Agilent Technologies

Advanced Design System 2002

AEL

February 2002

Notice

The information contained in this document is subject to change without notice.

Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Agilent Technologies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms that apply to this software product is available upon request from your Agilent Technologies representative.

Restricted Rights Legend

Use, duplication or disclosure by the U. S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Agilent Technologies
395 Page Mill Road
Palo Alto, CA 94304 U.S.A.

Contents

9 Math Functions

Chapter 1: Introduction to AEL

Application Extension Language (AEL) is a general purpose programming language, modeled after the popular C programming language. AEL is used to configure, customize and extend the capabilities of the design environment. Like C, AEL has an extensive set of built-in function libraries, including functions for file input/output, math, string manipulation, list handling, and database query.

In general, you can use AEL for these tasks:

- Organizing libraries and palettes of components.
- Defining the interface to new user-defined components.
- Creating new components with layout artwork.
- Defining custom layout artwork functions.
- Defining the interface to discrete-valued simulation components.
- Creating custom utility functions, such as parts list generators and bill of materials.
- Automating routine tasks, such as repetitive command sequences, batch analysis, or optimizations.

Note Only a small subset of the language is used for component, library, or palette definitions. If you just want to customize a library or palette, you can skip the general language description and start at the section "Using AEL for Library Definition" on page 3-2.

General AEL Structure

The AEL language is similar to the C language in many ways. AEL is a procedural language, with many of the same language components as C, such as the use of variables and functions, arithmetic expressions, program flow, and logic statements. However, there are significant differences between AEL and the C language:

- AEL has no pre-processor; therefore, the *#if*, *#ifdef*, *#ifndef*, *#endif*, *#define*, *#undef* and *#include* directives are not supported. However, AEL files can reference other AEL files using the load() function, which has an effect similar to *#include*.

- Variables, pass parameters and function return values are not typed; therefore, the C keywords *int*, *char*, *float*, *double*, *long*, *short*, *unsigned*, *auto*, *register*, and *typedef* are not supported. Although a value has a particular type, the type of a variable, pass parameter, or function return is dependent only on the most recently associated value, not on the declaration of the variable, parameter, or function. Supported values are long, double, string, and boolean.
- AEL also supports complex numbers and lists which are not supported by C.
- AEL does not support structures or unions which are part of standard C, nor does it support external or static variables. All variables defined outside of a function are global.
- Variables can be declared globally (outside a function definition), local to a function, or declared within a compound statement.
- AEL supports a large subset of the C operators and logic and control constructs. However, AEL does not support the right arrow (->) and period (.). Brackets ([ ]) are supported for list access only. For example:

  ```
  a=list("yes", "no")
  b=[0] - "yes"
  b=[1] - "no"
  ```

  The *switch*, *case*, and *default* logic and control constructs are supported.
- AEL does not support casting; however, there are several type conversion functions.
- AEL features automatic *garbage collection*, a language feature that automatically re-claims storage when it is no longer needed.

Note If you are unfamiliar with the C language, refer to *The C Programming Language* by Kernighan and Ritchie.

Language Specifics

The AEL language can be described in terms of the six classes of tokens: comments, identifiers, keywords, constants, operators, and other separators. Like the C language, blanks, tabs, new lines, and comments are ignored except as separators for other tokens.

Comments

The characters /* or // introduce a comment. If the characters /* are used, the comment ends with the next occurrence of the characters */. If the characters // are used, the comment continues to the end of the line.

Identifiers

An identifier is a sequence of letters, digits, and underscore, where the first character must not be a digit. Case is significant for identifiers. There is no limit to the length of an identifier. Identifiers are used to represent function names, variable names, or statement labels.

Keywords

These identifiers are reserved for use as keywords:

Keyword	Description
break	leave a loop, exit switch statement
continue	start next iteration of a loop
decl	define a variable
defun	define a function
do	begin a do {} while (); loop
else	begin alternate action of an if()-conditional or of an inline if()-then-else conditional
for	begin for (;;) loop
goto	goto a labelled statement
if	begin if() conditional
NULL	constant NULL
return	return from a function
while	begin while(){} loop, or end do{} while(); loop
__FILE__	current filename
__LINE__	current line number
default	defines default case for switch(){} statement
switch	begin a switch (){} statement
case	defines a case 1A switch (){case: }

Keyword	Description
elseif	begin alternate if() conditional to an if() conditional
then	defines the action for inline if() then else statements
AND	logical and
OR	logical or
EQUALS	logical equal operator
NOT	logical not
list	define a list

Constants

AEL supports two constant forms not supported by C: the null value and the imaginary number. These constants represent internal data types not found in C. The list is also an internal data type, but there is no constant expression for a list. The supported forms of constants are:

Supported Constant	Description
null value	NULL
decimal integer constant	13
hexadecimal integer constant	0x3E (case is not significant)
octal integer constant	0377
string constant	“a string”
real number constant	10.3 or 25.4e-3 (case is not significant)
imaginary number constant	3.5i or 4+3.5i

A string constant consists of one or more characters enclosed in quotation marks (“ ”). Unlike C, AEL does not allow you to treat strings as arrays of characters. A string constant can have embedded non-printable characters, expressed using the backslash:

Non-Printable Character	Description
\n	new line
\r	return
\f	form feed

Non-Printable Character	Description
\b	backspace
\t	tab
\"	double quote
\\	backslash
\xNN	character in hexadecimal notation (N is 0 - 9 or A - F; case is not significant)
\0NNN	character in octal notation (N is 0 - 7)

Note If you do not want to convert control characters, use single quotes around the string instead of quotation marks; for example: `''\usr\dsn\nim.dsn''`.

Predefined Global Constants

To simplify the use of the database query functions, a set of symbolic constants, called global variables, have been predefined. The global boolean definitions are: TRUE, FALSE. Other global variables are shown with the applicable function definition.

Global Variable	Definition
TRUE	boolean true
FALSE	boolean false
stdin	standard in
stdout	standard out
stderr	standard error
hugeReal	3.4e +38
tinyReal	2.2e -308
e	2.718281828
ln10	ln(10) 2.302585093
c0	speed of light 2.997 924 58 e+08 m/s
e0	vacuum permittivity 8.8541878176204e-12 F/m
u0	vacuum permittivity 1.2566370614359e-06 H/m
boltzmann	Boltzmann's constant 1.380658e-23 J/K

Global Variable	Definition
qelectron	charge of an electron 1.60217733e-19 C
planck	Planck's constant 6.6260755e-34 J-sec
PI	= pi = 3.1415926535898
on_PC	TRUE if running on PC, else FALSE

Note that literally thousands of functions and global variables are defined in the Advanced Design System Design Environment (DE). To ensure that your functions and global variables do not interfere with those provided in the program, you should add a special prefix to your functions and variables. Further, different setups or users at your site should use unique definitions to distinguish newly-created sets of functions and variables (such as, by using a special prefix). For example:

```
mc1_varname
mimic_abc
```

Caution Do not use a dollar sign ($) as a special operator, since this character is used in expressions.

Operators

Constants and identifiers can be combined, by using operators, to form expressions. Expressions can be combined again, by using operators, to form more complex expressions. Expressions are the building blocks of AEL. Expressions can be used in assignments to variables and passed as parameters when calling AEL functions. AEL expressions are described and evaluated in the same way as in the C language, where operators are applied in order of their precedence. As with C, the default precedence applied in evaluating an expression can be overridden by grouping operands with parentheses. Table 1-1 lists (by operator precedence) the mathematical operators that are recognized in expressions.

Table 1-1. Mathematical Operators by Operator Precedence

Operator	Description
()	Expression grouping
!	Logical NOT (unary)
~	Bitwise one's complement (unary)

Table 1-1. Mathematical Operators by Operator Precedence (continued)

Operator	Description
++	Pre- and post-increment operator
--	Pre- and post-decrement operator
-	Negates a value (unary)
*	Used preceding an identifier to calculate value stored at an address (unary)
&	Used preceding an identifier to specify address of a variable (unary)
%	Integer remainder after division (modulus)
/	Division
+	Addition
-	Subtraction
>=	Test for first value greater than or equal to second
<=	Test for first value less than or equal to second
>	Test for first value greater than second
<	Test for first value less than second
==	Test for equal values
!=	Test for not equal values
&	Bitwise AND
^	Bitwise exclusive OR
|	Bitwise inclusive OR
&&	Logical AND
||	Logical OR
?:	Conditional evaluation operator (a ternary operator)
=	Assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Integer remainder assignment operator
+=	Addition assignment operator
^=	Bitwise exclusion or assignment operator
–=	Subtraction assignment operator
|=	OR assignment operator
&=	AND assignment operator

Table 1-1. Mathematical Operators by Operator Precedence (continued)

Operator	Description
<<	Bitwise left shift
>>	Bitwise right shift
<<=	Left shift assignment operator
>>=	Right shift assignment operator
,	Sequential evaluation operator
**	Power operator
::	Used to define sequences; for example, 1::10 or 1::10::2
$	Indicates substituting the variable for its value
AND	Logical AND
OR	Logical OR
EQUALS	Test for equal values (same as ==)
NOT	Logical NOT

Operators which have one operand are called unary operators. These include !, ~, ++, ––, –, *, and &. The operand follows the operator except for ++ and ––. The operator follows the operand in the post-increment and post-decrement use of ++ and ––. Unlike C, the unary * and & operators can be used only with identifiers. However, they generate the address or dereference an address much like C does.

Binary operators have two operands which are separated by the operator. Most of the operators fall into this category. Most of the binary operators perform mathematical operations on real numbers, but there are a few exceptional operators.

The operators !, &&, and || work with the logical values. The operands are tested for the values TRUE and FALSE, and the result of the operation is either TRUE or FALSE. In AEL a logical test of a value is TRUE for non-zero numbers or strings with non-zero length, and FALSE for 0.0 (real), 0 (integer), NULL or empty strings. Note that the right hand operand of && is only evaluated if the left hand operand tests TRUE, and the right hand operand of || is only evaluated if the left hand operand tests FALSE.

The operators >=, <=, >, <, ==, != , AND, OR, EQUALS, and NOT EQUALS also produce logical results, producing a logical TRUE or FALSE upon comparing the values of two expressions. These operators are most often used to compare two real numbers or integers. These operators operate differently in AEL than C with string

expressions in that they actually perform the equivalent of *strcmp()* between the first and second operands, and test the return value against 0 using the specified operator.

The operators ~, &, ^, |, <<, and >> are considered bitwise operators and work only with integers, manipulating the binary bits of the value.

There are several assignment operators: =, +=, -=. *=, /=, %=, ^=, |=, &=, <<=, and >>=. These operators always have an identifier as the left hand operand, and modify the value of the variable as well as returning its value after modification. Multiple assignments are supported (e.g., a=b=4) which evaluate right to left.

As in C, there is only one ternary operator, the conditional evaluation operator, which has the form:

expression ? expression : expression

This operator evaluates the first expression and tests it. If the value of the first expression is zero it evaluates the third expression, and if not zero it evaluates the second expression.

An expression can also be a function call, which takes the following form:

identifier(argument_list)

The *identifier* is the name of the function and *argument_list* is a list of comma separated expressions representing the values of the parameters passed to the function. The function can return a value for use as an operand in evaluating additional expressions.

The sequential evaluation operator is used to cause several comma separated expressions to be evaluated in left-to-right order, with only the last value returned.

Other Separators

In addition to the operators, several other separators are used in AEL to form statements:

Separator	Description
;	Terminates an AEL statement.
:	Terminates an identifier to define a statement label.
{}	Groups one or more statements together to form a compound statement. Variables declared within braces are defined only within the scope of the braces.

Building AEL Programs

AEL programs consist of a sequence of one or more of the following forms of AEL statements:

- Comment
- Simple statement
- Compound statement
- Variable declaration
- Function definition
- Control and logic statements

Comment

Comments are allowed anywhere, as long as the contents of the comment are not required to complete a statement or function definition. Comments begin with /* or //. In the first case, the comment ends with */ and in the second case, the comment continues to the end of the line.

Simple Statement

A simple AEL statement is an AEL expression terminated by a semicolon (;) where the resulting value of the expression is discarded. Expressions often have side effects caused by evaluation which can be more important than the resulting value. Some examples of simple AEL statements are:

AEL Statement	Description
a=5;	Value of a is set to 5
my_fun(1, 2);	Function *my_fun* is called, with the values 1 and 2 passed as parameters, return value of the function is discarded
b++;	Value of *b* is incremented by one

Compound Statement

A compound statement is several statements grouped together inside braces { }. Any type of statement can be contained in a compound statement. Compound statements

are used to define the body of a function, switch statement, or the action of a conditional or loop statement.

Variable Declaration

The keyword *decl* begins a variable declaration, followed by a comma-separated identifier list, and terminated by a semicolon. Each identifier can be given an initial value using the = assignment operator. An example of a variable declaration is:

```
decl a, b=5, cosA=cos(A);
```

A variable declaration without an initial value has the value NULL. A variable can be declared more than once, in which case the first declaration prevails and the identity of the variable is maintained through subsequent declarations.

Note If a variable is declared with an initial value, the 2nd declaration prevails. That is, if a global variable is re-declared *and* initialized, then the global variable writes over the previous value; if the global variable is declared *but is not initialized*, the previous value is retained.

A variable declaration outside of a function definition is considered global. Inside a function definition a variable is considered local to the function. Variable declarations contained inside a compound statement are hidden from expressions outside the compound statement.

AEL Lists

Declaration and Initialization of an AEL List:

An AEL list is a collection of AEL values. It is created by:

```
list(<val1>, <val2>, etc...);
```

where <valx> is any valid AEL value, including another AEL list.

Example:

```
decl a = list(1,2,3);
decl b = -3.44;
decl c = list(1,1.5, "hello",a, b);
```

Printing Lists:

The value of a list can be printed using the *identify_value()* function.

Example:

```
fputs(stderr,identify_value(c));
// output is:  list(1,1.5,"hello", list(1,2,3), -3.44)
```

See "List Management Functions" on page 8-1 for documentation on traversing lists and other operations on lists.

Function Definitions

The general form of a function definition is:

```
defun identifier (parameter_list)
{
  statements (body)
}
```

The keyword *defun* begins a function definition. The name of the function is the *identifier*. The *parameter_list* is a list of comma-separated identifiers defining the parameters of the functions. A return statement (consisting of a return keyword optionally followed by an expression and terminated by a semicolon) ends the function and specifies the return value. All AEL functions return a value. If the return statement is missing or no value was given for the return statement, the return value is NULL. If a function returning a NULL value is used in the context of an expression, the evaluation of the expression can fail, causing an AEL error.

Control and Logic Statements

AEL supports many of the C control and logic capabilities. The control and logic statements which can be used are:

goto label;

if (expression) block

else if (expression) block

else block

for (statement; expression; statement) block

do block *while* (expression);

while (expression) block

switch (expression) {case statements}

where:

label is a defined statement label;

expression is a combination of values, functions, and operators evaluating to a single value;

statement is also a single expression where the value is discarded; and

block is a single statement, or several statements enclosed in braces.

If *goto* is used within a function definition, the label must be defined in the same function definition. Switch and case statements are the same as case statements in C language.

AEL Arrays

Note If you plan to manipulate extremely large AEL arrays, you can use AELhpvars instead of AEL arrays. When using AELhpvars, you must include the genserver library, but the manipulation of the arrays is more efficient.

Declaration and Initialization

AEL supports multidimensional arrays. An array must be initialized in a declaration or in an assignment statement.

Note The number of elements in the initialization statement MUST be the same as the loop. You can initialize an array to one element, and then resize it later using the resize_array() function.

Array values are enclosed in curly braces "{}". Array elements are separated by ",". Array elements can be integers, doubles, or complex numbers. Sequences can also be used to initialize an integer or double array.

Example:

```
// one dimension (row vectors)
decl x = {1+3i, 4-5i, 9+3i, 10i};
x = [1+3i, 4-5i, 9+3i, 10i];
x = {1::10} // x = {1,2,3,4,5,6,7,8,9,10}
```

```
x = {1::0.5::2} // x = {1,1.5,2}

// column vectors
decl z;
z = {{1},{2},{3}};
z = [1;2;3];

// two dimension (matrix)
decl y;
y = { {1,2}, {3,4}, {5,6}, {7,8} };
y = [ 1,2; 3,4; 5,6; 7,8 ];

// three dimension
decl s;
s = {[0,1;2,3],[4,5;6,7]};
s = {{{0,1},{2,3}},
    {{4,5},{6,7}}};
```

Printing Arrays

The value of an array can be printed using the identify_value() function.

Example:

```
// this would output "{1+3i, 4-5i, 9+3i, 10i}"
fprintf(stdout, "%s\n", identify_value(x));

// this would output "{{1,2},{3,4},{5,6},{7,8}}"
fprintf(stdout, "%s\n", identify_value(y));
```

Operations Between Arrays

Some operations on a whole array can be executed. These include assignments, additions, subtraction, multiplication (which is essentially matrix multiplication), division, and negation.

When assignments are made, the receiver of the assignment will point to the same array as the original array. However, when a modification is made to an array, a new array is created so that other variables pointing to that array will not be modified. This is a significant difference between AEL arrays and C arrays.

Example:

```
decl x = {1,2,3,4};

// z = {1,2,3,4}
```

```
decl z = x;

// x = {1,2,3,4} and z = {5,5}
z = {5,5};
```

When addition or subtraction is executed between whole arrays, the math is executed one element at a time. Therefore, the array bounds must be exactly the same. Element by element multiplication, division and power are also supported by using the operators: .*, ./, and .**.

Example:

```
decl x = {1,2,3,4};
decl y = {9,8,7,6};

// z = {10,10,10,10};
decl z = x + y;

x = { {1,2}, {3,4} };
y = { {10,9}, {8,7} };

// z = { {-9,-7}, {-5,-3} };
z = x - y;

// z = {9,16,21,24}
z = x .* y;
```

When multiplication is executed between whole arrays, matrix multiplication is executed. Therefore, the arrays must be 2 dimensional and the array bounds must be compatible for matrix multiplication, which is row1 x col1 * row2 x col2 where the dimension of col1 must equal the dimension of row2. The new array will have the dimensions row1 x col2; that is, 4 x 2 * 2 x 8 = 4 x 8.

Example:

```
x = { {1,2}, {3,4} };
y = { {10,9}, {8,7} };

// z = { {26,23}, {62,55} };
z = x * y;
```

Note Currently, the following short hand notations are supported: +=, -=, *= .

Operations Between Scalar and Arrays

Any math operation between an array and scalar values is legal. The result of such expressions is an array.

Example:

```
decl x = {1,2,3,4};

// x = "{6,7,8,9}"
x += 5;

// x = "{32,28,24,22}"
x = 100 / x * 2;
```

Transpose

Transposition of arrays is supported. The syntax is:

Example:

```
decl x = {1,2,3,4};

// y = {{1},{2},{3},{4}}
y = x';
```

Access and Operations on Array Elements

Individual elements are accessed by specifying the array name followed by the index(s) enclosed in square brackets "[]". There are three ways to access elements of multidimensional arrays:

- The indices can be separated by commas and the whole sequence enclosed in square brackets; for example, y[1,3]
- Each index can be enclosed in square brakets; for example, y[1][3]
- An index can be an AEL sequence; for example, y[1::3]

When individual elements are accessed, they are treated as any other scalar variable. That is they can be assigned, printed, parameters, or simply accessed. Note that the indices begin with 0.

Example:

```
// x's 2nd element is modified to a new value
x[1] = 10-10i;
```

```
// an element in y is printed, the value printed is "4"
fprintf(stderr, "%s",identify_value(y[1,1]));


// an element in y is changed from "7" to "50"
y[3][0] = 50;

// this loop produces the sum of all elements of x
sum = 0;
for ( i=0; i<4; i++)
sum += x[i];
```

AEL Array Functions (Used in AEL Scripts)

The AEL array functions used in AEL scripts are:

“offset_array()” on page 10-23, “resize_array()” on page 10-34, “array_size()” on page 10-1, “array_type()” on page 10-2, “array_lowerBound()” on page 10-3, “array_upperBound()” on page 10-3, and “convert_array()” on page 10-5.

Writing, Loading and Testing AEL Functions

For component and artwork definitions, you can refer to the AEL files supplied with the program. These files, which define the entire element sets for each simulator, contain hundreds of examples. The files are located in the installation directory for your application:

$HPEESOF_DIR/circuit/ael (for Analog/Rf applications)

$HPEESOF_DIR/hptolemy/ael (for Signal Processing applications)

$HPEESOF_DIR/de/ael (for the Design Environment, all applications)

You can use any text editor to write AEL files.

Playing an AEL Macro

In the ADS Main window, you can replay an AEL macro using the Options > Playback Macro command. You can execute AEL statements interactively by typing the statement in the Options > Command Line dialog box. For example, the first value in a list could be examined by typing:

fputs(stderr, car(list(1,2,3)));

The value *1* would be displayed in a dialog.

Loading AEL Files

You can load AEL files by executing the *load()* command, either from the Command Line dialog box or as a statement in an AEL file. For example:

load("myfunc.ael");

Any number of functions can be declared in a single AEL file and any function in the loaded file can then be executed by entering the function name and parameters. For example:

testit(1,2, "myfunc");

You can load AEL files automatically when you start the program in one of these ways:

- By adding the name of the file containing the functions to one of the configuration variables LOCAL_AEL or USER_AEL and specifying the path to the file in the AEL_PATH variable. LOCAL_AEL is a project-specific variable that is reloaded every time a project is loaded (once per project). USER_AEL is

a user-specific variable that is loaded once per program session (once per user). For details on configuring these variables, refer to "*Customizing the ADS Environment*" in the *Customization and Configuration* manual.

- By using the *-m* option; for example, *hpads -m startup.ael*.
- By placing the file in the networks subdirectory of a project, which automatically loads the file when opening the project.

When files are loaded automatically by the program, these files are also compiled automatically. Any file loaded is re-compiled if it is modified or if its compiled version does not exist. Compiling greatly increases the speed in which the function is loaded and it also performs a simplified syntax check. When creating AEL files, you can perform a quick check of their syntax by pre-compiling the AEL file. For details on manually compiling AEL files, refer to the section "Using the AEL Compiler" on page 1-21 .

Testing AEL Functions

You can test most functions by loading the AEL file from the Command Line dialog box. Debugging of the program is limited to adding *fputs()* and *fprintf()* statements in the file to determine function values and code execution. When using *fputs()* and *fprintf()* statements be sure to direct the *fputs()* and *fprintf()* output to standard error, stderr, or to a file, since stdout is used by the project design environment to communicate with other programs.

On UNIX, the results of an *fputs()* or *fprintf()* statement sent to stderr will be displayed in the terminal window that the program was started in. On a PC, the program needs to be started as follows:

```
hpads -d daemon.log
```

This command opens the program with a window that contains all debugging statements and the results of *fputs()* and *fprintf()* statements sent to stderr. For example, to see something print to a UNIX terminal or PC window:

- In the Main window, choose **Options** > **Command Line**.
- In the dialog, type:

  ```
  fputs(stderr, "hello world!");
  ```

- Click Apply.

AEL Loading Context

When an AEL file is loaded, functions and variable references are resolved according to the loading context, which specifies the AEL vocabulary to search. New definitions in the AEL file are also added to the context vocabulary.

CmdOp and SimCmd Loading Contexts

The Advanced Design System contains two important contexts, called *SimCmd* and *CmdOp*. When AEL files are loaded at program start-up or when you enter AEL from the Command Line, the context is *CmdOp*. When the dialog callbacks are invoked, the context is *SimCmd*. *SimCmd* is a superset of *CmdOp*; that is, when the context is *SimCmd*, all of the definitions in *CmdOp* are accessible. But when the context is *CmdOp*, the definitions in *SimCmd* cannot be accessed and fail with the message: “could not find global word”.

An AEL file that contains new AEL functions or variables can be loaded automatically at program start up. The functions or variables are then accessable from the Comand Line and menu commands.

The default loading context is *CmdOp*, which is also the context in which all AEL definitions in ADS get loaded. The normal context for project’s networks .ael files is a project-specific context. This context is a superset of *SimCmd*. When the definitions in the file must be accessed while running the program, the context must be *CmdOp*.

The default loading context can be overwritten by specifying the optional second argument (the context name) for the *load()* function. This can be specified when loading the AEL file from another startup AEL file, or from the command line. For example, to load an AEL file “test” in the SimCmd, you can use the following load comand:

```
load (“test”, “SimCmd”);
```

The loading context within a particular AEL file can be fixed using the AEL directive *#voc()*. The argument to *#voc()* specifies the new loading context. If no argument is specified, the loading context is reset to the default for the file. The loading context fixed using *#voc()* overrides that set by the *load()* function. For example, when placed in an AEL file that is loaded in the *CmdOp* context, *#voc(SimCmd)* causes the lines in the AEL file following it to be loaded in *SimCmd* context. When a *#voc()* directive is encountered, or the end of file is reached, the loading context reverts back to *CmdOp*.

```
....        //Loading context is CmdOp
#voc(SimCmd) //Loading context is set to SimCmd
```

```
....        //AEL code is loaded in SimCmd
#voc()      //Loading context is reset to CmdOp
```

Using the AEL Compiler

When creating AEL functions, you can pre-compile the AEL file by using the AEL compiler, aelcomp, in a stand-alone mode. Any errors in syntax are reported by the compiler and a compiled atf AEL file is produced. The compiler is invoked automatically by the Design Environment whenever an AEL file is loaded that has been modified since the last compile or is missing its compiled atf counterpart. To run the AEL compiler, enter this instruction:

aelcomp <input>.ael <output>.atf

at the command line. The input file and output file names are required and must be specified with their extensions. Normally, the extensions ael and atf are used for the input and output files, respectively.

To compile an AEL file when using Sun 5.5 or 5.6, the LD_LIBRARY_PATH variable needs to be set before the compile is attempted. This is done by entering the following lines in your start-up file (.cshrc, .profile, etc.):

```
setenv HPEESOF_DIR /<your complete installation path>
setenv LD_LIBRARY_PATH /$HPEESOF_DIR/lib/<SunOS>
```

where

SunOS is the Sun operating system being used.

sun55 = Sun 5.5
sun56 = Sun 5.6

Customizing the Parts List

The Parts List command has extensive customization capability using AEL function calls. To configure the Pick and Place Options, simply modify the routines called each time a Pick and Place Report is generated. These routines are located in the de_parts.ael file located in your program installation directory under pde/ael.

Parts List Example Using an AEL Macro

In this example, the AEL macro traverses the database to query information about a design and generates a simple parts list (Figure 1-1). To run the macro enter this instruction at the command line:

```
plist();
```

```
/*write out a parts list file, each line contains "DesignName","InstID" */
decl overWrite;
overWrite = FALSE;

defun traverse_design ( designName, instName, fileHandle )
{
decl count;
decl dgHandle, repHandle, instHandle;
decl ignore, special;
decl designHandle;
/* traverse the design's instances-make sure not to output "special" (e.g. gnd), "deactivated", and
"no plot" instances. */
designHandle = db_get_design(designName, overWrite);
if (designHandle)
{
  repHandle = db_get_rep( designHandle, REP_SCHEM );
  if (repHandle)
  {
    instHandle = db_first_instance(repHandle);
    while (instHandle)
   {
    designName=db_get_instance_attribute(instHandle,INST_DESIGN_NAME);
     instName = db_get_instance_attribute(instHandle, INST_NAME);
     ignore = db_get_instance_attribute(instHandle, INST_NO_PLOT );
     special = db_get_instance_attribute(instHandle,INST_SPECIAL );
     if(designName &&!ignore&&(special==INST_NORMAL||special==INST_LINE))
      {
     fputs(fileHandle,fmt_tokens( list(designName, " ",instName)));
     traverse_design( designName, instName, fileHandle );
      }
     instHandle = db_next_instance( instHandle );
   }
  }
}
}
/* top level routine - to run just type: plist(); */
defun plist()
{
    decl designName;
    decl listItem;
    decl fileHandle;
    decl fileName;
    set_window( 1 ); /* set to schematic window */
    designName = current_design_name();
    fileName = strcat( designName, ".pl" );
    fileHandle = fopen( fileName, "W" );
    fputs( fileHandle, strcat( "Parts list for: ", designName ) );
    fputs( fileHandle, "\n" );
    $traverse_design( designName, "" , fileHandle );
    fclose( fileHandle );
    fputs( stderr, fmt_tokens( list( "Parts list", fileName, "created." ) ) );
}
```

Figure 1-1. Parts List Example

Chapter 2: Using AEL Database Retrieval Functions

This chapter describes how you can extract design information from the Design Environment (DE) by using the AEL database retrieval (db) functions.

Database Overview

The database is structured around a hierarchy of objects, with the design object at the top. Every design contains two representations, a schematic and layout. In turn, these representations are composed of masks, nodes, pins, and instances. Masks contain shapes and text (data groups). Nodes, pins, and instances maintain connectivity. Instances can refer to another design and/or contain their own masks. Properties may be added to each of these database objects.

Figure 2-1 shows the design hierarchy and relationships among the design objects. For all objects, except representations, there can be a *list* of zero or more occurrences of the object. For example, a representation or instance may have a list of zero or more mask objects, a mask may have zero or more data groups. Each type of object is described briefly in the following sections.

Design

A *design* represents a complete Analog/RF or Digital Signal Processing design. Each design is stored in a separate file. Every design has two representations: a layout and a schematic. Either representation can be empty. Designs can reference other designs hierarchically through instances. There is no limit to the number of designs loaded in the program. When a hierarchical design is loaded, all the referenced designs are loaded as well.

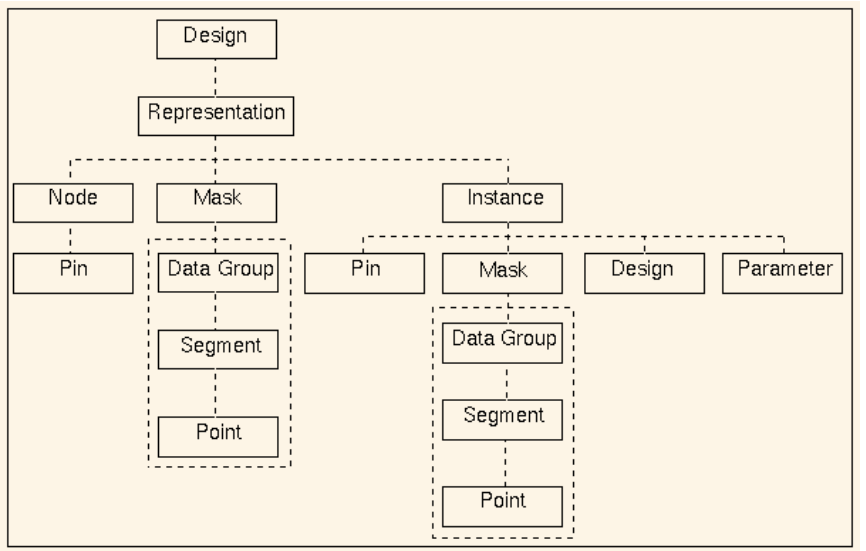


Figure 2-1. Design Hierarchy and Relationships

Representation

Representations describe either the layout or the schematic of an Analog/RF or Digital Signal Processing design. The design synchronization engine (DSE) ensures or checks that the two are equivalent. Representations own masks (also called layers), instances, and nodes. Representations also own symbols (not shown on the chart). Symbols, used in schematics only, display a schematic in a hierarchy. A symbol owns pins and masks.

Mask

Masks (sometimes termed layers) group shapes and text. The only attribute of interest in a mask object is its number. The number is used as an index into a layer table, which in turn tells the program what color, name, fill pattern, etc. to use when displaying the objects grouped by the mask.

Node

A node, also known as *net*, represents an electrical connection between pins. Nodes are established by abutting pins of instances or creating a wire, naming a node, or trace that connects one or more pins. Nodes contain pointers to the pins they interconnect.

Instance

An instance represents either a primitive simulation element or a reference to another design. The term *instance* is also called *component* in the program and documentation. A resistor is represented as a primitive instance, while a sub-network placed in another design is represented by a hierarchical instance. Regardless whether an instance is hierarchical or primitive, it can own a list of pins. Pins represent connection points for the instance. Interconnecting or abutting the pins of two instances forms an electrical connection (a node). Primitive instances can own masks. This is usually seen only in the layout representation. A microstrip transmission line instance owns the mask that describes its geometry. Instances can also own parameters. The type, name and number of parameters owned by an instance is dictated by an instance's component definition. Components are defined with the AEL *create_item()* and *create_param()* functions. When a new design is saved, a corresponding AEL definition is saved and used whenever an instance to the design is created.

Parameter

A parameter has a name and a value. The value of some parameters can be quite complex, so each value has a type. The simplest type is a single number, but some values are composed of a pair of numbers, a triple, a list of numbers, a string, etc. For example, the default value of a resistor's parameter *R* for resistance is the string “50 ohm.” Stored with each parameter is a form name that was specified in the *create_parm()* function. The form name indicates how to interpret the type of a parameter's value. A number of AEL functions are supplied that simplify the extraction of parameters and their values.

Pin

A pin represents an electrical connection point on an instance. Abutting or wiring pins together forms an electrical connection (called a *node*). In hierarchical layout representations, a pin is created for the instance for each port on the referenced design. In hierarchical schematic representations, the symbol for the referenced

design should have the same number of pins as the referenced design. The two are matched by pin number. Pins maintain a handle to any connecting wire or trace.

Data Group

A data group represents a shape or text. Shapes have a type that indicates whether the shape is a wire or trace, polygon, polyline, rectangle, circle, arc, construction line, or text. Polygons and polylines are composed of one or more segments.

Segment

A segment can be a list of points or an arc. Segments have the attribute of being *filled* or *empty*. An empty segment must be enclosed by a filled segment, since it represents a hole. Segments are composed of a list of one or more points.

Point

A point is an X,Y coordinate. These coordinates are integers in the range of roughly plus or minus 2 billion. Coordinates stored in the database are stored in database units. The number of database units to each user unit is controlled by a representation's precision. If mil is specified as the layout unit, the default is to store 1 mil as the integer 100, 0.5 mil would be stored as the integer 50.

Property

A property is a pair of a name and value. A property value type can be a long, double or string. An unlimited number of properties may be added to the following database objects: designs, representations, symbols, masks, nodes, instances, pins, data groups and segments.

Traversing a Design

For every object there is a traversal function that returns a handle to the object. Handles are like C pointers, in that they provide a way to uniquely identify an object and retrieve information about it. Generally these functions come in pairs, a *get_first()* and a *get_next()*. The *get_first()* function takes a handle to the object's parent, while the *get_next()* function takes a handle to the previous object in the list. This example uses these functions to traverse the list of data groups belonging to a mask object and count them.

```
decl cnt, maskHandle, dgHandle;
cnt = 0;
dgHandle = db_first_dg( maskHandle );
while(  dgHandle )
{
  cnt = cnt + 1;
  dgHandle = db_next_dg( dgHandle );
}
```

Besides traversal functions, each object has an attribute retrieval function. Attributes are the actual data, such as location, select status, bounding box, mask layer, pin number, etc.

The AEL function reference for an object's attribute function supplies a list of attributes that can be retrieved from each kind of object. After you have a handle to an object, you can retrieve any or all of its attributes. An attribute that is shared by both instances and data groups is the selected status attribute. By querying this attribute you can determine if an object has been selected.

Some objects, such as instances, have a large number of attributes. The example code segment demonstrates extracting information from an instance using the *db_get_instance_attribute()* query function.

```
instHandle = db_first_instance(repHandle);
while (instHandle)
{
designName=db_get_instance_attribute(instHandle,INST_DESIGN_NAME);
  instName   = db_get_instance_attribute(instHandle, INST_NAME);
  fputs( stderr, fmt_tokens( list( designName, " ", instName) ) );
  instHandle = db_next_instance( instHandle );
```

Using the database query functions can be complex. For a guide, you can use one of the examples supplied, such as de_bom.ael, which traverses the instances of a design. This file can be found in the installation sub-directory $HPEESOF_DIR/de/ael.

Note Be cautious when mixing query functions with database modification functions. As a design is modified, handles to the object in the database change and existing handles become invalid. After any modification, you should re-traverse the database from the representation level on down to get new handles. Even modifications, such as *move* and *copy*, invalidate old handles.

Besides the list traversal functions, some functions retrieve design and instance information by name, rather than by handle. Examples are:

db_find_instance()
db_find_property()
db_get_instance_parm()

Details on these functions can be found in Chapter 14, Database Query and Manipulation Functions.

Chapter 3: Using AEL with Component Libraries

Each design type comes with a pre-defined component set. The set includes all the basic components that a simulator understands, such as transmission line components, lumped components, filters, active devices, etc. For each component, an AEL description exists which describes the component to the design environment and defines the component's interface to the simulator.

In the design environment, the simulator and the front-end schematic and layout capabilities are separate programs which communicate by sending data back and forth through an Interprocess Process Communication (IPC) protocol. The data that is simulated by the simulator is sent as a netlist. A netlist describes the schematic and its simulation controls in a form understood by the simulator. When you create new parts, you must create a netlist format string for the component that corresponds to the syntax expected by the simulator. This syntax is described in Chapter 2, Using AEL Database Retrieval Functions.

Some of the new parts that you can create are:

- Components linked to the simulator using the user-compiled model feature of the simulator.
- Components that reference specific measured data files, such as in-house or custom active devices.
- Components with specific artwork requirements, such as chip capacitors, printed resistors and etched inductors.
- Parametric subnetworks, which model commonly-used devices.

In order to access the new components through the program, you must add the components to an existing library or create a new library. The design environment becomes aware of new parts through a combination of configuration file changes.

Note For details, refer to the section "Building User-Compiled Models" in the *Circuit Simulation* manual.

Using AEL for Library Definition

The libraries of simulation components are defined for the program through AEL. Adding new libraries to the program requires creating or changing AEL files. AEL definition files are interpreted when the program is invoked or after attaching to a project. The AEL files contain the definitions for the new library parts and for the groups used to present the parts through the palettes and library lists.

The design environment refers to parts as *components*. The definition for a new component specifies information such as:

- Component name
- Descriptive label
- Component parameters
- Symbol used to represent the component in the schematic
- Artwork used to represent the component in the layout
- Syntax used to represent the component in the netlist
- How the components parameters are displayed in the schematic
- Image (icon) used to represent the component in a palette (if the part is used in a palette)

The program refers to libraries or palettes as *groups*. The definition of a group specifies:

- Group name
- Descriptive label
- List of components in the group

The AEL definitions for components and groups take the form of function calls that are interpreted sequentially from the AEL file. The AEL functions required to define the program components and groups are described in a later section.

You should not modify the AEL files that are shipped with your product. Instead, create new AEL files to hold the AEL definitions for new or altered parts and libraries. You can use any text editor to create and modify the AEL files. New files can have any name you choose, but you must use the extension ael; for example *steves.ael* is an appropriate name for a new AEL file.

Note If you do modify the shipped AEL files, create backups of the original files first, in case you need to restore them.

Although AEL files can be located anywhere in the file system, usually these files are located in a directory convenient to all users. Since the $HPEESOF_DIR/custom directory is not modified by a new release installation, all system-wide customized component definitions can be stored there. You can locate the files in an application-specific sub-directory, such as $HPEESOF_DIR/custom/circuit/ael, $HPEESOF_DIR/custom/hptolemy/ael, or $HPEESOF_DIR/custom/de/ael.

If the files are used by a particular user (or all users of a particular project), locate the files in the $HOME/hpeesof/circuit/ael, $HOME/hpeesof/hptolemy/ael, or $HOME/hpeesof/de/ael directory. For a particular project, locate the files in the project's networks directory.

The environment variables must be set properly to allow the program to find and read the new AEL files. For details on the environment variables that are used, refer to the section, "Environment Configuration Directories" on page 3-3.

When adding parametric subnetworks as new parts, the design environment does much of the AEL work for you. The program creates an AEL definition file in the current project when a network design is saved. This file defines the parametric subnetwork component and assigns it to a library. New parametric subnetworks automatically become part of the current working project. Assignment of the symbol, artwork, and library list group can be made from within the program before the network is saved. Parameters can be added to a network from within the program. However, you can not define palettes for parametric subnetworks from within the program. If you want to add your parametric subnetworks to a palette, you must alter the AEL files created by the program or create your own AEL definition file.

Environment Configuration Directories

The environment configuration file, de_sim.cfg, contains many definitions required by the program. Directories that contain the environment file are:

$HPEESOF_DIR/config Environment variables defined at this level are effective for all sessions unless overridden by custom, user or project definitions.

$HPEESOF_DIR/custom/config Environment variables defined at this level are effective for all sessions unless overridden by user or project definitions. This

directory allows site-wide modifications without changing the installation directory $HPEESOF_DIR/config.

$HOME/hpeesof/config Environment variables defined at this level override those defined in the global environment files, but only for the current user.

Project Environment variables defined at this level override those defined in the both the user and global environment files.

Using Environment Variables

Environment variables tell the program which AEL files to read, the directories where they are located, and directories where associated design files are located.

AEL files are located and read by the program according to the values of environment variables that are defined in environment file, de_sim.cfg. The names of new AEL files must be added to the definitions in de_sim.cfg for the program to know about them and read them. These environment variable definitions can be changed by editing the de_sim.cfg file.

Each definition has the form of a variable name followed by an equal sign and a value (MY_NAME=my_value). The value is text, possibly a single name or a list of names separated by colons or semi-colons. Values should not have any embedded spaces.

The important configuration variables are: SIMULATOR_AEL, AEL_PATH, USER_AEL, LOCAL_AEL, USER_DSN_PATH, and SITE_AEL. For detailed information on these variables, refer to *"Customizing Configuration Variables"* in the *Customization and Configuration* manual.

Chapter 4: Creating New Component Definitions

The program learns about new components through component definitions in AEL files. An example of a new component definition is shown in Figure 4-1. The example is a definition of a microstrip bend component that has five parameters.

```
/* MBEND */
create_item(                "MBEND",                    // name
            "Microstrip Bend (Arbitrary Angle/Miter",       // label
            "BEND",                                     // prefix
            0,                                          // attribute
            NULL,                                       // priority
            "MBEND",                                    // iconName
            standard_dialog,                            // dialogName
            "*",                                        // dialogData
            ComponentNetlistFmt,                        // netlistFormat
            "MBEND",                                    // netlistData
            ComponentAnnotFmt,                          // displayFormat
            "SYM_BendAngle",                            // symbolName
            macro_artwork,                              // artworkType
            "MBendAngle",                               // artworkData
            ITEM_PRIMITIVE_EX,                          // extraAttrib
           create_parm ("Subst", "Substrate instance name", 0, "StringAndReferenceFormSet",
UNITLESS_UNIT, prm("StringAndReference", "\"MSub1\"")),
           create_parm ("W", "Conductor width", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", LENGTH_UNIT, prm("StdForm", "25.0 mil"), list(dm_create_cb(PARM_DEFAULT_VALUE_CB,
"get_default_length_value_cb", "25.0", TRUE))),
           create_parm ("Angle", "Angle of bend", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", ANGLE_UNIT, prm("StdForm", "90")),
           create_parm ("M", "Miter fraction", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", UNITLESS_UNIT, prm("StdForm", "0.6")),
           create_parm ("Temp", "Physical temperature", PARM_NO_DISPLAY PARM_OPTIMIZABLE |
PARM_STATISTICAL, "StdFileFormSet", TEMPERATURE_UNIT, prm("StdForm", "27.0")));
```

Figure 4-1. New Component Definition Example

The example definition uses the AEL functions *create_item()* and *create_parm()*.

The *create_item()* function creates a component definition with a specific name, associated parameters and other characteristics.

The *create_parm()* function creates and returns a parameter definition and is used to create parameter definitions for *create_item()*, where the *create_parm()* function return values become pass parameters for *create_item()*, with each use of *create_parm()* defining a parameter for the component.

The function *prm()* creates and returns a parameter value and is used to generate a default value where the form of the value is something other than a normal number

or string. The forms of a parameter value are explained more completely in the section “Creating and Using Custom Forms” on page 4-3.

For details on these functions, see “Component Definition Functions” on page 15-1.

Form Sets

When components and commands are defined with their associated parameters, the allowable forms for each parameter are specified through a named form set. A form set describes the *types* of values a parameter can take.

A form set is composed of one or more forms, each representing a value option for a parameter. Several forms are associated together in sets for use with a parameter, where a parameter value can assume any of the forms in the set. To be valid, the parameter definitions must specify the name of a defined form set. In some cases, this can require the definition of a form set of one form.

For example, the Parameter Entry Mode menu in the Component Parameters dialog box is configured by the form set for the selected parameter. The name of the form is used to configure the standard Component Parameters dialog box.

Forms and form sets are all defined using these AEL functions:

create_form_set()

create_text_form()

create_constant_form()

create_compound_form()

The following example defines a form set:

```
create_form_set("StdFileFormSet", "StdForm", "FileBasedForm");
```

Certain primitive forms are pre-defined and do not need to be created explicitly. Some of the pre-defined forms are:

StdFormSet

StdFileFormSet

ReadFileFormSet

StringAndReferenceFormSet

SingleTextLineFormSet

SingleTextLineIntegerFormSet

SingleTextLineDoubleFormSet

The files *stdforms.ael*, *stditems.ael*, and *pde_gemini.ael* contain many of the standard form and component definitions used by the program. These files can be found in the *<$HPEESOF_DIR>/de/ael* directory.

For details on forms and form set functions, see "Component Definition Functions" on page 15-1.

Creating and Using Custom Forms

In some cases a non-standard form is required to support unique syntax or to limit the options presented in a dialog box. Some of the necessary forms are unique to a simulator or to a particular design type. The program provides the ability to define new forms and their characteristics.

