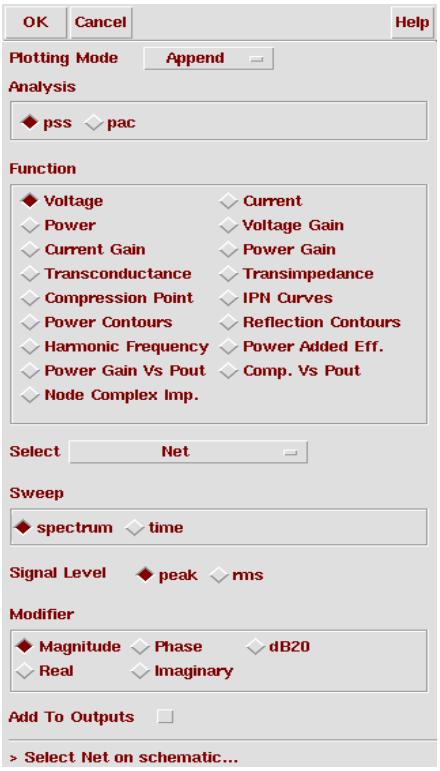
- Returns the list of harmonic indices available in the resultName or current result data:

```
harmonicList( ?result "pac" )
```

- Returns the spectral power given the spectral current and voltage:

```
spectralPower( i("/PORT0/PLUS") v("/net28") )
```

Finding Expressions for SpectreRF Functions



A general method for finding RF (and other) expressions is to bring up the Direct Plot form, and then check the **Add To Outputs** checkbox (or click the **Add To Outputs** button, if there is one).

Once the expression is in the ADE Outputs Pane, double click on the output line in the Outputs Pane, and cut and paste the expression to your code.

SpectreRF OCEAN Plotting Commands

Examples of plotting commands for SpectreRF results:

- Plots the phase noise close to the oscillator fundamental:

```
plot( phaseNoise( 1 "pss_fd" ?result "pnoise" ) )
```

- Plots and calculates the 1dB compression point for harmonic 3 using -25dBm as the extrapolation point:

```
dbCompressionPlot( v("/Pif") 3 -25 )
```

- Plots and calculates IP3 for 3rd order harmonic 5 and 1st order harmonic 3 using -25dBm as the extrapolation point:

```
ip3Plot(v("/net28") 5 3 -25)
```

SpectreRF OCEAN Printing Commands

- Prints a report about the noise contributions:

```
noiseSummary( 'spot ?result 'pnoise ?frequency 100k )
```

Lab Overview

Lab 10-1 Using SpectreRF OCEAN Commands

Module Summary

In this module, we discussed

- SpectreRF functions
- SpectreRF plotting functions

OCEAN and Mixed-Signal Simulation

Module 11

October 5, 2005

Module Objectives

- Choosing a mixed-signal simulator.
- Setting up the mixed-signal simulation.
- Setting mixed-signal and digital options.
- Plotting analog and digital signals.
- Debugging using the SimVision graphical debugger with OCEAN.

Terms and Definitions

SimVision	A Cadence graphical debugging tool for digital simulators
Verimix	A special build of the Verilog-XL simulator that can interface to the Virtuoso® Spectre® Circuit Simulator using interprocess communication (IPC) to run mixed-signal simulations

Setting Up OCEAN for Mixed-Signal

- You need to specify a mixed-signal simulator to run mixed-signal simulations in OCEAN:

  ```
  simulator( 'ams)
  ```

  or

  ```
  simulator( 'spectreVerilog )
  ```

  or

  ```
  simulator( 'spectreSVerilog )
  ```

- You need to set up the *$PATH* and *$LD_LIBRARY_PATH* variables to include search paths to Verilog-XL (in the LDV or IUS release streams).
- Virtuoso® AMS Designer (shown above) is supported from IC5141 USR1 onwards, in conjunction with IUS54.

If you are using a third-party analog simulator that supports Verimix in the Virtuoso Analog Design Environment, it can be used in OCEAN.

Analog and Digital Netlists (Verimix)

- Verimix is a special build of the Verilog-XL simulator that interfaces with the Virtuoso® Spectre® Circuit Simulator to run mixed-signal simulations.
- The analog and digital netlists are separated in Verimix.
- In OCEAN you specify the design by pointing to the analog netlist in the analog directory:

```
design( "~/oceansim/netlist/analog/netlist" )
```

- The digital netlist in the parallel *../netlist/digital* directory will be fed to the digital simulator Verilog-XL.
- The recommended procedure is to first set up the mixed-signal simulation in the Analog Design Environment and then save the OCEAN script.