The characteristics of a form govern its handling in a Component Parameters dialog box, on-screen editing in the schematic and in the netlist. The form contains the instructions for formatting an instance parameter value for use in the display annotation and in the netlist. It also contains information to generate a list of value options for selection from a list in a Component Parameters dialog box. Also, the form can check the typed input for validity using the validate function.

Forms come in several classes that are provided with the program. A new form can be defined using one of these classes, where the new definition gives the form characteristics specific to the simulator/syntax requirements. The following built-in classes are characterized.

Constant Class

Used where the value is a constant and cannot be changed. Examples of this class are the forms for the SinE parameter of the LPF_RaisedCos component, where the allowed values are YES and NO. Two forms would be created, one for each of the possible values, and a form set would be created that referenced both forms. The definition for the SquareRoot parameter of the LPF_RaisedCosineTimed component would reference the form set defined. Selection would be made between the two allowed forms from an option menu in a dialog box, with no typing allowed.

Compound Class

Used to define a hierarchical parameter, where the parameter value is defined by several sub parameter values combined. Examples of this class are VarFormStdForm and DACForm. For this class, parameters are defined for the form using create_parm(), each of which references a form set. Since this form is indirectly referenced through another parameter definition, it creates a recursive definition (only one level deep). The parameter's value is composed of the multiple fields set in the form's parameter values.

Text Class

Used where a text value is required, but an arbitrary string is not appropriate. This occurs for parameters where a list of possible values should be generated at run time and presented in a dialog box, or where additional verification must be performed on the typed response in the dialog box. An example of this class is the msub parameter of the mlin element, where the text required is the instance name of an msub element. Optionally, a dialog box would provide a list of msub components available, and the selection could be made from the list or a name could be entered. In this case, checking would not be necessary, but a check function could be applied to verify that the entry was indeed a valid msub name. Although not supported by the user interface anymore, most parameters in the program are using this class of form.

Two rules should be applied to the naming of a form:

- All form names should be unique.
- If the form needs to be identified in the netlist, the name of the form should be the same as the identification string desired in the netlist.

The AEL functions that create new forms are *create_constant_form()*, *create_compound_form()*, and *create_text_form()*.

The forms created by create_constant_form() represent a fixed value, which is selected from a list of the possible values (normally from an option box) but does not edit textually. An example of this might be the definition of the Yes form:

```
create_constant_form("Yes", "YES", 0, "yes", "yes");
```

The forms created by create_compound_form() represent values that contain one or more parameters, some of which represent a value more complicated than a string. Each parameter has its own set of forms for the values it can accept. For example:

```
create_compound_form("LinearStart", "LinearStart", 0,
  "Start=%0s, Stop=%1s Step=%2s Lin=%3s",
  "Start=%0s, Stop=%1s Step=%2s Lin=%3s"

  create_parm("Start", "Start Value", 0 "FSTextForms",
    FREQUENCY_UNIT, prm("FSTextForm", "0")),

  create_parm("Stop", "Stop Value", 0 "FSTextForms",
    FREQUENCY_UNIT, prm("FSTextForm", "0")),

  create_parm("Step", "Step Value", 0 "FSTextForms",
    FREQUENCY_UNIT, prm("FSTextForm", "0")),

  create_parm("Lin", "Points in a Linear Sweep", 0 "FSTextForms",
    UNITLESS_UNIT, prm("FSTextForm", "0")));
```

The forms created by create_text_form() represent values that accept a string, but not just any string will do. A dialog box to present options for the string and additional checking of the typed string can be specified. A data string can be provided for use in the option list generation and value verification. An example might be the definition of the SingleTextLineIntegerForm form, used to represent a value that is an integer.

The value stdforms_validate_integer specifies a function that checks a value typed in by the user. If the value is an integer, the function returns 1 and the user-value is acceptable. If the value is not an integer, the function returns 0 and the user-value is not acceptable.

```
create_text_form("SingleTextLineInteger", "Integer Value",
  "SingleTextLine", 0, "%v", "%v", NULL, stdforms_validate_integer,
  NULL);
```

For details on these functions, see “Component Definition Functions” on page 15-1.

Format Strings

Format strings are used by a number of functions to construct a single string out of a number of different components. They are modeled on the basic printf capability in the C programming language, but include a large number of additional conversion specifiers. These strings provide a flexible way to control the formatting of complex value strings. Format strings control the way component parameters are displayed on the schematic. A standard format string is used by most components. A standard format string, called *ComponentNetlistFmt*, is used for the formatting of a component’s netlist line.

The basic form of these strings is a pair of quotation marks enclosing any number of strings and conversion specifiers. Conversion specifiers are preceded by the percent sign (%) and take the following form:

%nc

where n represents an optional integer value and c represents the instruction code. The default value of the optional integer value is 0.

Available Specifiers

The available specifiers are Loops, Conditionals, and Operators. The available specifiers and what each does (with optional integer after % and before the single letter instruction code) are listed in the following sections.

Loops

%b Begin parameter iteration loop, which shifts through each parameter in the list of parameters of a component instance or a compound form. The optional integer value (the *n* in %nc) indicates the initial parameter index, which sets the value of j counter. The j counter is incremented at the end of each parameter iteration.

%r Begin repeated parameter iteration loop, which checks if the current parameter (j counter) is a repeated parameter and iterates through each value in the list of values for the repeated parameter. The optional integer value (the *n* in %nc) indicates the initial repeat parameter index, which sets the value of i counter. The i counter is incremented at the end of each repeated parameter iteration. The value of a repeated parameter is stored as a list of values, where a normal parameter has a single value. For details on how to define a repeated parameter, refer to *create_parm()*. An example of a component with repeated parameters is SDD1P.

%h Begin hierarchical parameter iteration loop for compound form, where the parameter list is retrieved from the current parameter's compound form. The optional integer value (the *n* in %nc) indicates the initial parameter index, which sets the value of j counter for the hierarchical loop. The j counter for the hierarchical loop is incremented at the end of each hierarchical parameter iteration. If this is in a nested loop, the j counter value for the original parameter iteration loop (%b) is stored and retrieved as a top counter value.

%# Begin node iteration loop, which shifts through each node in the list of nodes of a component instance. The optional integer value (the *n* in %nc) indicates the initial

node index, which sets the value of l counter. The l counter is incremented at the end of each node iteration.

%> Begin global node iteration loop, which shifts through each global node in the global node list for the design hierarchy. The optional integer value (the *n* in %nc) indicates the initial global node index, which sets the value of g counter. The g counter is incremented at the end of each global node iteration.

%i Transfer the sum of repeated parameter iteration index (current i counter value) and the optional integer value (the *n* in %nc), to the output string.

%j Transfer the sum of parameter iteration loop index (current j counter value) and the optional integer value (the *n* in %nc) to the output string.

%l Transfer the sum of node iteration loop index (current l counter value) and the optional integer value (the *n* in %nc) to the output string.

%e End the most recent iteration loop. The iteration loop ends when the iteration counter for the loop is the same as the optional integer value (the *n* in %nc), or when the end of the list being iterated on is reached. The appropriate iteration counter is modified depending on the type of loop in progress. In addition, the format string is modified to facilitate the next iteration of the current loop or to end the current loop. Loops can be nested.

Conditionals

%F Set the fieldIndx so the next %29? test will apply to that field. The optional integer value (the *n* in %nc) indicates the field number. %2F … %29? Will test if 3rd field is an empty value. After %29? the fieldIndx is reset to –1.

%? Begin a conditional test, where the optional integer value (the *n* in %nc) indicates the type of test performed:

0 = TRUE if inside a repeated parameter loop (%r)

1 = TRUE if form of current parameter value is tunable

2 = TRUE if form of current parameter value has sub parameters (compound form)

3 = TRUE if form of current parameter value has special attributes

5 = TRUE if form of current parameter value is an equation

6 = TRUE if nominal value of current parameter is non-zero

7 = TRUE if component instance is mirrored

8 = TRUE if parameter is netlistable

9 = TRUE if component instance has a layout location available

20 = TRUE if design is a network

24 = TRUE if form of current parameter value is binary

25 = TRUE if form of current parameter value is octal

26 = TRUE if form of current parameter value is hexadecimal

27 = TRUE if form of current parameter value is numeric

28 = TRUE if form of current parameter value is string

29 = TRUE if current parameter has no value

30 = TRUE if PARM_RIGHT_HAND_ONLY attribute is set for the parameter.

31 = TRUE if node name for the specified node index (l counter) exists in the node name list.

32 = TRUE if external port name for the specified external port index (k counter) exists in the external port name list.

33 = TRUE if global node name for the specified global node index (g counter) exists in the global node name list.

35 = TRUE if any global node exists in the design.

37 = TRUE if the number of parameters for the instance is not zero.

38 = TRUE if PARM_NO_PLOT attribute is set for the parameter.

39 = TRUE if the global parameter AllParams is set for the instance.

41 = TRUE if INST_SCOPE_LOCAL attribute is set for the instance.

42 = TRUE if INST_SCOPE_ NESTED attribute is set for the instance.

43 = TRUE if INST_SCOPE_GLOBAL attribute is set for the instance.

44 = TRUE if the node is a ground node (node number = 0)

45 = TRUE if the value string starts with ";"

The format string following a test is optionally interpreted depending on a TRUE result of the test. The %: and %; delimit the end of the conditional part of the format string.

%: End TRUE branch of conditional format begun with %? And begin FALSE branch.

%; End of conditional format begun with %?

Operators

%g Transfer the current design name to the output string. The optional integer value (the *n* in %nc) is ignored.

%n Transfer the component name to the output string. The optional integer value (the *n* in %nc) is ignored.

%t Transfer the instance name (formerly instance tag) to the output string. The optional integer value (the *n* in %nc) is ignored.

%x Transfer the instance location value to the output string. The optional integer value (the *n* in %nc) indicates:

0 = X coordinate
1 = Y coordinate
2 = Orientation angle (degrees)

%d Transfer the data string to the output string. Data string can be component netlist data (*netlistData* argument in *create_item()*) or an array of form data strings (*dataValue* argument in *create_text_form()*). The optional integer value (the *n* in %nc) specifies the index into the array of form data. Index value other than 0 is invalid for netlist data. The default index value is 0.

%k Transfer the parameter name to the output string. The optional integer value (the *n* in %nc) indicates the desired parameter index relative to the current parameter index in the parameter list, with 0 for the current parameter. The current parameter index is determined by the value of j counter.

%p Transfer the parameter value, formatted using the netlistFormat string from the parameter's form definition, to the output string. This works like %s except that the netlistFormat is used. The optional integer value (the *n* in %nc) indicates the desired parameter index relative to the current parameter index in the parameter list, with 0 for the current parameter. The current parameter index is determined by the value of j counter.

%s Transfer the parameter value, formatted using the displayFormat string from the parameter's form definition, to the output string. This works like %p except that the displayFormat is used. The optional integer value (the *n* in %nc) indicates the desired parameter index relative to the current parameter index in the parameter

list, with 0 for the current parameter. The current parameter index is determined by the value of j counter.

%v Transfer the current parameter value to the output string. The optional integer value (the *n* in %nc) indicates the field index for forms having more than one field, with 0 for the first field (see example later). For constant forms, special control over formatting may be exercised by calling the dataFunction specified in *create_text_form()*. This format is used only in netlistFormat string or displayFormat string of form definitions.

%o Transfer the nominal value of a parameter that can be optimized. The optional integer value (the *n* in %nc) indicates the desired parameter index relative to the current parameter index in the parameter list, with 0 for the current parameter. The current parameter index is determined by the value of j counter.

%q Transfer the nominal value, evaluating all variable references, of a parameter that can be optimized. This works like %o except that variable references are evaluated. The optional integer value (the *n* in %nc) indicates the desired parameter index relative to the current parameter index in the parameter list, with 0 for the current parameter. The current parameter index is determined by the value of j counter.

%f Transfer the parameter value form name to the output string. The optional integer value (the *n* in %nc) is ignored. (Use after %p or %s operators).

%u Transfer the parameter value unit name to the output string. The optional integer value (the *n* in %nc) indicates the desired parameter index relative to the current parameter index in the parameter list, with 0 for the current parameter. The current parameter index is determined by the value of j counter.

%z Transfer unit conversion factors, standard usage of optional integer value (the *n* in %nc):

0 = FREQUENCY_UNIT to MKS
1 = RESISTANCE_UNIT to MKS
2 = CONDUCTANCE_UNIT to MKS
3 = INDUCTANCE_UNIT to MKS
4 = CAPACITANCE_UNIT to MKS
5 = LENGTH_UNIT to MKS
6 = TIME_UNIT to MKS
7 = ANGLE_UNIT to MKS
8 = POWER_UNIT to MKS
9 = VOLTAGE_UNIT to MKS

10 = CURRENT_UNIT to MKS
11 = DISTANCE_UNIT to MKS
12 = UNIT length to layout database
13 = User length to layout database
20 = MKS to FREQUENCY_UNIT
21 = MKS to RESISTANCE_UNIT
22 = MKS to CONDUCTANCE_UNIT
23 = MKS to INDUCTANCE_UNIT
24 = MKS to CAPACITANCE_UNIT
25 = MKS to LENGTH_UNIT
26 = MKS to TIME_UNIT
27 = MKS to ANGLE_UNIT
28 = MKS to POWER_UNIT
29 = MKS to VOLTAGE_UNIT
30 = MKS to CURRENT_UNIT
31 = MKS to DISTANCE_UNIT
32 = Layout database to UNIT length
33 = Layout database to User length

%c Transfer the node number to the output string. The optional integer value (the *n* in %nc) indicates the desired node index relative to the current node index in the node number list for the instance, with 0 for the current node number. The current node index is determined by the value of l counter.

%C Transfer the node name to the output string. The optional integer value (the *n* in %nc) indicates the desired node index relative to the current node index in the node name list for the instance, with 0 for the current node name. The current node index is determined by the value of l counter.

%m Transfer the external port number to the output string. The optional integer value (the *n* in %nc) indicates the desired external port index relative to the current external port index in the external port number list for the design, with 0 for the current external port number. The current external port index is determined by the value of k counter.

%M Transfer the external port name to the output string. The optional integer value (the *n* in %nc) indicates the desired external port index relative to the current external port index in the external port name list for the design, with 0 for the current external port name. The current external port index is determined by the value of k counter.

%w Transfer the global node number to the output string. The optional integer value (the *n* in %nc) indicates the desired global node index relative to the current global node index in the global node number list for the design hierarchy, with 0 for the current global node number. The current global node index is determined by the value of g counter.

%W Transfer the global node name to the output string. The optional integer value (the *n* in %nc) indicates the desired global node index relative to the current global node index in the global node name list for the design hierarchy, with 0 for the current global node name. The current global node index is determined by the value of g counter.

%% Transfer a single % character to the output string, integer value (the *n* in %nc) ignored.

Netlist Format String Examples

These examples illustrate the use of netlist format string in components and forms.

Example 1: Simple format string

A Compound form netlist format string "Start=%0v Stop=%1v Step=%2v"

Can be interpreted as follows:

Transfer the string “Start=” to output string (Start=)

Transfer the first field in parameter value to output string (%0v).

Transfer the string “Stop=” to output string (Stop=)

Transfer the second field in parameter value to output string (%1v).

Transfer the string “Step=” to output string (Step=)

Transfer the third field in parameter value to output string (%2v).

An example output string with this format string, for a compound form component parameter value list of 10, 20, 2 is:

“Start=10 Stop=20 Step=2”

Example 2: Simple iteration loop and conditionals

A node iteration loop format string %# %44?0%:%31?%C%:_net%c%;%;%e

Can be interpreted as follows:

Start node iteration loop at node index 0 (%#)

(For each node iteration perform all of the following)

If the node is a ground node (%44?)

Transfer “0” to output string (0)

Else (%:)

If there is a node name entry for the node index (%31?)

Transfer the node name to output string (%C)

Else (%:)

Transfer the string “_net” and the node number to the output string (_net%c)

End condition (%;)

End condition (%;)

End loop (%e)

An example output string with this format string for a component with four nodes, one connected to the ground and another named “Vcc” is:

“0 _net2 Vcc _net4”

Example 3: Nested iteration loops

A simplified nested parameter iteration loop(%b) and repeated parameter loop(%r) format string “ %b%r%k%?[%1i]%;=%p %e%e”

Can be interpreted as follows:

Start parameter iteration loop at parameter index 0 (%b)

(For each parameter iteration, perform all of the following)

Start a repeated parameter iteration loop at repeated parameter index 0 (%r)

(For each repeated parameter iteration perform all of the following)

Transfer the parameter name to the output string (%k)

If the parameter is a repeated parameter (%?)

```
        Transfer "[" string to the output string ( [ )
        Transfer the sum of 1 and the repeated parameter index to the output
      string (%1i)
        Transfer "]" string to the output string ( ] )
      End condition ( %; )
      Transfer the string "=" to output string ( = )
      Transfer the parameter value to the output string ( %p )
    End loop ( %e )
  End loop ( %e )
```

An example output string with this format string for a component with a repeated parameter S and another parameter Num is:

"S[1]=0 S[2]=1 Num=2"

Assigning Components to Groups

A new component must be assigned to a library group or palette group for placement. Library group and palette groups are associated with a particular type of design. The group definitions for each design type is distinct. Most likely, you will assign the new components you create to groups for the current network design, which are displayed in the schematic and layout windows in the program.

The functions used to assign components to groups are *set_design_type()*, *library_group()*, and *palette_group()*. For details on these functions, see "Component Definition Functions" on page 15-1.

An example of assignment of an component to a group is:

```
set_design_type(analogRFnet);
library_group("mstrip", "Microstrip components", "MLIN", "MBEND");
```

Two AEL functions are demonstrated in the example. The example assigns the components MLIN and MBEND to the library group *mstrip* and the library group is associated with analog/RF network designs.

Designing Component Schematic Symbols

Definition of a new component requires specification of a symbol to be used to represent the component when placed into a schematic. One of the standard symbols shipped with the product can be adequate for the new component. Usually, these are located in the application directories *<installation directory>/circuit/symbols* or *<installation directory>/hptolemy/symbols*.

Often a new symbol must be created to properly represent the part. The symbol is constructed in the symbol view for the schematic, where the name of the design is used as the name of the symbol. For a parametric subnetwork design, the symbol view for the design itself can contain the desired symbol. After switching to the symbol view, the commands on the Drawing palette or Draw menu are used to create the symbol shape. Symbol pins representing Ports must be placed into the symbol, with the first pin located at the origin (0, 0).

Designing Component Palette Buttons

When new components are defined, a bitmap name can be specified to represent the component button in a palette. If the named icon does not exist, the component name is displayed on the button. To add a customized button to the component pallette, a graphic to represent the component needs to be created, saved in the correct form, and placed in the appropriate directory.

On UNIX systems, any visual tool such as Icon Editor or xv will suffice and the bitmaps are saved as X pixmap files. On a PC, they can be created using any visual tool such as Paint or Microsoft Visual C++ and are saved in PC bitmap format. Regardless of the platform type or the tool used to create them, all bitmaps must have the following attributes:

- Size of 32 x 32 pixels
- 16 colors
- .bmp filename extension

The bitmap search path is defined at runtime by ADS via the environment variables XBMLANGPATH for UNIX platforms and WBMLANGPATH for PCs. Users can place bitmaps in predefined directories in the search path where they will be automatically found and displayed.

On UNIX platforms, the predefined directories are:

$HOME/hpeesof/custom/bitmaps - for individual user customization

$HPEESOF_DIR/custom/bitmaps - for site-wide customization

$HOME/hpeesof/circuit/bitmaps - for analog/RF customization

$HOME/hpeesof/de/bitmaps - for general PDE customization

$HOME/hpeesof/hptolemy/bitmaps - for DSP releated customization

On PCs the predefined directories are:

%HOME%\hpeesof\custom\bitmaps - for individual user customization

%HPEESOF_DIR%\custom\bitmaps - for site-wide customization

%HOME%\hpeesof\circuit\bitmaps - for analog/RF customization

%HOME%\hpeesof\de/bitmaps - for general PDE customization

%HOME%\hpeesof\hptolemy\bitmaps - for DSP releated customization

Developing Measured Component Libraries

A set of linear S-parameter measurement files can represent a library of parts for the simulator. This is accomplished by representing these as a library of parts that can be placed in a schematic. The complete set of S-parameter files would reside together in a directory. For example, the directory <*home directory*>*/hpeesof/myparts/parts/circuit* would hold a set of S-parameter files. In the same directory an AEL definition file, possibly named *circuitcomponents*, defines the components for the project environment, one component for each part, and the library group. The name of the directory is added to the AEL path for the SIMULATOR_AEL, directing the program to search and read the definition file when attaching to a project. The next time program is run, the new group name appears in the list of libraries in the schematic and layout windows. Any parts library would be handled in the same way.

The environment variable value for CIRCUIT_AEL is:

HOME_CIRCUIT_AEL={$HOME}/hpeesof/{%PROJECT1}/ael

where: PROJECT1=circuit and $HOME is your home directory on a UNIX platform.

Circuit components are defined in the file gemini.ael.

The circuit example, shown in Figure 4-2, creates a library of S-parameter file components. The AEL definition file and the S-parameter files can be stored in the same directory and the directory is added to the CIRCUIT_AEL path. First, a message that the library is being loaded is displayed on the terminal window. Then a form set is created specifying that the value of the S-parameter component will be an S-parameter file name. Finally, a component definition is created for each S-parameter file. The standard EEFET1 symbol is used for the schematic symbol artwork. The component is presented to the simulator as an S2P element with the name of the S-parameter file as the value of the FILE parameter. The component description field is used to display the *Vds* and *Ids* values for the S-parameter file. This line, which clearly identifies these values, appears in the library list dialog box. The contents of the *linear.ael* file are shown in Figure 4-2 and the contents of the *d_0777.s2p* file are shown in Figure 4-3.

```
fputs(stderr,"Reading S-Parameter Library: sp_lin_40");

create_file_name_form_set("sp_lin_40","s-parameter files", "SP_LIN_40", "s2p" );

create_item( "sp40_lin_d_0777", "Vds=3.5 Ids=15", "S",NULL, NULL, NULL, standard_dialog, "d_0777
fet", special_netlist, "ELEMENT S2P", standard_symbol, "EEFET1", no_artwork, NULL, create_parm(
"File", "Tdata model file", 2, "sp_lin_40", -2, prm( "sp_lin_40" , "d_0777" ) ), create_parm( "TEMP",
"TEMP name", 0, "tempname", -2, prm( "def*" )));

create_item( "sp40_lin_d_1503", "Vds=3.5 Ids=15", "S", NULL, NULL, NULL, standard_dialog, "d_1503
fet", special_netlist, "ELEMENT S2P", standard_symbol, "EEFET1", no_artwork, NULL, create_parm(
"File", "Tdata model file", 2,"sp_lin_40", -2, prm( "sp_lin_40", "d_1503")), create_parm( "TEMP",
"TEMP name", 0, "tempname", -2, prm("def*")));

create_item( "sp40_lin_d_2501", "Vds=3.5 Ids=15", "S", NULL, NULL, NULL, standard_dialog, "d_2501
fet", special_netlist, "ELEMENT S2P", standard_symbol, "EEFET1", no_artwork, NULL, create_parm(
"File", "Tdata model file", 2, "sp_lin_40", -2, prm( "sp_lin_40", "d_2501")), create_parm(
"TEMP","TEMP name", 0, "tempname", -2, prm("def*")));

create_item( "sp40_lin_d_2502", "Vds=3.5 Ids=15", "S", NULL, NULL, NULL, standard_dialog, "d_2502
fet", special_netlist, "ELEMENT S2P", standard_symbol, "EEFET1", no_artwork, NULL,
create_parm("File","Tdata model file", 2, "sp_lin_40", -2, prm( "sp_lin_40", "d_2502")),
create_parm("TEMP", "TEMP name", 0, "tempname", -2, prm("def*")));

 library_group("sp_lin_40","S-parameters",
"sp40_lin_d_0777",
"sp40_lin_d_1503",
"sp40_lin_d_2501",
"sp40_lin_d_2502"
```

Figure 4-2. Measured Component Library Example

```
D-0777 (CHIP) ULTRA LOW NOISE FET
! VDS= 3.5V, IDS= 15mA
! NOTE: S-PARAMETERS INCLUDE 0.7 mil BOND WIRES
# GHZ S MA R 50
! FREQ      S11          S21          S12          S22
! GHZ   MAG    ANG    MAG    ANG    MAG    ANG    MAG    ANG
  2     .97     -22    3.40   161    .03    77     .64    -13
  4     .93     -41    3.11   145    .06    66     .62    -21
  6     .88     -55    2.84   133    .08    60     .59    -26
  8     .82     -67    2.66   122    .09    54     .56    -32
  10    .75     -82    2.55   110    .10    48     .51    -38
  12    .68     -100   2.42    97    .10    40     .45    -48
  14    .61     -118   2.22    84    .10    38     .43    -59
  16    .61     -141   2.02    71    .10    35     .43    -75
  18    .61     -157   1.74    59    .10    32     .43    -84
```

Figure 4-3. Measured Component Example Database

Developing Discrete Valued Part Libraries

The Advanced System Design simulators have the capability of using discrete measurement data to simulate manufactured components with discrete values. Discrete measured data is provided in a Microwave Data Interchange Format (MDIF) file. The file organization allows the description of named sets of data, where each set represents a discrete value of a part. You can create a component definition in AEL that allows you to pick only valid discrete values for a part. For example, you can have measured data for a capacitor in the form of capacitance and series resistance for several nominal values of the capacitor, such as 20, 22, 24, 27, and 30 pf. These values are organized into a MDIF file, a subnetwork is created to model the physical part, and an AEL component definition is created that allows only the discrete values to be chosen for analysis in the subnetwork. Also, the simulator's discrete value optimizer uses only the valid discrete values.

To create a set of discrete values components for simulation:

- Develop a model for the part and measure the model parameters for the desired nominal values of the part.
- Format the measured values into a MDIF file.
- Build a parametric subnetwork for the part model, where one of the pass parameters represents the nominal value.

- Create the AEL component definition for use of the parametric subnetwork as a discrete valued part.

Developing Discrete Valued Parts MDIF File

The values available for discrete valued parts are defined in a file using a subset of the Microwave Data Interchange Format (MDIF). This file format is used to define a variety of data for use by the simulators. The file to define discrete values contains a table of values, which are accessed by row index and column name. An example of a simple MDIF file format for discrete valued parts is:

```
REM This is an example file for discrete capacitors
BEGIN DSCRDATA
% PART  C   RS
20pF    20  2.1
22pF    22  1.9
24pF    24  1.75
27pF    27  1.56
30pF    30  1.4
END DSCRDATA
```

This example has five sets of values, where each set consists of two values. The REM line at the top of the file is an optional comment. The BEGIN DSCRDATA line marks the beginning of the data table and gives the table the required name DSCRDATA. The next line, beginning with a percent sign, names the columns in the data table. The name for the first column (PART) is not significant. This column contains the names for the rows used by the program when accessing a particular set of values. The names of the remaining columns are used to access the individual data values for a particular row. There must be at least one column of data values beyond the first column. The lines following define sets of data values, one row at a time, until the END DSCRDATA line ends the table. Only one data table is allowed in a file. For use by the local project, the MDIF file should be placed in the project's data subdirectory.

Designing a Discrete Valued Parts Parametric Subnetwork

The model for the discrete valued part is contained in a parametric subnetwork. Figure 4-4 shows the schematic for the discrete valued parts parametric subnetwork.

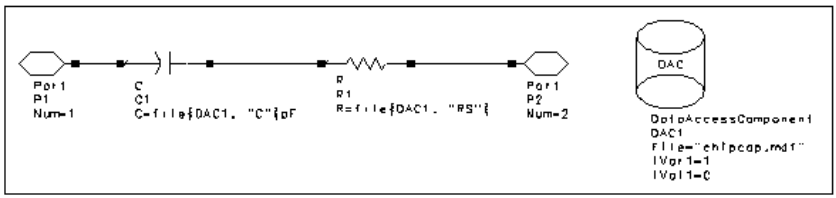


Figure 4-4. Discrete Valued Parts Parametric Subnetwork Schematic

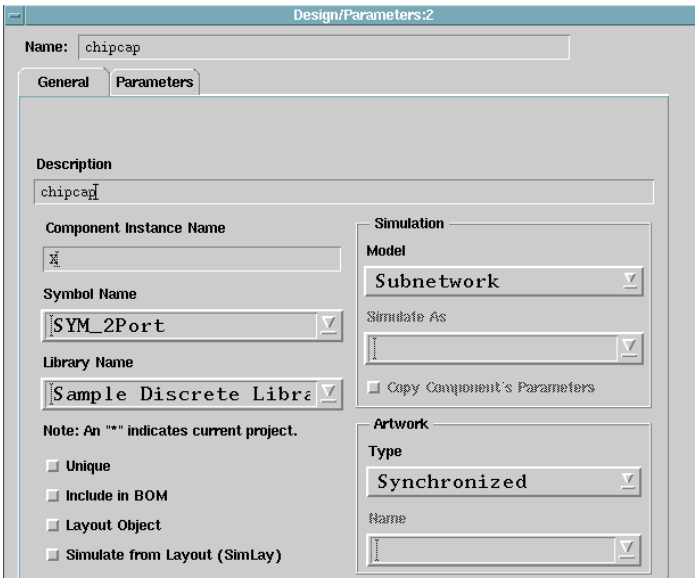
Model components are placed and connected in the network in the usual way, except for the addition of references to the discrete data file. This involves adding a DataAccessComponent (DAC) and setting certain model component parameter values to refer to the DAC. One parameter for the subnetwork is used to represent the nominal value of the part.

Note The units of the components placed in the schematic are assumed to be in SI units. To enter the value of capacitance in pF, you must edit the unit in the schematic and add pF to “file{DAC1, “C”}”, as shown in Figure 4-4.

The basic requirements for designing a discrete valued parts parametric subnetwork are to define the subnetwork component, the symbol, and the subnetwork parameter for use as the nominal value.

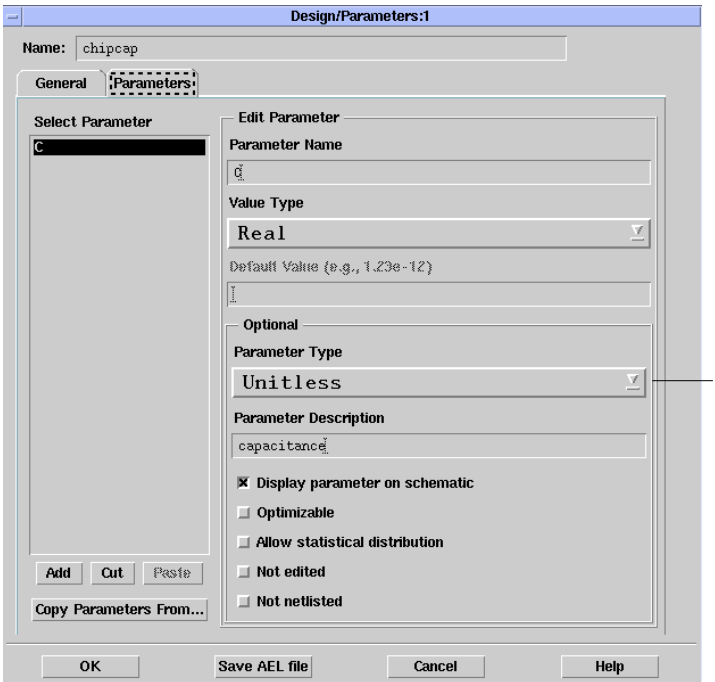
1. Create the subnetwork by placing and connecting the model components and the ports, as shown in Figure 4-4. Save the design. In this example, chipcap is used as the design name.
2. Select **File** > **Design/Parameters** and click **General**.
   - To save the new item in an existing custom library group, select the group from the **Library Name** list.

- To save the new item in a new custom library group, select the Library Name field and enter the name. The default Library Name is “*”, which indicates the current project. In this example, **Sample Discrete Library Group** is entered as the custom library.



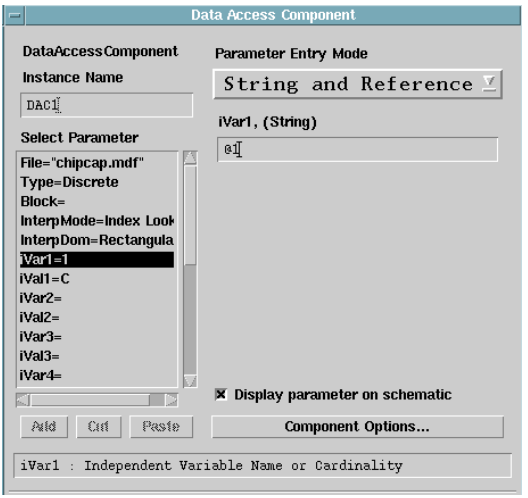
3. Select **Parameters**.

- Select the Parameter Name field and enter the name. In this example C is entered as a parameter.
- Set Parameter Type to **Unitless**.
- The Value Type and Default Value are not important.
- Save the AEL file by clicking **Save AEL File**. Then close the dialog by clicking OK.



4. In the schematic window, place a **DataAccessComponent (DAC)** from the Data Items library. Open the Edit Component dialog by double-clicking on the DataAccessComponent (DAC) instance.

   - Set the **File** parameter to the name of the MDIF data file. Include the *mdf* extension.
   - Confirm that **File Type** = **Discrete,** to set the parameter values for certain components of the model which reference individual values from the data file. When this type is used, the instance name of the DataAccessComponent (DAC) and the name of the data column provide the link to the data file.
   - Confirm that **InterpMode=Index Lookup** and **InterpDom=Rectangular**.
   - Set the index parameter (**iVal1**) to the subnetwork parameter used for the nominal value. When entering the Independant Variable Name or Cardinality value of iVar1, enter as @1 so that it will not appear in quotes.



5. Open the Edit Component dialogs for components of the model and reference individual values from the data file, by selecting File Based Parameter Entry Mode and entering the appropriate Dependent Parameter Name, as shown in Figure 4-4.

6. Save the design.

After the parametric subnetwork has been created and saved, you must modify the AEL definition to make the part usable. When any design is saved, an AEL definition is created for the subnetwork so that it can be placed in another design. For parametric subnetworks that represent discrete parts, the AEL definition created above does not allow the part to be handled as a discrete valued part. You must modify the AEL definition each time the parameters for the network are modified. The modification is described in the section "Discrete Valued Parts Required AEL Definitions" on page 4-23.

Discrete Valued Parts Required AEL Definitions

The AEL component definition created when saving a parametric subnetwork must be modified in order for the network to be used as a discrete valued part. Figure 4-5. shows the modified AEL component definition for the subnetwork design in Figure 4-4. In the figure, the lines you must modify are highlighted. In addition, you must remove the line:

library_group("*", "*", 1, "chipcap");

if you do not want the item to appear in the current project, but only to appear in the designated library group (in this example, Sample Discrete Library Group).

```
set_simulator_type(-1);
set_design_type(1);

create_constant_form ("chipcap_20pF", "20pF", 68, "0", "20pF");
create_constant_form ("chipcap_22pF", "22pF", 68, "1", "22pF");
create_constant_form ("chipcap_24pF", "24pF", 68, "2", "24pF");
create_constant_form ("chipcap_27pF", "27pF", 68, "3", "27pF");
create_constant_form ("chipcap_30pF", "30pF", 68, "4", "30pF");

create_form_set ("chipcap_values", "chipcap_20pF", "chipcap_22pF", "chipcap_24pF",
"chipcap_27pF", "chipcap_30pF");

create_compound_form ("chipcap_index", "Discrete optimize", "DistLibForm", 68, "%p opt{discrete
%1p to %2p by 1}", "%s,%1s to %2s",
   create_parm ("NOM", "Nominal", PARM_DISCRETE_VALUE, "chipcap_values", UNITLESS_UNIT,
prm("chipcap_20pF")),
         create_parm ("MIN", "Minimum", PARM_DISCRETE_VALUE, "chipcap_values", UNITLESS_UNIT,
prm("chipcap_20pF")),
   create_parm ("MAX", "Maximum", PARM_DISCRETE_VALUE, "chipcap_values", UNITLESS_UNIT,
prm("chipcap_30pF")));

create_form_set ("chipcap", "chipcap_20pF", "chipcap_22pF", "chipcap_24pF", "chipcap_27pF",
"chipcap_30pF", "chipcap_index", "value");

create_item("chipcap", "chipcap", "X", 16, -1, NULL, "Component Parameters", "",
    "%d:%t %f %44?0%:%31?%C%:_net%c%;%;%e %b%r%8?%29?%:%30?%p %:%k%?[%1i]%;=%p %;%;%;%e%e",
    "chipcap", "%t%b%r%38?%:\n%39?all_parm%A%:%30?%s%:%k%?[%1i]%;=%s%;%;%;%e%e%;",
    "SYM_C", 3, NULL, 0,
   create_parm("C", "capacitance", PARM_DISCRETE_VALUE, "chipcap", UNITLESS_UNIT,
prm("chipcap_20pF")));
```

Figure 4-5. AEL Definition for Discrete Value Component

Supporting User-Defined Simulator Components

The Advanced Design System simulators support the capability for augmenting the built-in simulator element set with new components that you define by writing C language functions that are linked with the simulator object archive to produce a simulator executable.

Note For details, refer to the section "*Building User-Compiled Models*" in the *Model Builder* manual.

After the user-defined modules have been linked to the simulator, you need to create an AEL definition to place and use the element in a schematic. You accomplish this by creating a component for each user-defined element.

In AEL, user-defined components are described in the same way as built-in components. For each parameter of the element, a parameter definition is created using the *create_parm()* function. The parameters are added to the component using *create_item()*, along with the basic component definitions. Finally, the component is specified in a library group, or optionally, assigned to a palette group and given a bitmap.

An important consideration in describing the user-defined component is assuring the correct parameters. The parameters should match in order, name and type, between the AEL component description and those being used in the user-defined module. In particular the *UserParamType* array definition needs to match the parameters described by *create_parm()*. Also, the netlist string provided in the *create_item()* call needs to produce the proper simulator syntax. Unless the component has unusual parameters, the pre-defined *standard_netlist* string can be used.

Note The netlist data string for all user-defined components should be either ELEMENT or DATA. ELEMENT should be used for user-defined components with one or more ports/pins, and DATA used for components with none.

In the following example, the standard *U2PA* element is defined. Only the definitions for the user-defined element are described here:

```
LOCAL UserParamType
U2PA[] =
```

```
{
{"R1", REAL_data}, {"R2", REAL_data}, {"R3", REAL_data}
};

LOCAL UserElemDef user_components[] =
{
{"U2PA", 2, sizeof(U2PA), U2PA, NULL, u2pa_y, thermal_n, NULL, NULL,
NULL},
};
```

The corresponding AEL component definition for this element would be:

```
create_item( "U2PA", "User Defined Element", "I", NULL,NULL, "U2PA",
  standard_dialog, "*", standard_netlist, "ELEMENT", standard_symbol,
  "ARES", no_artwork, NULL,
create_parm( "R1", "Resistance of R1", 0, "rvopt",1,1.000000),
create_parm( "R2", "Resistance of R2", 0, "rvopt",1,1.000000),
create_parm( "R3", "Resistance of R3", 0, "rvopt",1,1.000000),
));
```

In the component definition file, the component is defined as U2PA and the component ID prefix is I. The component uses the standard netlist format for components and the ARES symbol, which needs to be created. It has no associated artwork and has three optimizable parameters with the default value of 1.0.

Note For details on creating a symbol, refer to "*Chapter 10, Working with Symbols*" in the *User's Guide*.

Defining Artwork Creation Functions

If you have purchased the Layout option, you have the ability to define parameterized custom layout functions using AEL. The full power of the language is available, including artwork generation functions, math libraries, and data query functions.

Several AEL functions are especially useful for generating layout artwork:

```
db_factor()
mks_factor(u)
ang_factor()
de_set_layer(n)
```

de_set_global_db_factor()
de_draw_rect(x, y, l, h)
de_draw_circ(x, y, r)
de_draw_text(f, x, y, h, a, string)
de_draw_port(x, y, a, n)
de_add_path()
de_add_polygon()
de_add_polyline()
de_end()
de_end_command()
de_draw_point(x, y)
de_draw_arc(x, y, a, f)

Artwork functions are assigned in the *create_item()* function. The name of the artwork function is set as the artwork data and the artwork type is set to *artwork_macro* or the integer 2.

Artwork functions use parameters passed into them to control the dimensions and features of the layout geometry. The basic requirement in all AEL artwork creation macros is that the parameters defined for the macro match the exact order of the parameters defined for the component. That is, the names of the parameters used in the macros *defun* statement do not need to match the component's, but the order of the parameters needs to match the order in which they are defined in the *create_item()* function. If W is defined before L in *create_item()*, then W is the first parameter passed in and L the second.

An AEL artwork macro can use all available AEL commands, including file I/O, string manipulation, data base query, lists, math functions, and can also call other AEL artwork functions. If you are constructing a library of artwork generation routines, you can want to build a set of AEL utilities to simplify tasks that are repeated in more that one function.

The basic format of the artwork function is similar to any AEL function. It is composed of a declaration section (*defun*, followed by a list of parameters), followed by the function body. The function body can contain any number of AEL statements and usually contains a call to a draw function, as well as calls to draw port to create the component's ports. The number of ports needs to match that of any schematic symbol defined for the component. Further, port 1 should always be placed at the 0,0 point.

When using the polygon or polyline command functions to draw complex shapes, follow these by two or more *de_draw_point()* functions. The *de_draw_point()*

functions should always be terminated with a call to the *de_end()* function, followed by the *de_end_command()*. This example draws a simple polygon:

```
.
.
.
de_set_layer(iDrawLayer);
de_draw_rect(0, -fWidth * 0.5, fLength, fWidth * 0.5 );
de_draw_port(0, 0, -90.0);
de_draw_port(fLength, 0, 90.0);
.
.
.
```

The simple artwork macro shown next creates a microstrip transmission line. The function is passed in the width and length of the transmission line as the second and third arguments (note the order of the parameter's *fWidth* is the same as in the *create_item()* function for the MLIN). The width and length parameters are then used to create a rectangle and specify the port 2 location. A more complete version of this macro can be found for in the <installation directory>/circuit/ael/ckt_linear_art.ael file.

```
defun MLIN(w,l)
{
de_set_layer(1);
de_draw_rect(0,-w*.5,l,w*.5);
de_draw_port(0,0,-90.0);
de_draw_port(l,0,90.0);
}
```

Creating a Library of Artwork Objects

Typically, many users have developed a library of artwork to use in designs, such as simple shapes used for drill holes, cover alignment, and mask alignment or parts developed for resistors, capacitors, FETs and other components.

Often, the former are not part of the active circuitry and are not simulated; however, the later are active components which need to be simulated. The AEL definitions for these two types of objects differ, and the method for creating the supporting AEL can differ as well.

AEL for Simulated Components with Artwork

Generally, you would create new components that can be simulated by following the directions in the *Layout* manual. The method outlined there involves creating a schematic to represent the simulation model for the artwork.

However, a number of components have a one-to-one correspondence between a single simulator element and the artwork for the element. This is especially true with FETs, but is often true with lumped components and others. In this case it is simpler to create a new component definition using AEL, than to create a schematic with just this one element placed in it.

The basic idea is to copy the AEL *create_item()* definition for the built-in simulator element, rename and modify the definition to associate it with a artwork file or AEL artwork generation macro. Though the *create_item()* definition for a simulated component can look imposing, you only need to modify a few fields. Then, you add your new AEL files to the appropriate AEL configuration variable so the new components can appear in the library.

An example of the *create_item()* definition for a capacitor is described in the next section, "Example Artwork Creation Functions" on page 4-30.

Note the changed parameters in *create_item()*: the name, label, artwork type and artwork name in the fixed artwork example, plus the addition of a new parameter in the macro example. The artwork types are fixed, macro and none, the type can be set with the pre-defined variables *no_artwork*, *fixed_artwork* or *macro_artwork*.

If the type is fixed or macro, you need to supply the design file name (minus the *.dsn* extension) or the macro function name. If the type is a macro, be sure that you load the AEL file containing the macro, or you can include it with the component definition, as is done here.

For details on creating artwork macros, refer to the *Circuit Components* manual. Note that the macro's parameters are exactly the same as the capacitor's. The names do not have to match, but the position of each parameter (their order in the list) does. You can use this example parameter definition for all layout-only parameters you wish to add to an component. Most likely, you will only need to change the parameter name and default value.

Finally, a *library_group* statement is added to associate the new components with a new library group. You can associate new components with an existing group or name a new group. You can have any number of library groups. If you want the components

to show up in a palette as well, you can use the *palette_group* function to define a similar group for palettes.

All existing AEL definitions for a simulator are stored in the install directory. You should not modify these files directly, but copy the information you need to a new file and add the new file to the AEL search path.

Example Artwork Creation Functions

This example is a more complex artwork generation function that creates the CAPP2_Pad1 component. A number of supporting routines are included to demonstrate the use of generalized utility functions when creating a library of components. This function and the supporting code for Analog/RF designs can be found in these files:

- $HPEESOF_DIR/circuit/ael/ckt_lumped_item.ael file (section of CAPP2_PAD1)
- $HPEESOF_DIR>/circuit/ael/ckt_linear_art.ael file
- $HPEESOF_DIR/de/ael/destdart.ael (section of defun pad1)

where $HPEESOF_DIR is your installation directory (on UNIX platform).

The sample code is shown in Figures 4-6, 4-7, 4-8, and 4-9.

```
// sample code in $MPEESOF_DIR/circuit/ael/ckt_lumped_item.ael
// CAPP2 - Pad1
create_item ("CAPP2_Pad1",              // name
             "Chip Capacitor (Pad Artwork)",          // label
             "C",                       // prefix
             0,                         // attribute
             -1,                        // priority
             "CAPP2_Pad1",              // iconName
             standard_dialog,           // dialogName
             "*",                       // dialogData
             ComponentNetlistFmt,       // netlistFormat
             "CAPP2",                   // netlistData
             ComponentAnnotFmt,         // displayFormat
             "SYM_C",                   // symbolName
             macro_artwork,             // artWorkType
             "CP2_Pad1",                // artworkData
             ITEM_PRIMITIVE_EX,         // extraAttrib
             create_parm("C", "Capacitance", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", CAPACITANCE_UNIT, prm("StdForm", "1.0 pF")),
             create_parm ("TanD", "dielectric loss tangent", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", UNITLESS_UNIT, prm("StdForm","0.001")),
             create_parm ("Q", "quality factor", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", UNITLESS_UNIT, prm("StdForm","50.0")),
             create_parm ("FreqQ", "reference frequency for q", PARM_OPTIMIZABLE |
PARM_STATISTICAL, "StdFileFormSet", FREQUENCY_UNIT, prm("StdForm","300.0 MHz")),
             create_parm ("FreqRes", "resonance frequency", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", FREQUENCY_UNIT, prm("StdForm","500.0 MHz")),
             create_parm ("Exp", "exponent for frequency dependance of q", PARM_OPTIMIZABLE |
PARM_STATISTICAL, "StdFileFormSet", UNITLESS_UNIT, prm("StdForm","2.0")),
             create_parm ("W",  "W", PARM_NOT_NETLISTED,  "StdFormSet", LENGTH_UNIT,
prm("StdForm","25.0 mil"), list(dm_create_cb(PARM_DEFAULT_VALUE_CB,
"get_default_length_value_cb", "25.0", TRUE))),
             create_parm ("S",  "S", PARM_NOT_NETLISTED,  "StdFormSet", LENGTH_UNIT,
prm("StdForm","10.0 mil"), list(dm_create_cb(PARM_DEFAULT_VALUE_CB,
"get_default_length_value_cb", "10.0", TRUE))),
             create_parm ("L1", "L", PARM_NOT_NETLISTED,  "StdFormSet", LENGTH_UNIT,
prm("StdForm","50.0 mil"), list(dm_create_cb(PARM_DEFAULT_VALUE_CB,
"get_default_length_value_cb", "50.0", TRUE))));
```

Figure 4-6. Artwork Generation Function Example (ckt_lumped_item.ael)

```
// sample code in $HPEESOF_DIR/circuit/ael/ckt_lumpedart_def.ael
library_group("lumpedart",
              "Lumped Components (with artwork)",
              "SMT_Pad",
              "C_Pad1",
              "C_Space",
              "C_Conn",
              "CAPP2_Pad1",
              "CAPP2_Space",
              "CAPP2_Conn",
              "CQ_Pad1",
              "CQ_Space",
              "CQ_Conn",
              "L_Pad1",
              "L_Space",
              "L_Conn",
              "LQ_Pad1",
              "LQ_Space",
              "LQ_Conn",
              "R_Pad1",
              "R_Space",
              "R_Conn");
```

Figure 4-7. Artwork Generation Function Example (ckt_lumped_item.ael)

```
// sample code in $HPEESOF_DIR/circuit/ael/ckt_linear_art.ael
defun CP2_Pad1 (c,tand,q,fq,fr,exp,w,s,l)
{
    elem = "CP2";
    pad1(w,s,l,xlatorLayer);
}
```

Figure 4-8. Artwork Generation Data Example (ckt_linear_art.ael)

```
// sample code in $HPEESOF_DIR/de/ael/destdart.ael
//PAD1
defun pad1(w,s,l,layer)
{
  de_set_global_db_factor();
  if(w <= 0.0)
    error(DeStdArtErrClass,2,fmt_tokens(list(DeStdArtElem,"Width <= 0.0
:",w)),"");
  if(s <= 0.0)
    error(DeStdArtErrClass,3,fmt_tokens(list(DeStdArtElem,"Spacing <= 0.0
:",s)),"");
  if(l <= 0.0)
    error(DeStdArtErrClass,1,fmt_tokens(list(DeStdArtElem,"Length <= 0.0
:",l)),"");
  if (s + s > l )
    error(DeStdArtErrClass,4,fmt_tokens(list(DeStdArtElem,"Two Spacings >
Length :",s,l)),"");

  de_set_layer(layer);
  de_draw_rect(0,-w*.5, s,w*.5);
  de_draw_rect(l-s,-w*.5, l,w*.5);
  de_draw_port(0,0, -90, FALSE,1);
  de_draw_port(l, 0, 90, FALSE,2);
}
```

Figure 4-9. Artwork Data Example (destdart.ael)

Modifying AEL Configuration Variables

After you have created AEL definitions for a new library, you must modify the appropriate configuration variables for the file with the new definitions to be loaded automatically. Which variable you modify depends on who will use the definitions.

Chapter 5: AEL Function Reference

The chapters comprising the AEL function reference provide complete details of all functions available from the environment.

Note For a complete list of available functions, see “List of Functions” on page 5-3. For a list of obsolete functions, see “Obsolete Functions” on page 5-15.

The available functions are categorized as:

Common AEL Functions

Design Environment Functions

The synopsis of each function consists of the same format: a function name, followed by an open parenthesis, followed by zero or more arguments separated by commas, followed by a closing parenthesis, and finally a semi-colon. For example,

```
fputs(stderr, “hello world!”);
```

The arguments may be in the form of a string, number (real or integer), list, or variable reference. For detailed information on formats, refer to the section "Language Specifics" on page 1-2.

You can type these functions directly into the *Command Line* dialog box, which is accessed from the Options menu in the Main window. Or, you can add functions to a file, then load and execute the file using *Options > Playback Macro* from the Main window. For additional options, see the section "Writing, Loading and Testing AEL Functions" on page 1-18.

Many of the functions refer to the active window or representation. Any function that modifies a design representation, acts on the active representation. An active representation is any design in a currently-open window. You can select a window using the *de_set_window()* function.

Several functions refer to simulator or user units. User units are set in the schematic or layout *preference* files. These units control the dimensions of geometric forms. Simulator units are always MKS units.

List of Functions

This section provides a complete list of active and obsolete functions, arranged alphabetically.

A

B

C

D

De_

De

Dm

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Obsolete Functions

activate()

See “de_activate()” on page 12-1.

add_layer()

See “de_add_layer()” on page 13-1.

add_point()

See “de_add_point()” on page 12-6.

add_vertex()

See “de_add_vertex()” on page 12-10.

add_wire()

See “de_add_wire()” on page 12-10.

analyze()

See “de_analyze()” on page 17-3.

ang_factor()

See “de_ang_factor()” on page 11-1.

arc()

See “de_add_arc()” on page 12-1.

attach_project()

See “de_open_project()” on page 12-57.

bell()

See “de_warning_bell()” on page 10-8, “de_set_warning_bell()” on page 10-7, “de_error_bell()” on page 10-6, “de_set_error_bell()” on page 10-7.

break_connection()

See “de_break_connection()” on page 12-13.

change_annotation_layer()

See “de_change_annotation_layer()” on page 12-13.

change_units()

See “de_change_units()” on page 12-14.

circle()

See “de_add_circle()” on page 12-5.

clear_all()

See “de_close_all()” on page 12-16.

clear_all_grids()

Deleted

clear_design()

See “de_close_design()” on page 12-16.

clear_highlighting()

See “de_clear_highlighting()” on page 12-15.

clear_rep()

See “de_clear_rep()” on page 12-16.

close_window()

See “de_close_window()” on page 12-17.

config_window()

See “de_config_window()” on page 12-17.

connect()

See “de_connect()” on page 12-18.

convert_to_polygon()

See “de_convert_to_polygon()” on page 12-18.

convert_traces_to_instances()

See “de_convert_traces_to_instances()” on page 12-19.

copy()

See “de_copy()” on page 12-19.

copy_design()

See “de_copy_design()” on page 12-20.

copy_project()

See “de_copy_project()” on page 12-20.

copy_to_buffer()

See “de_copy_to_buffer()” on page 12-21.

copy_to_layer()

See “de_copy_to_layer()” on page 12-21.

create_array_form

Deleted

create_project()

See “de_new_project()” on page 12-57.

create_window()

See “de_create_window()” on page 12-22.

current_design_name()

See “de_current_design_name()” on page 11-1.

current_design_type()

See “de_current_design_type()” on page 11-2.

data_dialog()

See “de_data_dialog()” on page 18-6.

default_design_name()

Deleted

define_npport()

See “de_define_npport()” on page 12-23.

define_port()

See “de_define_port()” on page 12-24.

delete()

See “de_delete()” on page 12-24.

delete_all_orphaned_instances()

See “de_delete_all_orphaned_instances()” on page 12-25.

delete_design()

See “de_delete_design()” on page 12-25.

delete_project()

See “de_delete_project()” on page 12-25.

delete_vertex()

See “de_delete()” on page 12-24.

demand_library_group()

Deleted

demand_palette_group()

Deleted

deselect_all()

See “de_deselect_all()” on page 12-26.

deselect_all_force()

See “de_deselect_all_force()” on page 12-26.

deselect_by_name()

See “de_deselect_by_name()” on page 12-27.

deselect_item()

Deleted

deselect_window()

See “de_deselect_window()” on page 12-27.

de_attach_project()

See “de_open_project()” on page 12-57.

de_clear_all_grids()

Deleted

de_create_project()

See “de_new_project()” on page 12-57.

de_delete_vertex()

See “de_delete()” on page 12-24.

de_init_iteminfo()

See “de_init_item()” on page 12-47.

de_init_tune()

“de_tune()” on page 17-4.

de_iteminfo_edit_instance()

See “de_edit_item()” on page 12-35.

de_iteminfo_new_instance()

See “de_place_item()” on page 12-64.

de_load_design()

See “de_open_design()” on page 12-57.

de_load_grid()

See “de_open_datadisplay()” on page 17-3.

de_move_vertex()

See “de_move()” on page 12-54.

de_named connection()

Deleted

de_new_text()

Deleted

de_open_grid()

See “de_new_datadisplay()” on page 12-56.

de_optimization_continue()

Deleted

de_performance_optimization

Deleted

de_place_orphan()

Deleted

de_query_iteminfo_attr()

Deleted

de_redisplay()

See “de_refresh_view()” on page 12-67.

de_remove_prop()

See “de_remove_properties()” on page 12-68.

de_restore_all_grids

See “de_restore_all_datadisplay()” on page 17-4.

de_save_grid()

See de_save_datadisplay().

de_send_netlist()

Deleted

de_set_add_optional_parameters()

Deleted

de_set_iteminfo_attr()

Deleted

de_set_named connection()

Deleted

de_set_part_size()

See “de_set_port_size()” on page 13-28.

de_set_trace_mcover_id()

Deleted

de_set_trace_msub_id()

Deleted

de_set_trace_mwall_id()

Deleted

de_set_trace_sub_id()

Deleted

de_set_trace_tand_id()

Deleted

de_set_trace_temp_id()

Deleted

de_sim_file_command()

Deleted

de_statistical_analysis

Deleted

de_translate_design()

Deleted

de_tune_item()

See “de_tune()” on page 17-4.

de_unmap_grids

See “de_close_all_datadisplay()” on page 17-3.

de_viewAll()

See “de_view_all()” on page 12-91.

de_yield_optimization()

Deleted

dm_num_parm_definition()

See “dm_num_parm_definitions()” on page 16-7.

draw_arc()

See “de_draw_arc()” on page 12-27.

draw_circ()

See “de_draw_circ()” on page 12-31.

draw_point()

See “de_draw_point()” on page 12-31.

draw_port()

See “de_draw_port()” on page 12-32.

draw_rect()

See “de_draw_rect()” on page 12-32.

draw_text()

See “de_draw_text()” on page 12-33.

dse_l2s()

See “de_dse_l2s()” on page 12-33.

dse_place_orphan()

See de_place_orphan().

dse_s2l()

See “de_dse_s2l()” on page 12-34.

edit_annotation_attribute()

See “de_edit_annotation_attribute()” on page 12-34.

edit_instance()

See “de_edit_item()” on page 12-35.

edit_item_parameters()

See “de_set_item_parameters()” on page 12-78.

edit_path_trace()

See “de_edit_path_trace()” on page 12-35.

edit_symbol_pin()

See “de_edit_symbol_pin()” on page 12-36.

edit_text_attribute()

See “de_edit_text_attribute()” on page 12-36.

edit_text_string()

See “de_edit_text_string()” on page 12-37.

empty()

See “de_empty()” on page 12-38.

end()

See “de_end()” on page 12-38.

end_command()

See “de_end_command()” on page 12-39.

fill()

See “de_fill()” on page 12-40.

fix_instance()

See “de_fix_instances()” on page 12-42.

flatten()

See “de_flatten()” on page 12-42.

get_data_parm()

See “de_get_data_parm()” on page 12-44.

get_design_instances()

See “de_get_design_instances()” on page 11-2.

get_design_list()

Deleted

get_file_names()

See “de_get_file_names()” on page 11-3.

get_layer_attribute()

See “de_get_layer_attribute()” on page 13-3.

get_preference()

See “de_get_preference()” on page 13-4.

get_string_list()

Deleted

get_variable_names()

See “de_get_variable_names()” on page 11-3.

get_window()

See “de_get_window()” on page 11-4.

highlight_instance()

See “de_highlight_instance()” on page 12-45.

import_design()

See “de_import_design()” on page 12-46.

info()

See “de_info()” on page 18-7.

init_tune()

See “de_tune()” on page 17-4.

instantiate()

See “de_instantiate()” on page 12-48.

lock_project()

Deleted

mask_read()

Deleted

mask_write()

Deleted

merge_and()

See “de_merge_and()” on page 12-50.

merge_diff()

See “de_merge_diff()” on page 12-50.

merge_or()

See “de_merge_or()” on page 12-50.

mirror_x()

See “de_mirror_x()” on page 12-50.

mirror_y()

See “de_mirror_y()” on page 12-51.

miter_vertex()

See “de_miter_vertex()” on page 12-51.

mks_factor()

Deleted

modify_break()

See “de_modify_break()” on page 12-52.

modify_explode()

See “de_modify_explode()” on page 12-53.

modify_join()

See “de_modify_join()” on page 12-53.

move()

See “de_move()” on page 12-54.

move_annotation()

See “de_move_annotation()” on page 12-54.

move_break()

See “de_move_break()” on page 12-54.

move_to_layer()

See “de_move_to_layer()” on page 12-55.

move_vertex()

See “de_move()” on page 12-54.

netlist()

See “de_netlist()” on page 12-56.

new_design()

See “de_new_design()” on page 12-56.

open_design()

See “de_open_design()” on page 12-57.

open_grid()

See “de_open_datadisplay()” on page 17-3.

open_window()

See “de_open_window()” on page 12-58.

optimization_continue()

Deleted

oversize()

See “de_oversize()” on page 12-58.

palette_group()

See “de_define_palette_group()” on page 15-14.

pan_window()

See “de_pan_window()” on page 12-59.

paste_from_buffer()

See “de_paste_from_buffer()” on page 12-64.

path()

See “de_add_path()” on page 12-6.

performance_optimization

Deleted

place_instance()

See “de_place_item()” on page 12-64.

place_port()

See “de_place_port()” on page 12-65.

place_unplaced()

See “de_place_unplaced()” on page 12-65.

playback_macro()

See “de_playback_macro()” on page 12-66.

plot()

See “de_plot()” on page 12-66.

polygon()

See “de_add_polygon()” on page 12-7.

polyline()

See “de_add_polyline()” on page 12-7.

pop_outof_instance()

See “de_pop_outof_instance()” on page 12-67.

pop_window()

Deleted

prompt()

See “de_prompt()” on page 18-9.

push_into_instance()

See “de_push_into_instance()” on page 12-67.

push_window()

Deleted

question()

See “de_question()” on page 18-10.

read_layer()

See “de_read_layer()” on page 13-9.

read_preference()

See “de_read_preferences()” on page 13-10.

rectangle()

See “de_add_rectangle()” on page 12-8.

redisplay()

See “de_refresh_view()” on page 12-67.

refresh_view()

See “de_refresh_view()” on page 12-67.

remove_all_layers()

See “de_remove_all_layers()” on page 13-10.

restore_all_grids()

See “de_restore_all_datadisplay()” on page 17-4.

restore_grids()

See “de_open_datadisplay()” on page 17-3.

restore_status()

See “de_restore_status()” on page 17-4.

rotate()

See “de_rotate()” on page 12-69.

rotate_90()

See “de_rotate_90()” on page 12-69.

rotate_image()

See “de_rotate_image()” on page 12-70.

save_all()

See “de_save_all_designs()” on page 12-70.

save_design()

See “de_save_design()” on page 12-70.

save_grids()

See de_save_grid().

scale()

See “de_scale()” on page 12-71.

screen_dump()

Deleted

search_and_replace()

See “de_search_and_replace()” on page 12-72.

select_all()

See “de_select_all()” on page 12-72.

select_all_force()

See “de_select_all_force()” on page 12-72.

select_by_name()

See “de_select_by_name()” on page 12-73.

select_item()

See “de_select_item()” on page 12-73.

select_point()

See “de_select_range()” on page 12-74.

select_point_range()

Deleted

select_range()

See “de_select_range()” on page 12-74.

select_unplaced()

See “de_select_unplaced()” on page 12-74.

select_window()

See “de_select_window()” on page 12-75.

send_netlist()

Deleted

set_add_optional_parameters()

See de_set_add_optional_parameters().

set_annotation_font()

See “de_set_annotation_font()” on page 13-10.

set_annotation_height()

See “de_set_annotation_height()” on page 13-11.

set_annotation_id_layer()

See “de_set_annotation_id_layer()” on page 13-11.

set_annotation_name_layer()

See “de_set_annotation_name_layer()” on page 13-11.

set_annotation_parameters_layer()

See “de_set_annotation_parameters_layer()” on page 13-12.

set_annotation_precision()

See “de_set_annotation_precision()” on page 13-12.

set_annotation_rows()

See “de_set_annotation_rows()” on page 13-12.

set_arc_radius()

See “de_set_arc_radius()” on page 13-13.

set_auto_update_opt()

Deleted

set_background_color()

See “de_set_background_color()” on page 13-13.

set_backup_count()

See "de_set_backup_count()" on page 13-13.

set_check_self_intersection()

See "de_set_self_intersect()" on page 13-35.

set_coordinate_readout_mode()

See "de_set_coordinate_readout_mode()" on page 13-14.

set_curve_radius()

See "de_set_curve_radius()" on page 13-15.

set_dse_start()

See "de_set_dse_start()" on page 13-16.

set_dual_placement()

See "de_set_dual_placement()" on page 13-17.

set_edit_text()

See "de_set_edit_text()" on page 12-76.

set_entry_mode()

See "de_set_shape_entry_mode()" on page 13-35.

set_foreground_color()

See "de_set_foreground_color()" on page 13-17.

set_grid_color()

See "de_set_grid_color()" on page 13-18.

set_grid_display_type()

See "de_set_grid_display_type()" on page 13-18.

set_grid_snap()

See "de_set_grid_snap()" on page 13-18.

set_grid_snap_type()

See "de_set_grid_snap_type()" on page 13-19.

set_highlight_color()

See "de_set_highlight_color()" on page 13-20.

set_instance()

See "de_init_item()" on page 12-47.

set_instance_id()

See "de_set_item_id()" on page 12-77.

set_instance_parameters()

See "de_set_item_parameters()" on page 12-78.

set_layer()

See "de_set_layer()" on page 13-20.

set_major_grid_display()

See "de_set_major_grid_display()" on page 13-21.

set_minor_grid_display()

See “de_set_minor_grid_display()” on page 13-21.

set_miter_cutoff()

See “de_set_miter_cutoff()” on page 13-22.

set_origin()

See “de_set_origin()” on page 12-79.

set_oversize()

See “de_set_oversize()” on page 13-22.

set_path_corner()

See “de_set_path_corner()” on page 13-23.

set_path_width()

See “de_set_path_width()” on page 13-23.

set_pin_color()

See “de_set_pin_color()” on page 13-24.

set_pin_size()

See “de_set_pin_size()” on page 13-24.

set_place_popup_mode()

See “de_set_place_popup_mode()” on page 13-26.

set_plotting_depth()

See “de_set_plotting_depth()” on page 13-28.

set_plot_pin_mode()

See “de_set_plot_pins()” on page 13-27.

set_plot_pin_names()

See “de_set_plot_pin_names()” on page 13-26.

set_plot_pin_num_mode()

See “de_set_plot_pin_numbers()” on page 13-27.

set_port()

See “de_set_port()” on page 12-79.

set_reroute_wires()

See “de_set_reroute_wires()” on page 13-29.

set_rotation_increment()

See “de_set_rotation_increment()” on page 13-30.

set_scale()

See “de_set_scale()” on page 13-32.

set_select_box_size()

See “de_set_select_box_size()” on page 13-32.

set_select_color()

See “de_set_select_color()” on page 13-33.

set_select_filter()

See “de_set_select_filter()” on page 13-33.

set_select_inside_polygon()

See “de_set_select_inside_polygon()” on page 13-34.

set_swap_template_instances()

See “de_set_swap_template_instance()” on page 12-80.

set_tap_length()

See “de_set_tap_length()” on page 13-36.

set_tee_color()

See “de_set_tee_color()” on page 13-37.

set_tee_size()

See “de_set_tee_size()” on page 13-37.

set_text_absolute()

See “de_set_text_absolute()” on page 13-38.

set_text_angle()

See “de_set_text_angle()” on page 13-38.

set_text_font()

See “de_set_text_font()” on page 13-38.

set_text_height()

See “de_set_text_height()” on page 13-39.

set_text_justification()

See “de_set_text_justification()” on page 13-39.

set_text_string()

See “de_set_text_string()” on page 13-40.

set_trace_mcover_id()

Deleted

set_trace_msub_id()

Deleted

set_trace_mwall_id()

Deleted

set_trace_sim_mode()

See “de_set_trace_sim_mode()” on page 13-40.

set_trace_single_elem()

See “de_set_trace_single_elem()” on page 13-41.

set_trace_tand_id()

Deleted

set_trace_tech()

See “de_set_trace_tech()” on page 13-41.

set_trace_temp_id()

See de_set_trace_temp_id().

set_trace_traverse()

See “de_set_trace_traverse()” on page 13-42.

set_window()

See “api_set_current_window()” on page 18-4.

show_connected()

See “de_show_connected()” on page 12-82.

show_equiv_inst()

See “de_show_equiv_inst()” on page 12-83.

show_fixed()

See “de_show_fixed()” on page 12-83.

show_unmatched()

See “de_show_unmatched()” on page 12-83.

show_unplaced()

See “de_show_unplaced()” on page 12-84.

sim_file_command()

Deleted

snap()

See “de_snap()” on page 12-84.

split_tlin()

See “de_split_tlin()” on page 12-84.

statistical_analysis

Deleted

stretch()

See “de_stretch()” on page 12-85.

stretch_tlin()

See “de_stretch_tlin()” on page 12-86.

string_list()

Deleted

swap_instances()

See “de_swap_instances()” on page 12-87.

switch_view()

See “de_switch_view()” on page 12-87.

tap_tlin()

See “de_tap_tlin()” on page 12-88.

text()

See “de_add_text()” on page 12-9.

trace()

See “de_add_trace()” on page 12-9.

translate_design()

Deleted

tune_item()

See “de_tune()” on page 17-4.

undo()

See “de_undo()” on page 12-89.

undo_vertex()

See “de_undo_vertex()” on page 12-89.

unhighlight_instances()

See “de_unhighlight_instances()” on page 12-89.

unix_system()

Deleted

unmap_grids

See “de_close_all_datadisplay()” on page 17-3.

update_optimization_values()

See “de_update_optimization_values()” on page 17-6.

vertex_to_arc()

See “de_vertex_to_arc()” on page 12-91.

view_all()

See “de_view_all()” on page 12-91.

window_is_open()

See “de_window_is_open()” on page 11-5.

write_preference()

See “de_write_preferences()” on page 13-43.

yield_optimization()

Deleted

zoom_in()

See “de_zoom_in_scale()” on page 12-92.

zoom_out()

See “de_zoom_out_scale()” on page 12-93.

zoom_window()

See “de_zoom_window()” on page 12-93.

Chapter 6: File Handling Functions

This chapter describes each File Handling function in detail. The functions are listed in alphabetical order.

ael_gfile_hasext()

Takes a string parameter and returns an integter that indicates where an extension exists in that string. An extension is found if there is a "." in the string. Returns: -1 if no extension is found. Otherwise it returns the position at which the extension occurs. If more than one "." is found, it assumes that the last "." found indicates the extension follows. Note that the position is 1-based.

Syntax:

ael_gfile_hasext(filename);

where

filename is a string; the name of the file.

Example:

```
fputs(stderr, ael_gfile_hasext("design.ael");   //7 is the output
```

chdir()

Changes the current directory to the specified directory. Changes the program's reference directory, making the "." directory be the new directory. Any configuration files are re-read. Returns: TRUE if a directory is created successfully; else FALSE.

Syntax:

chdir(directoryName);

where

directoryName is a string; name of the directory to change to.

Example:

```
decl a;
a = chdir("myDir");
```

fclose()

Closes a file opened with *fopen()* command. Returns: none.

Syntax:

fclose(fileID);

where

fileID is the file identifier returned by *fopen()*.

Example:

In this example “hello there” is written in a file as “hello”.

```
decl fID;
fID = fopen("hello", "W");
  fputs(fID, "hello there");
fclose(fID);
```

fflush()

Flushes buffers to disk when writing a file opened with *fopen()* for write or append. Returns a value indicating whether buffers were successfully flushed, where:
0 = successful, and 1 = not successful

Syntax:

fflush(file);

where

fileID is the file value, created with *fopen()*.

Example:

```
decl fID;
fID = fopen("hello", "W");
  fflush(fID, "hello there");
fclose(fID);
```

fgets()

Reads a line from a file. Returns a string; the current line is returned and the line advanced. At end-of-file, returns NULL.

Syntax:

fgets(fileID);

where

fileID is the file value, created with *fopen()*.

Example:

```
decl fID, fLine;
fID = fopen("hello", "R");
fLine = fgets(fID);
fLine = "hello there"
fclose(fID);
```

fopen()

Opens a file and returns a file identifier. At some point the file must be closed using *fclose()*.

Syntax:

fopen(fileName, mode);

where

fileName is a string; name of file to open.

mode is a string; type of action to perform, where:

"W" = write
"R" = read
"A" = append

Example:

```
decl fid;
fid = fopen("atest.txt", "W");
  fputs(fid, "hello");
fclose(fid);
```

fprintf()

Print formatted text to a file. This function works like the C language counterpart. An argument is required for each format code in the format string. Unlike the function *fputs()*, newline is not automatically appended to the printed text. Returns: none.

See also: *sprintf()*, *fputs()*

Syntax:

fprintf(file, fmt, args...);

where

file is the open file handle for the output file.

fmt is the output format string.

args are the values to be included in the formatted output, as required by the format string. The format codes recognized are:

%c is the single ASCII character
%d is the signed decimal integer
%e is the floating point value in scientific format
%E is the same as %e but with E in exponent
%f is the floating point value in fixed decimal format
%g is the floating point value in flexible format, most compact of %e or %f
%G is the same as %g but with G printed in exponent instead of e
%i is the integer value
%o is the octal integer
%s is the string value
%u is the unsigned integer value
%x is the hexadecimal integer, using lower case abcdef
%X is the hexadecimal integer, using upper case ABCDEF

Example:

Prints to the standard error stream: *The value of x is 5.5*.

```
x=5.5;
fprintf (stderr, “The value of %s is %f\n”, “x”, x);
```

fputs()

Writes a value to the file or device specified. Then the file must be closed using *fclose()*. Returns: none.

Syntax:

fputs(fileID, value);

where

fileID is the file identifier returned from *fopen()*.

value is a string or a variable name. If a string is desired, it should be placed within quotes; if a variable name is desired, no quotes should be used.

Example:

```
decl fid;
```

```
fid = fopen("a test", "W");
  fputs(fid, "hello");
fclose(fid);
```

freopen()

Opens the named file for reading or writing, attaching an existing file handle to it. Returns a file value; NULL if the file cannot be opened (that is, file does not exist). Binary mode is not currently supported for files.

See also: *fopen()*.

Syntax:

freopen(name, mode, handle);

where

name is a string; name of file to open.

mode is a string; type of action to perform, where:

"W" = write
"R" = read
"A" = append

handle is a string; handle to be reopened.

Example:

```
file = freopen ("myfile.dat", "R", fid);
fclose(fid);
```

remove()

Deletes a given file. Returns: none.

Syntax:

remove(filename);

where

filename is the name of file to be deleted.

Example:

```
remove("myfile.txt");
```

Note Designs should be removed with *de_delete_design()*.

rename()

Renames a file. Returns FALSE if successful, TRUE if error.

Syntax:

rename(name, new);

where

name is current name of file.

new is new name of file.

Example:

```
rename("myfile.dat", "newfile.dat");
```

sprintf()

Format text values into a string. This function works like *fprintf()* except that the output is formatted into a string rather than a file. The same format specifications are valid for the format string. Returns a formatted string.

See also: *fprintf()*.

Syntax:

sprintf(fmt, args...);

where

fmt is output format string.

args is the values to be included in the formatted output, as required by the format string.

Example:

Creates and stores the following string in the variable *s*: *The value of x is 5.5*.

```
x=5.5;
s=sprintf (“The value of %s is %f”, “x”, x);
```

Chapter 7: String Functions

This chapter describes each String function in detail. The functions are listed in alphabetical order.

fmt()

Converts a float to a string. Returns a string in which the given value is converted to an ASCII string according to the width and precision specified.

Syntax:

fmt(realNum [, width, precision]);

where

realNum is the real number to convert.

width is optional; default = 10. The string width (number of characters in converted string). Must be equal to or larger than the total size of the resulting string plus a string termination character.

precision is optional; default = 6. The number of characters to the right of the decimal point.

Example:

The example returns the string “10.134599”.

```
decl str;
str = fmt(10.134599);
```

fmt_tokens()

Creates a string from a list of tokens. Concatenates the list items into a single string. This is a convenient method for printing out the contents of a list. Returns: A string, the concatenation of the list items.

Syntax:

fmt_tokens(list);

where

list is a list of items to format into a string.

Example:

```
warning(errclass, 20, fmt_tokens(list(elem, "ANG not valid", ang)));
```

index()

Returns the position of the first occurrence of a string or character in a string; such as, returns the position of the first occurrence of str2 in str1 (where, n is an integer >= 0, where 0 is the first character and –1 = is not found).

Syntax:

index(str1, str2);

where

str1 is a string.

str2 is a string to search for.

Example:

```
decl pos;
pos = index("hello", "e");   //returns 1
```

leftstr()

Returns the left portion of a string, truncating or padding on the right with blanks if necessary, until the length specified is obtained.

Syntax:

leftstr(string, length);

where

str is the string.

length is an integer, length to truncate or pad.

Example:

The example creates the string “hel”.

```
decl nstr;
nstr = leftstr("hello", 3);
```

midstr()

Returns a piece of a string (the dissected string), starting and ending at the given indices. An end index of –1 indicates the end of the string. If the start index is also negative, it indicates a count from the end of the string.

Syntax:

midstr(str, startIndex, endIndex);

where

str is the string to operate on.

startIndex is the starting position.

endIndex is the ending position.

Example:

The example returns the string “ll”.

```
decl mstr;
mstr = midstr("hello", 2,3);
```

parse()

Parses a string into tokens, where each token is delimited by a blank, tab, or operator. The following interpretations are made:

- Alphabetic characters include underscore (_) and dollar sign ($) as well as uppercase A through Z, lowercase a through z, and numbers 0 through 9.
- Numbers are interpreted as expected without separation into tokens because of sign or exponent notation.
- The characters period and E (or e) are interpreted as expected in the context of a number.
- A number followed by a string is interpreted as two tokens: the first numeric and the second a string.
- A string followed by a number (without an intervening operator or delimiting punctuation) is interpreted as a single string token.
- Operators are interpreted as string tokens and adjacent operators are merged into a single token. The other normal delimiters are blank and tab, which are treated as white space and are ignored except to separate tokens.

Special handling is provided for strings enclosed in double quotes, where the entire string (including the quotes) is interpreted as a single token. However, there is no provision for handling strings with embedded quotes.

If the optional delimiter and operator strings are used, stricter parsing rules are enabled. Only the listed characters will cause separation of the text into tokens, and these will always cause separation into tokens, regardless of the context. Delimiters will be discarded but will cause separation of tokens, while operators will be returned as single character strings, causing the special handling for real numbers, operators and strings to be unavailable when the delimiter or operator strings are specified. Returns: A list constructed from the tokens. A token may be returned as an integer, real, or string member in the list.

See also: *list()*.

Syntax:

parse(text[, delim][, ops][, str]));

where

text is the string of text to be parsed.

delim is optional. String of delimiter characters. Recognizes normal arithmetic operators unless alternate characters are given in delimiter or special character strings. Default delimiters are space and tab.

ops is optional. String of special operator characters, each to be interpreted as an individual token. Default special operators are:

+ - * / = , ~ & ^ < > ! # % ' () : ; ? @ [] \ ` {}

str is optional. Specifies to only return strings, where:

FALSE = (default) each element is appropriately converted to int, real, or strings

TRUE = returns a list of strings, no conversion takes place

Example:

The example returns a list containing “hello”, 1, 7.8, “*”, and 32.6.

```
decl mylist;
mylist = parse("hello 1 7.8 * 32.6");
```

ALSO

```
parse("123.456", ".", NULL)    // returns list(123,456)
parse("123.456", ".", NULL, TRUE)    // returns list("123","456")
```

parse_blank()

Parses a string of tokens separated by spaces or tabs into a list of strings. Returns a list of tokens.

Syntax:

parse_blank(str);

where

str a character string to be parsed.

Example:

```
decl a;
a = parse_blank("1.5 2.3 4.1");
```

This example creates the same list:

```
a = list("1.5", "2.3", "4.1");
```

rightstr()

Returns the right portion of a string value, truncating or padding on the left with blanks if necessary until the length specified is obtained.

Syntax:

rightstr(str, length);

where

str is the string to truncate or pad.

length is the new string length.

Example:

```
decl mystr;
mystr = rightstr("hello", 2);   //creates the string "lo"
```

strcasecmp()

Compares two strings, ignoring differences in case between them. If the first and second strings are equivalent (except for case differences), this function returns the integer zero. If the first string is higher than the second in the ASCII collating

sequence, this function will return an integer greater than zero. If the first string is lower than the second in the collating sequence, this function returns an integer less than zero.

See also: *strcmp()*.

Syntax:

strcasecmp(str1, str2);

where

str1 is the first string to compare.

str2 is the second string to compare.

Example:

```
fputs(stderr, strcasecmp(“A”, “a”));
// will print 0
```

strcat()

Appends two or more strings to the end of the first, resulting in a single string. Returns a new string which is composed of the second string (and possibly additional strings) appended to the end of the first string. If any argument is not a string, the function fails and reports an error.

See also *leftstr()*, *midstr()*, *rightstr()*.

Syntax:

strcat(text1, text2 ...);

where

text1 is the first string to concatenate.

text2 is the second string, optionally followed by additional strings separated by commas.

Example:

```
decl str;
str = strcat("hello ", "there"); // returns "hello there"
```

strcmp()

Compares two strings. If the first and second strings are the same, this function returns the integer zero. If the first string is higher than the second in the ASCII

collating sequence, this function returns an integer greater than zero. If the first string is lower than the second in the collating sequence, this function returns an integer less than zero. Returns an integer indicating the results.

See also: *strcasecmp()*.

Syntax:

strcmp(str1, str2);

where

str1 is the first string to compare.

str2 is the second string to compare.

Example:

```
fputs(stderr, strcmp(“a”, “b”));
// will print -1
```

stripstr()

Returns a string with leading and trailing blanks and tabs removed.

Syntax:

stripstr(str);

where

str is the string to strip.

Example:

```
decl sstr;
sstr = stripstr("    hi there"); // returns "hi there"
```

strlen()

Returns the length of a string, as an integer.

Syntax:

strlen(str);

where

str is a string.

Example:

```
decl len;
len = strlen("hi"); // returns 2
```

tolower()

Converts a string to lowercase and return the converted string. If an integer having the value of an ASCII character is passed, then the integer is converted to represent the lowercase value for the ASCII character and returned. Returns a string (if a string is passed) or an integer (if an integer is passed).

See also: *toupper(), strcasecmp()*.

Syntax:

tolower(str);

where

str is a string to convert.

Example:

```
fputs(stderr, tolower(“aBcD”));
// will print “abcd”
```

toupper()

Converts a string to uppercase and return the converted string. If an integer having the value of an ASCII character is passed, then the integer is converted to represent the uppercase value for the ASCII character and returned. Returns a string (if a string is passed) or an integer (if an integer is passed).

See also *tolower(), strcasecmp()*.

Syntax:

toupper(str);

where

str is a string to convert.

Example:

```
fputs(stderr, toupper(“aBcD”));
// will print “ABCD”
```

val()

Returns a real number, the numeric value of an ASCII string.

Syntax:

val(string);

where

string is an ASCII string, representing a numeric value.

Example:

```
decl v;
v = 10 + val("2.5");
v = 12.5
```

Chapter 8: List Management Functions

This chapter describes each List Management function in detail. The functions are listed in alphabetical order.

append()

Appends a new list to the end of an existing list. Lists are created with the *list()* function. Returns a new list with the appended items.

Syntax:

append(list, newList);

where

list is a list created with the *list()* function.

newList is a new list of items to append to the first list.

Example:

```
decl mylist;
mylist = list("one", "two");
mylist = append(mylist, list("three"));
//The result is mylist = ("one", "two", "three")
```

car()

Returns the first item from a given list. Does not change the list, is passed as a parameter.

Syntax:

car(list);

where

list is a list created with the *list()* function.

Example:

The example returns the string item *a* (see comment).

```
decl c;
c = car(list("a", "b", "c"));   //the result is c="a"
```

cdr()

Returns the remainder of the list, after the first item is removed.

Syntax:

cdr(list);

where

list is a list created with the *list()* function.

Example:

The example returns the list containing one item: *there*.

```
decl shortList;
shortList = cdr(list("hello", "there"));
```

cons()

Constructs a list cell from the member value to be returned by *car()* and the next element value to be returned by *cdr()*. The first argument can be any value, and the second argument is normally a list, or NULL. If the second argument is a list, then the value returned is a list with new member inserted at the beginning. If the second argument is NULL, then the value returned is a list with one member. If the second argument is neither a list nor NULL, then an improper list is returned. Returns a list cell; that is, returns a list with one element.

Syntax:

cons(carValue, cdrValue);

where

carValue is a member value.

cdrValue is the next element value.

Example:

```
x = cons( 5, NULL );   //the result is x=(5)
```

Also:

```
x = cons( 5, list(6,7) );   //the result is x=(5,6,7)
```

delete_nth()

Deletes an item in a list, given an integer index into the list and an existing list. Returns a new list with a deleted item. If index >= list length, then NULL is returned.

Syntax:

delete_nth(index, list);

where

index is an integer index into list. If index = list length, then NULL is returned.

list is the list to delete an item from.

Example:

```
decl a = list(1,2,3,4,5);
decl b;

b = delete_nth(0,a);
fputs(stdout, identify_value(b));    // deletes the 1

b = delete_nth(1,a);
fputs(stdout, identify_value(b));    // deletes the 2

b = delete_nth(4,a);
fputs(stdout, identify_value(b));    // deletes the 5

b = delete_nth(5,a);
fputs(stdout, identify_value(b));    // returns NULL
```

insert()

Returns a new list with an inserted item, given an existing list and item to insert into the beginning of the list.

Syntax:

insert(list, listItem);

where

list is the list to insert the item into.

listItem is the item to insert.

Example:

```
decl list1;
list1 = list(1, 2, 3);
// The result is list1 = 1, 2, 3
list1 = insert(list1, 0);
// The result is list1 = 0, 1, 2, 3
=======================================
To produce an empty list:
insert_nth(0, list(), 10) produces list(10) )
```

insert_nth()

Returns a new list with an inserted item, given an index into the list, an existing list and an item to insert into the list.

Syntax:

insert_nth(index, list, listItem);

where

index is an integer index into list. If index = list length, then NULL is returned.

list is the list to insert the item into.

listItem is the item to insert.

Example:

```
decl x = list(1,2,3,4,5);
decl y;

y = insert_nth(3,x,10);
fputs(stdout, identify_value(y)); // output: list(1,2,3,10,4,5)
fputs(stdout, identify_value(x)); // output: list(1,2,3,4,5)

y = insert_nth(0,x,10);
fputs(stdout, identify_value(y)); // output: list(10,1,2,3,4,5)

y = insert_nth(5,x,10);
fputs(stdout, identify_value(y)); // output: list(1,2,3,4,5,10)

y = insert_nth(7,x,10);
fputs(stdout, identify_value(y)); // output: NULL

y = insert_nth(-1,x,10);
fputs(stdout, identify_value(y)); // output: NULL
```

list()

Creates a list of items. The list can be composed of any type of item. Returns a list of given items.

Syntax:

list(item1, item2,..., itemN);

where

itemN is an item.

Example:

```
decl list1;
list1 = list(1, 34.5, 5, 7, "hello", 3+4i);
```

listlen()

Returns an integer representing the length of a list; that is, the number of items in the list.

Syntax:

listlen(list);

where

list is a list of items created with the list function.

Example:

The example returns the integer 3:

```
decl len;
len = listlen(list(1,2,3));
```

member()

Tests whether an item is a member of a list. Returns a list consisting of the original list from the member on or NULL.

Syntax:

member(m, list);

where

m is the item you are testing for.

list is a list of items.

Example:

In this example, member(“def”, L) returns the list (“def”, “ghi”).

```
decl L;
L=list("abc", "def", "ghi");
if(member("def", L))
  fputs(stderr, "of is");
```

nth()

Takes a list and an integer index into the list. Returns the corresponding *nth* list item.

Syntax:

nth(n, list);

where

n is an integer greater than or equal to 0; represents position of item in list to retrieve.

list is the list of items.

Example:

```
decl sixth;
sixth=nth(5, list(0, 1, 2, 3, 4, 5, 6, 7, 8));
// returns the number 5 (5th position)
```

nthcdr()

Takes a list, and an integer index into the list. Returns a list consisting of the members of the original list from the *nth* item on.

Syntax:

nthcdr(n, list);

where

n is the ndex (integer ≥ 0) into list.

list is a list of items.

Example:

```
decl j, L;
```

```
L = list("a", "b", "c");
j=nthcdr(1, L);
// j ==list("b", "c")
```

remov()

Returns a new list with all the items copied from the original list except for the item specified in the first parameter.

Syntax:

remov(item, alist);

where

item is a member of a list to be deleted from the new list.

alist is the original list that will be copied, except for item.

Example:

```
decl a,b;
a = list(1,5.8,4,10);
b = remov(4, a);    // returns the list (1,5.8,10)
```

repla()

Replaces an item in a list. Returns the original list, modified with the replaced item.

Syntax:

repla(list, item, index);

where

list is a list created with the list function.

item is the new list item.

index is the position in the list of the item to replace, numbered from 0 to n–1.

Example:

```
decl list1;
list1 = list(1, 2, 3);
repla(list1, 5, 2);
// The result is list1 = 1, 2, 5
```

sort_list()

Returns a new list with members of a given list in sorted order. If the compare function is not given and the list is homogeneous, a built-in function will be applied if:

- List has only integer values, or
- List has only real values, or
- List has only string values, or
- List has only list values where each list is comparable on a value by value basis (has the same structure of integer, real, string, and list members).

For other cases, a *compare()* function must be supplied. The *compare()* function is called with two member values as the argument and must return a value greater than, equal to, or less than zero, depending on whether the first member is greater than, equal to, or less than the second member. If the first argument is not a list, the function fails and reports an error.

Syntax:

sort_list(list, [, func]);

where

list is a list to be sorted.

func is optional. Address of function to perform member comparison.

Example:

This example redefines l to the list 1, 2, 3, 4, 9.

```
decl l = list(2, 4, 1, 3, 9);
defun my_compare(first, second)
{decl neg;
if (first < second)
  neg = -1;
else if (first > second)
  neg = 1;
else
  neg = 0;
return (neg);
}
l = sort_list(l, my_compare);
```

Chapter 9: Math Functions

This chapter describes each Math function in detail. The functions are listed in alphabetical order.

Note For math functions used in simulations, refer to the *Expressions, Measurements, and Simulation Data Processing* manual.

---

abs()

Returns the absolute value of a real number or an integer. In the case of a complex number, the abs function:

- accepts one complex argument.
- returns a positive real number.
- returns the magnitude of its complex argument.

Syntax:

abs(num);

where

num is any number.

Example:

```
decl a;
a = abs(-12.3);    // a=12.3
```

acos()

Returns the inverse cosine, in radians, of a real number or complex number.

Syntax:

acos(realNum);

where

realNum is a real or complex number.

Example:

```
decl a;
```

```
a = acos(0.5);    // a=1.0471975511966
```

acosh()

Returns the inverse hyperbolic cosine of an integer, real, or complex number.

Syntax:

acosh(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = acosh(1.5);
```

acot()

Returns the inverse cotangent of an integer, real, or complex number.

Syntax:

acot(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = acot(1.5);
```

acoth()

Returns the inverse hyperbolic cotangent of an integer, real, or complex number.

Syntax:

acoth(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = acoth(1.5);
```

asin()

Returns the inverse sine, in radians, of a real number, an integer, or complex number.

Syntax:

asin(realNum);

where

realNum is a real number, an integer, or complex number.

Example:

```
decl a;
a = asin(0.5);
```

asinh()

Returns the inverse hyperbolic sine of an integer, real, or complex number.

Syntax:

asinh(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = asinh(.5);
```

atan()

Returns the inverse tangent, in radians, of a real number, π/2, an integer, or complex number.