Setting Mixed-Signal and Digital Options (Verimix)

You can set several mixed-signal and digital options using OCEAN:

```
option(?categ 'mixed 'maxDCIter x_count)
option(?categ 'mixed 'dcInterval f_interval)

option(?categ 'digital 'accelerationNormal {t|nil})
option(?categ 'digital 'libraryFile "t_library_file1 t_library_file2")
option(?categ 'digital 'verimixBinary "t_nameOfVerilogVmxExecutable")
```

To find out all the default analog, digital, and mixed option settings:

```
ocnDisplay( 'option
```

AMS Designer Details

- If the design is specified as a netlist, then it will be something like:

  ```
  design("simulation/block/ams/config/netlist/netlist")
  ```

  The netlist is not really a netlist, but a control file produced by the netlister to indicate the top-level configuration being used. The design itself may consist of a mix of Verilog®, Verilog-AMS, VHDL, VHDL-AMS and Spectre netlists.

- An additional OCEAN function is introduced to specify the connect rules being used:

  ```
  connectRules("my5V_basic" ?lib "connectLib" ?view "connect")
  ```

  Connect rules control which Verilog-AMS models are used to handle the interface between the digital and the analog domains.

Plotting Analog and Digital Signals in OCEAN

There is no difference in plotting analog or digital signals in OCEAN. Both are handled as if the simulation were purely analog.

Debugging Using SimVision in OCEAN (Verimix)

You can start the SimVision interactive debugger when you run the analysis in OCEAN by setting additional options:

```
option(?categ 'digital 'stopCompile t)
option(?categ 'digital 'simVision t)
```

The first option stops after compiling the digital circuit parts and waits for a Tcl command from SimVision.

The second options opens the SimVision debugger.

Lab Overview

Lab 11-1 Running Mixed Signal Simulations using OCEAN

Module Summary

In this module, we discussed

- Choosing a mixed-signal simulator
- Setting up the mixed-signal simulation
- Setting mixed-signal and digital options
- Plotting analog and digital signals
- Debugging using SimVision with OCEAN

Waveform Data Objects and Custom Calculator Functions

Module 12

October 5, 2005

Module Objectives

- Accessing individual point data in waveforms
- Creating new waveform objects
- Registering special functions in the calculator

Waveform Data Objects

- SKILL represents the waveform objects returned by data access functions as a special object type:

  ```
  v("op") => drwave:43347992
  ```

  The displayed return value is the print representation of the object; the numerical part is a unique identifier for the object. The print representation cannot be used as the input syntax for the object.

- Waveforms are made up of an X vector and a Y vector.
- Normal arithmetic operators are overloaded to be able to handle these as well as atomic types (integers, floating point numbers, etc).
- This module explores the SKILL to access and create waveform objects.

Waveform Object Functions

- Checking whether the argument is a waveform object:

  ```
  drIsWaveform(wave) => t/nil
  ```

- Get the type of the X vector:

  ```
  drGetWaveformXType(wave) => s_type
  ```

  The data types may be one of *'intlong*, *'double*, *'doublecomplex* or *'string*.

- Get the type of the Y vector:

  ```
  drGetWaveformYType(wave) => s_type
  ```

- Get the X vector:

  ```
  drGetWaveformXVec(wave) => drVectorObj
  ```

- Get the Y vector:

  ```
  drGetWaveformXVec(wave) => drVectorObj
  ```

Waveform Object Functions (continued)

- Creating a new waveform:

```
drCreateWaveform(xVector yVector) => drWaveObj
```

- Create a new waveform object with no vectors assigned yet:

```
drCreateEmptyWaveform() => drWaveObj
```

- Set the X vector for an existing waveform object:

```
drPutWaveformXVec(wave xVector)
```

- Set the Y vector for an existing waveform object:

```
drPutWaveformYVec(wave yVector)
```

- Example—transposing the X and Y axes on a waveform:

```
procedure(TrTranspose(wave)
  drCreateWaveform(
    drGetWaveformYVec(wave)
    drGetWaveformXVec(wave)
  )
)
```

Vector Object Functions

- Finding the type of a waveform vector:

  ```
  drType(vector) => s_type
  ```

- Create a new vector with initial length (the vector can grow):

  ```
  drCreateVec(s_type length) => vectorObj
  ```

- Create a vector, initialized with a list of values.

  ```
  drCreateVec(s_type l_values) => vectorObj
  ```

- Get an element from a vector:

  ```
  drGetElem(vector index)
  ```

  The index starts at 0. This retrieves the value of an index in the vector.