Syntax:

atan(realNum);

where

realNum is a real number, an integer, or complex number.

Example:

```
decl b;
b = atan(0.5);
```

atan2()

Returns the arc tangent of the rectangular coordinates y and x.

Syntax:

atan2(y,x);

where

realNum is a real number, an integer, or complex number.

Example:

```
decl a;
a = atan2(10,15);
```

atanh()

Returns the inverse hyperbolic tangent of an integer, real, or complex number.

Syntax:

atanh(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = atanh(.5);
```

ceil()

Given a real number, returns the smallest integer not less than its argument; that is, its argument rounded to the next highest number.

Syntax:

ceil(realVal);

where

realVal is a real number.

Example:

```
a = ceil(5.27);    // a=6
```

chr()

Returns the character representation of an integer.

Syntax:

chr(code);

where

code is a valid ASCII string representing a character.

Example:

The example returns the character @.

```
decl aChar;
aChar = chr(64);
```

cint()

Given a noninteger real number, returns a rounded integer value.

Syntax:

cint(realVal);

where

realVal is a real number.

Example:

The example returns the integer *46*.

```
decl d;
d = cint(5.27);    // d=5
```

Also:

```
decl d;
d = cint(5.6);    // d=6
```

cmplx()

Given two real numbers representing the real and imaginary components of a complex number, returns a complex number.

Note Use the *real* and *imag* functions to retrieve the real and imaginary components, respectively. The basic math functions operate on complex numbers.

Syntax:

cmplx(realPart, imagPart);

where

realPart is the real component of the number.

imagPart is the imaginary component.

Example:

```
decl cmpx;
cmpx = cmplx(10.0, 3);    // returns the complex number (10.0+3i)
```

conj()

Returns the conjugate of a complex number.

Syntax:

conj(num);

where

num is a complex number.

Example:

```
decl a;
a = conj(3 + 4i);    // returns the complex number a=3-4i
```

convBin()

Returns a binary string of an integer with n-digits.

Syntax:

convBin(val, num);

where

val is any integer to be converted to binary string.

num is an integer to specify the number of digits in the binary string.

Example:

```
decl a;
a = convBin(1064, 8);    // returns a="00101000"
```

convHex()

Returns a hexidecimal string of an integer with n-digits.

Syntax:

convHex(val, num);

where

val is any integer to be converted to hexidecimal string.

num is an integer to specify the number of digits in the hexidecimal string.

Example:

```
decl a;
a = convHex(1064, 8);    // returns a="00000428"
```

convOct()

Returns an octal string of an integer with n-digits

Syntax:

convOct(val, num);

where

val is any integer to be converted to octal string.

num is an integer to specify the number of digits in the octal string.

Example:

```
decl a;
a = convOct(1064, 8);    // returns returns a="00101000"
```

cos()

Returns the cosine of a real number (in radians).

Syntax:

cos(realNum);

where

realNum is a real number, in radians.

Example:

```
decl c;
c = cos(PI/4);    // where PI is the AEL constant 3.1415926535898
```

cosh()

Returns the hyperbolic cosine of an integer, real, or complex number.

Syntax:

cosh(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = cosh(1.5);
```

cot()

Returns the cotangent of an integer, real, or complex number.

Syntax:

cot(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = cot(1.5);
```

coth()

Returns the hyperbolic cotangent of an integer, real, or complex number.

Syntax:

coth(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = coth(1.5);
```

dB()

Converts a voltage ratio to decibels. Returns a real number that represents a voltage ratio in decibels.

Syntax:

dB(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = dB(1.5);     // a=3.5218251811136
```

dBm()

Converts a voltage to decibels referenced to miliwatt. Returns a real number that represents a voltage ratio in miliwatt.

Syntax:

dBm(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = dBm(1.5);     // a=13.521825181114
```

deg()

Converts an angle from radians to degrees. Returns a real number or NULL if an error occurs.

Syntax:

deg(num);

where

num is any integer or real number that represents an angle in radians.

Example:

```
decl a;
a = deg(1.5);    // a=85.943669269623
```

exp()

Given an integer or real number as an exponent, returns e (~2.7183) raised to that exponent.

Syntax:

exp(realVal);

where

realVal is the exponent of *e*, where *e* is the AEL constant 2.718281828459.

Example:

```
decl b;
b = exp(10);
```

fix()

Takes a real number argument, truncates it, and returns an integer value.

Syntax:

fix(realVal);

where

realVal is a real number.

Example:

```
decl b;
b = fix(5.9);    // returns 5
```

float()

Converts an integer to a floating decimal number. Returns a real (floating-point) number.

Note To convert a real number to an integer, use *int*.

Syntax:

float(intNum);

where

intNum is the integer to convert.

Example:

```
decl r;
r = float(10);    // r=10.0 or r is the floating point number 10
```

floor()

Returns the largest integer not more than its argument from a real number, that is, returns its argument rounded to the next highest number.

Syntax:

floor(realVal);

where

realVal is a real number to be rounded to an integer.

Example:

```
decl d;
d = floor(5.6)    // d=5
```

im()

Returns a real value, the imaginary component of a complex number.

Syntax:

im(complexNum);

where

complexNum is a complex number.

Example:

```
decl rv;
```

```
rv = im(1—1);    //rv=-1.0 or the real number -1
```

imag()

Returns a real value, the imaginary component of a complex number.

Syntax:

imag(complexNum);

where

complexNum is a complex number.

Example:

```
decl rv;
rv = imag(1—1);
```

int()

Returns the largest integer not greater than a given floating decimal number.

Syntax:

int(realVal);

where

realVal is a real value.

Example:

```
decl i;
i = int(4.3); // returns integer 4
```

ln()

Formerly log() in Series IV. Returns the natural logarithm of an integer, real, or complex number.

Syntax:

ln(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = ln(1.5);
```

log()

Returns the base 10 logarithm of an integer or real number.

Note log10(x) performs the same operation.

Syntax:

log(rvalue);

where

rvalue is an integer or real number.

Example:

```
decl lval;
lval = log(10.4);
```

log10()

Returns the base 10 logarithm of an integer or real number.

Note log(x) performs the same operation.

Syntax:

log10(rvalue);

where

rvalue is an integer or real number.

Example:

```
decl lval;
lval = log10(10.4);    // 1val=1.0170333392988
```

mag()

Returns the absolute/magnitude value of an integer, real, or complex number.

Syntax:

mag(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = mag(-4-6i);    // a=7.211102550928
```

max2()

Returns the larger value of two numeric values, or NULL if parameters are invalid..

Syntax:

max2(val1, val2);

where

val1 is any integer or real number.

val2 is any integer or real number.

Example:

```
decl a;
a = max2(1.5, -1.5);    // a=1.5
```

min2()

Returns the leser value of two numeric values, or NULL if parameters are invalid.

Syntax:

min2(val1, val2);

where

val1 is any integer or real number.

val2 is any integer or real number.

Example:

```
decl a;
a = min2(1.5, -1.5);    // a=-1.5
```

num()

Returns an integer that represents an ASCII numeric value of the first character in the specified string.

Syntax:

num(str);

where

str is a string.

Example:

```
decl a;
a = num("/users/myhome/fullpath");    // a=47
```

phase()

Returns a real number that represents the phase in degrees of the argument. NULL is returned in case of an error in the argument.

Note *phasedeg()* performs the same operation.

Syntax:

phase(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = phase(4+6i);    // a=56.30993247402
```

phasedeg()

Returns a real number that represents the phase in degrees of the argument. NULL is returned in case of an error in the argument.

Note *phase()* performs the same operation.

Syntax:

phasedeg(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = phasedeg(4+6i);    // a=56.30993247402
```

phaserad()

Returns a real number that represents the phase in radians of the argument. NULL is returned in case of an error in the argument.

Syntax:

phaserad(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = phaserad(1.5);    // a=1.0303768265243
```

polar()

Converts polar magnitude and angle into a complex number. Returns: a complex number or NULL if an error occurs.

Syntax:

polar(mag, ang);

where

mag is any integer or real number that represents polar magnitude.

ang is any integer or real number that represents an angle.

Example:

```
decl a;
a = polar(1.5, 90);    // a=9.18485e-17+1.5i (a complex number)
```

pow()

Returns an integer or real number raised to given power.

Syntax:

pow(num, exponent);

where

num is an integer or real number to raise.

exponent is an exponent to raise number by.

Example:

```
decl b;
b = pow(4, 10);    // b=1048576
```

rad()

Converts angle from degrees to radians. Returns: A real number that represents an angle in radians.

Syntax:

rad(angle);

where

angle is any integer or real number that represents an angle in degrees.

Example:

```
decl a;
a = rad(90);    // a=1.5707963267949
```

re()

Returns a real number, the real part of a complex value.

Note *real()* performs the same operation.

Syntax:

re(complexNum);

where

complexNum is a complex number.

Example:

```
decl r;
r = re(1–1);    // returns 1
```

real()

Returns a real number, the real part of a complex value.

Note *re()* performs the same operation.

Syntax:

real(complexNum);

where

complexNum is a complex number.

Example:

```
decl r;
r = real(1–1);    // returns 1
```

round()

Returns an integer, a real number rounded to nearest integer value.

Syntax:

round(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = round(1.5);    // returns 2
```

sgn()

Returns the integer sign of an integer or real number, as either 1 or –1.

Syntax:

sgn(number);

where

number is an integer or real number.

Example:

```
decl s;
s = sgn(-14.5);    // returns -1
```

sin()

Returns the sine of a real number (in radians).

Syntax:

sin(realNum);

where

realNum is a real number, in radians.

Example:

```
decl s;
s = sin(PI/4);
```

sinc()

Returns a real number that represents sin(num)/num in radians.

Syntax:

sinc(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = sinc(.5);
```

sinh()

Returns the hyperbolic sine of an integer, real, or complex number.

Syntax:

sinh(num);

where

num is any integer, real, or complex number.

Example:

decl a;

a = sinh(1.5);

sqrt()

Returns the square root of a positive integer or real number.

Syntax:

sqrt(x);

where

x is a positive integer or real number.

Example:

```
decl s;
s = sqrt(4);    // returns 2
```

step()

A step function that returns 0, 5, or 1. Returns: 0 if the argument < 0, .5 if argument == 0, or 1 if the argument > 0.

Syntax:

step(num);

where

num is any integer, real, or complex number.

Example:

```
decl increment;
```

```
increment = step(1.5);    // returns -1
```

tan()

Returns the tangent of a real number (in radians).

Syntax:

tan(realNum);

where

realNum is a real number, in radians.

Example:

```
decl t;
t = tan(PI/4);
```

tanh()

Returns the hyperbolic tangent of an integer, real, or complex number.

Syntax:

tanh(num);

where

num is any integer, real, or complex number.

Example:

```
decl a;
a = tanh(1.5);
```

xor()

Returns an integer that represents the exclusive OR between arguments.

Syntax:

xor(num1, num2);

where

num1 is any integer.

num2 is any integer.

Example:

```
decl a;
a = xor(16, 32);    // a=48
```

Chapter 10: Utility Functions

This chapter describes each Utility function in detail. The functions are listed in alphabetical order.

arg()

Returns the nth argument passed into a function. Arguments start at 0.

Syntax:

arg(n);

where

n is a position. Integer number of the argument to retrieve.

Example:

```
defun test(a, b, c)
{
  fputs(stderr, arg(2)); // prints value of c
};
```

arg_list()

Returns the arguments passed to a function as a list.

Syntax:

arg_list();

Example:

```
defun test(a, b, c)
{
decl l;
l = arg_list();
  fputs(stderr, identify_value(arg_list());
                // prints value of a, b, c as a list
};
```

In the above example, where test(1, 2, 3), prints list(1, 2, 3)

array_size()

Returns the size of each dimension in an AEL array.

See also: offset_array(), resize_array(), array_type(), array_lowerBound(), array_upperBound(), and convert_array()

Syntax:

size = array_size(<arr>)

where

arr is a valid AEL array.

size is defined by: if <arr> is one-dimensional array, an integer is returned; if <arr> is multi-dimensional array, an array is returned.

Example:

```
decl a = {{1,2,3},{4,5,6}};
decl b = {1,2,3};

fputs(stderr, identify_value(array_size(a))); // outputs {2,3}
fputs(stderr, identify_value(array_size(b))); // outputs 3
```

array_type()

Returns the type of elements in an AEL array.

See also: offset_array(), resize_array(), array_size(), array_lowerBound(), array_upperBound(), and convert_array()

Syntax:

type = array_type(<arr>)

where

arr is a valid AEL array.

type is one of the following AEL strings: "integer", "real", "complex".

Example:

```
decl a = {{1,2,3},{4,5,6}};

decl b = convert_array(a,"complex");

fputs(stderr, array_type(a)); // outputs "integer"
fputs(stderr, array_type(b)); // outputs "complex"
```

array_lowerBound()

Returns the lower bound of the specified dimension.

See also: offset_array(), resize_array(), array_size(), array_type(), array_upperBound(), and convert_array()

Syntax:

lb = array_lowerBound(<arr>,<dim>);

where

arr is a valid AEL array.

lb is an integer or NULL_VALUE if unsuccessful.

Example:

```
decl a = {{1,2,3},{4,5,6}};

fputs(stderr, array_lowerBound(a,1));  // outputs 0
fputs(stderr, array_lowerBound(a,2));  // outputs 0
```

array_upperBound()

Returns the upper bound of the specified dimension.

See also: offset_array(), resize_array(), array_size(), array_type(), array_lowerBound(), and convert_array()

Syntax:

ub = array_upperBound(<arr>,<dim>);

where

arr is a valid AEL array.

ub is an integer or NULL_VALUE if unsuccessful.

Example:

```
decl a = {{1,2,3},{4,5,6}};

fputs(stderr, array_upperBound(a,1));  // outputs 1
fputs(stderr, array_upperBound(a,2));  // outputs 2
```

call()

Invokes or calls a function, gives a function value and an argument list (as returned from *arglist()* or *list()*). Returns: none.

Syntax:

call(fnc, args);

where

fnc is the name of AEL function to execute.

args is the argument list to pass into the function.

Example:

```
defun myfnc(s1,s2) {return strcat(s1,s2);}
call (my_fnc, list("a", "b"));
```

is equivalent to:

```
my_fnc("a", "b");
```

call_depth()

Returns the depth of the active call stack. The call depth can be used in conjunction with what_file and what_line to produce a call trace back. Returns the depth of the active call stack, where: 1 = depth if the function is called directly from the command line or menu, 2 = depth if the function is called from within another function.

Syntax:

call_depth();

Example:

```
defun test()
{
   fputs(stderr, call_depth());
}
```

If this example is invoked from the command line, outputs 0; when test() is invoked, outputs 1.

check_syntax()

Interprets a string as an AEL command and returns whether or not the syntax is valid.

Note The function stores the resulting commands in a temporary ATF file and then removes that ATF file. Also, the string is checked for SYNTAX only. It does not check that all identifiers are defined! See *list_undefined()* for this type of check. Returns True or False.

Syntax:

check_syntax(string);

where

num is a string argument that contains the text to be checked as AEL syntax.

Example:

```
decl str = "cos(3+);";
if ( !check_syntax(str) )
  fputs(stdout, "Invalid syntax");
else
  execute(str);    // returns False
```

chmod()

Changes the mode of the specified file. Returns NULL if an error occurs, TRUE if the chmod occurs.

Syntax:

chmod(filename, int);

where

filename is a string that contains the name of the file to change.

int is an integer that specifies the mode to change to.

Example:

```
chmod(“myfile.txt”, 777);
          // grants read, write, and execute permission
          // to everyone for myfile.txt
```

convert_array()

Returns a new AEL array created from an existing array, but with the elements of the existing array converted to the specified type.

See also: offset_array(), resize_array(), array_size(), array_type(), array_lowerBound(), and array_upperBound()

Syntax:

newArr = convert_array(<arr>, <str>);

where

arr is a valid AEL array

str is a valid AEL string. Possible values:

"integer", "int", "long" will convert to integer

"real", "double" will convert to real

"complex" will convert to complex

Example:

```
decl a = {{1,2,3},{4,5,6}};

decl b = convert_array(a,"complex");

fputs(stderr, identify_value(a)); // outputs {1,2,3},{4,5,6}}
fputs(stderr, identify_value(b)); // outputs
{{1+0i,2+01,3+0i},{4+0i,5+0i,6+0i}}
```

date_time()

Returns the date and time as a formatted string. This string returned by this function is identical to the C language's ctime function.

Syntax:

date_time();

Example:

```
Sun Jan 3 15:24:13 1993 // returned string
```

de_error_bell()

Sounds the error bell. Returns: none.

Syntax:

de_error_bell();

Example:

```
de_error_bell();          // emits beep on error
```

de_get_env()

Retrieves environment file name.

See also: *getenv()*, *setenv()*.

Syntax:

de_get_env();

Example:

```
decl envFileName;
envFileName = de_get_env();
```

de_set_error_bell()

Enables or disables the error bell (on or off). Returns: none.

Syntax:

de_set_error_bell(bEnabled);

where

bEnable is 1 to turn on; 0 to turn off the error bell.

Example:

```
de_set_error_bell(1);     // enables the error bell
```

de_set_warning_bell()

Enables or disables the warning bell (on or off). Returns: none.

Syntax:

de_set_warning_bell(bEnabled);

where

bEnable is 1 to turn on; 0 to turn off the warning bell.

Example:

```
de_set_warning_bell(1);     // enables the warning bell
```

de_warning_bell()

Sounds the warning bell. Returns: none.

Syntax:

de_warning_bell();

Example:

```
de_warning_bell();    // beeps the bell as warning
```

delete_word()

Deletes a word from the current vocabulary, or from a specified vocabulary. Returns True if the name is found and deleted; False if cannot delete or if an invalid vocName was specified.

Syntax:

delete_word(name, vocName <opt>);

where

name is a string that contains the name of the word to delete.

vocName is a string that contains the name of the vocabulary from which to delete name.

Example:

```
decl a, b;
a = delete_word("b");
```

error()

Reports an error. Pops up the error dialog, displaying given error. Returns: none.

Syntax:

error(errorFileName, errorNum, defaultErrString, infoString);

where

errorFileName is the name of error message file (for internationalized messages). The file does not have to exist if default error string is given.

errorNum is the line number in the message file for this error string.

defaultErrString is a string used f no error found or error file not found.

infoString is the non-internationalized varying part of error string.

Example:

```
error("aelcmd", 2, "item error", "hello");
```

evaluate()

Evaluates an AEL expression and returns the result. The string passed must be a valid AEL expression. The value of the expression is returned as well as stored in the global variable EvaluationResult.

AEL is always interpreted in the context of a vocabulary of definitions which are used to resolve identifier references. Whenever a variable or function name is used in the AEL program, the context vocabulary is searched for an existing definition. New variables and function definitions are added to the context vocabulary as well. The optional second argument allows the context to be overridden for this function by providing the name of the desired vocabulary.

In the design environment, two contexts are important: SimCmd and CmdOp. Generally SimCmd is the context for any AEL files loaded during program initialization and when opening a project. The context is CmdOp when interpreting menu commands and in the Command Line dialog box. SimCmd is a superset of CmdOp, so not all definitions in SimCmd are available from the Command Line dialog without specifying the context SimCmd specifically. Returns: Value of the expression.

See also: *load(), execute()*.

Syntax:

evaluate(string, [context]);

where

string is the text of the AEL expression.

context is the name of vocabulary context for evaluation of the expression.

Example:

```
x = evaluate( “3 * 8” );     // sets x = integer value 24
                             // also sets EvaluationResult to 24
```

execute()

Interprets text as an AEL program. The text string must contain valid AEL statements.

AEL is always interpreted in the context of a vocabulary of definitions which are used to resolve identifier references. Whenever a variable or function name is used in the AEL program, the context vocabulary is searched for an existing definition. New variables and function definitions are added to the context vocabulary as well. The optional second argument allows the context to be overridden for this function by providing the name of the desired vocabulary.

In the design environment, two contexts are important: SimCmd and CmdOp. Generally SimCmd is the context for any AEL files loaded during program initialization and when opening a project. The context is CmdOp when interpreting menu commands and in the Command Line dialog box. SimCmd is a superset of CmdOp, so not all definitions in SimCmd are available from the Command Line dialog without specifying the context SimCmd specifically. Returns: None.

See also: *load()*, *evaluate()*.

Syntax:

execute(string, [context]);

where

string is the text of the AEL program.

context is the name of vocabulary context for evaluation of the expression.

Example:

```
decl x;
execute( “x=3*8;” ); //sets x = integer value 24
```

expandenv()

Returns a string with the environment variables and HP EEsof configuration variables fully expanded into their values, given a string that contains environment variable references.

Syntax:

expandenv(str);

where

str is a string that contains environment variable references.

Example:

```
decl path;
path = expandenv("$PATH");
```

file_loaded()

Tests to see whether an AEL file has been read. Returns TRUE if the file exist and has been read, otherwise FALSE.

Syntax:

file_loaded(name [, voc, searchpath]);

where

name is a string that contains the name of the file to test.

voc is optional. A string that contains the name of the vocabulary (not really used for anything).

searchpath is optional. A string that contains the search path to search for the specified file.

Example:

```
decl a;
a = file_loaded("myAELfile.ael");
```

filedate()

Returns an integer that represents a time stamp of a file.

Syntax:

filedate(filepath);

where

filepath is a string that represents the file to check or a file pointer that points to a file obtained via *fopen()*.

Example:

decl a;

a = filedate("myAELfile.ael");

filestat()

Returns a binary number that represents the status of a file. It is interpreted in the same way that UNIX interprets the permission bits for a file.

Syntax:

filestat(filepath);

where

filepath is a string that represents the file to check OR a file pointer that points to a file (obtained via *fopen()*).

The returned binary numbers can be interpreted as:

01 = Execute permission to others
02 = Write permission to others
04 = Read permission to others
010 = Execute permission to group
020 = Write permission to group
040 = Read permission to group
0100 = Execute permission to owner
0200 = Write permission to owner
0400 = Read permission to owner

Example:

```
decl stat;
stat = filestat("myAELfile.ael");

if (stat & 0200)
{
// writeable by owner
}

if ((stat & 0400) &&
(stat & 040) &&
    (stat & 04))
{
// readable by all
}
```

find_word()

Returns the address of an AEL word if it exists, otherwise NULL is returned.

Syntax:

find_word(name, voc);

where

name is a string that specifies the name of the AEL word to find.

voc is optional. A string that specifies what vocabulary to search for name. If it is not specified, the current vocabulary is searched. Note that parent vocabularies are searched if the word is not found at that level.

Example:

```
decl a;
a = find_word("cos");
```

find_word_voc()

Returns the address of a vocabulary in which a specified AEL word is found if it exists, otherwise NULL is returned.

Syntax:

find_word(name, voc);

where

name is a string that specifies the name of the AEL word to find.

voc is optional. A string that specifies what vocabulary to search for name. If it is not specified, the current vocabulary is searched. Note that parent vocabularies are searched if the word is not found at that level.

Example:

```
decl a;
a = find_word("cos", "AEL");
```

fix_path()

Adds the appropriate backslashes needed in a string to handle control characters correctly. This should be used when dealing with strings that represent paths. Returns a string that represents a path with backslashes added appropriately to handle control characters.

Syntax:

fix_path(path);

where

path is a string.

Example:

```
decl newPath;
newPath = fix_path("/users/myhome/project/template.ael");
```

format_date_time()

Returns the date and time as a formated string. The formatting can be specified using a format string. This string and formatting rules are identical to the C language's *strftime()* function. Note, this function is sensitive to the LANG language environment variable and returns internationalized date and time strings based on this variable.

Syntax:

format_date_time(format_string);

where

format_string is identical to the format string accepted by the C library function *strftime()*. Each *%c* is replaced using values appropriate for the local environment). No more than *smax* characters are placed into *s*. The *strftime()* returns the number of characters, excluding the *\0*, or zero if more than *smax* characters were produced.

where

%a is the abbreviated weekday name
%A is the full weekday name
%b is the abbreviated month name
%B is the full month name
%c is the local date and time representation
%d is the day of the month (01-31)
%H is the hour (24-hour clock) (00-23)
%I is the hour (12-hour clock) (01-12)
%j is the day of the year (001-366)
%m is the month (01-12)
%M is the minute (00-59)
%p is the local eqluivalent of AM or PM
%S is the second (00-61)
%U is the week number of the year (Sunday as first day of week) (00-53)
%w is the weekday (0-6, Sunday is 0)

%W is the week number of the year (Monday as first day of week) (00-53)
%x is the local date representation
%X is the local time representation
%y is the year without century (00-99)
%Y is the year with century

Example:

```
decl datetime;
datetime = format_date_time("%a %B"); //prints "Weds July"
```

get_dir_files()

Returns an AEL list of files found in a specified directory. The suffix of the files to list may be specified.

Syntax:

get_dir_files(dirName [, suffix]);

where

dirName is a string that contains the directory name to list its files.

suffix is optional. A string that contains the suffix of the files to list.

Example:

```
decl ls;
ls = get_dir_files("/users/myhome/dbproject", "atf");
```

getcwd()

Returns a string representing the path name of the current working directory. This is almost always the path name of the attached project.

Syntax:

getcwd();

Example:

```
decl prjName;
prjName = getcwd();
```

getenv()

Returns a string representing the configuration variable value. If the name of the configuration file is given, then the configuration variable values are set in the named

environment file. If the name of the configuration variable is not specified, an application-specific default is used.

See also: *expandenv()*, *setenv()*.

Syntax:

getenv(name[, conf]);

where

name is the name of configuration variable.

conf is optional. Name of configuration file to use. In an attempt to locate the file, the program searches the following directories, in this order: project directory, home directory, custom directory, installation directory. The file name should be specified without a trailing *.cfg*. Entering a path for this file is not allowed.

Example:

```
decl val;
getenv("LAYERS_PATH", "my_layers");
```

geterror()

Returns a string that represents the last error message that occurred.

Syntax:

geterror();

Example:

```
decl a;
a = geterror();
```

getppid()

Returns an integer that represents the parent process ID of the current process.

Syntax:

getppid();

Example:

```
deck al
a = getppid();
```

getsysenv()

Returns a string representing a system environment variable value.

Syntax:

getsysenv(name);

where

name is the name of a environment variable.

Example:

```
decl v;
getsysenv("PRINTER");    //Return PRINTER value
```

identify_value()

Returns a single string representing the value(s) of any valid AEL object. The object can be of any type.

See also: “Database Overview” on page 2-1.

Syntax:

identify_value(object);

where

object is any AEL object. The object can be of any type (such as list, handle, or variable), including complex lists (that is, lists containing other lists). List objects are enclosed in brackets [ ]. For non-printable objects, prints the name (such as handles).

Example:

```
// Shows 2 in an information dialog box
de_info(identify_value(2));

// Shows list ("a", "b", list(1,2.22,"c")) in an information dialog box
decl myList = list ("a", "b", list(1,2.22,"c"));
de_info(identify_value(myList));

// Sets myString to "12.3"
decl myString = identify_value(12.3)

// Shows <WinInst> in an information dialog box
// Window instances are not printable,
// so this just shows that it’s not NULL.
```

```
de_info(api_get_current_window());
```

is_complex()

Identifies whether a given value is a complex number. Returns TRUE if a given value is a complex number, NULL if given value is not a complex number.

Syntax:

is_complex(value);

where

value is any value.

Example:

```
is_complex(10.0); // returns NULL
```

is_dir()

Determines if a given path is a directory. Returns TRUE if path is a directory, otherwise FALSE.

Syntax:

is_dir(path);

where

path is a string that represents a path.

Example:

```
decl a;
a = is_dir("/users/myhome/project");
```

is_file()

Identifies whether a given value is a file identifier. Returns TRUE if a given value is a file identifier, NULL if given value is not a file identifier.

Syntax:

is_file(value);

where

value is any value.

Example:

```
decl f;
f = fopen("atest", "R");
is_file(f); // returns TRUE
```

is_function()

Identifies whether a given value is a function. Returns TRUE if a given value is a function, NULL if given value is not a function.

Syntax:

is_function(value);

where

value is any value.

Example:

```
is_function("abc"); // returns NULL
```

is_integer()

Identifies whether a given value is an integer number. Returns TRUE if a given value is an integer number, NULL if given value is not an integer.

Syntax:

is_integer(value);

where

value is any value.

Example:

```
is_integer(2.5); // returns NULL
```

is_list()

Syntax:

is_list(value);

where

value is any value.

Example:

```
decl list1;
```

```
list1 = list(1,2,3);
is_list(list1); // returns TRUE
is_list(4); // returns NULL
```

is_real()

Identifies whether a given value is a real number. Returns TRUE if a given value is a real number, NULL if given value is not a real number.

Syntax:

is_real(value);

where

value is any value.

Example:

```
is_real(12.4); // returns TRUE
```

is_string()

Identifies whether a given value is a list. Returns TRUE if a given value is a list, NULL if given value is not a list.

Syntax:

is_string(value);

where

value is any value.

Example:

```
is_string("abc"); // returns TRUE
```

is_type()

Identifies whether a given value is of a given type. Returns TRUE if a given value is of a given type, NULL if given value does not match the type.

Syntax:

is_type(typeStr, value);

where

typeStr is the type to check the value against. The valid typeStr are: “int”, “double”, “complex”, “list”, “string”, “file”, “function”, “vocabulary”, “word”.

value is any value.

Example:

```
is_type("int", 10); // returns TRUE
```

is_voc()

Identifies whether a given value is a vocabulary reference. Returns TRUE if a given value is a vocabulary reference, NULL if given value is not a vocabulary reference.

Syntax:

is_voc(value);

where

value is any value.

Example:

```
is_voc("not a voc"); // returns NULL
```

is_word()

Identifies whether a given value is a defined word in an AEL vocabulary. Returns TRUE if a given value is a defined word reference, NULL if given value is not a defined word reference.

Syntax:

is_word(value);

where

value is any value.

Example:

```
is_word(defun); // returns TRUE
```

list_undefined()

Interprets a string as an AEL command and returns an AEL list of undefined identifiers in that command.

Syntax:

list_undefined(str);

where

str is a string that represents an AEL command(s).

Example:

```
decl a;
a = list_undefined("a = b + c + d;");
                  // where c is not defined, returns list(c)
```

load()

Loads an AEL file and interprets its statements. If you load an AEL file from within another using *load()*, functions and variables defined in the loaded file cannot be accessed from the other file without a forward declaration preceding the call to *load()*. Use *decl name();* for forward declarations of variables and *defun name();* for forward declarations of functions.

AEL is always interpreted in the context of a vocabulary of definitions which are used to resolve identifier references. Whenever a variable or function name is used in the AEL program, the context vocabulary is searched for an existing definition. New variables and function definitions are added to the context vocabulary as well. The optional second argument allows the context to be overridden for this function by providing the name of the desired vocabulary.

In the design environment, two contexts are important: SimCmd and CmdOp. Generally SimCmd is the context for any AEL files loaded during program initialization and when opening a project. The context is CmdOp when interpreting menu commands and in the Command Line dialog box. SimCmd is a superset of CmdOp, so not all definitions in SimCmd are available from the Command Line dialog without specifying the context SimCmd specifically. Returns: none.

Syntax:

load(aelFileName, [context]);

where

aelFileName is the he name of an AEL file (without the extension).

context is the name of the vocabulary context for interpreting the file.

Example:

This example causes “I am a” to be printed.

```
File a.ael:
defun func_a()
{
fputs(stderr,"I am a");
}
Loading a.ael:
load("a");
func_a();
```

mkdir()

Creates a directory. Returns TRUE if a directory is created; FALSE if a directory is not created.

Syntax:

mkdir(str);

where

str is a string that contains the name of the directory.

Example:

```
decl a;
a = mkdir("myDir");
```

num_args()

Returns the number of arguments passed into a function.

Syntax:

num_args();

Example:

```
defun test(a, b, c)
{
  fputs(stderr, num_args());
}
test(10);    // should print out 1
test(10, 15);    // should print out 2
```

offset_array()

Creates a new array by copying an existing AEL array except that the lower bound of a specified dimension is modified to a given value.

See also: resize_array(), array_size(), array_type(), array_lowerBound(), array_upperBound(), and convert_array()

Syntax:

new = offset_array(<arr>, <dim>, <min>)

where

arr is a valid AEL array.

dim is an integer that specifies which dimension of <arr> will have its lower bound modified in the new array.

min is an integer that specifies what the lower bound of <dim> will be in the new array.

new is a new valid AEL array.

Example:

```
decl i,j;
    decl a = {{1,2,3},{4,5,6}};
    fputs(stderr, identify_value(a)); // outputs {{1,2,3},{4,5,6}}
    for (i=0;i<2;i++)                 // NOTE: access begins at 0
    for (j=0; j<3; j++)
    fputs(stderr,a[i,j]);          // loop output is: 1 2 3 4 5 6

    decl b = resize_array(a,2,-1,3);
    fputs(stderr, identify_value(b)); // outputs
{{<int>,1,2,3,<int>},{<int>,4,5,6,<int>}}
    // where <int> is uninitialized integer.
    for (i=0;i<2;i++)                // NOTE: access of 2nd dim begins at -1
    for (j=-1; j<4; j++)
    fputs(stderr,b[i,j]); // loop output is: <int> 1 2 3 <int> <int> 4 5 6
<int>

    decl b = offset_array(b,2,0);
    for (i=0;i<2;i++)                // NOTE: access of 2nd dim now begins
at 0
    for (j=0; j<5; j++)
   fputs(stderr,b[i,j]); // loop output is: <int> 1 2 3 <int> <int> 4 5 6
<int>
```

on_error()

Sets the AEL function to be called in case of an error. If the argument is omitted or is NULL, then the current error handler is removed. The address of the old error

handler function is returned. If the argument is not an AEL function address, the function fails and reports an error.

When an error occurs, the error handler function is called and passes the following:

- Error code, an integer indicating line in message file. Common run time error codes for the class “AEL” are listed in Table 10-1.
- Error class, a string indicating name of message file
- Operation code, an integer indicating the type of operation occurring. The AEL operation codes are listed in Table 10-2.
- File name, a string identifying the file where error occurred.
- Line, an integer indicating the line number where error occurred (starting at 1).
- Column, an integer indicating the column number where error occurred (starting at 1).

The error handling function should return a value of NULL to ignore the error (after taking appropriate corrective action) or a non-zero integer value to allow AEL to continue error processing. Returns a function address.

See also: *error()*.

Syntax:

on_error([func]);

where

func is the address of error handling function.

Example:

```
defun report_error( code, class, op, line, col)
{
fputs( stderr, strcat("Error occurred:", class, code, " at ", line, "/",
col);
return TRUE
}
on_error( report_error);
```

Table 10-1. Common Run Time Error Codes

Code	Definition
300	Integer parameter value was required
301	Real parameter value was required

Table 10-1. Common Run Time Error Codes (continued)

Code	Definition
302	Two numeric parameters are required
303	String parameter value was required
304	Incorrect type of parameter value
305	Parameters cannot be matched
306	Stack reference to value not on stack
307	Wrong number of parameters passed
308	Numeric parameter expected
309	Function address missing for call
310	Function address not executable
311	Word reference parameter expected
312	Bad frame reference */
313	Integer divide by zero
314	Real divide by zero
315	Complex divide by zero
316	Illegal return value from compare function
317	Values not comparable
318	Log of negative number
319	Log of zero
320	Square root of negative number

Table 10-2. AEL Operation Codes

Code	Definition	Code	Definition
0	discard return value	16	assign
1	logical OR	17	dereference word
2	logical AND	18	call function
3	logical NOT	19	call function, discard return value
4	logical test ==	20	return from function
5	logical test !=	21	branch on TRUE
6	logical test >=	25	bitwise AND

Table 10-2. AEL Operation Codes (continued)

Code	Definition	Code	Definition
7	logical test <=	26	bitwise exclusive OR
8	logical test >	27	bitwise OR
9	logical test <	28	bitwise ones compliment
10	add	29	bitwise left shift
11	subtract	30	bitwise right shift
12	multiply	31	pre-increment
13	modulus divide	32	pre-decrement
14	divide	33	post-increment
15	negate	34	post-decrement

pcb_get_form_value()

Retrieves the value of a parameter that is a constant form. This function must be called within a PARM_MODIFIED_CB callback function created by *dm_create_cb()*.

See also: *pcb_get_mks()*, *pcb_set_mks()*, *pcb_set_form_value()*, *pcb_get_string()*, *pcb_set_string()*.

Syntax:

pcb_get_form_value(callData, paramName)

where

callData is the third argument of the PARM_MODIFIED_CB callback function.

paramName is the name of the parameter to get the value from.

Example:

```
create_constant_form("first","first",0,"0","first");
create_constant_form("second","second",0,"1","second");
create_form_set("MyFormSet","first","second");

create_item(...
  create_parm("A",
              "A parameter value",
              NULL,
              "MyFormSet",
              UNITLESS_UNIT,
              prm("first", ""),
              list(dm_create_cb(PARM_MODIFIED_CB, "a_modified_cb", NULL, TRUE))),
  create_parm("B",
              "B parameter value",
              NULL,
```

```
              "MyFormSet",
              UNITLESS_UNIT,
              prm("second", "")));

defun a_modified_cb(cbP, clientData, callData)
{
    decl a_formValue=pcb_get_form_value(callData, "A");
    decl b_formValue;

    if (strcmp(a_formValue, "first")==0)
        b_formValue="second";
    else
        b_formValue="first";
    return(pcb_set_form_value(NULL, "B", b_formValue));
}
```

pcb_get_mks()

Retrieves the value of a parameter in MKS (unscaled) units. This function must be called within a PARM_MODIFIED_CB callback function created by *dm_create_cb()*.

See also: *pcb_set_mks()*, *pcb_get_form_value()*, *pcb_set_form_value()*, *pcb_get_string()*, *pcb_set_string()*.

Syntax:

pcb_get_mks(callData, paramName)

where

callData is the third argument of the PARM_MODIFIED_CB callback function.

paramName is the name of the parameter to get the value from.

Example:

```
create_item(...
  create_parm("A",
              "A parameter value",
              PARM_REAL | PARM_STATISTICAL | PARM_OPTIMIZABLE,
              "StdFormSet",
              0,
              prm("StdForm", "10.0"),
              list(dm_create_cb(PARM_MODIFIED_CB, "a_modified_cb", NULL, TRUE))),
  create_parm("B",
              "B parameter value",
              PARM_REAL | PARM_STATISTICAL | PARM_OPTIMIZABLE,
              "StdFormSet",
              0,
              prm("StdForm", "20.0")));

defun a_modified_cb(cbP, clientData, callData)
{
    decl a_mks=pcb_get_mks(callData, "A");
    return(pcb_set_mks(paramData, "B", a_mks*2));
}
```

pcb_get_parm_type()

Returns parameter type. This function must be called within a PARM_MODIFIED_CB callback function created by *dm_create_cb()*. Returns one of the following strings:

"NUMBER"
"EXPRESSION"
"CONSTANT_FORM"
"VARIABLE"
"UNKNOWN_FORM"
"UNKNOWN"
"STRING"
"DEFAULT_VALUE"
"DATA_FILE"
"ERROR"

See also: *pcb_get_mks()*, *pcb_set_mks()*, *pcb_get_form_value()*, *pcb_set_form_value()*, *pcb_get_string()*, *pcb_set_string()*.

Syntax:

pcb_get_parm_type(callData, paramName);

where

callData is the third argument of the PARM_MODIFIED_CB callback function.

paramName is the name of the parameter to get the type from.

Example:

```
defun a_modified_cb(cbP, clientData, callData)
{
   decl paramData=NULL;
   decl a_mks=pcb_get_mks(callData, "A");

   paramData=pcb_set_mks(paramData, "B", a_mks*2);
   return(pcb_set_mks(paramData, "C", a_mks*3));
}
```

pcb_get_string()

Retrieves the value of a parameter that is a constant form. This function must be called within a PARM_MODIFIED_CB callback function created by *dm_create_cb()*.

See also: *pcb_get_mks()*, *pcb_set_mks()*, *pcb_get_form_value()*, *pcb_set_form_value()*, *pcb_set_string()*.

Syntax:

pcb_get_string(callData, paramName)

where

callData is the third argument of the PARM_MODIFIED_CB callback function.

paramName is the name of the parameter to get the value from.

Example:

```
create_item(...
  create_parm("A",
              "A parameter value",
              PARM_STRING,
              "StringAndReferenceFormSet",
              STRING_UNIT,
              prm("StringAndReference", "myAString"),
              list(dm_create_cb(PARM_MODIFIED_CB, "a_modified_cb", NULL, TRUE))),
  create_parm("B",
              "B parameter value",
              PARM_STRING,
              "StringAndReferenceFormSet",
              STRING_UNIT,
              prm("StringAndReference", "myBString")));

defun a_modified_cb(cbP, clientData, callData)
{
    decl a_stringValue=pcb_get_string(callData, "A");
    return(pcb_set_string(NULL, "B", a_stringValue));
}
```

pcb_set_form_value()

Sets the value of a parameter that is a constant form. This function must be called within a PARM_MODIFIED_CB callback function created by *dm_create_cb()*. Returns data which must be returned by the PARM_MODIFIED_CB callback function.

See also: *pcb_get_mks()*, *pcb_set_mks()*, *pcb_get_form_value()*, *pcb_get_string()*, *pcb_set_string()*.

Syntax:

pcb_set_form_value(paramData, paramName, value)

where

paramData is parameter data. NULL the first time. In addition to this variable being a parameter to this function, the value returned by this function must also be assigned to it.

paramName is the name of the parameter to set the value of.

value is the new value in MKS units.

Example:

```
create_constant_form("first","first",0,"0","first");
create_constant_form("second","second",0,"1","second");
create_form_set("MyFormSet","first","second");

create_item(...
  create_parm("A",
              "A parameter value",
              NULL,
              "MyFormSet",
              UNITLESS_UNIT,
              prm("first", ""),
              list(dm_create_cb(PARM_MODIFIED_CB, "a_modified_cb", NULL, TRUE))),
  create_parm("B",
              "B parameter value",
              NULL,
              "MyFormSet",
              UNITLESS_UNIT,
              prm("second", "")));

defun a_modified_cb(cbP, clientData, callData)
{
    decl a_formValue=pcb_get_form_value(callData, "A");
    decl b_formValue;

    if (strcmp(a_formValue, "first")==0)
        b_formValue="second";
    else
        b_formValue="first";
    return(pcb_set_form_value(NULL, "B", b_formValue));
}
```

pcb_set_mks()

Sets the value of a parameter. The value must be in MKS (unscaled) units. This function must be called within a PARM_MODIFIED_CB callback function created by dm_create_cb(). Returns data which must be returned by the PARM_MODIFIED_CB callback function.

See also: *pcb_get_mks()*, *pcb_get_form_value()*, *pcb_set_form_value()*, *pcb_get_string()*, *pcb_set_string()*.

Syntax:

pcb_set_mks(paramData, paramName, value)

where

paramData is parameter data. NULL the first time. In addition to this variable being a parameter to this function, the value returned by this function must also be assigned to it.

paramName is the name of the parameter to set the value of.

value is the new value in MKS units.

Example:

```
create_item(...
  create_parm("A",
              "A parameter value",
              PARM_REAL | PARM_STATISTICAL | PARM_OPTIMIZABLE,
              "StdFormSet",
              0,
              prm("StdForm", "10.0"),
              list(dm_create_cb(PARM_MODIFIED_CB, "a_modified_cb", NULL, TRUE))),
  create_parm("B",
              "B parameter value",
              PARM_REAL | PARM_STATISTICAL | PARM_OPTIMIZABLE,
              "StdFormSet",
              0,
              prm("StdForm", "20.0")),
  create_parm("C",
              "C parameter value",
              PARM_REAL | PARM_STATISTICAL | PARM_OPTIMIZABLE,
              "StdFormSet",
              0,
              prm("StdForm", "30.0")));

defun a_modified_cb(cbP, clientData, callData)
{
    decl paramData=NULL;
    decl a_mks=pcb_get_mks(callData, "A");

    paramData=pcb_set_mks(paramData, "B", a_mks*2);
    return(pcb_set_mks(paramData, "C", a_mks*3));
}
```

pcb_set_string()

Sets the value of a parameter that is a constant form.This function must be called within a PARM_MODIFIED_CB callback function created by *dm_create_cb()*. Returns data which must be returned by the PARM_MODIFIED_CB callback function.

See also: *pcb_get_mks()*, *pcb_set_mks()*, *pcb_get_form_value()*, *pcb_set_form_value()*, *pcb_set_string()*.

Syntax:

pcb_set_string(paramData, paramName, value)

where

paramData is parameter data. NULL the first time. In addition to this variable being a parameter to this function, the value returned by this function must also be assigned to it.

paramName is the name of the parameter to set the value of

value is the new value in MKS units

Example:

```
create_item(...
  create_parm("A",
              "A parameter value",
              PARM_STRING,
              "StringAndReferenceFormSet",
              STRING_UNIT,
              prm("StringAndReference", "myAString"),
              list(dm_create_cb(PARM_MODIFIED_CB, "a_modified_cb", NULL, TRUE))),
  create_parm("B",
              "B parameter value",
              PARM_STRING,
              "StringAndReferenceFormSet",
              STRING_UNIT,
              prm("StringAndReference", "myBString")));

defun a_modified_cb(cbP, clientData, callData)
{
    decl a_stringValue=pcb_get_string(callData, "A");
    return(pcb_set_string(NULL, "B", a_stringValue));
}
```

rename_word()

Finds an AEL word and renames it. Returns TRUE if rename took place, FALSE if an error occured.

Syntax:

rename_word(orig, new [, voc]);

where

orig is a string that represents the original name of the word to be changed.

new is a string that represents the new name to be given to orig.

voc is optional. A string that represents what vocabulary to search for orig. If voc is not passed, the current vocabulary is searched. Note that the parent tree of the vocabulary is searched until orig is found.

Example:

```
decl a=10;
rename_word("a", "b");
               // a will no longer access a variable
               // the variable can now only be accessed with "b"
```

resize_array()

Creates a new array by copying an existing AEL array with resizing of the specified dimension. Note that if the dimension resize is bigger then the new elements of that dimension will be set to 0 or "".

See also: offset_array(), array_size(), array_type(), array_lowerBound(), array_upperBound(), and convert_array()

Syntax:

new = resize_array(<arr>,<dim>,<min>,<max>)

where

arr is a valid AEL array.

dim is an integer that specifies which dimension of <arr> to resize in new array (ranges from 1 to inf).

min is an integer that specifies the lower bound of <dim> of new array.

max is an integer that specifies the upper bound of <dim> of new array.

new is a new valid AEL array.

Example:

```
decl a = {{1,2,3},{4,5,6}};

decl b = resize_array(a,1,0,5);
decl c = resize_array(a,2,0,1);
decl d = resize_array(a,2,-1,1);
decl e = resize_array(a,2,-1,3);

    fputs(stderr, identify_value(a)); // outputs {{1,2,3},{4,5,6}}
```

```
    fputs(stderr, identify_value(b)); // outputs
{{1,2,3},{4,5,6},{<int>,<int>,<int>},{<int>,<int>,<int>},{<int>,<int>,<int>}
}  where <int> is uninitialized integer
    fputs(stderr, identify_value(c)); // outputs {{1,2},{4,5}}
    fputs(stderr, identify_value(d)); // outputs {{<int>,1,2},{<int>,4,5}}
    fputs(stderr, identify_value(e)); // outputs
{{<int>,1,2,3,<int>},{<int>,4,5,<int>}}
```

Note As demonstrated, you can add new elements to either the front or back of an existing array. In order to reset the offset back to 0 to max, see *offset_array()*.

---

setenv()

Sets the value of a named configuration variable. If the name of the configuration file is given, then the configuration variable values are set in the named environment file. If the name of the configuration variable is not specified, an application-specific default is used. Returns: none.

See also: *expandenv()*

Syntax:

setenv(name, value[, conf, dir, savetodisk]);

where

name is the name of configuration variable.

value is a new value of variable.

conf is optional. Name of configuration file to use.

dir is optional. Integer value that specifies the directory where setenv preferences are saved, where

ASTR_ENV_SAVE_CWD = current working directory
ASTR_ENV_SAVE_HOME = home directory
ASTR_ENV_SAVE_SYS = installation directory

savetodisk is optional. Integer value that controls whether the configuration is written to disk or not, where

0 = Do not write to disk
1 = Write to disk

Example:

```
setenv("LAYERS_PATH", "my_layers");
```

sleep()

Sets an idle time for the program. Returns: None.

Syntax:

sleep(time);

where

time is the time to idle between next command executed, for seconds.

Example:

```
sleep(10); //sleep 10 seconds
```

start_timer()

Syntax:

start_timer();

Records the current timestamp internally. The recording is used when *total_elapsed_time()* is called. Returns: NULL.

Example:

```
start_timer();
```

system()

Executes a system command and returns the *command* exit status. The argument passes directly to the UNIX command interpreter. This function works on both UNIX and PC systems, but the execution and results of the command may be different.

Syntax:

system(command[,mapStdErr,invokeDir,retValue]);

where

command is a system command. It can be a quoted string.

mapStdErr is an optional Boolean parameter (TRUE or FALSE). Specifies mapping stderr.

invokeDir is an optional string parameter. Specifies in which directory to invoke the command.

retValue is an optional Boolean parameter (TRUE or FALSE). Specifies whether to return the value of, and to specify to wait for, the command to finish before continuing AEL. If this is not passed or is FALSE, the return value is the program handle used by AEL, not usefile to users.

Note You cannot use redirect or pipes in command. Using a redirection or pipe causes an error.

Example:

```
system("mv abc.dsn def.dsn");  // invokes the mv system command
                               // with the parameters abc.dsn and def.dsn
```

You can obtain a return value from a command executed in system() and AEL will not continue until the command has been completed.

The value returned is sent to stdout. If you are running a shell script and have an exit code, that exit code is not what is returned. Therefore, you must output to stdout any information you want returned to AEL.

Example:

This example sets “a” to a string representing a “ls” of the current directory.

```
decl a = system("ls",FALSE,"",TRUE);
```

Example:

This example sets “a” to a string representing a “ls” of /users/kla/pde directory.

```
decl a = system("ls",FALSE,"/users/kla/pde",TRUE);
```

Running Shell Scripts

Suppose xxx.ksh:

```
#! /bin/ksh
ls > tmp
cat tmp
exit 1
```

In Unix:

```
decl a = system("xxx.ksh", FALSE, "", TRUE);
decl a = system("ksh xxx.ksh", FALSE, "", TRUE);
```

Both lines return “ls” of current directory (because of cat command sending the contents of tmp to stdout).

In PC:

```
decl a = system("ksh xxx.ksh", FALSE, "", TRUE);
```

Same results. However, not that “ksh” must be placed in front of the shell script name.

tmpnam()

Generates the name of a unique temporary file. Returns a string that represents a unique temporary file.

Syntax:

tmpnam();

Example:

```
decl a;
a = tmpnam();  // a is a string that represents a unique temporary file
```

total_elapsed_time()

Returns the time elapsed since the last *start_timer()* was called as an AEL list of length 2. The first item of the list is an integer that represents the seconds elapsed. The second item of the list is an integer that represents the useconds elapsed. If an error occurs (like *start_timer()* was not previously called), then NULL is returned.

Syntax:

total_elapsed_time();

Example:

```
decl time, sec, usec;
start_timer();
.....
time = total_elapsed_time();
sec = nth(0,time);
usec = nth(1,time);
```

validate_name()

Verifies that a string is a valid program file name; that is, that the string has no illegal punctuation characters or embedded spaces. Returns an integer indicating validity of file name, where: 0 = invalid, 1 = valid.

Syntax:

validate_name(string);

where

string is a string representing a file name.

Example:

Most often, this function is used in conjunction with *create_text_form()* to ensure that the file name entered is valid. In this example, the validate_name function will be called automatically to validate the value for the parameter that uses a text form, when the value is entered.

```
create_text_form("meas","Measurements", 0, "%v", "%v", get_measurement_list,
validate_name);
```

warning()

Issues a warning message which is then printed in the warning dialog. Returns: none.

Syntax:

warning(errorFileName, lineNum, defaultMsg, string);

where

errorFileName is the name of an error file containing error messages.

lineNum is an integer, line number in the error file for this warning message.

defaultMsg is the default message string to use if error file or message not found.

string is the string to print out at end of message.

Example:

```
warning("art", 180, fmt_tokens(list(elem, "W1, W2, W3 or W4< W/2")), NULL);
```

what_col()

Returns the column number of the caller of the current function as an integer.

Syntax:

what_col([call_depth]);

where

call_depth is optional. An integer that instructs the function to return the column number of the function at the specified call_depth. If not supplied, call_depth is defaulted to 1.

Example:

File x,ael:

```
defun funct()
{   // refers to funct() being called
fputs(stdout,what_col());      // prints 1
   // refers to what_col() being called
fputs(stdout,what_col(0));      // 14
   // refers to funct() being called
fputs(stdout,what_col(1));      // prints 1
   // refers to x.ael being called (or loaded)
fputs(stdout,what_col(2));      // prints 1
}

funct();
   // refers to x.ael being called (or loaded)
fputs(stdout,what_col());      // prints 1
```

what_file()

Returns the file name of the caller of the current function as a string when the optional argument, *call_depth()*, is supplied.

Syntax:

what_file([call_depth]);

where

call_depth is optional; default = 1. Instructs the function to return the file name of the file containing the calling function.

Example:

File x,ael:

```
defun funct()
{   // refers to funct() being called
fputs(stdout,what_file());      // prints ./x.ael
   // refers to what_file() being called
fputs(stdout,what_file(0));      // prints ./x.ael
   // refers to funct() being called
fputs(stdout,what_file(1));      // prints ./x.ael
   // refers to x.ael being called (or loaded)
```

```
fputs(stdout,what_file(2));      // prints ""
}

funct();
   // refers to x.ael being called (or loaded)
fputs(stdout,what_file());      // prints ""
```

what_function()

Takes an optional integer parameter to indicate what level of the function stack to extract.

Syntax:

what_function([level]);

where

level is optional; default = 1. Instructs the function which level of the function stack to extract.

Example:

```
defun x()
{
fputs(stdout,what_function(0));      // prints out "what_function"
fputs(stdout,what_function(1));      // prints out "x"
fputs(stdout,what_function(2));      // prints "", indicating main
fputs(stdout,what_function(3));      // ERROR
}
```

what_line()

Returns the line number of the caller of the current function as an integer.

Syntax:

what_line([call_depth]);

where

call_depth is optional; default = 1. Instructs the function to return the line number of the caller of the function at that call_depth.

Example:

File x,ael:

```
defun funct()
{   // refers to funct() being called
```

```
fputs(stdout,what_line());      // prints 20
   // refers to what_line() being called
fputs(stdout,what_line(0));      // 9
   // refers to funct() being called
fputs(stdout,what_line(1));      // prints 20
   // refers to x.ael being called (or loaded)
fputs(stdout,what_line(2));      // prints 1
}

funct();
   // refers to x.ael being called (or loaded)
fputs(stdout,what_line());      // prints 1
```

Chapter 11: Design Environment Query Functions

This chapter describes each Design Environment Query function in detail. The functions are listed in alphabetical order.

db_factor()

Returns a real value, a conversion factor to convert a value from simulator units to layout user units for the current window.

Syntax:

db_factor();

Example:

```
decl conFact, uUnit;
conFact = db_factor();
uUnit = 20*conFact;
```

de_ang_factor()

Returns a real number, a conversion factor to convert a value from simulator units to degrees.

Syntax:

ang_factor();

Example:

```
decl cfact, degree_angle;
cfact = de_ang_factor();
degree_angle = cfact * angle;
```

de_current_design_name()

Returns a string, the name of the current design (the design opened in the active window). The active window is set with *api_set_current_window()* or *api_set_current_window_by_seq_num()*. Returns NULL if if the current design does not exist.

Syntax:

de_current_design_name();

Example:

```
decl designName;
de_set_window(SCHEM_WIN);
open_design("abc", 1);
designName = de_current_design_name();
```

de_current_design_type()

Returns the current design type (the design opened in the active window). The active window set with *de_set_window()*. Design types are defined in the simulator ael definition files *stdcmds.ael* and *stddefs.ael*.

Syntax:

de_current_design_type();

Example:

```
decl b;
b = de_current_design_type();
```

de_get_design_instances()

Returns a list of instance names of a given element, belonging to the named design.

Syntax:

de_get_design_instances(designName[, elementName, attrib, stringList, separator]);

where

designName is optional; the design name.

elementName is optional; the element name (e.g., CAP). NULL if not used.

attrib is optional; the element attribute to search for. Values are the attributes used in *create_item()*. See “create_item() Attribute Choices” on page 15-9. NULL if not used.

stringList is optional; list of names to which new names are appended.

separator is optional; the separator inserted between the components. NULL if not used.

Example:

```
inst_list = de_get_design_instances(de_current_design_name());
inst_name = car(inst_list);
// e.g. inst_list=list("R1", "R2")
```

de_get_file_names()

Returns a list of files with given file extension. Used in conjunction with *create_text_form()*.

Syntax:

de_get_file_names(path, fileExt);

where

path is a string containing a list of colon-separated or semi-colon separated directories where files may be located. In most cases this will be obtained from configuration environment, using *getenv()*.

fileExt is the file extension (without the dot “.”).

Example:

```
decl filenames;
filenames = de_get_file_names(getenv("HPTOLEMY_TEMPLATE_PATH"), "tpl");
```

de_get_variable_names()

Returns a list of all variable names declared in a design.

Syntax:

de_get_variable_names(designName);

where

designName is the name of the design.

Example:

```
decl varNames;
varNames = de_get_variable_names("amp");
while(listlen(varNames) > 0)
{
   fputs(car(varNames));
   varNames=cdr(varNames);
}
```

de_get_variable_value()

For a given design name and variable name, returns the value of the variable.

Syntax:

de_get_variable_value(designName, varName);

where

designName is the name of the design.

varName is the name of the variable.

Example:

```
decl varValue;
varValue = de_get_variable_value("amp", "myVariable");
de_info(strcat("myVariable= ", varValue), 0);
```

de_get_window()

Returns the currently active window, where: MAIN_WIN = Main window, SCHEM_WIN = Schematic window, and LAYOUT_WIN = Layout window.

Syntax:

de_get_window();

Example:

```
decl win;
win = de_get_window();
if (win==SCHEM_WIN)
   fputs(stderr, "It's in schematic");
```

de_retrieve_version_info()

Returns a long string of product version information for the Design Environment. The first two lines relate to Schematic Capture version and license information.

See also: *de_version_number_int()*

Syntax:

de_retrieve_version_info()

Example:

```
decl versionInfo;
   versionInfo=de_retrieve_version_info();
   fputs(stderr, versionInfo);
   // will print out:
   // Design Environment (*) 170.day Aug 13 2001
```

```
   // FEATURE Schematic hpeesofd 1.700 1-oct-2001 0 4C7ADBC38E671FCAEEA3
"CYWBCUX RHSVSOUEYGOKE" b5688f44
```

de_version_number_int()

Returns the current version number as an integer. For example, version 1.9 is returned as 190.

See also: *de_retrieve_version_info()*

Syntax:

de_version_number_int()

Example:

One way to see the contents of each string is to execute the following function call in the Options > Command_Line window:

```
fputs(stderr, de_version_number_int());
```

which prints the returned string to the command window where ADS was started.

de_window_is_open()

Determines the status of a window. Returns the status of window, where: TRUE = window is open, FALSE = window is not open.

Syntax:

de_window_is_open(windowType);

where

windowType is the type of window where:

MAIN_WIN = Main window
SCHEM_WIN = Schematic window
LAYOUT_WIN = Layout window

Example:

```
if (de_window_is_open(SCHEM_WIN))
    fputs(stderr,"SCHEMATIC is open");
else
    fputs(stderr,"no SCHEMATIC window is open");
```

get_eqn_list()

Returns a list of variables and equations for the current design. The list is sorted in alphabetical order.

Syntax:

get_eqn_list();

Example:

```
decl vareqnList;
vareqnList = get_eqn_list()
while (listlen(vareqnList) > 0)
{
   fputs(car(vareqnList));
   vareqnList=cdr(vareqnList);
}
```

get_item_list()

Returns a string list of all components (instance names) matching a component name placed in the current design.

Syntax:

get_item_list(itemName);

where

itemName is the name of the component.

Example:

```
decl names;
names = get_item_list("R");
```

get_measurement_list()

Returns a string list of all measurement components active for the current design.

Syntax:

get_measurement_list();

Example:

```
decl mNames;
mNames = get_measurement_list();
```

get_parameter_names()

Any design may have an optional list of parameters associated with it. Returns a string list of parameter names for the current design.

Syntax:

get_parameter_names(designName[, stringList]);

where

designName is the name of the design whose parameters are to be returned.

stringList is optional; a string list, the parameters are appended to this list.

Example:

```
decl names;
names = get_parameter_names(current_design_name(), string_list);
```

get_push_history()

Returns a string list containing design push history of a given design. This is the list of designs that have been pushed into with the *de_push()* command.

Syntax:

get_push_history(designName, repType, separator, endDesign);

where

designName is the name of the design.

repType is the type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

separator is the separator character to insert between design names.

endDesign is optional. Terminates design when reached no other designs are listed.

Example:

```
out = get_push_history(current_design_name(), REP_SCHEM, "\\", NULL);
```

unit_name()

Returns a string; the unit name, given its code.

Syntax:

unit_name(unitCode);

where:

unitCode is a string, the unit name; one of these choices:

STRING_UNIT returns "string"
UNITLESS_UNIT returns "num"
FREQUENCY_UNIT returns "freq"
RESISTANCE_UNIT returns "res"
CONDUCTANCE_UNIT returns "cond"
INDUCTANCE_UNIT returns "ind"
CAPACITANCE_UNIT returns "cap"
LENGTH_UNIT returns "lng"
TIME_UNIT returns "time"
ANGLE_UNIT returns "ang"
POWER_UNIT returns "power"
VOLTAGE_UNIT returns "volt"
CURRENT_UNIT returns "cur"
DISTANCE_UNIT returns "dist"

Example:

```
decl unitStrg;
unitStrg = unit_name(FREQUENCY_UNIT);   // unitStrg="freq"
```

Chapter 12: Command Functions

This chapter describes each Command function in detail. The functions are listed in alphabetical order.

de_activate()

Activates an instance by setting the deactivate flag of an instance to false. Deactivated instances (which are commented out) are not simulated and are displayed with a box with an x-mark through them. Returns: none.

See also: *de_deactivate()*.

Syntax:

de_activate([x,y]);

where

x,y is optional. Coordinates within the select region of an instance to activate. If not given, activates previously-selected instances.

Example:

```
de_activate (10,20);
```

or

```
de_activate();
```

de_add_arc()

Adds a clockwise or counterclockwise, circular arc to a polygon or polyline in user units in the current representation. Before using an arc, you must define the starting point of a polygon or polyline. Returns: none.

See also: *de_add_arc1(), de_add_arc2(), de_add_arc3(), de_add_arc4()*.

Syntax:

de_add_arc(x,y, angle);

where

x,y is the center point of the arc; this becomes a vertex in the polygon or polyline.

angle is the sweep angle of the arc, where:

negative = clockwise
positive = counterclockwise

Example:

The example creates a polygon with a clockwise and counterclockwise arc.

```
de_add_polygon();
de_add_point(3.25,4.75);
de_add_point(4.125,5.625);
de_add_point(5.25,4.75);
de_add_point(6.25,4.75);
de_add_arc(7.125,4.75,180.0);
de_add_point(8.75,4.75);
de_add_point(8.75,3.875);
de_add_arc(8.75,3.125,-180.0);
de_add_point(3.75,3.125);
de_add_point(3.75,3.125);
de_end();
```

de_add_arc1()

Adds an arc using user-defined units.

- To create an arc using simulator units, use the *de_draw_arc1()* function.
- To create an arc embedded in a polygon or polyline, use the *de_add_arc()* function.

Returns: none.

See also: *de_draw_arc1()*, *de_add_arc()*.

Syntax:

de_add_arc1(x1, y1, x2, y2, x3, y3);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the circumference point of the arc.

x3, *y3* is the end point of the arc.

Example:

```
// A 180 degree clockwise standalone arc
de_add_arc1 (0.0, 100.0, -50.0, 50.0, 0.0, 0.0);

// A 180 degree counterclockwise standalone arc
```

```
de_add_arc1 (200.0, 0.0, 250.0, 50.0, 200.0, 100.0);
```

de_add_arc2()

Adds an arc using user-defined units.

- To create an arc using simulator units, use the *de_draw_arc2()* function.
- To create an arc embedded in a polygon or polyline, use the *de_add_arc()* function.

Returns: none.

See also: *de_draw_arc2()*, *de_add_arc()*.

Syntax:

de_add_arc2(x1, y1, x2, y2, x3, y3, direction[, thickness]);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the center point of the arc.

x3, *y3* is the end point of the arc.

direction is the direction of arc, where:

1 = clockwise
0 = counter clockwise

thickness is optional, where:

1 = thin, default
2 = medium
3 = thick

Example:

```
// A 180 degree clockwise standalone arc
de_add_arc2 (0.0, 0.0, 0.0, 50.0, 0.0, 100.0, 1);

// A 180 degree counter clockwise standalone arc
de_add_arc2 (200.0, 0.0, 200.0, 50.0, 200.0, 100.0, 0);
```

de_add_arc3()

Adds an arc using user-defined units.

- To create an arc using simulator units, use the *de_draw_arc3()* function.
- To create an arc embedded in a polygon or polyline, use the *de_add_arc()* function.

Returns: none.

See also: *de_draw_arc3()*, *de_add_arc()*.

Syntax:

de_add_arc3(x1, y1, x2, y2, sweepAngle, direction);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the center point of the arc.

sweepAngle is the arc angle in degrees.

direction is the direction of arc, where:

1 = clockwise
0 = counter clockwise

Example:

```
// A 180 degree clockwise stand alone arc
de_add_arc3 (0.0, 0.0, 0.0, 50.0, 180.0, 1);

// A 180 degree counter clockwise stand alone arc
de_add_arc3 (200.0, 0.0, 200.0, 50.0, 180.0, 0);
```

de_add_arc4()

Adds an arc using user-defined units.

- To create an arc using simulator units, use the *de_draw_arc4()* function.
- To create an arc embedded in a polygon or polyline, use the *de_add_arc()* function.

Returns: none.

See also: *de_draw_arc4()*, *de_add_arc()*.

Syntax:

de_add_arc4(x1, y1, x2, y2, chordLength, direction);

where

x1, y1 is the start point of the arc.

x2, y2 is the center point of the arc.

direction is the direction of arc, where:

1 = clockwise
0 = counter clockwise

chordLength is the arc angle expressed in terms of circumference.

Example:

```
// A 180 degree clockwise standalone arc
de_add_arc4 (0.0, 0.0, 0.0, 50.0, 157.08, 1);

// A 180 degree counter clockwise standalone arc
de_add_arc4 (200.0, 0.0, 200.0, 50.0, 157.08, 0);
```

de_add_circle()

Draws a circle in the current representation in user-defined units. To use simulator units, use the *de_draw_circ()* function. The *de_draw_circ()* function is used in artwork creation functions. Returns: none.

Syntax:

de_add_circle(x,y, radius[, thickness]);

where

x,y is the coordinates for the circle's center point.

radius is the radius of the circle.

thickness is optional, where:

1 = thin, default
2 = medium
3 = thick

Example:

```
set_window(1);
de_add_circle(10.0,10.0, 25.5);
```

See also: *de_add_arc()*, *de_add_arc1()*, *de_add_arc2()*, *de_add_arc3()*, *de_add_arc4()*.

de_add_construction_line()

Adds a construction line using user-defined units. The angle of a construction line is defined by any two distinct points. Construction lines have no bounding box and extend out to infinity. Returns: none.

Syntax:

de_add_construction_line(x1, y1, x2, y2);

where

x1, *y1* is the first point that defines the line.

x2, *y2* is the second point that defines the line.

Example:

The example creates two construction lines that intersect at 0,0.

```
de_add_construction_line(0, 0, 100, 0);
de_add_construction_line(0, 0, 0, 100);
```

de_add_path()

Starts a path command sequence. Adds a path to the current representation. Returns: none.

See also: *de_add_trace()*, *de_add_point()*, *de_end()*.

Syntax:

de_add_path();

Example:

```
de_add_path();
de_add_point(10, 20);
de_add_point(30, 40);
de_end();
```

de_add_point()

Adds a point to a polygon, polyline, or path using user-defined units. To draw a shape using simulator units, use the function *de_draw_point()*. The *de_draw_point()* function is typically used in artwork generation functions. Returns: none.

See also: *de_add_polyline()*, *de_add_polygon()*, *de_add_path()*.

Syntax:

de_add_point(x,y);

where

x,y is the coordinates, in user-defined units, for the new vertex.

Example:

```
//draws a polygon in the Schematic window
set_window(1);
de_add_polygon();
de_add_point(0.0, 5.625);
de_add_point(1.25, 6.125);
de_add_point(1.375, 5.0);
de_add_point(-0.125, 5.125);
de_add_point(-0.125, 5.125);
de_end();
```

de_add_polygon()

Starts a polygon command sequence. Adds a polygon to the current representation. Returns: none.

See also: *de_add_rectangle()*, *de_add_circle()*, *de_add_arc()*, *de_add_point()*, *de_end()*, *de_add_polyline()*.

Syntax:

de_add_polygon();

Example:

```
de_add_polygon();
de_add_point(0,0);
de_add_point(20,0);
de_add_point(20,40);
de_add_point(0,0);
de_end();
```

de_add_polyline()

Starts a polyline command sequence. Adds a polyline to the current representation. Returns: none.

See also: *de_add_polygon()*, *de_add_rectangle()*, *de_add_circle()*, *de_add_arc()*, *de_add_point()*, *de_end()*.

Syntax:

de_add_polyline();

Example:

```
de_add_polyline();
de_add_point(0,0);
de_add_point(10,20);
de_end();
```

de_add_property()

Adds property to data group, port, or instance. Returns: none.

See also: *de_remove_properties()*, *de_set_edit_property()*.

Syntax:

de_add_property(propName, propValue, editSelected);

where

propName is the name of property to be added.

propValue is the value of property to be added.

editSelected is the mode for selecting components to add, where:

TRUE = edit only selected components
FALSE = add to component selected by de_set_edit_property

Example:

```
de_add_property ("Prop1",5,FALSE);
```

de_add_rectangle()

Adds a rectangle to the current representation. Returns: none.

See also: *de_add_polygon()*, *de_add_circle()*.

Syntax:

de_add_rectangle(x1,y1, x2,y2[, thickness]);

where

x1,y1 is the first corner describing window.

x2,y2 is the second (opposite) selection corner.

thickness is optional, where:

1 = thin, default
2 = medium
3 = thick

Example:

```
de_add_rectangle(10,20, 30,40);
```

de_add_text()

Adds a text string to the current representation. The string can be set with the function *de_set_text_string()* is passed into this function; the text attributes are set with the functions *de_set_text_height(), de_set_text_font(), de_set_text_angle()*. Returns: none.

See also: *de_set_text_string()*.

Syntax:

de_add_text(x,y[, text]);

where

x,y is the point for the lower-left corner of first character in the string.

text is a text string.

Example:

```
de_set_text_string("This is a text string");
de_add_text(10,12);
de_add_text (13, 14);
```

or

```
de_add_text(10. 12, "This is a text string");
```

de_add_trace()

Starts trace command sequence for adding a trace to the current representation. This command is followed by two or more *de_add_point()* commands to describe the trace vertices and the *de_end()* command to terminate the trace. Traces must start and end at a pin or another trace. Returns: none.

See also: *de_add_point(), de_end(), de_add_path()*.

Syntax:

de_add_trace();

Example:

```
de_add_trace();
de_add_wire(10,10);
de_add_wire(30, 10);
de_end();
```

de_add_vertex()

Adds a vertex to an existing polygon, polyline, wire or trace in the current representation. Returns: none.

Syntax:

de_add_vertex(x,y, nx, ny);

where

x,y is the point between two existing vertices, between which a new vertex is added.

nx, ny is the coordinates for a new vertex.

Example:

```
de_add_vertex(5.0, 9.0, 5.0, 7.9);
```

de_add_wire()

Adds a wire vertex to a wire or trace connection in the current representation. Wires and traces must begin and end at another wire, trace or pin. Returns: none.

See also: *de_connect()*.

Syntax:

de_add_wire(x,y);

where

x,y is the coordinates of the wire vertex in user units.

Example:

```
de_connect();
de_add_wire(-0.25, 3.125);
de_add_wire(1.25, 3.25);
```

```
de_end();
```

de_add_wire_label()

Adds a label to a wire or pin at the x,y location. Any pins or wires with the same name are connected for simulations. If the label is in proper bus syntax, the wire or pin will be iterated appropriately.

Syntax:

de_add_wire_label(x, y, label);

where

x is the x coordinate of the label location

y is the y coordinate of the label location

label is label to give a wire

Example:

```
de_add_wire_label(1.0, 1.0, "A<0>");
```

de_archive_project()

Archives the given project in to a single file to be used for moving to another file system, onto disk, or for e-mailing. Returns: none.

Syntax:

de_archive_project(fromProjectName, toFileName, archiveHierarchy);

where

fromProjectName is the name of the project to archive. Should be a full path name.

toFileName is the name to save the archive as. Should be full path name.

archiveHierarchy is the choice for what to archive, where:

TRUE = archive the entire project hierarchy (if it exists)
FALSE = (default) archive the given project

Example:

```
de_archive_project("C:\users\default\MyAmps_prj",
"C:\users\temp\MyAmps_prj.zap", TRUE);
```

de_bom()

Generates a bill of material file for the current representation. Returns: none.

See also: *de_parts()*, *de_net()*.

Syntax:

de_bom(fileName, showDlg);

where

fileName is the name of file to store Bill of Materials list.

showDlg is whether or not to show the Bill of Materials list, where:

TRUE is show the list.
FALSE is do not show the list (it still records to the file).

Example:

```
de_bom ("my BOMList", FALSE);
```

de_boolean_logical()

Performs boolean operations on shapes placed on different layers in a layout presentation. Returns: none.

Syntax:

de_boolean_logical(operator, inLayer1, inLayer2, outLayer, rmLayer1, rmLayer2, applyToSelect[, winInstH]);

where

operator is a string that specifies the boolean operation to be performed. The choices are "AND", "OR", "XOR", and "DIFF".

inLayer1 and *inLayer2* are integers that specify the layers containing the shapes to perform the boolean operation on.

outLayer is an integer that specifies the layer to contain the results of the boolean operation.

rmLayer1 and *rmLayer2* are boolean flags that specify whether the input shapes are to be deleted or not. Choices are:

TRUE = delete the shapes.
FALSE = do not delete the shapes.

applyToSelect is a boolean flag that specifies whether the boolean operation is only to be performed on the input layers that have been selected. Choices are:

TRUE = use only those shapes in the operation that have been selected in the input layers.

FALSE = use all shapes in the input layers.

winInstH is an optional parameter. It is the handle to the window containing the layout representation to use for the boolean operation. If unspecified, the current window is used.

Example:

```
de_set_layer(1);
de_add_rectangle(-200, -50, 0, 50);
de_set_layer(2);
de_add_circle(0, 0, 50);
de_boolean_logical("AND", 1, 2, 3, TRUE, TRUE, FALSE);
```

de_break_connection()

Breaks connection to attached wire or trace from the selected instances in the current representation. Returns: none.

See also: *de_move_break()*, *de_select_range()*.

Syntax:

de_break_connection();

Example:

```
de_break_connection();
```

de_change_annotation_layer()

Changes the layer of the nearest annotation (within the select region) to the current layer. Returns: none.

See also: *de_edit_annotation_attribute()*.

Syntax:

de_change_annotation_layer(x,y);

where

x,*y* is the coordinates within the select region of an instance's annotation.

Example:

```
de_set_layer(5);
de_change_annotation_layer(10.4,3.4);
```

de_change_units()

Changes the units and precision of the current design and resets default design units. Returns: none.

Syntax:

de_change_units(units, prec, incr);

where

units is an integer, 1-7, where:

1 = mil
2 = inches
3 = microns
4 = mm
5 = cm
6 = meter
7 = feet

prec sets the precision of the design, within the range of 1 - 10. This is the exponent of 10 representing the number of data base units per user units. For example: 1000 data base units to a user unit is represented as precision = 3.

incr sets the increment of the design as a multiple of data base to user units. For example: 2000 data base units to a user unit is set with precision = 3, increment = 2. Data base units to a user unit = increment • $10^{(\text{precision})}$.

Example:

```
de_change_units(2, 3, 1);
//change data base units per inch to 1000
```

de_check_rep_options ()

Generates a check representation report. Checks for unconnected pins, nodal mismatch (layout vs. schematic), wires in layout (Layout only), component pin vs. symbol port mismatches (Schematic only), representation port vs. symbol port mismatches (Schematic only), and overlaid items. Returns: none.

Syntax:

de_check_rep_options(dispMode);

where

dispMode is the sum of desired checks, as one of the following modes:

DEINFO_DISP_CHECK_NONE
DEINFO_DISP_CHECK_UNCONNECTEDPIN
DEINFO_DISP_CHECK_NODALMISMATCH
DEINFO_DISP_CHECK_WIRELAYOUT (Layout only)
DEINFO_DISP_CHECK_PINVSPORT (Schematic only)
DEINFO_DISP_CHECK_PORTVSPORT (Schematic only)
DEINFO_DISP_CHECK_OVERLAIDITEMS
DEINFO_DISP_CHECK_OVERLAPWIRE

Example:

```
dispMODE=DEINFO_DISP_CHECK_UNCONNECTEDPIN
    +DEINFO_DISP_CHECK_NODALMISMATCH;
de_check_rep_options(dispMODE);
```

de_clear_dc_annotation()

Removes the annotated DC node voltages and branch currents on the current schematic.

See also: *de_dc_annotation()*

Syntax:

de_clear_dc_annotation();

Example:

```
de_clear_dc_annotation();
```

de_clear_highlighting()

Clears any highlighting from a design representation in a current window. Returns: none.

Syntax:

de_clear_highlighting();

Example:

```
de_clear_highlighting();
```

de_clear_rep()

Clears all data from the current design representation. This is equivalent to doing a *de_select_all()* and *de_delete()*. Returns: none.

Syntax:

de_clear_rep();

Example:

```
set_window(SCHEM_WIN);
de_clear_rep();
```

de_clear_show_connected()

Clears the graphics associated with graphically unconnected pins. Returns: none.

See also: *de_show_connected()*.

Syntax:

de_clear_show_connected();

Example:

```
de_clear_show_connected();
```

de_close_all()

Closes all designs from memory. Closes all windows. Does not prompt for unsaved changes. Returns: none.

Syntax:

de_close_all();

Example:

```
de_close_all();
```

de_close_design()

Closes the current design from memory and the user interface. Returns the handle of the next design.

Syntax:

de_close_design([dsnName]);

where

dsnName is optional. Name of design to clear. If not specified, closes current design.

Example:

```
newDsnH = de_close_design("Mydesign");
```

de_close_window()

Closes the current window instance. Returns: none.

See also: *de_create_window()*, *api_set_current_window_by_seq_num()*.

Syntax:

de_close_window();

Example:

```
// Closes the current window instance
api_set_current_window_by_seq_num (1);
de_close_window();
```

de_config_window()

Allow windows to be configured through AEL. This function can be used in conjunction with: *de_open_window()*, *de_close_window()*, *de_window_is_open()*. Returns: none.

See also: *api_set_current_window_by_seq_num()*, *api_set_window_instance_rect()*, *de_create_window()*.

de_config_window(windowType, xLoc, yLoc, width, height);

where

windowType is the type of window, where:

MAIN_WIN = Main window
SCHEM_WIN = Schematic window
LAYOUT_WIN = Layout window

xLoc, *yLoc* is the inital position for the window.

width is the width of window in user units.

height is the height of window in user units.

Example:

```
de_config_window(SCHEM_WIN, 150, 180, 300, 200, 1);
```

de_connect()

Starts a wire or trace connection. This command is followed by two or more calls to *de_add_wire()* and is terminated with the *de_end()* function. Returns: none.

Syntax:

de_connect([route]);

where

route is optional. Specifies choice for autoroute.

TRUE = use autoroute
FALSE = do not use autoroute; default value

Example:

```
de_connect();
de_add_wire(-0.25,3.125);
de_add_wire(1.25,3.25);
de_end();
```

de_convert_path_to_trace()

Converts selected paths to traces. Returns: none.

See also: *de_convert_trace_to_path(), de_edit_path_trace().*

Syntax:

de_convert_path_to_trace();

Example:

```
de_convert_path_to_trace()
```

de_convert_to_polygon()

Converts selected closed shapes to simple, single-segment polygons. Circles, rectangles and complex polygons (polygons with holes or arcs) are converted to simple, single-segment polygons. Returns: none.

Syntax:

de_convert_to_polygon([x,y]);

where

x,y is optional. Point within select region of object to convert.

Example:

```
de_convert_to_polygon();
```

or

```
de_convert_to_polygon(10,20);
```

de_convert_trace_to_path()

Converts selected traces to paths. Returns: none.

See also: *de_convert_path_to_trace()*, *de_edit_path_trace()*.

Syntax:

de_convert_trace_to_path();

Example:

```
de_convert_trace_to_path();
```

de_convert_traces_to_instances()

Converts selected traces to transmission line elements. The elements used are set using *de_set_trace()* commands. Returns: none.

Syntax:

de_convert_traces_to_instances();

Example:

```
de_set_trace_msub_id("TL4");
de_convert_traces_to_instances();
```

de_copy()

Copies the selected components in the current representation and places them at delta from the original position. Returns: none.

See also: *de_copy_to_buffer()*.

Syntax:

de_copy(dx,dy);

where

dx,dy is the delta from original position.

Example:

```
de_copy(15.9,17);
```

de_copy_design()

Copies the given design file to a new file. Copies the .dsn and .ael files. Also copies mom_dsn design directory, if one exists. Returns: none.

Syntax:

de_copy_design(fromName, toName, cpyHierarchy);

where

fromName is the name of design file to copy; should be full path name.

toName is the new design name; should be full path name.

copyHierarchy indicates the items to be copied, where:

TRUE = if entire project hierarchy is to be copied
FALSE (default) = if only specified project is to be copied

Example:

```
de_copy_design("c:\myamp", "c:\youramp");
```

de_copy_project()

Copies a complete project directory, with all associated sub-directories and files. Returns: none.

Syntax:

de_copy_project(fromName, toName, copyHierarchy, openProjectAfterCopy);

where

fromName is the name of the project to copy; should be full path name.

toName is the name of the new project; should be full path name.

copyHierarchy indicates the items to be copied, where:

TRUE = if entire project hierarchy is to be copied
FALSE (default) = if only specified project is to be copied

openProjectAfterCopy indicates whether or not to open the newly-copied project, where:

TRUE = open the project after successful copy
FALSE (default) = do not open the project

Example:

```
de_copy_project("c:\myprj", "c:\yourPrj", FALSE, TRUE);
```

de_copy_to_buffer()

Copies the selected group in the current representation to the copy buffer. Returns: none.

Syntax:

de_copy_to_buffer([x,y]);

where

x,y is optional. Coordinates used as the paste origin reference point when the Paste From Buffer command is executed. If no (x,y) value is specified, the paste origin defaults to the first unconnected pin or lower left corner of bounding box.

Example:

```
de_select_all();
de_copy_to_buffer();
de_paste_from_buffer(20.0,30.9);
```

de_copy_to_layer()

Copies the selected group in the current representation to the current active layer. Returns: none.

See also: *de_move_to_layer()*.

Syntax:

de_copy_to_layer([layerNo]);

where

layerNo is optional; layer to copy selected objects to. If layer not specified, object is copied to current layer.

Example:

```
de_select_all();
de_set_layer(4);
de_copy_to_layer();
```

de_create_window()

Creates and opens a window. Returns a NULL value.

See also: *de_close_window()*.

de_create_window(window_code[, winInstP, xLoc, yLoc, width, height]);

where

window_code is the code for type of window; one of the following:

SCHEM_WIN = Schematic window
LAYOUT_WIN = Layout window

winInstP is optional. NULL if no parent window; otherwise, winInst of parent window with design to show in newly-created window. The cfg env vars are:

SCHEMATIC/LAYOUT_WINDOW_X_LOC
SCHEMATIC/LAYOUT_WINDOW_Y_LOC
SCHEMATIC/LAYOUT_WINDOW_HEIGHT
SCHEMATIC/LAYOUT_WINDOW_WIDTH

If you don't specify the x, y, width, and height, then:

- If there are no windows open, then open the window with the coordinates, width, and height specified by the cfg env vars.
- If there are windows open, then instead of using the coordinates specified by the cfg env vars, open the window to the lower right of the most-recently opened window, but still use the width and height specified by the cfg env vars.
- If you are reattaching to a project which had windows open before you reattached to it (that is, save project state), then restore the windows, to the coordinates, width and height, that they had before.

xLoc, yLoc is optional. Initial position for the window.

width is optional. Width of window in user units.

height is optional. Height of window in user units.

Example:

```
// sets the layout window active and creates a rectangle in it.
de_create_window(LAYOUT_WIN);
rectangle (10.3,0,14.5,11);
```

de_dc_annotation()

Display DC node voltages and branch currents on the current schematic after a DC or Transient simulation.

See also: *de_clear_dc_annotation()*

Syntax:

de_dc_annotation();

Example:

```
de_dc_annotation;
```

de_deactivate()

Sets instance's deactivate flag to true (makes the instance deactivated). Deactivated instances are not simulated (they are commented out) and are displayed with a box and x-mark over them. Returns: none.

See also: *de_activate()*.

Syntax:

de_deactivate([x,y]);

where

x,y is optional. Coordinates within the select region of an instance to deactivate. If not given, deactivates selected instances.

Example:

```
de_deactivate(10,20);
```

de_define_npport()

Places a non-preferred port/pin in an artwork instance. This is used in layouts with multiple connection points per pin to define connection points. The preferred

connection point is used by design synchronization to connect to by default. Returns: none.

Syntax:

de_define_npport(x,y, angle, portNo, pinName);

where

x,y is the location of the port (in user-defined units).

angle is the angle of the port (used to determine abutting interconnected instance).

portNo is the port/pin number.

pinName is optional. The pin/part name; a string.

Example:

```
de_define_npport(10.2, 14.5, 90.0, 2, "input_sig");
```

de_define_port()

Places a single port/pin, with a unique pin number, in an artwork instance using user units. To specify location and angle in simulator units, use *de_draw_port()*. Returns: none.

Syntax:

de_define_port(x,y, angle, portNo, pinName[, pinPower]);

where

x,y is the port/pin location (in user-defined units).

angle is the angle of the port (used to determine abutting interconnected instance).

portNo is the unique port number.

pinName is optional; name of the pin; a string.

pinPower is optional; power of the pin.

Example:

```
de_define_port(10, 20, 0, 4,"input_sig");
```

de_delete()

Deletes selected objects, or object near given point in current representation. Returns: none.

Syntax:

de_delete([x,y]);

where

x,y is optional. Point within select region of object to delete.

Example:

```
de_delete();
```

or

```
de_delete(10.0, 34.68);
```

de_delete_all_orphaned_instances()

Deletes all the orphaned instances in the current representation. Returns: none.

Syntax:

de_delete_all_orphaned_instances();

Example:

```
de_delete_all_orphaned_instances();
```

de_delete_design()

Deletes given designs from disk. Deletes the .dsn, .ael, .atf , and related files. Returns: none.

Syntax:

de_delete_design(designName);

where

designName is the name of the design to delete; should be a full path name.

Example:

```
de_delete_design("c:\myproj\networks\mydesign.dsn");
```

de_delete_project()

Deletes given project, including the project directory and all of its contents. Returns: none.

Syntax:

de_delete_project(projectName);

where

projectName is the name of the project directory.

Example:

```
de_delete_project("c:\myproject");
```

de_delete_view()

Deletes a stored view with the matching name of view. Returns: none.

See also: *de_store_current_view()*, *de_restore_view()*.

Syntax:

de_delete_view(viewName);

where

viewName is the name of view to delete.

Example:

```
de_delete_view("myView");
```

de_deselect_all()

Deselects all objects in the current representation. Returns: none.

Syntax:

de_deselect_all();

Example:

```
de_deselect_all();
```

de_deselect_all_force()

Deselects all objects regardless of layer protection status. Returns: none.

Syntax:

de_deselect_all_force();

Example:

```
de_deselect_all_force();
```

de_deselect_by_name()

Deselects instances by instance name or ID. Returns: none.

Syntax:

de_deselect_by_name(instName, selectType);

where

instName is the instance name or ID.

selectType is the selection type code, where:

0 = selects by instance name
1 = selects by instance ID

Example:

```
de_deselect_by_name("MLIN",0);
```

or

```
de_deselect_by_name("TL1", 1);
```

de_deselect_window()

Deselects all objects in given window in current representation. Returns: none.

Syntax:

de_deselect_window(x1,y1, x2,y2);

where

x1,y1 is the first corner describing window.

x2,y2 is the second (opposite) selection corner.

Example:

```
de_deselect_window(10,15, 22,87);
```

de_draw_arc()

Adds a circular arc to a polygon or polyline in simulator units. Typically, this is used in artwork creation macros rather than in the *de_add_arc()* function. Returns: none.

See also: *de_draw_arc1(), de_draw_arc2(), de_draw_arc3(), de_draw_arc4()*.

Syntax:

de_draw_arc(x,y, angle);

where

x,y is the center of the arc.

angle is the sweep angle of the arc.

Negative = clockwise
Positive = counterclockwise

Example:

```
de_add_polyline();
de_draw_point(0,0);
de_draw_arc(20,0,90.0);
de_draw_point(40,60);
de_end();
```

de_draw_arc1()

Adds an arc using simulator units. Typically, used in artwork creation macros.

- To create an arc using user-defined units, use the *de_add_arc1()* function.
- To create an arc embedded in a polygon or polyline, use the *de_draw_arc()* function.

Returns: none.

See also: *de_add_arc1(), de_draw_arc()*.

Syntax:

de_draw_arc1(x1, y1, x2, y2, x3, y3);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the circumference point of the arc.

x3, *y3* is the end point of the arc.

Example:

```
// A 180 degree clockwise standalone arc
de_draw_arc1(0, 100, -50, 50, 0, 0);
```

```
// A 180 degree counterclockwise standalone arc
de_draw_arc1(200, 0, 250, 50, 200, 100);
```

de_draw_arc2()

Adds an arc using simulator units. Typically, used in artwork creation macros.

- To create an arc using user-defined units, use the *de_add_arc2()* function.
- To create an arc embedded in a polygon or polyline, use the *de_draw_arc()* function.

Returns: none.

See also: *de_add_arc2()*, *de_draw_arc()*.

Syntax:

de_draw_arc2(x1, y1, x2, y2, x3, y3, direction);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the center point of the arc.

x3, *y3* is the end point of the arc.

direction is the arc direction code, where:

1 = clockwise
0 = counter clockwise

Example:

```
// A 180 degree clockwise standalone arc
de_draw_arc2 (0.0, 0.0, 0.0, 50.0, 0.0, 100.0, 1);