- Put a new value after the last value in the vector:

  ```
  drAddElem(vector value)
  ```

- Set the value of a specified index:

  ```
  drSetElem(vector index value)
  ```

Example: An Eye Histogram

- Often waveforms with jitter are looked at using an eye diagram (which is available either natively in Wavescan or as a special function in the ADE calculator).
- You use the *TrEyeHisto* function to create the eye diagram.
- The example on the next three slides is an alternative way of looking at this kind of signal. What it does is:
  - Splits each period into a number of bins
  - Looks at the threshold crossings in each period
  - Depending on where the crossing occurred, increments the count for that bin
  - Makes a waveform from the bin counts
- As a result, you get a plot of the distribution of edges in the waveform.

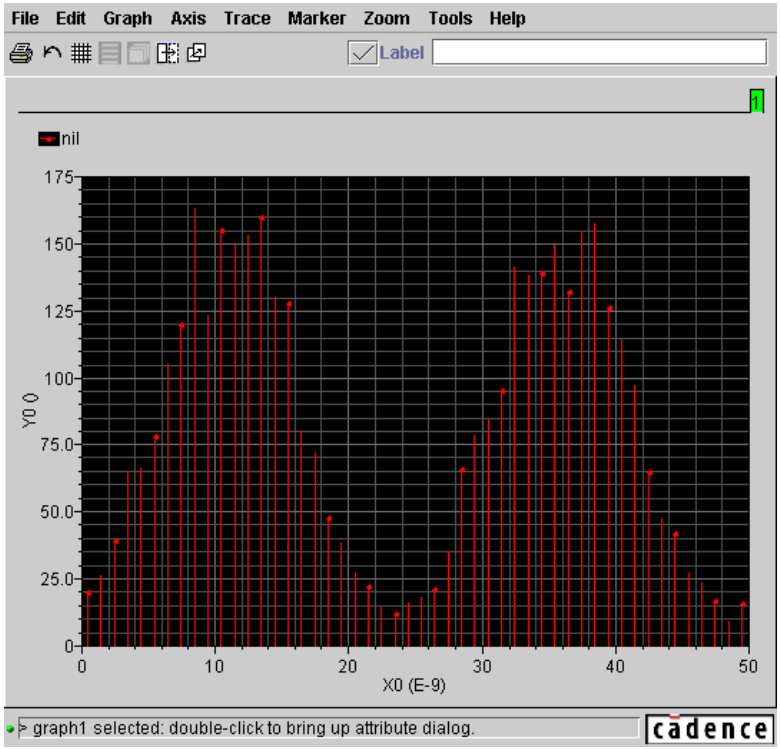
TrEyeHisto Code

```
procedure(TrEyeHisto(waveform start stop period threshold bins @optional
                     (edgeType "either"))
  cond(
   ;---------------------------------------------------------------------
   ; Handle ordinary waveform
   ;---------------------------------------------------------------------
   (drIsWaveform(waveform)
    let((xVec len clipped crosses cycleNum edgePos binArray binWidth bin
           outXVec outYVec)
         xVec=drGetWaveformXVec(waveform)
         len=drVectorLength(xVec)
         binArray=makeVector(bins+1 0)
         binWidth=period/bins
         ;---------------------------------------------------------------
         ; if no start given (start less or equal to 0), use first time point
         ;---------------------------------------------------------------
         when(start<=0
              start=drGetElem(xVec 0))
         ;---------------------------------------------------------------
         ; if no stop given (stop less or equal to 0), use last time point
         ;---------------------------------------------------------------
         when(stop<=0
              stop=drGetElem(xVec len-1))
         ;---------------------------------------------------------------
         ; cast everything to floats, just to make sure
         ;---------------------------------------------------------------
         start=float(start)
         stop=float(stop)
         period=float(period)
         ;---------------------------------------------------------------
         ; clip the waveform
         ;---------------------------------------------------------------
         clipped=clip(waveform start stop)
```

TrEyeHisto Code (continued)

```
        ;-------------------------------------------------------------
        ; And find the crossing points, and then bin them
        ;-------------------------------------------------------------
        crosses=cross(clipped threshold 0 edgeType)
        foreach(cross crosses
                cross=cross-start
                cycleNum=floor(cross/period)
                edgePos=cross-cycleNum*period
                bin=floor(edgePos/binWidth)
                binArray[bin]=binArray[bin]+1
                )
        ;-------------------------------------------------------------
        ; Build the waveform
        ;-------------------------------------------------------------
        outXVec=drCreateVec('double bins)
        outYVec=drCreateVec('intlong bins)
        for(binNum 0 bins-1
             drSetElem(outXVec binNum binWidth*(binNum+0.5))
             drSetElem(outYVec binNum binArray[binNum])
             )
        drCreateWaveform(outXVec outYVec)
        ))
    ;---------------------------------------------------------------------
    ; Handle family
    ;---------------------------------------------------------------------
    (famIsFamily(waveform)
     famMap('TrEyeHisto waveform start stop period threshold bins edgeType)
     ) ; family
    (t
     error("TrEyeHisto - can't handle %L\n" waveform)
     )
    ) ; cond
   ) ; procedure
```

TrEyeHisto Plot Results



OCEAN Script

The following OCEAN script was used to plot the above graph:

```
load( "TrEyeHisto.il" )
openResults( "test.raw" )
selectResult( 'tran )
plot( TrEyeHisto(v("op"),0,0,50n,2.5,50,"either") )
```

The simulation was performed using a Verilog-A block *randedges*, which produces a periodic signal with edges that have a random distribution.

The example can be found in the lab database, in *~/OCEAN/eyehisto* along with the code.

Handing Family Waveforms

The preceding example used a couple of functions to allow the function to deal properly with family objects.

- Determining if a object represents a family of waveforms:

```
famIsFamily(waveform)
```

- Iterating over the family automatically, building a family of results:

```
famMap('TrEyeHisto waveform start stop period threshold bins
edgeType)
```

Registering Special Functions in the Calculator

First a callback function creating the special function must be defined:

```
procedure(TrEyeHistoSpecialFunctionCB()
    calCreateSpecialFunction(
        ?formSym 'TrEyeHistoForm
        ?formInitProc 'TrCreateEyeHistoForm
        ?formTitle "Eye Histo"
        ?formCallback
        "calSpecialFunctionInput( 'TrEyeHisto '(start stop period threshold bins edgeType))"
        )
) ; procedure
```

This function uses the function in the notes below to create the form.

Then the function needs to be registered:

```
procedure(TrRegEyeHistoSpecialFunction()
    calRegisterSpecialFunction(
        list("eyeHisto..." 'TrEyeHistoSpecialFunctionCB)
        )
    t
) ; procedure

; call the function to register the special function
TrRegEyeHistoSpecialFunction()
```


Form Creation Code

```
procedure(TrCreateEyeHistoForm()
let((start stop period threshold bins edgeType)
    start=ahiCreateStringField(?name 'start ?prompt "Start Time" ?value "0")
    stop=ahiCreateStringField(?name 'stop ?prompt "Stop Time" ?value "0")
    period=ahiCreateStringField(?name 'period ?prompt "Period" ?value "")
    threshold=ahiCreateStringField(?name 'threshold ?prompt "Threshold" ?value "0.0")
    bins=ahiCreateStringField(?name 'bins ?prompt "Bins" ?value "10")
    edgeType=ahiCreateCyclicField( ?name 'edgeType ?prompt "Edge Type" ?value "either"
        ?choices '("either" "rising" "falling"))
    calCreateSpecialFunctionsForm(
            'TrEyeHistoForm
            list(
                list(start 0:0  180:20 90)
                list(stop 0:30 180:20 90)
                list(period 0:60 180:20 90)
                list(threshold 0:90 180:20 90)
                list(bins 0:120 180:20 90)
                list(edgeType 0:150 180:20 90)
                )
            )
    ) ; let
) ; procedure
```

Lab Overview

Lab 12-1 Writing a function to return the list of peaks in a waveform

Module Summary

In this module, we discussed

- Accessing individual point data in waveforms
- Creating new waveform objects
- An example of a more complex function using the waveform object API
- Registering special functions in the calculator