// A 180 degree counter clockwise standalone arc
de_draw_arc2 (200.0, 0.0, 200.0, 50.0, 200.0, 100.0, 0);
```

de_draw_arc3()

Adds an arc using simulator units. Typically, used in artwork creation macros.

- To create an arc using user-defined units, use the *de_add_arc3()* function.
- To create an arc embedded in a polygon or polyline, use the *de_draw_arc()* function.

Returns: none.

See also: *de_add_arc3()*, *de_draw_arc()*.

Syntax:

de_draw_arc3(x1, y1, x2, y2, sweepAngle, direction);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the center point of the arc.

sweepAngle is the arc angle in degrees.

direction is the arc direction code, where:

1 = clockwise
0 = counter clockwise

Example:

```
// A 180 degree clockwise stand alone arc
de_draw_arc3 (0.0, 0.0, 0.0, 50.0, 180.0, 1);

// A 180 degree counter clockwise stand alone arc
de_draw_arc3 (200.0, 0.0, 200.0, 50.0, 180.0, 0);
```

de_draw_arc4()

Adds an arc using simulator units. Typically, used in artwork creation macros.

- To create an arc using user-defined units, use the *de_add_arc4()* function.
- To create an arc embedded in a polygon or polyline, use the *de_draw_arc()* function.

Returns: none.

See also: *de_add_arc4()*, *de_draw_arc()*.

Syntax:

de_draw_arc4(x1, y1, x2, y2, chordLength, direction);

where

x1, *y1* is the start point of the arc.

x2, *y2* is the center point of the arc.

chordLength is the arc angle expressed in terms of circumference.

direction is the arc direction code, where:

1 = clockwise
0 = counterclockwise

Example:

```
// A 180 degree clockwise standalone arc
de_draw_arc4 (0.0, 0.0, 0.0, 50.0, 157.08, 1);

// A 180 degree counter clockwise standalone arc
de_draw_arc4 (200.0, 0.0, 200.0, 50.0, 157.08, 0);
```

de_draw_circ()

Draws a circle in simulator units. Typically, used in artwork creation macros rather than in the *de_add_circle()* function. Returns: none.

Syntax:

de_draw_circ(x,y,radius);

where

x,y is the center of the circle.

radius is the radius of the circle.

Example:

```
de_draw_circ(10,45, 36.7);
```

de_draw_point()

Adds a point to a polygon or polyline to the current representation using simulator units. The polygon or polyline is started with the *de_add_polygon()* or *de_add_polyline()* command and it must be terminated with an *de_end()*. Typically, used in artwork creation macros rather than in the *de_add_point()* function. Returns: none.

Syntax:

de_draw_point(x,y);

where

x,y is the coordinates of the point to add.

Example:

```
de_add_polyline();
de_draw_point(0,0);
de_draw_point(25,56);
de_end();
```

de_draw_port()

Adds a port to an artwork instance. Returns: none.

Syntax:

de_draw_port(x, y, angle[, power, portNumber, portName]);

where

x, y is the port location.

angle is the port angle; used to determine the angle of a connecting instance.

power is optional; power of pin.

portNumber is optional; next available unique port number used if not supplied.

portName is optional; unique port name.

Example:

```
de_draw_port(0, 0, -90);
```

de_draw_rect()

Draws a rectangle in simulator units. Typically, used in artwork creation macros rather than in the *de_add_rectangle()* function. Returns: none.

Syntax:

de_draw_rect(x1,y1, x2,y2);

where

x1,y1 is the lower left corner of the rectangle.

x2,y2 is the upper right corner of the rectangle.

Example:

```
de_draw_rect(0,0, 10,20);
```

de_draw_text()

Draws text at given location, with given font, height and angle, height and angle are in simulator units. Typically, used in artwork creation macros rather than in the *de_add_text()* function. Returns: none.

Syntax:

de_draw_text(font, x,y, height, angle, string);

where

font is an integer indicating text font number.

0 = Hershey Roman
1 = Hershey Roman Narrow

x,y is the location for the text string (lower left corner of first character).

height is the height of the text in user units.

angle is an angle in simulator angle units.

string is a text string.

Example:

```
de_draw_text(1, 12.3,45.6, 44, 90.0, "this is a text string");
```

de_dse_l2s()

Synchronizes the schematic with the layout, using the layout as the reference representation. Returns: none.

See also: *de_dse_s2l()*.

Syntax:

de_dse_l2s(starting_item, x,y, ang, x_spac,y_spac);

where

starting_item is the name of the component to start the synchronization from. Usually set to “P1” for port 1.

x,y is the location to place the starting element when creating a layout or schematic the first time.

ang is the angle to place the starting element when creating a layout or schematic the first time.

x_spac, y_spac is the spacing to use between components. A wire will be drawn to connect components that are spaced > 0 units apart. Schematic components are usually spaced 0.5 inches apart, layout 0.

Example:

```
de_dse_l2s("P1", x,y, ang, 0,0 ,0,0);
```

de_dse_s2l()

Synchronizes the layout with the schematic, using the schematic as the reference representation. Returns: none.

See also: *de_dse_l2s()*.

Syntax:

de_dse_s2l(starting_item, x,y, ang, x_spac, y_spac, layoutWinInstP);

where

starting_item is the name of the component to start the synchronization from. Usually set to “P1” for port 1.

x,y is the location to place the starting element when creating a layout or schematic the first time.

ang is the angle to place the starting element when creating a layout or schematic the first time.

x_spac, y_spac is the spacing to use between components. A wire will be drawn to connect components that are spaced > 0 units apart. Schematic components are usually spaced 0.5 inches apart, layout 0.

layoutWinInstP is the handle to the layout window to synchronize.

Example:

```
de_dse_s2l("P1", x,y, ang, 0,0, 0,0, layoutWinInstP);
```

de_edit_annotation_attribute()

Edits attributes of selected instance’s parameter annotation. Returns: none.

See also: *de_change_annotation_layer()*.

Syntax:

de_edit_annotation_attribute(font, height, maxRows, isPoint);

where

font is a string indicating text font.

height is the height of text in user units or points.

maxRows is the number of rows per column of text.

isPoint informs whether height is in user units or point size, where:

TRUE = height is in points
FALSE = height is in user units

Example:

```
de_edit_annotation_attribute("Arial",10,15,TRUE);
```

de_edit_item()

Allows a given item to be selected for editing. Returns: a pointer to the instance.

See also: *de_end_edit_item()*, *de_set_item_id()*, *de_set_item_parameter()*.

Syntax:

de_edit_item(id);

where

id is a string indicating unique id of the instance to edit.

Example:

```
// Select the instance with id R1
decl itemInfoOSP = de_edit_item ("R1");
// Change the instance id to R11
de_set_item_id(itemInfoOSP, "R11");
// Finish the editing (commit and display the editing changes)
de_end_edit_item(itemInfoOSP);
```

de_edit_path_trace()

Modifies attributes of a selected path or trace in the current representation. Returns: none.

Syntax:

de_edit_path_trace(cornerType, width, cutoffRatio, curveRadius);

where

cornerType is the corner type.

1 = mitered
2 = square
3 = curved

width is the width of path or trace in user units.

cutoffRatio is the angle of miter (used only if *cornerType* is mitered).

curveRadius is the radius of the curve (used only if *cornerType* is curve).

Example:

```
de_edit_path_trace(1, 12.4, 20.0, 0);
```

de_edit_symbol_pin()

Sets up the editing of a symbol pin. Returns: none.

Syntax:

de_edit_symbol_pin(name,number, angle, type[, pinPower]);

where

name is the name of pin.

number is an integer pin number. Each pin of a symbol must have a unique number.

angle is the pin angle. Used only in design synchronization when creating/updating schematic from layout. Determines the angle of the connecting part.

type is the pin direction: Input, Output, or Input/Output.

pinPower is optional; power of the pin.

Example:

```
de_set_edit_symbol_pin(10, 20);
de_edit_symbol_pin("Input", 1, 90.0, 1);
```

de_edit_text_attribute()

Edits the attributes of selected text components in the current representation. Returns: none.

Syntax:

de_edit_text_attribute(font, height, angle, just, absolute, isPoint);

where

font is a string indicating text font.

height is the height of the text in user units or points.

angle is the angle of text in degrees.

just is a field composed of the *or* of the first four bits of an integer. The first two bits represent the top, center, bottom vertical justification, while the next two represent the left, middle, right horizontal justification. The default justification is left, bottom (9 or 1001 binary).

absolute defines mode of text rotation, where:

TRUE = text on a symbol or design does not rotate when the symbol or design is rotated

FALSE = text on a symbol or design rotates when the symbol or design is rotated; default

isPoint informs whether height is in user units or point size, where:

TRUE = height is in points
FALSE = height is in user units

Example:

```
de_edit_text_attribute("Ariel For CAE", 10, 0, 9, 0, TRUE);
```

de_edit_text_string()

Edits text string in the current representation that was set with *de_set_edit_text()*. Returns: none.

Syntax:

de_edit_text_string(newString);

where

newString is the new text string which replaces selected text strings.

Example:

```
de_set_edit_text(0,0,1);
```

```
de_edit_text_string("this is the new string");
```

de_empty()

Empties an enclosed, filled shape in the current representation, creating a hole. Returns: none.

See also: *de_fill()*.

Syntax:

de_empty(innerX, innerY, outerX, outerY);

where

innerX, *innerY* is the coordinate within select region of inner shape (hole).

outerX, *outerY* is the coordinate within select region of outer shape (enclosing shape).

Example:

```
de_empty(10, 20, 30, 60);
```

de_end()

Completes a polygon, polyline, wire or trace command sequence. This command completes a shape. Use the *de_end_command()* to terminate the repeating command. Returns: none.

Syntax:

de_end([thickness]);

thickness is optional, where:

1 = thin, default
2 = medium
3 = thick

Example:

```
de_add_polyline();
de_add_point(10,20);
de_add_point(40, 60);
de_add_point(20, 30);
de_end();
de_end_command();
```

de_end_command()

Terminates a repeating command sequence. To properly playback any macro with repeating commands, the macro needs to terminate the command sequence with this function. Returns: none.

Syntax:

de_end_command();

Example:

```
de_add_polygon();
de_add_point(2.75,3.5);
de_add_point(3.5,3.5);
de_add_point(3.5,3.75);
de_add_point(4.0,3.75);
de_add_point(4.0,3.0);
de_add_point(2.75,3.0);
de_add_point(2.75,3.0);
de_end();
de_end_command();
```

de_end_edit_item()

Commits and displays the editing changes of a given instance. Returns: none.

See also: *de_edit_item()*.

Syntax:

de_end_edit_item(itemP);

where

itemP is a pointer to the item structure (the return value of de_edit_item()).

Example:

```
// Select the instance with id R1
decl itemInfoOSP = de_edit_item ("R1");
// Change the instance id to R11
de_set_item_id(itemInfoOSP, "R11");
// Finish the editing (commit and display the editing changes)
de_end_edit_item(itemInfoOSP);
```

de_export_design()

Exports the current layout or schematic in given format. Options are set via appropriate options file for the export format. Returns: none.

See also: *de_import_design()*.

Syntax:

de_export_design(designType, “outputFileName”);

where

designType is a numerical value indicating type of output design, where:

DE_HPIFF_FILE = HP IFF
DE_GDSII_FILE = GDSII Stream Format
DE_IGES_FILE = IGES
DE_HPGL2_FILE = HPGL/2
DE_MASK_FILE = Mask File (.msk)
DE_EGSGEN_FILE = EGS Generate Format
DE_EGSARC_FILE = EGS Archive Format
DE_GERBER_FILE = ACS MTOOLS Gerber
DE_GERBER_VIEWER = ACS MTOOLS Gerber Viewer
DE_DXF_FILE = ACS MTOOLS DXF
DE_MGCPCB_FILE = MGC/PCB

outputFileName is the file name where output will be written

Example:

The example outputs the current design in GDSII format.

```
de_export_design(DE_GDSII_FILE, “mmicAmp”);
```

de_fill()

Converts an empty shape (a hole) in the current representation into a filled shape (normal closed shape). Returns: none.

See also: *de_empty()*.

Syntax:

de_fill(x,y);

where

x,y is the point within select region of empty object to fill.

Example:

```
de_fill(20,30);
```

de_find_arc_center()

Moves the cursor to the center of the arc or circle selected within the select region and enters the coordinates to the event that was previously installed. Returns: none.

Syntax:

de_find_arc_center(x,y);

where

x,y is the point within select region of arc center to find.

Example:

```
de_find_arc_center(20,30);
```

de_find_line_center()

Moves the cursor to the pin closest to the center of the line selected within the select region and enters that point to the event that was previously installed. Returns: none.

Syntax:

de_find_line_center(x,y);

where

x,y is the point within select region of line center to find.

Example:

```
de_find_line_center(20,30);
```

de_find_pin()

Moves the cursor to the pin closest to the point selected within the select region and enters that point to the event that was previously installed. Returns: list (x,y), the coordinates of the pin within the pick region, else NULL.

Syntax:

de_find_pin(x,y);

where

x,y is the point within select region of pin to find.

Example:

```
de_find_pin(20,30);
```

de_find_vertex()

Moves the cursor to the pin closest to the vertex selected within the select region and enters that point to the event that was previously installed. Returns: none.

Syntax:

de_find_vertex(x,y);

where

x,y is the point within select region of vertex to find.

Example:

```
de_find_vertex(20,30);
```

de_fix_instances()

Fixes the position of an instance in the current representation and prevents design synchronization from re-positioning it. Returns: none.

See also: *de_free_instances()*.

Syntax:

de_fix_instances([x,y]);

where

x,y is optional. Point within select region of instance to fix; if not specified, uses selected instances.

Example:

```
de_fix_instances(15, 20);
```

de_flatten()

Removes a single level of hierarchy. Returns: none.

See also: *de_instantiate()*.

Syntax:

de_flatten();

Example:

```
de_flatten();
```

de_free_instances()

Frees the position of an instance in the current representation and allows design synchronization to re-position it. Returns: none.

See also: *de_fix_instance()*.

Syntax:

de_free_instances([x,y]);

where

x,y is optional. Point within select region of instance to free; if not specified, uses selected instances.

Example:

```
de_free_instances(15, 20);
```

de_free_item()

Frees the data structure created by *de_init_item()* when it is no longer needed. Returns: NULL.

See also: *de_init_item()*, *de_place_item()*, *de_set_item_id()*, *de_set_item_parameters()*, *de_edit_item()*, *de_end_edit_item()*.

Syntax:

de_free_item(itemP);

where

itemP is a pointer to the item structure (the return value of *de_init_item()*).

Example:

```
// Initialize a resistor item
decl itemInfoOSP = de_init_item("R");
// Place the resistor
de_place_item(itemInfoOSP, 0.125, -1.625);
// Free the resistor item
```

```
itemInfoOSP = de_free_item(itemInfoOSP);
/* You should set the variable initialized in the de-init_item step to the
return value of the de_free_item so it resets to NULL. This allows future
de_place_item()'s with the no longer valid variable to behave as NOPs (not do
anything). */
```

de_generate_symbol()

Generates a schematic symbol with the number of pins determined from the number of points on the schematic. Returns: none.

Syntax:

de_generate_symbol(leadLen, leadSpacing[, symbolType, replace, order]);

where

leadLen is the lead length, usually 0.25.

leadSpacing is the distance between pins, usually .25.

symbolType is optional; TRUE if a Dual line symbol is desired; FALSE if a Quad line symbol is desired

replace is optional; TRUE if the existing symbol should be replaced; FALSE for otherwise.

order is optional; FALSE if the pins should be ordered by location; TRUE if the pins should be ordered by their pin numbers.

Example:

```
de_generate_symbol (.25,.25);
```

de_get_data_parm()

Returns a string, the value of a given data reference or a named parameter belonging to a default component. Returns NULL if reference is not located.

Syntax:

de_get_data_parm(name, key, type);

where

name is the name of the default component (instance name).

key is the name of the component's parameter.

type is the default component's prefix, such as: MSUB, TEMP, TAND, PERM, etc.

Example:

The example retrieves the height parameter of the default MSUB.

```
hm = de_get_data_parm("MSUB10", "h", "MSUB");
```

de_group_edit_parameter_value()

Modifies selected instances with the parameter name to the value given. Returns: none.

Syntax:

de_group_edit_parameter_value(paramName. paramValue);

where

paramName is the name of the parameter.

paramValue is a string or real value of the parameter.

Example:

```
de_group_edit_parameter_value("L", "25 mil");
```

de_highlight_instance()

Highlights an instance using the highlight color. Returns: none.

Syntax:

de_highlight_instance(designName, repType, instName, toggle);

where

designName is the name of design the instance is in.

repType is the type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

instName is the unique instance name.

toggle is optional. If present toggle highlight state, otherwise set highlighting.

Example:

```
de_highlight_instance("amp", REP_SCHEM, "M1");
```

de_import_design()

Imports a foreign design format. Translates a foreign file into a design file. Returns: none.

See also: *de_export_design()*.

Syntax:

de_import_design(designType, overwrite, “inputFileName”, “designFileName” [, “defaultDesignName”]);

where

designType is a numerical value indicating type of output design, where:

DE_SPICE_FILE = Spice File. The spice dialect for translation is set using the SpiceDialect Option in the options file *spice.opt*, where:

1 = SPICE2G
2 = SPICE3
3 = PSPICE (default)
4 = HSPICE
5 = HPSPICE

DE_HPIFF_FILE = HP IFF

DE_GDSII_FILE = GDSII Stream Format

DE_IGES_FILE = IGES

DE_HPGL2_FILE = HPGL/2

DE_MASK_FILE = Mask File (.msk)

DE_EGSGEN_FILE = EGS Generate Format

DE_EGSARC_FILE = EGS Archive Format

overwrite is a numerical value indicating whether design file is overwritten.

0 = do not overwrite design file
1 = overwrite design file

inputFileName is a string, enclosed in quotes, indicating the file name of the file to be translated.

designFileName is a string, enclosed in quotes, indicating the file name of the design file to be generated.

defaultDesignName is a string, enclosed in quotes, indicating the default design for Series IV migration. Optional.

Example:

The example imports the current design in GDSII format.

```
de_import_design(DE_GDSII_FILE, 1, "graph1.hpg", "graph1.dsn");
```

de_init_item()

Initializes the instance, readying it for placement. Returns: a pointer to the instance.

See also: *de_place_item()*, *de_free_item()*, *de_set_item_id()*, *de_set_item_parameters()*, *de_edit_item()*, *de_end_edit_item()*.

Syntax:

de_init_item(itemName);

where

itemName is the name of the instance to initialize.

Example:

```
// Initialize a resistor
decl itemInfoOSP = de_init_item("R");
// Place the resistor
de_place_item(itemInfoOSP, 0.125, -1.625);
// Free the resistor item
itemInfoOSP = de_free_item(itemInfoOSP);
```

de_insert_arrow()

Inserts an arrow draw as a polyline or polygon item.

Syntax:

de_insert_arrow(arrowNum, length, width, solid, x2, y2, x1, y1);

where

arrowNum is the number of arrowheads to be drawn.

length is the length of the arrowhead.

width is the width of the arrowhead.

solid is TRUE if the arrow should be drawn as a polygon; FALSE if the arrow should be drawn as a polyline.

x1, *y1* and *x2*, *y2* are the coordinates of the end points of the arrows.

Example:

```
de_insert_arrow(2, 20, 6.67, FALSE, -220, -165, 20, 10);
```

de_insert_dimlin()

Inserts a dimension line.

Syntax:

de_insert_dimlin(x1, y1, x2, y2);

where

x1, *y1* and *x2*, *y2* are the coordinates of the end points of the dimension line.

Example:

```
de_insert_dimlin(-220, -155, -70, 50);
```

de_instantiate()

Creates a new design from the selected group. The design is not saved to disk and it is not placed. This is the opposite of *de_flatten()*. Returns: none.

See also: *de_flatten()*.

Syntax:

de_instantiate(newDesignName);

where

newDesignName is the name of the design to create.

Example:

```
de_instantiate("sub_amp");
```

de_last_view()

Recalls last view into the current window. Returns: none.

See also: *de_restore_view(), de_store_current_view(), de_delete_view()*.

Syntax:

de_last_view();

Example:

```
de_last_view();
```

de_load_item_artwork_image()

Generates the component artwork based on the values stored in the given data structure. This function should be called if the operation of placing new components is in the layout representation. Returns the handle to the artwork image buffer. This handle can be used to set up the mouse tracking image.

See also: *de_init_iteminfo(), de_query_iteminfo_attr(), de_query_iteminfo_attr(), de_iteminfo_edit_instance(), de_free_iteminfo(), set_instance(), edit_instance(), place_instance(), set_instance_parameters(), set_instance_id().*

Syntax:

de_load_item_artwork_image(itemDataP);

where

itemDataP is the handle to the data structure set up by the function *de_init_iteminfo()*.

Example:

```
decl itemDataP;

// initialize the operation of placing new components
itemDataP = de_init_iteminfo ("MLIN", EDIT_ITEM_NEW);
// generate the artwork based on the default parameter values
de_load_item_artwork_image (itemDataP);
// set the component instance name to "MYMLIN1"
de_set_iteminfo_attr (itemDataP, ITEMINFO_INST_NAME, "MYMLIN1");
// set the parameters values
de_set_iteminfo_attr (itemDataP, ITEMINFO_TO_APPLY_DB_PARAMLIST,
                      "\"MSUB2"\", "10 mil", "100 mil",
                      "", "", "");
// re-generate the artwork based on the new parameter values
de_load_item_artwork_image (itemDataP);
// place a new MLIN at location (100, 0)
de_iteminfo_new_instance (itemDataP, 100, 0, 1);
// free the data structure
de_free_iteminfo (itemDataP);
```

de_merge_and()

Performs an “and” among selected overlapping closed shapes. Performs a logical “and”. Returns: none.

See also: *de_merge_or(), de_merge_diff()*.

Syntax:

de_merge_and();

Example:

```
de_merge_and();
```

de_merge_diff()

Performs a difference between selected overlapping closed shapes. Performs a logical “and not”. Returns: none.

See also: *de_merge_and(), de_merge_or()*.

Syntax:

de_merge_diff();

Example:

```
de_merge_diff();
```

de_merge_or()

Merges selected overlapping, abutting closed shapes. Performs a logical “or”. Returns: none.

See also: *de_merge_diff(), de_merge_and()*.

Syntax:

de_merge_or();

Example:

```
de_merge_or();
```

de_mirror_x()

Mirrors selected objects around the X-axis, given a reference point. Returns: none.

Syntax:

de_mirror_x(x,y);

where

x,y is the reference point to mirror around.

Example:

```
de_mirror_x(10,20);
```

de_mirror_y()

Mirrors selected objects around the Y-axis, given a reference point. Returns: none.

Syntax:

de_mirror_y(x,y);

where

x,y is the reference point to mirror around.

Example:

```
de_mirror_y(10,30);
```

de_miter_vertex()

Creates a mitered edge on a polygon or polyline. Returns: none.

Syntax:

de_miter_vertex([x,y]);

where

x,y is optional. The point to convert into a mitered corner. If not specified, previously-selected point is converted.

Example:

```
de_set_miter_length(10);
de_miter_vertex();
```

de_modify_arc_resolution()

Modifies the resolution of an arc. Returns: boolean true or false.

Syntax:

de_modify_arc_resolution(resolution);

where

resolution is the resolution in degrees.

Example:

```
de_modify_arc_resolution(5);
```

de_modify_break()

Converts selected polygons into polylines; that is, breaks a closed shape into an open shape. Returns: none.

See also: *de_modify_join(), de_modify_explode()*.

Syntax:

de_modify_break([x,y]);

where

x,y is optional. The point within selected region of object to break.

Example:

```
de_modify_break();
```

or

```
de_modify_break(10,20);
```

de_modify_circle_radius()

Modifies the radius of a circle. If the absolute radius is less than or equal to zero, use delta radius instead. Returns: boolean true or false.

Syntax:

de_modify_circle_radius(absRadius, deltaRadius);

where

absRadius is the radius as given.

deltaRadius is the change in radius.

Example:

```
de_modify_circle_radius(0, 0.5);
```

de_modify_explode()

Converts selected polygons and polylines into two-point polyline segments. Each pair of vertices form a new shape. Returns: none.

See also: *de_modify_join()*, *de_modify_break()*.

Syntax:

de_modify_explode([x,y]);

where

x,y is optional. The point within selected region of object to explode.

Example:

```
de_modify_explode();
```

or

```
de_modify_explode(10,20);
```

de_modify_join()

Joins selected polylines with coincident end points into a single polyline or polygon. If selected polylines form a closed shape, a polygon will be created. Returns: none.

See also: *de_modify_explode()*, *de_modify_break()*.

Syntax:

de_modify_join([x,y]);

where

x,y is optional. The point within selected region of object to join.

Example:

```
de_modify_join();
```

or

```
de_modify_join(10,20);
```

de_move()

Moves selected items in the current representation by a given amount. Returns: none.

See also: *de_move_break()*.

Syntax:

de_move(dx, dy);

where

dx,dy is the delta to move the selected objects by.

Example:

```
de_move(20,-30);
```

de_move_annotation()

Moves parameter annotation of a selected instance to a new location. Returns: none.

See also: *de_set_move_annotation()*, *de_change_annotation_layer()*, *de_edit_annotation_attribute()*.

Syntax:

de_move_annotation(dx,dy);

where

dx,dy is the delta to move the annotation by.

Example:

```
de_set_move_annotation(10,10);
de_move_annotation(10,20);
```

de_move_break()

Moves selected instances in the current representation, breaking any wire or trace connection. Returns: none.

See also: *de_break_connection()*, *de_move()*.

Syntax:

de_move_break(dx,dy);

where

dx,dy is the delta to move the selected instances.

Example:

```
de_move_break(10.5,12.3);
```

de_move_to_layer()

Moves selected objects in the current representation to the current entry layer. Returns: none.

See also: *de_copy_to_layer()*.

Syntax:

de_move_to_layer([layerNo]);

where

layerNo is optional; layer to move selected objects to. If layer is not specified, object is moved to current layer.

Example:

```
de_select_all()
de_move_to_layer();
```

de_net()

Creates a netlist report file. This is similar to a parts list in netlist format. Returns: none.

See also: *de_bom()*, *de_parts()*.

Syntax:

de_net(fileName, showDlg);

where

fileName is the name of report file to be created.

showDlg is whether or not to show the parts list report where:

TRUE = show the report
FALSE = do not show the report (the file is still recorded)

Example:

```
de_net("net");
```

de_netlist()

Creates a netlist file starting at the current design, moving through all referenced designs. The netlist file is named according to the value of “NETLIST_FILE_NAME” set in an ADS configuration file. Returns: none.

Syntax:

de_netlist();

Example:

```
de_netlist();
```

de_new_datadisplay()

Sends the new window command to the data display server. Opens a data display window. Returns: none.

Syntax:

de_new_datadisplay([dataSetName]);

where

dataSetName is the name of the dataset that will most likely be used. Optional.

Example:

```
de_new_datadisplay();
```

de_new_design()

Creates a new, empty design. Depending on its type, different extensions are added automatically. When saved, the extension *.dsn* is added to all designs. Returns a handle to the new design, NULL if not successful.

Syntax:

de_new_design(designName, designCode);

where

designName is the name of the design to create

designCode is the unique code for each design type, as defined in stddefs.ael, such as *libranet*, *analogRFnet* or *sigproc_net*.

Example:

```
decl designP=de_new_design("aAmp", sigproc_net);
if(designP)
    de_show_design_in_window(designP, api_get_current_window());
```

de_new_project()

Creates a new project directory and all of its sub-directories. Returns: none.

Syntax:

de_new_project(projectName);

where

projectName is the name of the project to create. A *_prj* extension is appended if it is not specified. The feature of automatically appending _prj extension can be turned off in the Options > Preferences dialog available from the Main window.

Example:

```
de_new_project("c:\sub_assembly");
```

de_open_design()

Opens a design (and its AEL definition, if it exists). Returns a handle to the design, NULL if design is not found.

Syntax:

de_open_design(dsnName, overwrite);

where

dsnName is the name of design file.

overwrite is the choice for overwriting existing design, where:

FALSE = don't overwrite if design exists in memory
TRUE = overwrite

Example:

```
decl designP=de_open_design("tee5", FALSE);
```

de_open_project()

Opens the program to the given project. The program has its reference directory changed to the new project directory. The old project (if any) is cleaned out of memory

and the new project's environment variables and libraries are initialized. Returns: none.

Syntax:

de_open_project(projectName);

where

projectName is the name of the project directory to attach to; should be a full path name.

Example:

```
de_open_project("c:\myamp_prj");
```

de_open_window()

Opens a Schematic or Layout window. Returns: none.

See also: *de_create_window()*, *de_set_window()*, *de_close_window()*.

Syntax:

de_open_window(windowType, designName);

where

windowType is the type of window, where:

SCHEM_WIN = Schematic window
LAYOUT_WIN = Layout window

designName is the name of a design

Example:

```
de_open_window(SCHEM_WIN);
de_set_window(SCHEM_WIN);
de_close_window();
```

de_oversize()

Creates a new oversized or undersized shape from the selected closed shapes in the current representation. The shape will be oversized or undersized by the amount specified by *de_set_oversize()*. Returns: none.

See also: *de_set_oversize()*, *de_scale()*, *de_set_scale()*.

Syntax:

de_oversize();

Example:

```
de_oversize();
```

de_pan_window()

Re-centers the viewing window to the given point. Returns: none.

See also: *de_zoom_window(), de_zoom_in_point(), de_zoom_out_point(), de_view_all(), de_zoom_in_scale(), de_zoom_out_scale().*

Syntax:

de_pan_window(x,y);

where

x,y is the point for new viewing window center.

Example:

```
de_pan_window(10, 20);
```

de_parts()

Generates a parts list for the current representation and stores in the specified file. Opens the Parts List UI. Returns: none.

See also: *de_net(), de_bom().*

Syntax:

de_parts(fileName, showDlg);

fileName is the name of file for storing parts list.

showDlg is whether or not to show the parts list report where:

TRUE = show the report
FALSE = do not show the report (the file is still recorded)

Example:

```
de_parts ("myParts");       //creates file myParts.p1
```

de_parts_option_add_exclusion_items()

Adds an Exclusion List to the parts list options.

Syntax:

de_parts_option_add_exclusion_items (list ("MLIN"));

where:

list is a list of items that will not appear in the parts list. This list is useful if parts have not been consistently flagged as BOM items. For this case, you wish to include everything except items in the exclusion list. In order to include everything, do not check the BOM flag.

Example:

```
de_parts_option_check_bom (FALSE);
```

or

```
de_parts_option_add_exclusion_items (DePartsLumpedWithArtworkElements);
```

de_parts_option_add_inclusion_items()

Adds an Inclusion List to the parts list options.

Syntax:

de_parts_option_add_inclusion_items (list ("res_smt"));

where:

list is a list of items that will appear in the parts list. This list is useful if parts have not been consistently flagged as BOM items. For this case, specify to include only items flagged as BOM items, and add additional items in the inclusion list.

Inclusion items are treated as leaf-level parts and do not get flattened. For example, if an inclusion item is a hierarchical part, its subelements will not be included in the parts list.

Example:

```
de_parts_option_check_bom (TRUE);
```

or

```
de_parts_option_add_inclusion_items (list ("res_smt"));
```

de_parts_option_check_bom()

Checks the BOM Flag.

Syntax:

de_parts_option_check_bom (TRUE | FALSE);

where:

TRUE Only include instances with attribute INST_SPECIAL set as ITEM_BOM_ITEM

FALSE Do not test for ITEM_BOM_ITEM (default)

de_parts_option_include_header()

Sets the parts list option to Include Header.

Syntax:

de_parts_option_include_header (TRUE | FALSE);

where:

TRUE is the output header information (default).

FALSE is the output part data only.

de_parts_option_set_attribute_columns()

Sets the User Attribute Columns parts list options.

Syntax:

de_parts_option_set_attribute_columns (list ("INST_SPECIAL", “PART_NUM", "Price"));

where:

list is the list of attributes that will appear as columns in the parts list. The attributes can be user properties, user parameters, or instance attributes. The following instance attributes can appear in the report:

INST_TYPE

INST_SPECIAL

INST_NAME

INST_DESIGN_NAME

INST_SYMBOL_NAME

INST_BBOX

INST_PROPERTY

de_parts_option_set_center_placement()

Sets the Component Placement X,Y Coordinates parts list options.

Syntax:

de_parts_option_set_center_placement (TRUE | FALSE);

where:

TRUE Coordinates represent the center point of the instance bounding box. The bounding box does not include the annotation text.(default)

FALSE Coordinates represent the location of pin one.

de_parts_option_set_delimeter()

Sets the Delimeter Character parts list options.

Syntax:

de_parts_option_set_delimeter (delimeter);

where:

delimeter is used to separate column data (i.e. " ", ","). The default is NULL. If a NULL delimeter is specified, column widths will be determined by the longest data field and all data will be left justified.

Example:

```
/* Separate columns with commas */
de_parts_option_set_delimeter (",");
```

or, for example:

```
/* Auto-format */
de_parts_option_set_delimeter (NULL);
```

de_parts_option_set_hierarchical()

Sets Hierarchical Reporting parts list option.

Syntax:

de_parts_option_set_hierarchical (TRUE | FALSE);

where:

TRUE Produces a parts list containing instances from all levels of the hierarchy (default).

FALSE Produces a parts list containing instances from only the top level of hierarchy.

de_parts_option_set_package_offset()

Sets the Package Offsets parts list option.

Syntax:

de_parts_option_set_package_offset (packageAttributeName, packageName, xOffset, yOffset);

where:

packageAttributeName is the name of the user attribute.

packageName is the value of the user attribute.

xOffset is the amount the origin on the x coordinate will be offset to make the placement coordinate.

yOffset is the amount the origin on the y coordinate will be offset to make the placement coordinate.

Example:

For each instance which has a user attribute named "Package", with attribute value "P1", the placement coordinate will be the origin offset by xOffset, yOffset.

de_parts_option_set_package_offset ("Package", "P1", 15, 0);

de_parts_option_sort_by_component()

Sets the Sort by Component Name parts list option.

Syntax:

de_parts_option_sort_by_component (TRUE | FALSE);

where:

TRUE Sort the parts list by the component name (default).

FALSE Parts are listed as they appear in the database.

de_paste_from_buffer()

Copies the contents of a buffer to the current representation. Note that schematic instances cannot be copied to layout and vice-versa. Returns: none.

Syntax:

de_paste_from_buffer(x,y);

where

x,y is the point to paste origin of paste buffer.

Example:

```
de_paste_from_buffer(23.4, 56.98);
```

de_place_design_template()

Inserts the design template setup with *de_set_design_template()* into current design. Returns: none.

See also: *de_set_design_template()*.

Syntax:

de_place_design_template(x,y);

where

x,y is the X and Y location to place the template.

Example:

```
de_place_design_template(1.2, -1.2);
```

de_place_item()

Places the instance that was initialized by *de_init_item()*. Returns: none.

See also: *de_init_item()*, *de_free_item()*, *de_set_item_id()*, *de_set_item_parameters()*, *de_edit_item()*, *de_end_edit_item()*.

Syntax:

de_place_item(itemP, xLoc, yLoc);

where

itemP is a pointer to the item structure (the return value of *de_init_item()*).

xLoc and *yLoc* are the x and y locations on the layout or schematic drawing area where the instance is to be placed.

Example:

```
// Initialize a resistor item
decl itemInfoOSP = de_init_item("R");
// Place three resistors in a row
de_place_item(itemInfoOSP, 0, 0);
de_place_item(itemInfoOSP, 1, 0);
de_place_item(itemInfoOSP, 1, 0);
// Free the resistor item
itemInfoOSP = de_free_item(itemInfoOSP);
```

de_place_port()

Places a symbol pin in the current representation's symbol view. Adds a pin to a symbol. The pin/ports attributes are set with *de_set_port()* command. Returns: none.

See also: *de_set_port()*.

Syntax:

de_place_port(x,y);

where

x,y is the port location.

Example:

```
de_switch_view(SYMBOL_VIEW);
de_set_port("",1,0.0);
de_place_port(0.75,3.25);
```

de_place_unplaced()

Places an instance that has not yet been placed in one representation. Returns: none.

Syntax:

de_place_unplaced(x,y);

where

x,y is the location in the other representation to place the instance.

Example:

```
de_select_unplaced(15, 25);
de_set_window(LAYOUT_WINDOW);
de_place_unplaced(55.0,-45.0);
```

de_playback_macro()

Executes an AEL or DEM file. Returns: none.

Syntax:

de_playback_macro(macroName);

where

macroName is the name of the macro file.

Example:

```
de_playback_macro("abc.ael");
de_playback_macro("def.dem");
```

de_plot()

Creates a plot of the active design in the current window and sends it to the default printer or plotter. Returns: none.

Syntax:

de_plot();

Example:

```
de_set_window(SCHEM_WIN); // set schematic window
de_plot();
```

de_plot_to_file()

Creates a plot of the active design and saves it to a file in a format that can be sent to a printer or plotter. The format of the file is HPGL2. The file adds the extension *.hgl*. Returns: none.

Syntax:

de_plot_to_file(fileName);

where

fileName is the filename for saving the plot.

Example:

```
de_set_window(SCHEM_WIN);
de_plot_to_file("myFile");
```

de_pop_outof_instance()

Returns to the previous design before a *de_push_into_instance()* command. Returns: none.

See also: *de_push_into_instance()*.

Syntax:

de_pop_outof_instance();

Example:

```
de_pop_outof_instance();
```

de_push_into_instance()

Pushes into hierarchical instance reference. Opens the design referred to by the instance and sets it active. Returns: none.

See also: *de_pop_outof_instance()*.

Syntax:

de_push_into_instance(x,y);

where

x,y is the point within select region of instance to push into.

Example:

```
de_push_into_instance(10,20);
```

de_refresh_view()

Redraws the screen of the active window. Returns: none.

Syntax:

de_refresh_view();

Example:

```
de_refresh_view();
```

de_release_simulator()

Cancels the simulation and makes the simulator license available for other users on the network. Returns: none.

See also: *de_analyze()*

Syntax:

de_release_simulator();

Example:

```
de_release_simulator();
```

de_remove_properties()

Removes properties of design group, port, or instance. Returns: none.

See also: *de_set_edit_property()*, *de_add_property()*.

Syntax:

de_remove_properties(editSelected);

where

editSelected is the mode for selecting items to remove.

TRUE = edit only selected items
FALSE = edit item selected by de_set_edit_property()

Example:

```
de_remove_properties(FALSE);
```

de_restore_view()

Recalls specified view into the current window. Returns: none.

See also: *de_store_current_view()*, *de_delete_view()*.

Syntax:

de_restore_view(viewName);

where

viewName is the name of the view to recall.

Example:

```
de_restore_view("myView");
```

de_rotate()

Rotates selected items in the current representation around a given point. Returns: none.

Syntax:

de_rotate(x,y, angle);

where

x,*y* is the point to rotate around.

angle is the angle to rotate.

Example:

```
de_rotate(10, 20, 45);
```

de_rotate_90()

Rotates an item being placed 90 degrees clockwise around pin 1. Returns: none.

Syntax:

de_rotate_90();

Example:

```
de_rotate_90();
```

de_rotate_center()

Rotates selected items in the current representation around the center of gravity if more than one object is selected. If only one object is selected, rotates around pin 1. Returns: none.

Syntax:

de_rotate_center(angle);

where

angle is the angle in degrees.

Example:

```
de_rotate_center(45);
```

de_rotate_image()

Rotates an instance (element) being placed in a specified direction around pin 1. This command must be specified between the function *set_instance()* and the function *place_instance()*. Returns: none.

Syntax:

de_rotate_image(direction);

where

direction is the string “UP”, “DOWN”, “LEFT”, “RIGHT”.

Example:

```
decl itemInfoOSP=de_init_item("R");
de_rotate_image("UP");
de_place_item(itemInfoOSP, 10, 20);
itemInfoOSP=de_free_item(itemInfoOSP);
```

de_save_all_designs()

Saves all designs in memory. Does not save untitled designs. Returns: none.

Syntax:

de_save_all_designs();

Example:

```
de_save_all_designs();
```

de_save_design()

Saves the schematic and layout representations of a design either as the name the design currently has or as a different name. Returns the name that the design was saved as.

Syntax:

de_save_design(designName, designHandle);

where

designName is the name to save the design as; NULL if saving design as same name.

designHandle is the handle of the design to save.

Example:

```
de_save_design("test1", designHandle);
```

de_save_design_template()

Saves the schematic representation of a design (or a design template) as a design template. The name of design template can be the name the design (or design template) currently has or can be a different name. Design templates are stored in special directories for access from any project. These directories are: CIRCUIT_TEMPLATE_DIR for Analog/RF Designs and HPTOLEMY_TEMPLATE_DIR for DSP Designs. Returns the name the design template was saved as.

Syntax:

de_save_design_template(templateName, designHandle);

where

templateName is the name to save the design template as; NULL if saving template as same name.

designHandle is the handle of the design (or design template) to save.

Example:

```
de_save_design_template("test1", designHandle);
```

de_scale()

Scales items in the current representation by a scale factor. Returns: none.

See also: *de_set_scale(), de_set_oversize(), de_oversize()*.

Syntax:

de_scale(x,y);

where

x,y is the X and Y location to scale around. The scale factor is set with *de_set_scale()*.

Example:

```
de_scale(10, 15);
```

de_search_and_replace()

Finds all references (depending on refType), given a variable or item reference, and highlights the item. Optionally, if searchOnlyFlag is set to 0, replaces the reference with the reference specified by replaceName. Returns: none.

Syntax:

de_search_and_replace(searchName, replaceName, refType, searchOnlyFlag);

where

searchName is the name of variable or item reference.

replaceName is the name of variable or item to search for or replace.

refType is the choice of reference type, where:

0 = item reference
1 = variable reference

searchOnlyFlag is the signal for searching and replace options, where:

0 = search and replace reference
1 = search for only

Example:

```
de_search_and_replace("TL1", "TL14", 0, 0);
```

de_select_all()

Selects everything matching the select filter in the current window. Returns: none.

Syntax:

de_select_all();

Example:

```
de_select_all();
```

de_select_all_force()

Selects all items regardless of a layers protection status. Used in macros that need to reliably modify selected items regardless of the layer state. Returns: none.

de_select_all_force();

Example:

```
de_select_all_force();
```

de_select_by_name()

Selects instances by instance name or ID. Returns: none.

Syntax:

de_select_by_name(name, selectType);

where

name is the instance name or ID.

selectType is the choice for type of selection, where:

0 = selects by instance ID
1 = selects by instance name

Example:

```
de_select_by_name("MLIN", 0);
```

or

```
de_select_by_name("TL1", 1);
```

de_select_item()

Toggles a selection of an item within the select region in the current representation. Returns: none.

Syntax:

de_select_item(x,y);

where

x,y is the location of item to toggle.

Example:

```
de_select_item(10,20);
```

de_select_range()

Selects all items in the current representation enclosed by a given window range. Returns: none.

Syntax:

de_select_range(x1,y1, x2,y2, [deselectFlag, buttonState]);

where

x1,y1 is the first corner of the select window.

x2,y2 is the second, opposite corner.

deselectFlag is optional. Signals whether to deselect all objects first, where:

0 = default; do not deselect all objects first
1 = deselect all objects first

buttonState is optional. Specification of select mode, where:

0 = unknown. If x1==x2 and y1==y2 a single item is selected; otherwise, all objects in window range are selected

1 = Single click to select a single object

2 = Double click to select a single object; deselect flag forced to 1

3 = Press and drag to select all objects in window range

Example:

```
de_select_range(3.25,2.75, 3.25,2.75);
```

de_select_unplaced()

Selects an unplaced instance in the current representation to place in the other representation (from schematic to layout or layout to schematic). Returns: none.

Syntax:

de_select_unplaced(x,y);

where

x,y is the point within select region of an unplaced instance.

Example:

```
de_select_unplaced(3.375,4.125);
```

```
de_set_window(LAYOUT_WINDOW);
de_place_unplaced(35.0,-40.0);
```

de_select_window()

Selects all items (matching the select filter) in the current representation enclosed by given window. Returns: none.

Syntax:

de_select_window(x1,y1, x2,y2);

where

x1,y1 is the point describing first corner selection window.

x2,y2 is the point describing opposite corner of selection window.

Example:

```
de_select_window(10,20, 40,60);
```

de_set_design_template()

Sets up the design template for insertion into current design by *de_place_design_template()*. Returns: none.

See also: *de_place_design_template()*.

Syntax:

de_set_design_template(templateName);

where

templateName is the name of the design template to be inserted.

Example:

```
de_set_design_template("mytemplate");
```

de_set_edit_property()

Sets the data group, port, or instance for property editing. Returns: none.

See also: *de_remove_properties()*, *de_add_property()*.

Syntax:

de_set_edit_property([x,y]);

where

x,y is optional. Location coordinates of data group, port, or instance for property editing. If if not specified, uses first selected data group, port, or instance that is found.

Example:

```
de_set_edit_property();
```

de_set_edit_symbol_pin()

Sets the symbol pin whose attributes are to be edited. Returns TRUE if symbol pin is found, FALSE if symbol pin is not found.

See also: *de_edit_symbol_pin()*.

Syntax:

de_set_edit_symbol_pin(x,y);

where

x,y is the location of symbol whose atributes are to be edited.

Example:

```
de_set_edit_symbol_pin(3.5, 4.125);
```

de_set_edit_text()

Sets the text string in the current representation to be edited by *de_edit_text_string()*. Returns: none.

Syntax:

de_set_edit_text(x,y, useSelected);

where

x,y is the point within select region of text to edit. If not given, first selected text is used.

useSelected is TRUE to set selected text, FALSE to use x,y location entered.

Example:

```
de_set_edit_text(10,20, 0);
de_edit_text_string("this is the new string");
```

de_set_instance_path_to_design()

Sets the Path to Specific Instance of Design to Layout for the design pointed to by designP which is used during the search for variable declarations for the purpose of artwork generation in layout. If designP is omitted or NULL, the value for the design in the current window is set.

Syntax:

de_set_instance_path_to_design(designP, pathToDsn);

where

designP is a handle to a valid design.

pathToDsn is the component path indicating which instance of the component in the top design has actual parameter values wanted for the layout/artwork generation. If the design is not used as a subcircuit of the top design, this field need not be filled in.

Example:

```
designName=de_current_design_name();
designP=db_get_design(designName);
de_set_instance_path_to_design(designP, "myVars.varset1");
```

de_set_item_id()

Sets the ID of the item that is either being readied to be placed (from *de_init_item()*) or is being placed (from *de_edit_item()*). Returns: none.

See also: *de_init_item()*, *de_place_item()*, *de_free_item()*, *de_set_item_parameters()*, *de_edit_item()*, *de_end_edit_item()*.

Syntax:

de_set_item_id(itemP, id);

where

itemP is a pointer to the item structure (the return value of *de_init_item()* and *de_edit_item()*).

id is the new unique id for the instance.

Example:

```
// Initialize a resistor
decl itemInfoOSP = de_init_item("R");
```

```
// Set the resistor id
de_set_item_id(itemInfoOSP, "R99");
// Place the resistor
de_place_item(itemInfoOSP, 0.125, -1.625);
// Free the resistor item
itemInfoOSP = de_free_item(itemInfoOSP);
```

de_set_item_parameters()

Sets the parameter values for a given item that has been selected by the *de_init_item()* or *de_edit_item()*. Returns: none.

See also: *de_init_item()*, *de_place_item()*, *de_free_item()*, *de_set_item_id()*, *de_edit_item()*, *de_end_edit_item()*.

Syntax:

de_set_item_parameters(itemP, listOfParameters);

where

itemP is a pointer to the item structure (the return value of *de_init_item()* and *de_edit_item()*).

listOfParameters is a complete list of parameters in the order they are to be set.

Example:

```
// Select the instance with id R1
decl itemInfoOSP = de_edit_item ("R1");
// Change the instance parameters to 99 Ohm (from 50 Ohm)
de_set_item_parameters(itemInfoOSP, list(prm("StdForm", "99 Ohm"),
prm("StdForm", ""), prm("Yes"), prm("StdForm", ""), prm("StdForm", ""),
prm("StdForm", ""), prm("StdForm", ""));
// Finish the editing (commit and display the editing changes)
de_end_edit_item(itemInfoOSP);
```

de_set_move_annotation()

Selects an instance to have its annotation moved. Returns: none.

See also: *de_move_annotation()*.

Syntax:

de_set_move_annotation(x,y);

where

x,y is the point within the instance to have its annotation moved.

Example:

```
de_set_move_annotation(10,10);
de_move_annotation(10,20);
```

de_set_origin()

Resets the origin of a representation (resets the 0,0 point). Returns: none.

Syntax:

de_set_origin(x,y);

where

x,y describes the new origin point for the representation.

Example:

```
de_set_origin(25, 30);
```

de_set_port()

Sets attributes of a symbol port (pin) before it is created for the current window. Returns: none.

See also: *de_place_port()*.

Syntax:

de_set_port(portName, portNum, portAngle[, type, power]);

where

portName is the port name (unused—reserved for future use).

portNum is the port number (integer > 0).

portAngle is the angle to use for instance connected by abutment to this pin.

type is optional; INPUT=0, OUTPUT=1, IN/OUT=2 (default is 0)

power is optional; power of pin.

Example:

```
de_set_port("", 1, 0.0);
```

de_set_simulation_dataset()

Sets the name of the dataset used for simulation. Returns: none.

See also: *de_set_simulation_host()*.

Syntax:

de_set_simulation_dataset(datasetName);

where

datasetName is the dataset name.

Example:

```
de_set_simulation_dataset("myDataSet");
de_set_simulation_host("local");
de_analyze();
```

de_set_simulation_host()

Sets the name of the host computer to be used for simulation. Returns: none.

See also: *de_set_simulation_dataset()*.

Syntax:

de_set_simulation_host(hostName);

where

hostName is the host computer name; “local” for local computer.

Example:

```
de_set_simulation_dataset("myDataSet");
de_set_simulation_host("local");
de_analyze();
```

de_set_swap_template_instance()

Sets the name of instance to swap with in the function *de_swap_instance()*. Returns: none.

Syntax:

de_set_swap_template_instance(instName);

where

instName is a string; the name of an instance.

Example:

```
//Select all MLINs, replace with TLIN.
```

```
de_select_by_name("MLIN", 0);
de_set_swap_template_instance("TLIN");
de_swap_instances(TRUE);
```

de_set_top_design_name()

Sets the Top Design for Variables for the design pointed to by designP. This value is used to determine what design is used as the starting point during a search for variable declarations for the purpose of artwork generation in layout. If designP is omitted or NULL, the design in the current window is used.

Syntax:

de_set_top_design_name(designP, designName);

designP is a handle to a valid design.

designName is the name of a design.

Example:

```
designName=de_current_design_name();
designP=db_get_design(designName);
de_set_top_design_name(designP, designName);
```

de_set_top_design_rep_type()

Sets the Top Design Representation Type for the design pointed to by designP. This value is used to determine whether the schematic or layout representation of the Top Design for Variables should be used during the search for Variable declarations for the purpose of artwork generation in layout. If designP is omitted or NULL, the value for the design in the current window is used.

Syntax:

de_set_top_design_name(designP, repType);

designP is a handle to a valid design.

repType is the type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

Example:

```
designName=de_current_design_name();
designP=db_get_design(designName);
de_set_top_design_name(designP, REP_SCHEM);
```

de_shove()

Shove by a distance, vertices or whole polygons, whole text, and whole instances that are selected and lie on vector side of dividing line formed perpendicular to the vector. Returns: none.

Syntax:

de_shove(x, y, angle, dist);

where

x,y is the point on dividing line.

angle is the angle of vector that indicates direction or movement.

dist is the distance to shove.

Example:

```
de_shove(0, 0, 90, 10);
```

de_show_connected()

Displays graphically unconnected pins and the interconnection needed. This feature is known as a *rats nest* display. Returns: none.

See also: *de_clear_show_connected()*.

Syntax:

de_show_connected();

Example:

```
de_show_connected();
```

de_show_design_in_window()

Displays the given design in the specified window. Returns: none.

Syntax:

de_show_design_in_window(designHandle, windowHandle);

where

designHandle is the handle of a design to display.

windowHandle is the handle of a window to display design.

Example:

```
decl designP=de_load_design("myAmp", 0);
if(designP)
    de_show_design_in_window(designP, api_get_current_window());
```

de_show_equiv_inst()

Highlights the equivalent instance in another representation to a given instance. Returns: none.

See also: *de_show_connected(), de_show_fixed(), de_show_unmatched(), de_show_unplaced()*.

Syntax:

de_show_equiv_inst(x,y);

where

x,y is the point within select region of instance.

Example:

```
de_show_equiv_inst(10,20);
```

de_show_fixed()

Highlights any instances placed with the position fixed attribute set in the current window. Returns: none.

Syntax:

de_show_fixed();

Example:

```
de_show_fixed();
```

de_show_unmatched()

Highlights all instances in the current representation that have no equivalent placed in the other representation. Returns: none.

Syntax:

de_show_unmatched();

Example:

```
de_show_unmatched();
```

de_show_unplaced()

Highlights all instances that remain unplaced in the current representation after design synchronization has been run. Returns: none.

Syntax:

de_show_unplaced();

Example:

```
de_show_unplaced();
```

de_snap()

Force the vertices of selected shapes to the nearest snap grid; forces pin 1 of selected instances to nearest grid point. Returns: none.

Syntax:

de_snap([x,y]);

where

x,y is optional. Point within select region of an object to snap.

Example:

```
de_snap();
```

or

```
de_snap(10,20);
```

de_split_tlin()

(For Layout only.) Splits a transmission line element into two of the same elements at a given point, adjusting the new elements length parameters. Works with MLIN and SLIN components. Returns: none.

See also: *de_tap_tlin()*, *de_stretch_tlin()*.

Syntax:

de_split_tlin(x,y);

where

x,y is the point enclosed by a transmission line element (SLIN or MLIN) indicating the element to split as well as the point where the split should occur.

Example:

```
de_split_tlin(10,20);
```

de_step_and_repeat()

Copies and places selected items multiple times in rows and columns. Returns: none.

Syntax:

de_step_and_repeat(x,y);

where

x,y is the location to place copies.

Example:

```
de_set_step_and_repeat(0.5, 0.5, 2, 2, FALSE);
de_step_and_repeat (10, 20);
```

de_store_current_view()

Assigns a name to a view and stores view window coordinates. The view may be recalled by *de_restore_view()*. Returns: none.

See also: *de_restore_view()*, *de_delete_view()*.

Syntax:

de_store_current_view(viewName);

where

viewName is the name assigned to view.

Example:

```
de_store_current_view("myView");
```

de_stretch()

Stretches or moves an edge of a given shape in the current representation. Returns: none.

Syntax:

de_stretch(oldX, oldY, newX, newY, maintainAngle);

where

oldX,oldY is the point on the edge to stretch.

newX,newY indicates the delta and direction to stretch the edge.

maintainAngle is TRUE if the adjacent angles of the edge to stretch are to be kept, FALSE otherwise.

Example:

```
de_stretch(0,0, 20,20, TRUE);
```

de_stretch_dimlin()

Stretches a dimension line. Works with MLIN and SLIN elements. Returns: none.

See also: *de_tap_tlin()*, *de_split_tlin()*.

Syntax:

de_stretch_dimlin(x1, y1, x2, y2);

where

x1, *y1* is the coordinate of the endline to stretch from.

x2, *y2* is the coordinate of the new position for the endline to stretch to.

Example:

```
de_stretch_dimlin(-65, 55, -13.9933, 19.2953);
```

de_stretch_tlin()

(For Layout only) Stretches (increases or decreases its length) a given transmission line element in the current representation. Works with MLIN and SLIN elements. Returns: none.

See also: *de_tap_tlin()*, *de_split_tlin()*.

Syntax:

de_stretch_tlin(oldX, oldY, newX, newY);

where

oldX, *oldY* is the point on edge of transmission line element to stretch.

newX, *newY* is the delta and direction to stretch the edge is determined from this point.

Example:

```
de_stretch_tlin(19.8,18.45, 30.4, 18.45);
```

de_swap_instances()

Replaces all selected instances in the current window with the instance set with the function *de_set_swap_template_instance()*. Returns: none.

See also: *de_set_swap_template_instance()*.

Syntax:

de_swap_instances(replaceID);

where

replaceID signals whether to replace ID of selected instances, where:

FALSE = preserve the item ID
TRUE = create a new ID

Example:

```
//Select all MLINs, replace with TLIN with the same instance name
de_select_by_name("MLIN", 1);
de_set_swap_template_instance("TLIN");
de_swap_instances(FALSE);
```

de_switch_view()

Toggles the view between design view and symbol view for the current Schematic window. Returns: none.

Syntax:

de_switch_view(view[, winInst]);

where

view is the view in current window, where:

DESIGN_VIEW = design view
SYMBOL_VIEW = symbol view

winInst is the window in which the view is to be switched. Optional.

Example:

```
de_switch_view(SYMBOL_VIEW);
```

de_tap_tlin()

(For Layout only) Taps a transmission line element (MLIN or SLIN) in the current representation, and inserts a tee element (MTEE or STEE). The W3 parameter's value is set with the function *de_set_tap_length()*. Returns: none.

See also: *de_stretch_tlin()*, *de_split_tlin()*.

Syntax:

de_tap_tlin(x,y);

where

x,y is the point on a transmission line to split element and insert tee.

Example:

```
de_tap_tlin(23,34);
```

de_unarchive_project()

Unarchives the given file. Takes the given ADS archived file and unarchives it into one or more projects. Returns: none.

Syntax:

de_unarchive_project(fromFileName, toPathName, openProjectAfterUnarchive);

where

fromFileName is the name of the file to unarchive. Should be full path name and should be an ADS archived file.

toPathName is the name of the directory to unarchive the project to. Should be a full path name.

openProjectAfterUnarchive indicates whether to open the project after successfully unarchiving,

where

TRUE = open project
FALSE = do not open project

Example:

```
de_unarchive_project("C:\users\temp\MyAmps_prj.zap",
"C:\users\MyProjectsDir\", TRUE);
```

de_undo()

Undoes the effect of the last draw or edit command in the current window. Returns: none.

Syntax:

de_undo();

Example:

```
de_undo();
```

de_undo_vertex()

Undoes or removes the last vertex entered with the *de_add_point()* command. Returns: none.

Syntax:

de_undo_vertex();

Example:

```
de_undo_vertex();
```

de_unhighlight_instances()

Removes the highlighting of any highlighted instances in the given design. Opposite of *de_highlight_instance()*. Returns: none.

Syntax:

de_unhighlight_instances(designName, repType);

where

designName is the name of the design.

repType is the type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

Example:

```
de_unhighlight_instances("myamp", 1);
```

de_update_tune_parameters()

Modifies tuned parameter values. Returns: none.

See also: *de_analyze_tune()*, *de_tune_deinit()*.

Syntax:

de_update_tune_parameters(designName, repType, valueList list(instName1, paramName1, value1, ...));

where

designName is the name of the design.

repType is the type of representation where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

valueList is a list of triplets list(instName1, paramName1, value1, ...) where:

instName1 = instance ID
paramName1 = parameter name
value1 = new value of parameter as string or real

Example:

```
de_update_tune_parameters("MyDesign", REP_SCHEM, list("TL3", "2", "50 ohm",
"Term1", "2", "50 ohm"));
```

de_variables()

Sets the Top Design for Variables, Top Design Representation Type, and Path to Specific Instance of Design to Layout values for the design pointed to by designP. These values are used during the search for variable declarations for the purpose of artwork generation in layout. If designP is omitted or NULL, the value for the design in the current window is set.

Syntax:

de_variables(designName, repType, pathToDsn[, designP]);

where

designName is the name of a design.

repType is an enumerated type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

pathToDsn is the component path indicating which instance of the component in the top design has actual parameter values wanted for the layout/artwork generation. If the design is not used as a subcircuit of the top design, this field need not be filled in.

designP is a handle to a valid design.

Example:

```
designName=de_current_design_name();
designP=db_get_design(designName);
pathToDsn="MyVars.varset1";
de_variables(designName, REP_SCHEM, pathToDsn, designP);
```

de_vertex_to_arc()

Converts a vertex on a shape to an arc. The arc radius is set with the *de_set_arc_radius()* command. Returns: none.

Syntax:

de_vertex_to_arc([x,y]);

where

x,y is optional. Point within select region of the vertex to convert.

Example:

```
de_vertex_to_arc(23,45);
```

de_view_all()

Expands the view window to view all data in the current window. Returns: none.

Syntax:

de_view_all();

Example:

```
de_view_all();
```

de_zoom_in_point()

Zooms in (double magnification) at the location specified on the current window. Returns: none.

See also: *de_zoom_out_point()*.

Syntax:

de_zoom_in_point(x,y);

where

x,y is the point to zoom in to.

Example:

```
de_zoom_in_point(0,0);
```

de_zoom_in_scale()

Zooms in by factor specified on the current window. Returns: none.

See also: *de_zoom_out_scale()*.

Syntax:

de_zoom_in_scale(scaleFactor);

where

scaleFactor is the integer specifying zoom factor

Example:

```
de_zoom_in_scale(2);
```

de_zoom_out_point()

Zooms out (double magnification) at the location specified on the current window. Returns: none.

See also: *de_zoom_in_point()*.

Syntax:

de_zoom_out_point(x,y);

where

x,y is the point to zoom out from.

Example:

```
de_zoom_out_point(0,0);
```

de_zoom_out_scale()

Zooms out by factor specified on the current window. Returns: none.

See also: *de_zoom_in_scale()*.

Syntax:

de_zoom_out_scale(scaleFactor);

where

scaleFactor is the integer specifying zoom factor

Example:

```
de_zoom_out_scale(2);
```

de_zoom_window()

Describes a region to display in the current window. Returns: none.

Syntax:

de_zoom_window(x1,y1, x2,y2);

where

x1,y1 is the first corner of zoom window.

x2,y2 is the second, opposite corner of zoom window.

Example:

```
de_zoom_window(-20, -30, 50, 60);
```

Chapter 13: Preference Functions

This chapter describes each Preference function in detail. The functions are listed in alphabetical order.

de_add_layer()

Adds a layer definition to the current set of layers. A layer definition consists of a layer name, number, color, fill pattern, line style, plotting mode, visible/invisible flag, protected/unprotected flag, IGES number, GDSII Stream number, layer binding list, and layer type. The order in which layers are added, using this command, determines their priority. Returns: none.

See also: *de_remove_all_layers()*, *de_set_layer()*, *de_get_layer_attribute()*.

Syntax:

de_add_layer(layerName, num, gds2num, igesNum, color, pattern, lineStyle, plotMode, protected, visible, bindingList, layerType);

where

layerName is the layer name. This must be unique—the name *defaults* is reserved. The name can be any number of alpha-numeric characters. The only punctuation character that can be used is an underscore.

num is the layer number. This must be a unique, positive integer. Layer 0 is reserved.

gds2num is the GDSII Stream layer number. This must be a positive integer between 0 and 63.

igesNum is the IGES layer number. This must be a positive integer between 1 and 256.

color is the layer color number. This must be a positive integer. This should be between 0 and the number of colors defined in the *eecolor.cfg* file.

pattern is the layer pattern number. This must be a positive integer. This should be between 0 and the number of fill patterns defined in the *eefill.cfg* file.

lineStyle is the layer line style number; an integer between 0 and 6, where:

0 = solid line
1 = dotted line
2 = double dotted line

3 = short dash line
4 = short dot dash line
5 = long dash line
6 = long dot dash line

plotMode is the layer plot mode number; an integer between 0 and 2, where:

0 = outline plot mode
1 = filled
2 = both

protected is the layer protect status, where:

0 = unprotected
1 = protected

visible is the layer visibility status, where:

0 = the layer is invisible
1 = the layer is visible

bindingList is a comma-separated list of quoted strings which are the names of all layers that a component on the layer being defined are allowed to attach to, where:

Wildcard "*" = any or all layers
Null = no layers

layerType is the type of layer, where:

1=Physical (normal layer)
2=Notes
3=DRC (layer for DRC results)
4=LVS (layer for LVS results)

Example:

```
de_add_layer ("metal2", 10, 14, 14, 6, 3, 2, 1, 0, 0, "*", 1);
```

de_get_layer_names()

Returns the list of layer names of the current window or given design.

Syntax:

de_get_layer_names([designHandle]);

where

designHandle is optional; handle to a design. If not given, the current window is used.

Example:

```
decl layerList;
layerList=de_get_layer_names();
```

de_get_layer_attribute()

Given a layer name in that set, returns a layer attribute value integer or string for the current layer set, which represents the layer attribute value requested.

See also: *de_add_layer()*.

Syntax:

de_get_layer_attribute(layerName, layerAttribute);

where

layerName is the name of a layer defined in the current layers file.

layerAttribute is the layer name. Can be one of these layer attributes:

LAYER_NUMBER = Returns the layer number.

LAYER_STREAM_NUM = Returns the GDSII stream layer number.

LAYER_IGES_NUM = Returns the IGES layer/level number.

LAYER_COLOR = Returns the layer color index (into eecolor.cfg).

LAYER_FILL = Returns the layer fill index (into eepattern.cfg).

LAYER_LINE_TYPE = Returns the line type number, where:

0 = solid line
1 = dotted line
2 = double dotted line
3 = short dash line
4 = short dot dash line
5 = long dash line
6 = long dot dash line

LAYER_PLOT_MODE = Returns the plot mode, where:

0 = outline plot mode
1 = filled
2 = both

LAYER_PROTECTED = The layer protection flag, where:

1 = protected
0 = not protected.

LAYER_VISIBLE = The layer visible flag, where:

1 = visible
0 = not visible.

LAYER_BINDING_LIST = Returns a string which is the list of layers that components on this layer can attach to.

LAYER_TYPE = The layer type, where:

1 = Physical (normal layer)
2 = Notes
3 = DRC (layer for DRC results)
4 = LVS (layer for LVS results)

Example:

```
decl color;
color = de_get_layer_attribute( "cond1", LAYER_COLOR);
```

de_get_preference()

Returns the value of a current preference, the preference setting.

See also: *de_add_layer()*.

Syntax:

de_get_preference(preference, [repType | repHandle]);

where

preference is an integer or predefined preference variable, as listed in Table 13-1.

repType is the type of representation, where:

REP_SCHEM = schematic representation
REP_LAY = layout representation

repHandle is the handle of a design representation

Table 13-1. Preference Variables for de_get_preference()

Path corner types: PREF_MITERED_PATH PREF_SQUARE_PATH PREF_CURVED_PATH
Entry modes: PREF_ENTRY_SNAP_NONE PREF_ENTRY_90_SNAP PREF_ENTRY_45_SNAP
Select filter (bits can be or'd together): PREF_NONE_SELECT_FILT PREF_ELEMENT_SELECT_FILT PREF_WIRE_SELECT_FILT PREF_POLYGON_SELECT_FILT PREF_POLYLINE_SELECT_FILT PREF_PATH_SELECT_FILT PREF_TEXT_SELECT_FILT PREF_ARC_SELECT_FILT PREF_CIRCLE_SELECT_FILT PREF_PORT_SELECT_FILT PREF_FORMAT_SELECT_FILT PREF_POINT_SELECT_FILT PREF_ALL_SELECT_FILT
Grid snapping types: PREF_SNAP_TO_GRID PREF_SNAP_TO_PIN PREF_SNAP_TO_EDGE PREF_SNAP_TO_VERTEX PREF_SNAP_TO_ARC_CENTER PREF_SNAP_TO_INTERSECT PREF_SNAP_TO_MIDPOINT
Grid display types: PREF_GRID_DOTS PREF_GRID_LINES
Placement modes: PREF_DUAL_PLACE_OFF PREF_DUAL_PLACE_SINGLE PREF_DUAL_PLACE_DSE
Selection mode types: PREF_SELECT_UNSEL_FIRST PREF_SELECT_OUTSIDE_WIN

Table 13-1. Preference Variables for de_get_preference() (continued)

Selection mode for polygons: PREF_SELECT_MODE_ON PREF_SELECT_MODE_INSIDE
Trace conversion simulation mode types: PREF_TRACE_SIM_TLIN PREF_TRACE_SIM_SINGLE PREF_TRACE_SIM_NODAL
Trace conversion technology types: PREF_TRACE_TECH_MICROSTRIP PREF_TRACE_TECH_STRIPLINE PREF_TRACE_TECH_PCB

Table 13-1. Preference Variables for de_get_preference() (continued)

Preference attribute types:	
CLEARANCE_P	MAJOR_GRID_DISPLAY_P
STEP_REPEAT_XSPACE_P	GRID_DISPLAY_P
STEP_REPEAT_YSPACE_P	GRID_DISPLAY_MODE_P
STEP_REPEAT_NUMROWS_P	GRID_COLOR_P
STEP_REPEAT_NUMCOLS_P	GRID_SNAP_X_P
TAP_LENGTH_P	GRID_SNAP_Y_P
SCALE_X_P	GRID_DISPLAY_X_P
SCALE_Y_P	GRID_DISPLAY_Y_P
OVERSIZE_P	MAJOR_GRID_DISPLAY_X_P
MITER_ANGLE_P	MAJOR_GRID_DISPLAY_Y_P
TO_ARC_RADIUS_P	PIN_SNAP_SIZE_P
MITER_VERTEX_LENGTH_P	PIN_SNAP_UNITS_P
PORT_NAME_P	GRID_SNAP_P
PORT_NUMBER_P	GRID_SNAP_MODE_P
PORT_ORIENT_P	TRACE_SIM_MODE_P
PORT_TYPE_P	TRACE_SINGLE_ELEM_P
PATH_WIDTH_P	TRACE_TECH_P
PATH_BEND_P	TRACE_MSUB_ID_P
PATH_END_P	DUAL_PLACEMENT_P
PATH_RADIUS_P	PLACE_POPUP_P
PATH_MITER_PERCENT_P	PLACE_POPUP_ON_ZERO_PARM_P
SELECT_FILTER_P	SWAP_KEEP_INST_NAME
SELECT_MODE_P	PORT_SIZE_P
SELECT_BOX_SIZE_P	PORT_SIZE_UNITS_P
SELECT_BOX_UNITS_P	PIN_SIZE_P
SELECT_POINT_SIZE_P	PIN_SIZE_UNITS_P
SELECT_POINT_UNITS_P	TEE_SIZE_P
SELECT_COLOR_P	TEE_SIZE_UNITS_P
PIN_COLOR_P	FG_COLOR_P
TEE_COLOR_P	BG_COLOR_P
NODE_VOLT_COLOR_P	HIGHLIGHT_COLOR_P
PIN_CURRENT_COLOR_P	DVE_REAL_MEMORY_P
NODE_NAME_COLOR_P	DVE_STORAGE_PER_AREA_P
PLOT_PINS_P	DVE_EPSILON_P
PLOT_PIN_NUMBERS_P	DVE_FRINGE_P
PLOT_PIN_NAMES_P	DVE_BIN_WIDTH_P
	(continued)

Table 13-1. Preference Variables for de_get_preference() (continued)

Preference attribute types: (continued)	INSTANCE_NAME_P
ENTRY_MODE_P	INSTANCE_ID_P
COORD_ENTRY_POPUP_P	INSTANCE_ANGLE_P
CHECK_INTERSECTION_P	BBOX_COLOR_P
KEEP_NODE_NAMES_P	VIEW_TYPE_P
REROUTE_WIRES_P	PORT_COLOR_P
ROUTE_AROUND_INST_TEXT_P	SCHEM_PREC_P
CHECK_BINDING_P	SCHEM_INCR_P
MIN_VERTEX_DIST_P	SCHEM_UNITS_P
GLOBAL_ARC_RESOLUTION_P	LAYOUT_PREC_P
BACKUP_COUNT_P	LAYOUT_INCR_P
UNDO_EDIT_COUNT_P	LAYOUT_UNITS_P
ROTATION_INC_P	DVE_MAX_ERROR_P
ROUTE_DIST_SIZE_P	INSTANCE_MIRROR_P
ROUTE_DIST_UNITS_P	INSTANCE_AUTO_ROTATE_P
DRAG_MOVE_P	EQUIV_ELEM_NAME_P
DRAG_MOVE_THRESHOLD_SIZE_P	EQUIV_ELEM_ID_P
DRAG_MOVE_THRESHOLD_UNITS_P	ELEM_FIXED_P
INST_TEXT_FONT_P	GDS_NUM_PNTS_ARC_P
INST_TEXT_HEIGHT_P	WINDOW_LL_X
INST_NAME_LAYER_P	WINDOW_LL_Y
INST_ID_LAYER_P	WINDOW_UR_X
INST_PARAM1_LAYER_P	WINDOW_UR_Y
INST_TEXT_PREC_P	AUTOMATIC_DSE_P
INST_TEXT_ROWS_P	DSE_SYMB_X_DISTANCE_P
TEXT_FONT_P	DSE_SYMB_Y_DISTANCE_P
TEXT_HEIGHT_P	DSE_ART_X_DISTANCE_P
TEXT_JUST_P	DSE_ART_Y_DISTANCE_P
TEXT_ANGLE_P	DSE_S2L_REPORT_P
TEXT_ABSOLUTE_P	DSE_L2S_REPORT_P
PLOTTING_DEPTH_P	DISP_SUBNET_INST_NAMES_P
FORCE_DELETE_P	CHECK_UNCONNECTED_PINS_ P
AUTO_UPDATE_OPT_P	CHECK_NODAL_MISMATCH_P
SIMULATOR_WARNINGS_OPT_P	CHECK_WIRES_IN_LAYOUT_P
SHOW_CONNECTED_SCHEM_P	CHECK_PIN_VS_PORT_P
SHOW_CONNECTED_LAY_P	(continued)

Table 13-1. Preference Variables for de_get_preference() (continued)

Preference attribute types: (continued) TUNE_MODE_P TUNE_HISTORY_SIZE_P TUNE_RANGE_P TUNE_STEP_SIZE_P TUNE_SCALE_P **Preference value types (units):** UNITS_FREQ_P UNITS_RES_P UNITS_COND_P UNITS_IND_P UNITS_CAP_P UNITS_LNG_P UNITS_TIME_P UNITS_ANG_P UNITS_POWER_P UNITS_VOLT_P UNITS_CUR_P UNITS_DIST_P
Preference value types (int, float, string): PREF_TYPE_FLOAT_VALUE PREF_TYPE_INT_VALUE PREF_TYPE_STRING_VALUE

Example:

```
decl cornerType;
cornerType = de_get_preference(PATH_BEND_P);
if (cornerType==PREF_MITERED_PATH)
    fputs(stderr, “mitered”);
else if (cornerType==PREF_SQUARE_PATH)
    fputs(stderr, “square”);
else
    fputs(stderr, “curved”);
```

de_read_layer()

Reads a layer file from disk, re-setting the layers in the current window. Returns: none.

Syntax:

de_read_layer(layerFileName);

where

layerFileName is a string; name of the layer file to read.

Example:

```
de_read_layer("mylayer.lay");
```

de_read_preferences()

Reads a preference file from disk, re-setting the preferences in the current window. Returns: none.

Syntax:

de_read_preferences(preferenceFileName);

where

preferenceFileName is a string; name of the preference file to read.

Example:

```
de_read_preferences("myprefs.prf");
```

de_remove_all_layers()

Deletes all current layer attribute descriptions for the active window. Returns: none.

See also: *de_add_layer()*, *de_set_layer()*.

Syntax:

de_remove_all_layers();

Example:

```
de_remove_all_layers();
```

de_set_annotation_font()

Sets the font for instance parameter annotation for the current window. Returns: none.

Syntax:

de_set_annotation_font(fontNumber);

where

fontNumber is an integer indicating text font number, where:

0 = Hershey Roman
1 = Hershey Roman Narrow

Example:

```
de_set_annotation_font(1);
```

de_set_annotation_height()

Sets the text height used for component parameter annotation. Returns: none.

Syntax:

de_set_annotation_height(textHeight);

where

textHeight is the text height in user units.

Example:

```
de_set_annotation_height(0.1);
```

de_set_annotation_id_layer()

Sets the layer number for instance name (ID) parameter annotation for the current window. The layer instance ID, design name and parameters are all separately specified. Returns: none.

Syntax:

de_set_annotation_id_layer(layerNum);

where

layerNum is the layer number (≥ 0).

Example:

```
de_set_annotation_layer(4);
```

de_set_annotation_name_layer()

Sets the layer for instance name annotation for the current window. Returns: none.

Syntax:

de_set_annotation_name_layer(layerNum);

where

layerNum is the layer number (≥ 0).

Example:

```
de_set_annotation_name_layer(5);
```

de_set_annotation_parameters_layer()

Sets the layer number for instance parameter annotation for the current window. Returns: none.

Syntax:

de_set_annotation_parameters_layer(layerNum);

where

layerNum is the layer number to place parameter annotation on (≥ 0).

Example:

```
de_set_annotation_parameters_layer(6);
```

de_set_annotation_precision()

Sets the display precision of real numbers displayed in schematic annotation. This is the number of digits displayed to the right of the decimal point for real numbers. This does not affect precision transmitted to the simulator. Returns: none.

Syntax:

de_set_annotation_precision(precision);

where

precision is the number of digits to the right of the decimal point.

Example:

```
de_set_annotation_precision(3);
```

de_set_annotation_rows()

Sets the number of rows for instance parameter annotation for the current window. Instances with more parameters than this number have parameters placed in columns set the left of the first column. Returns: none.

Syntax:

de_set_annotation_rows(numRows);

where

numRows is an integer (> 0), describing number of rows per column of instance parameter annotation.

Example:

```
de_set_annotation_rows(12);
```

de_set_arc_radius()

Set the radius for any arcs created using the *de_vertex_to_arc()* command, which converts a vertex of a shape to an arc for the current window. Returns: none.

Syntax:

de_set_arc_radius(radius);

where

radius is the radius of arc in degrees.

Example:

```
de_set_arc_radius(20.0);
```

de_set_background_color()

Sets the background color of the drawing area in the current window. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_background_color(colorNum);

where

colorNum is the number of defined color (integer > 0).

Example:

```
de_set_background_color(14);
```

de_set_backup_count()

Sets the number of edits that must be performed before a backup file is made. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_backup_count(count);

where

count is an integer; the number of editing operations to perform before automatically writing a .bak backup file of the current design.

Example:

```
de_set_backup_count(10);
```

de_set_coord_entry_popup()

Sets whether the Coordinate Entry dialog will be displayed for draw and place commands. Returns: none.

Syntax:

de_set_coord_entry_popup(onOffFlag);

where

onOffFlag signals displaying Coordinate Entry dialog, where:

TRUE = display
FALSE = do not display

Example:

```
de_set_coord_entry_popup(TRUE);
```

de_set_coordinate_readout_mode()

Sets the display of coordinate readout in the status panel of the active window on or off. Returns: none.

Syntax:

de_set_coordinate_readout_mode(onOffFlag);

where

onOffFlag is the setting for display of coordinate readout, where:

0 = off
1 = on

Example:

```
de_set_coordinate_readout_mode(0);
```

de_set_curve_radius()

Sets the curve radius in the current window used for paths and traces with rounded corners. Returns: none.

Syntax:

de_set_curve_radius(radius);

where

radius is the radius in degrees.

Example:

```
de_set_curve_radius(20.4);
```

de_set_drag_move()

Sets the ability to drag and move objects with the left mouse button. Returns: none.

See also: *de_set_drag_move_size()*, *de_set_drag_move_units()*.

Syntax:

de_set_drag_move(mode);

where

mode is the type of ability to drag and move objects with left mouse button, where:

0 = off
1 = on

Example:

```
de_set_drag_move(1);
```

de_set_drag_move_size()

Sets the minimum distance an object must be dragged before the object is moved. Returns: none.

See also: *de_set_drag_move_units()*.

Syntax:

de_set_drag_move_size(size);

where

size is the size for distance (in user units).

Example:

```
de_set_drag_move_size(12.4);
```

de_set_drag_move_units()

Sets the units of the drag move. Returns: none.

See also: *de_set_drag_move_size()*.

Syntax:

de_set_drag_move_units(unit);

where

unit is the type of units, where:

0 = user units
1 = screen pixels

Example:

```
de_set_drag_move_units(1);  //sets to screen pixels
```

de_set_dse_start()

Sets the starting instance for design synchronization in the current representation. Returns: none.

Syntax:

de_set_dse_start(x,y);

where

x,y is the point within select region of starting instance.

Example:

```
de_set_dse_start(10,20);
```

de_set_dual_placement()

Sets the mode so that when enabled, causes any component placed in one representation to be automatically placed in the other representation. Returns: none.

Syntax:

de_set_dual_placement(dualPlaceMode);

where

dualPlaceMode is the placement choice, where:

PREF_DUAL_PLACE_DSE = automatically perform a design synchronization after each part is placed

PREF_DUAL_PLACE_SINGLE = automatically place connected equivalent component in other representation

PREF_DUAL_PLACE_OFF = do not place automatically

Example:

```
de_set_dual_placement(PREF_DUAL_PLACE_SINGLE);
```

de_set_foreground_color()

Sets the sketch color of the active window. Sets the color of the sketch mode for polygons, polylines, wires, traces, selection outlines, etc. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_foreground_color(color);

where

color is the color number, an integer ≥ 0.

Example:

```
de_set_foreground_color(2);
```

de_set_global_db_factor()

Sets the simulator units to layout user unit conversion factor used during rendering of AEL artwork objects.

See also: Figure 4-9.

Syntax:

de_set_global_db_factor();

Example:

```
de_set_global_db_factor();
```

de_set_grid_color()

Sets the color of the visible grid in the current window. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_grid_color(color);

where

color is the color number, an integer ≥ 0.

Example:

```
de_set_grid_color(5);
```

de_set_grid_display_type()

Sets the grid display to dots or dotted line for the current window. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_grid_display_type(type);

where

type is the type of grid display, where;

PREF_GRID_DOTS = dotted
PREF_GRID_LINES = lines

Example:

```
de_set_grid_display_type(PREF_GRID_DOTS);
```

de_set_grid_snap()

Sets grid snap increments in X and Y direction and turns grid snapping for the current window on or off . Returns: none.

Syntax:

de_set_grid_snap(SnapX, SnapY, onOffFlag);

where

SnapX is the cursor snapping measurement in X direction.

SnapY is the cursor snapping measurement in Y direction.

onOffFlag signals grid snap setting, where:

0 = Off
1 = On

Example:

```
de_set_grid_snap(1);
```

de_set_grid_snap_mode()

Turns grid snapping for the current window on or off . Returns: none.

Syntax:

de_set_grid_snap(mode);

where

Mode enables or disables grid snapping, where:

TRUE = Enable
FALSE = Disable

Example:

```
de_set_grid_snap_mode(TRUE);
```

de_set_grid_snap_type()

Sets the snap type to one or more of grid, pin, or vertex for the current window. Returns: none.

Syntax:

de_set_grid_snap_type(type);

where

type is the type of snap, where:

PREF_SNAP_TO_GRID = cursor will snap to the nearest grid point

PREF_SNAP_TO_PIN = cursor will snap to the nearest pin

PREF_SNAP_TO_VERTEX = cursor will snap to the nearest vertex on a shape

PREF_SNAP_TO_EDGE = cursor will snap and slide along the nearest edge of a shape

PREF_PREF_SNAP_TO_MIDPOINT = cursor will snap to the midpoint of a line if the midpoint is within the snap radius

PREF_SNAP_TO_ARC_CENTER = cursor will snap to the center of an arc or circle

SNAP_TO_INTERSECT = cursor will snap to the intersection of lines or edges of two different objects

---

Note For details on grid snap mode behavior, refer to the *Layout* manual.

Example:

```
de_set_grid_snap_type(PREF_SNAP_TO_GRID|PREF_SNAP_TO_PIN);
```

de_set_highlight_color()

Sets the highlight color for the current window (used for errors, and show commands). Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_highlight_color(color);

where

color is the color for highlighting, an integer ≥ 0.

Example:

```
de_set_highlight_color(13);
```

de_set_layer()

Sets the entry layer for the current layer for the current window. All subsequent shapes and text are added to this layer, until reset. Returns: none.

See also: *de_add_layer()*, *de_remove_all_layers()*.

Syntax:

de_set_layer(layerNum);

where

layerNum is the layer to set, an integer ≥ 0.

Example:

```
de_set_layer(15);
```

de_set_major_grid_display()

Sets the major grid display factor. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_major_grid_display(xFactor, yFactor, onOff);

where

xFactor is the multiple of the snap grid in X.

yFactor is the multiple of the snap grid in Y.

onOff is the grid display options, where:

TRUE = Display major grid
FALSE = Do not display major grid

Example:

```
de_set_major_grid_display(8, 8, TRUE);
```

de_set_minor_grid_display()

Sets the minor grid display factor to *xFactor* multiple of the snap grid in X and *yFactor* multiple of the snap grid in Y. Returns: TRUE if successful, FALSE if not successful.

Syntax:

de_set_minor_grid_display(xFactor, yFactor, onOff);

where

xFactor is a multiple of the snap grid in X.

yFactor is a multiple of the snap grid in Y.

onOff is the grid display options, where:

TRUE = Display minor grid
FALSE = Do not display minor grid

Example:

```
de_set_minor_grid_display(10, 10, TRUE);
```

de_set_miter_cutoff()

Sets the miter cutoff angle for the path and trace command. Angles less than the given number, result in the point being mitered. Returns: none.

Syntax:

de_set_miter_cutoff(miterAngle);

where

miterAngle is the cutoff angle in degrees.

Example:

```
de_set_miter_cutoff(30.0);
```

de_set_miter_length()

Sets the miter length. It is often used in the *de_miter_vertex()* command, where both edges of a vertex need to be larger than the miter length for the command to be successful. Returns: none.

Syntax:

de_set_miter_length(length);

where

length is the length in user units (> 0).

Example:

```
de_set_miter_length(12.4);
```

de_set_oversize()

Sets the amount to oversize or undersize closed shapes for the *de_oversize()* command for the current window. Returns: none.

Syntax:

de_set_oversize(oversizeAmount, angle);

where

oversizeAmount is the amount to size shapes, where:

positive numbers = oversize shapes
negative numbers = undersize shapes

angle is the angle in degrees.

Example:

```
de_set_oversize(-5, 45);
```

de_set_path_corner()

Sets the corner type used for paths and traces in the current window. Types can be square, mitered or curved. Returns: none.

Syntax:

de_set_path_corner(type);

where

type is the corner type where:

PREF_MITERED_PATH = mitered
PREF_SQUARE_PATH = square
PREF_CURVED_PATH = curved

Example:

```
de_set_path_corner(PREF_MITERED_PATH);
```

de_set_path_width()

Sets the width for paths and traces used by the *de_add_path()* and *de_add_trace()* entry commands for the current window. Returns: none.

Syntax:

de_set_path_width(width);

where

width is the path width in user units (> 0).

Example:

```
de_set_path_width(10.8);
```

de_set_pin_color()

Sets the color used for instance pins for the current window. Returns: none.

Syntax:

de_set_pin_color(colorNum);

where

colorNum is the color number; an integer ≥ 0.

Example:

```
de_set_pin_color(12);
```

de_set_pin_size()

Sets the size used to display instance pins of the current window. Instance pins are drawn as squares; this controls their size. Returns: none.

See also: *de_set_pin_size_units()*.

Syntax:

de_set_pin_size(size);

where

size is the size to draw pins.

Example:

```
de_set_pin_size(12.4);
```

de_set_pin_size_units()

Sets the units for the pin size set by *de_set_pin_size()*. Returns: none.

See also: *de_set_pin_size()*.

Syntax:

de_set_pin_size_units(unit);

where

unit is the type of units for pin size setting, where:

0 = user units
1 = screen pixels

Example:

```
de_set_pin_size_units(1);  //sets to screen pixels
```

de_set_pin_snap()

Sets the maximum distance at which the cursor snaps to a pin when snap type includes PREF_SNAP_TO_PIN. Returns: none.

See also: *de_set_pin_snap_units()*.

Syntax:

de_set_pin_snap(snapDist);

where

snapDist is the maximum distance at which cursor snaps to a pin.

Example:

```
de_set_pin_snap(10.0);
```

de_set_pin_snap_units()

Sets the units for the pin snap distance set by *de_set_pin_snap()*. Returns: none.

See also: *de_set_pin_snap()*.

Syntax:

de_set_pin_snap_units(unit);

where

unit is the type of units for snap distance.

0 = user units
1 = screen pixels

Example:

```
de_set_pin_snap_units(1);  //sets to screen pixels
```

de_set_place_popup_mode()

Sets mode for displaying the component Parameter dialog box for the current window. Returns: none.

Syntax:

de_set_place_popup_mode(onOffFlag);

where

onOffFlag signals dialog popup when instance is placed, where:

0 = do not popup dialog
1 = popup dialog

Example:

```
de_set_place_popup_mode(1);
```

de_set_place_popup_on_zero_parm()

Sets mode for the current window for displaying the Component Parameter dialog box for components with no parameters. Returns: none.

See also: *de_get_preference()*.

Syntax:

de_set_place_popup_on_zero_parm(onOffFlag);

where

onOffFlag signals dialog popup when a component with no parameters is chosen, where:

TRUE = popup dialog
FALSE = do not popup dialog

Example:

```
de_set_place_popup_on_zero_parm(TRUE);
```

de_set_plot_pin_names()

Turns the display of pin names on or off. Returns: none.

Syntax:

de_set_plot_pin_names(onOff);

where

onOff signals displaying of pins, where:

FALSE = pin names do not display
TRUE = pin names display

Example:

```
de_set_plot_pin_names("TRUE");
```

de_set_plot_pin_numbers()

Turns the plotting of pin numbers on and off for the current window. Returns: none.

Syntax:

de_set_plot_pin_numbers(onOffFlag);

where

onOffFlag signals plotting of pins, where:

0 = do not plot pin numbers
1 = plot them

Example:

```
de_set_plot_pin_numbers(0);
```

de_set_plot_pins()

Turns plotting of connected pins on or off for the current window. Returns: none.

Syntax:

de_set_plot_pins(onOffFlag);

where

onOffFlag signals plotting of pins, where:

0 = plot all pins
1 = plot only unconnected pins

Example:

```
de_set_plot_pins(1);
```

de_set_plotting_depth()

Sets the hierarchical level at which component instances in Layout are drawn in outline. Returns: none.

Syntax:

de_set_plotting_depth(depth);

where

depth is the depth is an integer greater than 1.

Example:

```
de_set_plotting_depth(4);
```

de_set_port_size()

Sets the size of ports in the Layout representation. Returns: none.

See also: *de_set_port_size_units()*.

Syntax:

de_set_port_size(size);

where

size is the size to draw ports.

Example:

```
de_set_port_size(12.4);
```

de_set_port_size_units()

Sets the units for the port size set by *de_set_port_size()*. Returns: none.

See also: *de_set_port_size()*.

Syntax:

de_set_port_size_units(unit);

where

unit is the type of units for port size setting, where:

0 = user units
1 = screen pixels

Example:

```
de_set_port_size_units(1);  //sets to screen pixels
```

de_set_preference()

Sets the value of any preference. Refer to *de_get_preference()* for preferences and values. Returns: none.

See also: *de_get_preference()*, *de_set_resolution_for_arc()*, *de_set_undo_edit_count()*.

Syntax:

de_set_preference(preference, prefValue, [repType | repHandle]);

where

preference is an integer or predefined preference variable.

prefValue is an integer or predefined preference value variable.

repType is the type of representation, where:

REP_SCHEM = schematic representation
REP_LAY = layout representation

repHandle is the handle of a design representation

Example:

```
// Set the Trace conversion technology type to PCB
// In Schem/Lay window Options > Preferences Trace tab,
// under Element Set
de_set_preference(TRACE_TECH_P, PREF_TRACE_TECH_PCB);
```

de_set_reroute_wires()

Sets wire or trace editing route mode for the current window. When a wire or trace is re-routed after an instance is moved, the wire or trace can be completely re-routed or re-routed only from its end point. Returns: none.

Syntax:

de_set_reroute_wires(routeMode);

where

routeMode signals routing of wires or traces, where:

0 = reroute entire wire
1 = route from end point only

Example:

```
de_set_reroute_wire(1);
```

de_set_resolution_for_arc()

Sets the resolution that is used for approximating an arc when shapes with true arcs are converted to polygons with the *de_convert_to_polygon()* command for the current window. Returns: none.

Syntax:

de_set_resolution_for_arc(degree);

where

degree is the number for resolution, in degrees.

Example:

```
de_set_resolution_for_arc(5.0);
```

de_set_rotation_increment()

Sets the step increment used by the *de_rotate()* command for the current window. Returns: none.

Syntax:

de_set_rotation_increment(angle);

where

angle is the angle in degrees for the step (≥ 0, < 360).

Example:

```
de_set_rotation_increment(45.0);
```

de_set_route_around_annot()

When the Wire Route command is in progress, this mode determines if the wire should route around or through annotations. Returns: none.

Syntax:

de_set_route_around_annot(routeAroundMode);

where

routeAroundMode is the mode for routing wire, where:

0 = route through annotation
1 = route around annotation

Example:

```
de_set_route_around_annot(0);
```

de_set_route_dist()

Sets the minimum distance from obstacles that the Wire Route command will place wires. Returns: none.

See also: *de_set_route_dist_units()*.

Syntax:

de_set_route_dist(routeDist);

where

routeDist is the minimum distance from obstacles.

Example:

```
de_set_route_dist(10.0);
```

de_set_route_dist_units()

Sets the units for the minimum distance for the Wire Route command to be in user units or screen pixels. Returns: none.

See also: *de_set_route_dist()*.

Syntax:

de_set_route_dist_units(unit);

where

unit is the type of units for route distance setting, where:

0 = user units
1 = screen pixels

Example:

```
de_set_route_dist_units(1);  //sets to screen pixels
```

de_set_scale()

Set the X- and Y-scale factors used by the *de_scale()* command in the current window. Returns: none.

Syntax:

de_set_scale(sx, sy);

where

sx,sy is the X and Y scale factors (> 0).

Example:

```
de_set_scale(.5,.5);
```

de_set_select_box_size()

Sets the select region size for the current window. Objects within a square of the size specified here are selected when any selection command is given. Returns: none.

See also: *de_set_select_box_units()*.

Syntax:

de_set_select_box_size(size);

where

sized describes size of select region (side of select region square); size > 0.

Example:

```
de_set_select_box_size(4.0);
```

de_set_select_box_units()

Sets the units for the select region size, set by *de_set_select_box_size()*, at the current window. Returns: none.

See also: *de_set_select_box_size()*.

Syntax:

de_set_select_box_units(unit);

where

unit is the type of units for select region setting, where:

0 = user units
1 = screen pixels

Example:

```
de_set_select_box_units(1);  //sets to screen pixels
```

de_set_select_color()

Sets the color used to display selected objects for the current window. Returns: TRUE if successful; FALSE if not successful.

Syntax:

de_set_select_color(color);

where

color is the color to use for selected objects; an integer ≥ 0.

Example:

```
de_set_select_color(10);
```

de_set_select_filter()

Sets the select filter mask for the current window. This controls what type of objects can be selected by the select commands. Returns: none.

Syntax:

de_set_select_filter(mask);

where

mask is one or more of the following:

PREF_ARC_SELECT_FILT
PREF_CIRCLE_SELECT_FILT
PREF_ELEMENT_SELECT_FILT
PREF_FORMAT_SELECT_FILT
PREF_PATH_SELECT_FILT
PREF_POINT_SELECT_FILT
PREF_POLYGON_SELECT_FILT
PREF_POLYLINE_SELECT_FILT
PREF_PORT_SELECT_FILT
PREF_TEXT_SELECT_FILT

PREF_WIRE_SELECT_FILT
PREF_NONE_SELECT_FILT
PREF_ALL_SELECT_FILT

Example:

```
de_set_select_filter
  (PREF_POLYGON_SELECT_FILT|PREF_POLYLINE_SELECT_FILT);
// this means only polygons and polylines can be selected
```

de_set_select_inside_polygon()

Sets the selection mode. Objects can be selected when the given point is enclosed by any closed shape (the default in Schematic window) or when any edge of an object is within the select region. Returns: none.

Syntax:

de_set_select_inside_polygon(onOffFlag);

where

onOffFlag signals selection of objects, where:

PREF_SELECT_MODE_ON = 0
PREF_SELECT_MODE_INSIDE = 1

Example:

```
de_set_select_inside_polygon(PREF_SELECT_MODE_INSIDE);
```

de_set_select_point_size()

Sets the size of the selected vertices in the current window. Returns: none.

See also: *de_set_select_point_size_units()*.

Syntax:

de_set_select_point_size(size);

where

size describes size of selected vertex (2 • size = side of select region square); size > 0.

Example:

```
de_set_select_point_size(4.0);
```

de_set_select_point_size_units()

Sets the units for the selected vertices in the current window. Returns: none.

See also: *de_set_select_point_size()*.

Syntax:

de_set_select_point_size_units(unit);

where

unit is the type of units for select point size setting, where:

0 = user units
1 = screen pixels

Example:

```
de_set_select_point_size_units(1);  //sets to screen pixels
```

de_set_self_intersect()

Sets the polygon self intersection checking for the current window. Returns: none.

Syntax:

de_set_self_intersect(onOffFlag);

where

onOffFlag is the setting for polygon self intersection checking, where:

0 = off
1 = on

Example:

```
de_set_self_intersect(1);
```

de_set_shape_entry_mode()

Sets the entry mode in the current window for polygons and polylines to orthogonal or non-orthogonal. Returns: none.

Syntax:

de_set_shape_entry_mode(mode);

where

mode is the type of entry mode, where:

0 = non-orthogonal
1 = orthogonal

Example:

```
de_set_shape_entry_mode(1);
```

de_set_step_and_repeat()

Sets variables for the Step and Repeat command. Returns: none.

Syntax:

de_set_step_and_repeat(xSpacing, ySpacing, numRows, numCols[, connectFlag]);

where

xSpacing is the space between columns.

ySpacing is the space between rows.

numRows is the number of rows.

numCols is the number of columns.

connectFlag is an optional parameter, defaulted to FALSE. Set to TRUE if connection of overlapping pins and wire ends is desired. Note: in a large design, this increases the processing time.

Example:

```
de_step_and_repeat(0.125, 0.125, 2, 2, FALSE);
```

de_set_tap_length()

Sets the length (w3 parameters) of the tee element produced when a transmission line element is tapped with the *de_tap_tlin()* command. Returns: none.

Syntax:

de_set_tap_length(length);

where

length is the length in simulator units (> 0). For valid ranges for the M3 parameter, refer to the data sheet for *mtee* or *stee* elements.

Example:

```
de_set_tap_length(12.4);
```

de_set_tee_color()

Sets the color of schematic tees (interconnect dots) for the current window. Returns: none.

Syntax:

de_set_tee_color(colorNum);

where

colorNum is the color to use for tees; an integer ≥ 0.

Example:

```
de_set_tee_color(12);
```

de_set_tee_size()

Sets the size of tees (interconnect dots) used in schematics (for the current window). Returns: none.

See also: *de_set_tee_size_units()*.

Syntax:

de_set_tee_size(size);

where

size is the size (tees are squares, this specifies length of a side) in user-defined units.

Example:

```
de_set_tee_size(50.0);
```

de_set_tee_size_units()

Sets the type of units of the tee size. Returns: none.

See also: *de_set_tee_size()*.

Syntax:

de_set_tee_size_units(unit);

where

unit is the type of units for tee size setting, where:

0 = user units
1 = screen pixels

Example:

```
de_set_tee_size_units(1);  //sets to screen pixels
```

de_set_text_absolute()

Sets absolute characteristics for text.

Syntax:

de_set_text_absolute(onOFF);

where

onOFFsignals absolute characteristics for text, where:

0 = text rotates relative to custom symbol
1 = text does not rotate relative to custom symbol

Example:

```
de_set_text_absolute(1);
//text does not rotate when custom symbol is rotated when placed in design
```

de_set_text_angle()

Sets the angle in which text is placed for the current window. Returns: none.

Syntax:

de_set_text_angle(angle);

where

angle is the angle in degrees (≥ 0).

Example:

```
de_set_text_angle(12.7);
```

de_set_text_font()

Sets the font type when placing text for the current window. Returns: none.

Syntax:

de_set_text_font(fontNum);

where

fontNum is the integer indicating text font number, where:

0 = Hershey Roman
1 = Hershey Roman Narrow

Example:

```
de_set_text_font(0);
```

de_set_text_height()

Sets the height for placed text (in the current window). Returns: none.

Syntax:

de_set_text_height(height);

where

height is the height in user units (> 0).

Example:

```
de_set_text_height(12.4);
```

de_set_text_justification()

Sets the default text justification. Used for all subsequent text placement. Returns: none.

Syntax:

de_set_text_justification(justification);

where

justification is an “or’d” combination of one of these:

DB_BOT_JUST
DB_MID_JUST
DB_TOP_JUST

with one of these:

DB_LEFT_JUST
DB_CENTER_JUST
DB_RIGHT_JUST

Example:

```
de_set_text_justification(DB_BOT_JUST | DB_LEFT_JUST);
```

de_set_text_string()

Sets the text string used by the *de_add_text()* command for the current window. This is the string that is placed when the *de_add_text()* command is issued without a text string. Returns: none.

See also: *de_add_text()*

Syntax:

de_set_text_string(string);

where

string is the text string enclosed in quotes.

Example:

```
de_set_text_string("This is a text string");
```

de_set_trace_sim_mode()

Sets the trace simulation mode for the current window. Traces can be simulated as a simple interconnection (short), as a single element or as a series of interconnected transmission line elements with discontinuity. Currently, this command has no effect. Traces are simulated as simple nodal interconnections (shorts). Returns: none.

Syntax:

de_set_trace_sim_mode(mode);

where

mode is the choice for trace simulation, where:

0 = short
1 = single element
2 = full discontinuities

Example:

```
de_set_trace_sim_mode(1);
```

de_set_trace_single_elem()

Sets the name of the element used when trace simulation is set to a single element. Currently, this command has no effect. All traces are simulated as simple nodal interconnections (shorts). Returns: none.

Syntax:

de_set_trace_single_elem(itemName);

where

itemName is the name of the component to simulate trace as. This component can have a width (W) and length (L) parameter. If they exist, L is set to the total trace length and W is set to the total trace width. This applies to traces connected pin-to-pin only (without tees).

Example:

```
de_set_trace_single_elem("MLIN");
```

de_set_trace_tech()

Sets the technology type for converting/simulating traces to microstrip, stripline, or PCB for the current window. Currently, this only affects the conversion of traces to transmission line elements. It is ignored for the simulation of traces. Returns: none.

Syntax:

de_set_trace_tech(technology);

where

technology is the choice for technology type, where:

PREF_TRACE_TECH_MICROSTRIP = microstrip
PREF_TRACE_TECH_STRIPLINE = stripline
PREF_TRACE_TECH_PCB = printed circuit board

Example:

```
de_set_trace_tech(PREF_TRACE_TECH_STRIPLINE);
```

de_set_trace_traverse()

Sets the trace traversal mode. When traces are converted to transmission line elements, complex traces (traces with tee junctions) can be fully converted, or the trace converted only up to the tee. Returns: none.

Syntax:

de_set_trace_traversal(mode);

where

mode is the choice for trace conversion, where:

0 = partial trace conversion
1 = full trace conversion

Example:

```
de_set_trace_traverse(0);
```

de_set_undo_edit_count()

Sets the number of edit commands that can be undone. Returns: none.

Syntax:

de_set_undo_edit_count(count);

where

count is the number of edits that can be undone.

Example:

```
de_set_undo_edit_count(100);
```

de_write_layer()

Writes the current layer settings of the active window to a file. Returns: none.

Syntax:

de_write_layer(fileName);

where

fileName is the name of the layer file.

Example:

```
de_write_layer("myLayer.lay");
```

de_write_preferences()

Writes the current preference settings of the active window to a file. Preferences include grid, object (not layer) colors, text attributes, select filters, modes. Returns: none.

Syntax:

de_write_preferences(fileName);

where

fileName is the name of the preference file.

Example:

```
de_write_preferences("myPreferences.prf");
```

ly_find_layer_by_gds_num()

Retrieves a layer number by its GDSII stream layer number, for the active layer set. Returns an integer, the first layer defined with the given GDSII layer number (-1 if the layer is not defined).

Syntax:

ly_find_layer_by_gds_num(gds2 layer number);

where

gds2 layer number is the GDSII stream layer number; an integer.

Example:

```
decl num;
num = ly_find_layer_by_gds_num(12);
```

ly_find_layer_by_name()

Retrieves a layer number given its name, for active layer set. Returns an integer, the layer number (–1 if the layer is not defined).

See also *ly_find_layer_name_by_num()*.

Syntax:

ly_find_layer_by_name(layerName);

where

layerName is the name of a layer; a string.

Example:

```
decl num;
num = ly_find_layer_by_name("conn1");
```

ly_find_layer_name_by_num()

Retrieves a layer name by its layer number, for the active layer set. Returns a string, the first layer defined with the given layer number (–1 if the layer is not defined).

Syntax:

ly_find_layer_name_by_num(layerNumber);

where

layerNumber is the layer number; an integer.

Example:

```
decl name;
name = ly_find_layer_name_by_num(12);
```

set_num_pnts_for_arc()

Sets the number of points used for approximating an arc when shapes with true arcs are converted to polygons with the *convert()* command for the current window. Returns: none.

Syntax:

set_num_pnts_for_arc(numPoints);

where

numPoints is the number of points to approximate converted arcs. Integer > 2.

Example:

```
set_num_pnts_for_arc(100);
```

set_part_size_units()

Sets the part size units in the Layout representation. Returns: none.

Syntax:

de_set_part_size_units(unit);

where

unit is the type of units for part size setting.

0 = user units
1 = screen pixels

See also: *de_set_part_size()*.

Example:

```
de_set_part_size_units(1);  //sets to screen pixels
```

Chapter 14: Database Query and Manipulation Functions

This chapter describes each Database Query and Manipulation functions in detail. The functions are listed in alphabetical order.

db_add_symbol_properties()

Adds one or more user-defined properties to a symbol. Returns: none.

Syntax:

db_add_symbol_properties(symbolHandle, [propertyName, propertyValue],...);

where

SymbolHandle is the handle to a schematic representation's symbol.

propertyName is a string; the name for the property.

propertyValue is a string, real or integer; a value for the property.

Example:

```
// add two properties to my new symbol
db_add_symbol_properties( symbolH, "partNum", "10345", "partID", "resCap" );
```

db_clear_map()

Clears the transformation mapping established by *db_setup_transform()*, *db_set_map()*, or *db_setup_map()*. Returns: none.

Syntax:

db_clear_map();

Example:

```
db_clear_map();
```

db_current_instance()

Returns a handle to the current instance. This is used in artwork macros to retrieve the instance handle. Returns: none.

Syntax:

db_current_instance();

Example:

```
decl instH;
instH = db_current_instance();
```

db_find_instance()

Returns a handle (pointer) of a named instance of a given design. Returns NULL if not found.

Syntax:

db_find_instance(designName, repType, instName);

where

designName is the name of the design to search.

repType is the type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

instName is a nique instance name.

Example:

```
decl myinst;
myinst = db_find_instance("amp", REP_SCHEM, "TL1");
```

db_find_property()

Returns a handle to the first property in a property list with a matching name, given a handle to any property in the list, and a property name.

Syntax:

db_find_property(propertyHandle, propertyName);

where

propertyHandle is the property handle, as returned from any *db_get_xxxx_attribute()* function.

propertyName is the name of a property; supplied when the property was created.

Example:

```
decl propHandle;
propHandle =  db_find_property( propHandle, "myproperty" );
```

db_first_dg()

Returns a handle to the first data group component (shape or text) of a given mask.

Syntax:

db_first_dg(maskHandle);

where

maskHandle is the handle of a mask component returned from a function such as *db_first_mask()* or *db_next_mask()*.

Example:

```
decl msk, dg;
msk = db_first_mask(repP);
dg = db_first_dg(msk);
```

db_first_instance()

Returns a handle to the first instance in a given design representation.

Syntax:

db_first_instance(repHandle);

where

repHandle is a handle to a representation returned from a function such as *db_get_rep()*.

Example:

```
decl rep, inst;
rep = db_get_rep(design, REP_SCHEM);
inst = db_first_instance(rep);
```

db_first_mask()

Returns A handle to the first mask of a given representation.

Syntax:

db_first_mask(repHandle);

where

repHandle is a handle to a representation returned from a function such as *db_get_rep()*.

Example:

```
decl rep, mask;
design = db_get_design("myamp", 1);
rep = db_get_rep(design, REP_SCHEM);
mask = db_first_mask(rep);
```

db_first_node()

Returns a handle to the first node of a given representation.

Syntax:

db_first_node(repHandle);

where

repHandle is a handle to a representation returned from a function such as *db_get_rep()*.

Example:

```
decl repHandle, nodeHandle;
repHandle = db_get_rep("abc",1);
nodeHandle = db_first_node(repHandle);
```

db_first_parm()

Returns a handle to a parameter or instance identified by an instance handle or the first parameter of a list of parameters identified by a parameter handle.

Syntax:

db_first_parm(instOrParmHandle);

where

instOrParmHandle is a handle to a parameter or instance.

Example:

```
decl instH, parmH;
parmH = db_first_parm(instH);
```

db_first_point()

Returns a handle to the first point of a data group component (shapes only).

Syntax:

db_first_point(dataGroupHandle);

where

dataGroupHandle is a handle of a data group returned from a function such as *db_first_dg* or *db_next_dg*().

Example:

```
decl x, y, dg, pnt, prevDG, coord;
dg = db_next_dg(prevDG);
pnt = db_first_point(dg);
coord = db_get_coord(pnt);
x = db_get_x(coord); y = db_get_y(coord);
```

db_first_property()

Returns a handle to the first property in a property list, given a handle to any property in the list.

Syntax:

db_first_property(propertyHandle);

where

propertyHandle is the property handle returned from any *db_get_xxxx_attribute()* function.

Example:

```
decl propHandle;
propHandle = db_first_property( db_get_instance_attribute(instHandle,
INST_PROPERTY));
```

db_first_segment()

Returns a handle to the first segment of the given segment list.

Syntax:

db_first_segment(segHandle);

where

segHandle is the handle to a segment list returned from a function such as *db_get_dg_attribute*.

Example:

```
decl segHandle, dgHandle, dgType;
dgType = db_get_dg_attribute (dgHandle, DG_TYPE);
if (dgType == POLYGON_DG_TYPE);
segHandle = db_get_dg_atribute (dgHandle, DG_DATA);
db_first_segment (segHandle);
```

db_free_points()

Frees the list of points created with db_segment_to_points. This should be done when the point list is no longer needed. Returns: none.

Syntax:

db_free_points(pointHandle);

where

pointHandle is a list of points returned from *db_segment_to_points()* or *db_first_point()*.

Example:

```
defun free_points()
{
   decl designHandle,dgtype,rep,mask,dg,segH,pntH,fpoint;

   designHandle = db_get_design( current_design_name(), TRUE );
   rep=db_get_rep(designHandle,REP_SCHEM);
   mask=db_first_mask(rep);
   dg=db_first_dg(mask);
   dgtype=db_get_dg_attribute(dg,DG_TYPE);
   if (dgtype==POLYGON_DG_TYPE)
   {
      segH=db_get_dg_attribute(dg,DG_DATA);
      db_first_segment(segH);
      pntH=db_segment_to_points(segH);
      db_free_points(pntH);
   }
}
```

db_get_arc_segment_attribute()

Returns an arc_segment attribute, given a handle to the arc_segment and an attribute key.

Syntax:

db_get_arc_segment_attribute(arcsegHandle, arcsegAttribute);

where

arcsegHandle is the handle to a arc segment.

arcsegAttribute is the arc segment attribute, where:

ARC_START_POINT = Handle to a coordinate holding x, y starting point of the arc

ARC_CENTER_POINT = Handle to a coordiante holding x,y center point of arc

ARC_ANGLE = The ending angle (*1000) in degrees of the arc

ARC_NUM_POINTS = The number of points used to approx the arc when converted to a polygon

Example:

```
decl id;
id = db_get_arc_segment_attribute(arcsegH, arcseg_ID);
```

db_get_bbox_x1()

Returns an integer, the x component of the lower-left corner of an object's bounding box. The bounding box is the maximum extent of any object, and is described as a rectangle.

Syntax:

db_get_bbox_x1(bboxHandle);

where

bboxHandle is the handle of an object's bounding box.

Example:

```
decl repBBox, x, repHandle;
repBBox = db_get_rep_bbox (repHandle);
x = db_get_bbox_x1(repBBox);
```

db_get_bbox_x2()

Returns an integer, the x component of the upper-right corner of an object's bounding box. The bounding box is the maximum extent of any object, and is described as a rectangle.

Syntax:

db_get_bbox_x2(bboxHandle);

where

bboxHandle is the handle of an object's bounding box.

Example:

```
decl repBBox, x, repHandle;
repBBox = db_get_rep_bbox (repHandle);
x = db_get_bbox_x2(repBBox);
```

db_get_bbox_y1()

Returns an integer, the y component of the lower-left corner of an object's bounding box. The bounding box is the maximum extent of any object, and is described as a rectangle.

Syntax:

db_get_bbox_y1(bboxHandle);

where

bboxHandle is the handle of an object's bounding box.

Example:

```
decl repBBox, y, repHandle;
repBBox = db_get_rep_bbox (repHandle);
y = db_get_bbox_y1(repBBox);
```

db_get_bbox_y2()

Returns an integer, the y component of the upper-right corner of an object's bounding box. The bounding box is the maximum extent of any object, and is described as a rectangle.

Syntax:

db_get_bbox_y2(bboxHandle);

where

bboxHandle is the handle of an object's bounding box.

Example:

```
decl repBBox, y, repHandle;
repBBox = db_get_rep_bbox (repHandle);
y = db_get_bbox_y2(repBBox);
```

db_get_coord()

Syntax:

db_get_coord(pointHandle);

where

pointHandle is a handle to a point returned from as function such as *db_first_point()* or *db_next_point()*.

Returns a coordinate point from current point in a list.

Example:

```
decl dgHandle, pntHandle, coord;
pntHandle = db_first_point (dgHandle);
if (pntHandle) coord = db_get_coord (pntHandle);
```

db_get_design()

Returns a handle to the named design. If the design is not in memory, it is read in. Returns NULL if design is not found.

Syntax:

db_get_design(designName);

where

designName is the name of the design to read in. If a full path is not supplied, the program searches the PDE_LIB environment path for the named design.

Example:

```
decl designHandle;
designHandle = db_get_design ("myamp");
```

db_get_design_attribute()

Returns an attribute of a design, given the design handle and an attribute key.

Syntax:

db_get_design_attribute(designHandle, designAttribute);

where

designHandle is the handle to a design, as returned from *db_get_design()*.

designAttribute is an attribute code, where:

DESIGN_TYPE is the design type (circuit, system, etc.); an integer.

DESIGN_MODIFIED is a flag indicating modification status, where:

0 = not modified
1 = modified

DESIGN_REP_SCHEM is the name of representation's schem file.

DESIGN_REP_LAY is the name of representation's layout file.

DESIGN_NAME is the name of design.

DESIGN_PROPERTY is the head of property list.

DESIGN_STAMP is the last-saved Time design.

DESIGN_COMMENT is the comment at top of design file.

Example:

```
decl attrib, dHandle;
attrib = db_get_design_attribute(dHandle, DESIGN_TYPE);
```

db_get_dg_attribute()

Returns an attribute of the data group, given a handle to the data group and an attribute key.

A DG or Data Group is a primitive (non-hierarchical) shape or text string. A polygon, polyline, rectangle, etc., is a data group.

Retrieving the DG_TYPE returns an integer code for the kind of data group the record represents. If DG_TYPE is a polygon, polyline, wire, or circle, DG_DATA is a handle to a segment list. If DG_TYPE is a path, DG_DATA is a handle to a path record; if text, to a text record; if a wire, to a wire record. Use the corresponding

get_attribute functions for retrieving additional attributes unique to these data types. DG_DATA is null if type is rectangle (all coordinates are described in the dg's bounding box).

Syntax:

db_get_dg_attribute(dgHandle, dgAttribute);

where

dgHandle is the handle to data group, as returned from *db_first_dg()*.

dgAttribute is the attribute code, where:

DG_SELECT is the select flag

DG_TYPE is the DG type; returns one of the following:

POLYGON_DG_TYPE
POLYLINE_DG_TYPE
PATH_DG_TYPE
WIRE_DG_TYPE
CIRCLE_DG_TYPE
TEXT_DG_TYPE
CONSTRUCTION_LINE_DG_TYPE
ARC_DG_TYPE
RECTANGLE_DG_TYPE

DG_SEG_SELECTED is a flag, any dg segment selected; an integer.

DG_PNT_SELECTED is a flag, any dg point selected; an integer.

DG_NUM_HOLES is the number of holes in dg; an integer.

DG_HIGHLIGHT is a flag, dg highlighted; an integer.

DG_NO_PLOT is a flag; dg to be plotted; an integer.

DG_DATA is the data handle; depends on DG_TYPE.

DG_BBOX is the bounding box handle; 4 coordinates.

DG_PROPERTY is the head of property list.

Example:

```
decl selected, dgHandle, maskH;
dgHandle = db_first_dg(maskH);
selected = db_get_dg_attribute(dgHandle, DG_SELECT);
```

db_get_instance_attribute()

Returns an attribute (a string, integer, or handle) of a given instance given the instance handle and an attribute key. Except for the case of the INST_SPECIAL flags, the data base accessed with the *db_get_instance_attribute()* routine stores only unique data regarding an instance (for example, the instance ID such as C1). For the permanent or non-unique data use the dm_ routines, such as *dm_get_item_definition_attribute()*. These routines reference a component's AEL description and return data, such as the component parameter names, component description string, component netlisting information, etc.

Syntax:

db_get_instance_attribute(instanceHandle, Attribute);

where

instHandle is a handle to an instance returned from a function, such as *db_first_instance()* or *db_next_instance()*.

instanceAttribute is the data for the instance, where:

INST_ID is a unique ID number for instance

INST_TYPE is a type of instance; where:

INST_SYMB_TYPE = instance is a symbol

INST_REP_TYPE = instance is a representation

INST_PSN_TYPE = instance is a parameterized subnetwork

INST_ART_TYPE = instance is an artwork design

INST_MACRO_TYPE = instance is an AEL artwork macro to generate layout data

INST_NULL_TYPE = is an invisible instance used as a place holder or the end of an unconnected wire. Not netlisted.

INST_SPECIAL is a flag indicating special status (port, gnd, etc); represents an “or” of the component attributes. Note: To use these you must bitwise & (AND) this against the INST_SPECIAL flag, such as:
if(special & INST_GLOBAL

INST_EQUIV_CREATED = a flag indicating equivalent instance created.

INST_READIN = a flag indicating instance reference in memory.

INST_SELECT = a flag indicating instance selected.

INST_VISITED = a flag indicating instance visited.

INST_FROZEN = a flag indicating instance position frozen.

INST_TRANSFORM_VALID = a flag indicating instance's transform valid.

INST_HIGHLIGHT = a flag indicating instance highlighted.

INST_DEACTIVATE = a flag indicating instance deactivated.

INST_TUNE = a flag indicating instance can be tuned.

INST_DRC = a flag indicating DRC error instance.

INST_NO_PLOT = a flag indicating instance should not be plotted.

INST_PARAM_ROWS = the number of rows per column.

INST_TRANSFORM = the transformation matrix.

INST_PARAM_HEAD = the head of the parameter list.

INST_PIN_HEAD = the head of the pin list.

INST_MASK_HEAD = the head of the mask list.

INST_NAME = a unique name of the instance.

INST_DESIGN_NAME = the name of design referenced.

INST_SYMBOL_NAME = the name of the symbol file.

INST_ART_NAME = the name of AEL artwork gen.

INST_BBOX = the bounding box: 4 coordinates.

INST_PROPERTY = the head of the property list.

Example:

```
decl xformHandle, instHandle;
xformHandle = db_get_instance_attribute (instHandle, INST_TRANSFORM)
```

db_get_instance_bbox()

Returns a handle to the bounding box of an instance. This is the smallest rectangle that completely encloses all data of an instance.

Syntax:

db_get_instance_bbox(instHandle, type);

where

InstHandle is the handle to an instance.

type indicates the type of bounding box, where:

0 = symbol bounding box
1 = annotation bounding box
2 = bounding box to include both symbol and annotation

Example:

```
decl bbox, x1;
bbox = db_get_instance_bbox(instHandle, 0);
x1 = db_get_bbox_x1(bbox);
```

db_get_instance_description()

Returns a string, the AEL component description string for an instance.

Syntax:

db_get_instance_description(instHandle);

where

InstHandle is the handle to an instance.

Example:

```
decl string, instH;
string = db_get_instance_description(instH);
```

db_get_instance_parm()

Returns a real number, the nominal value of an instance parameter given the parameter name or index (position in list of parameters).

Syntax:

db_get_instance_parm(designName, instHandle [, parameterReference]);

where

designName is the name of the design the instance is in.

instHandle is the instance value handle returned from a function such as *db_find_instance()*.

parameterReference is optional; the name of the parameter. If not supplied, 0.

Example:

```
decl inst, nomVal;
inst = db_find_instance(de_current_design_name(), 0, "TL1"); /* find TL1 in
current schematic design */
nomVal = db_get_instance_parm(de_current_design_name(), inst, "W"); /*
retrieve nominal value of W (width)*/
```

db_get_location_angle()

Returns an integer, the angle given a location handle in .001 of a degree. Location handles can be retrieved from pins using *db_get_pin_attribute()* with the PIN_LOCATION attribute.

Syntax:

db_get_location_angle(locationHandle);

where

locationHandle is the handle to a location.

Example:

```
decl ang, loc;
ang = db_get_location_angle(loc);
```

db_get_location_x()

Returns an integer, the x position given a location handle. Location handles can be retrieved from pins using *db_get_pin_attribute()* with the PIN_LOCATION attribute.

Syntax:

db_get_location_x(locationHandle);

where

locationHandle is the handle to a location.

Example:

```
decl x, loc;
x = db_get_location_x(loc);
```

db_get_location_y()

Returns an integer, the y position given a location handle. Location handles can be retrieved from pins using *db_get_pin_attribute()* with the PIN_LOCATION attribute.

Syntax:

db_get_location_y(locationHandle);

where

locationHandle is the handle to a location.

Example:

```
decl y, loc;
x = db_get_location_y(loc);
```

db_get_map()

Returns a handle to the current transformation mapping. Used in conjunction with *db_setup_map()* and *db_set_map()*.

Syntax:

db_get_map();

Example:

```
decl saveMap;
saveMap = db_get_map();
db_setup_transform(xformHandle);
transform_points();
db_set_map(saveMap); // restore the original mapping
```

db_get_map_attribute()

Returns an attribute of a transformation mapping, as retrieved by *db_get_map()*. Values are returned in database units, angle in 0.001 degrees, scale as a real number, and mirror flag as an integer 1 or 0.

Note that a mirror about the Y axis is stored in the transformation mapping as a combination of a mirror about the X axis followed by a 180 degree rotation, so query for the DB_MAP_MIR_Y attribute always returns 0. If a mirror about the Y axis was part of the transformation mapping, query for DB_MAP_MIR_X returns 1, and query for DB_MAP_ANG returns 180000 (or 180 degrees). If a rotation was involved as well as mirror, then query for the DB_MIR_ANG includes the rotation angle also.

Syntax:

db_get_map_attribute(mapAttribute [, map]);

where

mapAttribute is the attribute code for the map, where:

DB_MAP_X = Horizontal translation value
DB_MAP_Y = Vertical translation value
DB_MAP_ANG = Rotation angle
DB_MAP_SCL_X = Horizontal scale factor
DB_MAP_SCL_Y = Vertical scale factor
DB_MAP_MIR_X = Flag for mirror about X axis
DB_MAP_MIR_Y = Flag for mirror about Y axis (always 0)

map is optional; the handle of transformation mapping. If omitted, assumes current mapping as set by *db_set_map()*, *db_setup_transformation()*, or *db_setup_map()*.

Example:

```
decl x;
x = db_get_map_attribute( DB_MAP_X );
```

db_get_mask_attribute()

Returns an attribute of a given mask.

Syntax:

db_get_mask_attribute(maskHandle, maskAttribute);

where

maskHandle is a handle to a mask.

maskAttribute is the data for the mask, where:

MASK_NUMBER = Mask number; an integer
MASK_DG = Data group handle: dgs owned by mask
MASK_PROPERTY =Head of property list

Example:

```
decl maskNum;
maskNum = db_get_mask_attribute(maskH,MASK_NUMBER);
```

db_get_node_attribute()

Returns an attribute of a given node.

Syntax:

db_get_node_attribute(nodeHandle, nodeAttribute);

where

nodeHandle is a handle to a node.

nodeAttribute is the data for the node, where:

NODE_NUMBER = Node number
NODE_NAME = Name of node
NODE_PIN_HEAD =Head of pin list

Example:

```
decl nodeNumber;
nodeNumber = db_get_node_attribute(nodeHandle, NODE_NUMBER);
```

db_get_node_number()

Returns an integer, the node number of a given node.

Syntax:

db_get_node_number(nodeHandle);

where

nodeHandle is the handle to a node returned from a function such as *db_first_node()* or *db_next_node()*.

Example:

```
decl nodenum;
nodenum = db_set_node_number(nodeHandle);
```

db_get_node_wires()

Returns a handle to the segment representing a wire for a given node.

Syntax:

db_get_node_wires(nodeHandle);

where

nodeHandle is the handle to a node returned from a function such as *db_first_node()* or *db_next_node()*.

Example:

```
segHandle = db_get_node_wires(nodeHandle);
```

db_get_parm_attribute()

Returns an attribute of a parameter.

Syntax:

db_get_parm_attribute(parmHandle, parmAttribute [,index]);

where

parmHandle is the handle to a parameter.

parmAttribute is the data for the parameter, where:

PARM_VALUE_CODE = Type of parameter, where

0 = no value
1,2 = double
3,4 = parameter list
5,7 = string
6 = no value, but form name is important

PARM_NAME = Name of parameter

PARM_FORM_NAME = Name of the parameter form

PARM_NUM_DVALS = Number of double(real) values 0-3

PARM_VALUE_DVALUE = Real value

PARM_VALUE_SVALUE = String value (set if var reference)

PARM_VALUE_LIST = List of parameters

PARM_NO_PLOT = Flag indicating parameter visibility. Parameters are owned by instances. A parameter has a name and a value. The value can be a string, an array of 1-3 doubles or a further list of parameters. For example, in W=3, there is one dvalue (3); in W=gain, there is one svalue (gain). The value code indicates what the value type is. If the value is a double, the PARM_NUM_DVALS indicates how many make up the value; for example, constrained optimization has 3 dvalues (min, max and nominal). For complex values, it is often necessary to look up the form name and then the

form definition to determine how the value is represented in the data base. The AEL function "format_instance_data()" on page 14-42 can convert complex parameter values into strings.

index is optional. If attribute is PARM_VALUE_DVALUE, the index can be 0, 1, or 2 to retrieve first, second, or third real value. If the index is not given, the first real value is returned.

Example:

```
decl fname;
fname = db_get_parm_attribute(pHandle, PARM_FORM_NAME);
```

db_get_parm_nominal_value()

Returns the nominal value of a parameter, given a parameter value handle.

Syntax:

db_get_parm_nominal_value(parmHandle);

where

parmHandle is the handle to a parameter.

Example:

```
decl v;
v = db_get_parm_nominal_value(pHandle);
```

db_get_path_attribute()

Returns a path attribute, given a handle to the path and an attribute key.

Syntax:

db_get_path_attribute(pathHandle, pathAttribute);

where

pathHandle is a handle to a path.

pathAttribute is the path attribute can be one of the following:

PATH_WIDTH = Width of the path in user units

PATH_BEND= Type of path corner; returns one of the following

DB_MITERED_CORNER
DB_SQUARE_CORNER
DB_CURVED_CORNER

PATH_MITERRADIUS = If curved, the curve radius angle*1000

PATH_SEGMENT = Handle to segment list

Example:

```
decl corn;
corn = db_get_path_attribute(pathH, PATH_BEND);
```

db_get_pin_attribute()

Returns the attribute value of a given pin.

Syntax:

db_get_pin_attribute(pinHandle, pinAttribute);

where

pinHandle is the handle to a pin.

pinAttribute is the data for the pin, where:

PIN_LOCATION = Location handle: for x,y and angle of pin

PIN_INST_PTR = Instance handle: for instance owning this pin

PIN_NODE_PTR = Node handle: for the node owning this pin

PIN_NUMBER = Pin number; an integer

PIN_PROPERTY = Head of property list

PIN_NAME = Name of the pin

PIN_DIRECTION = Pin direction: Input, output, or input/output

PIN_WIRE_PTR = Wire handle for wire that is connected to pin, where the attributes for layer binding in layout are:

PIN_MASK_NUMBER = A pin gets the mask number from the layer of the component it is attached to. Used to enforce layer compatibility when a trace connects to a component, or between traces or between components.

PIN_BINDING_LIST = List of layers that this pin is allowed to attach to.

PIN_INHERIT_BINDING = Behavior of connectivity in hierarchy, where: TRUE = a port pin, FALSE = all other connections

Example:

```
decl pinNo, pinHandle;
pinNo = db_get_pin_attribute(pinHandle, PIN_NUMBER);
```

db_get_port_attribute()

Returns the value of the power pin.

Syntax:

db_get_port_attribute(portH, PORT_POWER);

where

portH is (tbd)

PORT_POWER is (tbd)

db_get_port_number()

Returns an integer, the port number of a given Port instance.

Syntax:

db_get_port_number(instHandle);

where

instHandle is a handle to a Port instance returned from a function call such as *db_find_instance()*.

Example:

```
decl pn, portHandle;
portHandle = db_find_instance("myDesign", REP_SCHEM, "P1");
pn = db_get_port_number(portHandle);
```

db_get_property_attribute()

Returns a property attribute, given a handle to the property and an attribute key.

Syntax:

db_get_property_attribute(propHandle, propAttribute);

where

propHandle is the handle to a property.

propAttribute is the property attribute can be one of the following:

PROPERTY_NAME = Name of the property; a string

PROPERTY_TYPE = The property type; indicates how the value is represented, where:

0 = unknown type
1= long value
2 = double value
3 = string value

PROPERTY_VALUE_LONG = Property value = long

PROPERTY_VALUE_STRING = Property value = a string; a string

PROPERTY_VALUE_DOUBLE = Property value = double; real

Example:

```
decl propType;
propType=db_get_property_attribute(propHandle, PROPERTY_TYPE);
```

db_get_rep()

Returns the handle to a design representation.

Syntax:

db_get_rep(designHandle, repType);

where

designHandle is a handle to a design, as returned from *db_get_design()*.

repType is the type of representation, where:

REP_SCHEM = Schematic representation
REP_LAY = Layout representation

Example:

```
decl repH;
repH = db_get_rep(db_get_design("amp.dsn", 0),REP_SCHEM);
// return schematic rep
```

db_get_rep_attribute()

Retrieves a representation's attribute, given a representation handle and an attribute key.

Syntax:

db_get_rep_attribute(repHandle, repAttribute);

where

repHandle is the handle of a design representation, as returned from *db_get_rep()*.

repAttribute is the representation attribute, where:

REP_TYPE = Design type code, either REP_SCHEM or REP_LAY
REP_NAME = Name of design
REP_PRF_FILE = Name of rep's preference file
REP_LAY_FILE = Name of rep's layers file
REP_SYMBOL = Used to retrieve rep's symbol handle
REP_BBOX = Bounding box handle
REP_PROPERTY = Head of property list

Example:

```
decl fileName;
fileName = db_get_rep_attribute( repH, REP_LAY_FILE );
```

db_get_rep_bbox()

Returns a handle to the bounding box of a given design representation.

Syntax:

db_get_rep_bbox (repHandle);

where

repHandle is the handle of a design representation, as returned from *db_get_rep()*.

Example:

```
decl bbox, x;
bbox = db_get_rep_bbox(repH);
x = db_get_bbox_x1(bbox);
```

db_get_rep_db_factor()

Converts a data base unit to a user unit. Returns a real value, the data base conversion factor of a given representation.

Syntax:

db_get_rep_db_factor(repHandle);

where

repHandle is the handle of a design representation, as returned from *db_get_rep()*.

Example:

```
decl conFact;
conFact = db_get_rep_db_factor(repH);
```

db_get_rep_unit_mks()

Converts a unit to mks. Returns a real value, the design representation's user unit to mks conversion factor.

Syntax:

db_get_rep_unit_mks(repHandle);

where

repHandle is the handle of a design representation, as returned from *db_get_rep()*.

Example:

```
decl conFact;
conFact = db_get_rep_unit_mks(repH);
```

db_get_rep_unit_name()

Returns a string, the name of the units used by a design representation. The string can be “mil”, “in”, “um”, “mm”, “cm”, “meter”, or “ft”.

Syntax:

db_get_rep_unit_name(repHandle);

where

repHandle is the handle of a design representation, as returned from *db_get_rep()*.

Example:

```
decl uName, repH;
repH = db_get_current_design_rep();
uName = db_get_rep_unit_name(repH)
```

db_get_segment_attribute()

Returns a segment attribute, given a handle to the segment and an attribute key. A segment is a list of x,y coordinates. Shapes can be composed of one or more segments (for example, a polygon with an arc or hole in it). For a circle or arc segment, the list of points is the polygonal approximation of the arc. The number of points used to approximate the arc is determined when the arc is created. To retrieve an arc as a starting, center point, and ending angle, use the *db_get_arc_segment_attribute()* function. You can also use this function to retrieve the number of points used to approximate the arc as a set of points. All other segments are list of x,y coordinates. The ortho types are orthogonal shapes where, other than the first point, only every other x or y point is stored (since you can always infer the other coordinate). The h is for segment lists that start off horizonally, the v type is for those whose next point lies vertically. The non-ortho type is for shapes with one or more non-orthogonal edges and thus every x,y point is stored in the segment list. However, the point list returned is always normalized to include every x,y point; you do not need to determine orthogonal from non-orthogonal segments. To retrieve the points use *db_first_point()*, *db_next_point()*, *db_transform_points()*, and *db_free_points()* to traverse and transform the points to user units.

Syntax:

db_get_segment_attribute(segHandle, segAttribute);

where

segHandle is the handle to a segment.

segAttribute is the segment attribute, where:

SEG_TYPE = The type of segment; returns one of these types:

ARC_TYPE
ORTHO_H_TYPE
ORTHO_V_TYPE

SEG_INSIDE = Flag, where:

0 = outside segment
1 = hole
2 = edge connecting inside to outside

SEG_NUM_PNTS = Number of x,y points in segment

SEG_PNT_LIST = A handle to the list of points

SEG_SELECT = Flag, where:

0 = not selected
1 = selected

SEG_PNT_SELECTED = Flag indicating any points selected

SEG_BBOX = Bounding box handle: 4 coordinates

SEG_PROPERTY = Head of the property list

Example:

```
decl id;
id = db_get_segment_attribute(segmentH, segment_ID);
```

db_get_symbol_attribute()

Retrieves an attribute of a symbol attribute, given a handle to the symbol and an attribute key.

Syntax:

db_get_symbol_attribute(symbolHandle, symbolAttribute);

where

symbolHandle is the handle to a symbol.

symbolAttribute is the symbol attribute can be one of the following:

SYMB_MASK_HEAD = Head of mask list
SYMB_PORT_HEAD = Head of port list
SYMB_BBOX = Bounding box handle
SYMB_PROPERTY = Head of property list

Example:

```
decl mask;
symbH = db_get_rep_attribute(db_get_current_design_rep(), REP_SYMBOL);
mask = db_get_symbol_attribute( symbH, SYMB_MASK_HEAD);
```

db_get_text_attribute()

Returns a text attribute, an integer code for the kind of text. Normal text is simple free standing text, not associated with components/instances. Annotation text is associated with an instance and is used to display the instance's name, design name and parameters. You must retrieve an instance's mask record to obtain access to these types of text strings. The TEXT_JUST field retrieves an “or'd” integer representing the text justification.

Syntax:

db_get_text_attribute(textHandle, textAttribute);

where

textHandle is a handle to a text, as returned from *db_get_dg_attribute()*.

textAttribute is the text attribute, where:

TEXT_TYPE = Type of text; returns one of these types:

NORMAL_TEXT
ANNOT_DESIGN_NAME
ANNOT_INST_NAME
ANNOT_INST_PARAMETER

TEXT_FONT = Font number

TEXT_FONT_NAME = Font name

TEXT_HEIGHT = Height of the text in user units

TEXT_JUST= Text justification, an “or’d” combination of one or more of these:

DB_BOT_JUST
DB_MID_JUST
DB_TOP_JUST with one of these:
  DB_LEFT_JUST
  DB_CENTER_JUST
  DB_RIGHT_JUST

TEXT_ABSOLUTE = Flag indicating whether the text can rotate

TEXT_LOCATION = Location handle: x,y and angle

TEXT_PARAM_SEQ_NO = Parameter index indicating which parameter

TEXT_STRING = Actual text string

Example:

```
decl type;
type = db_get_text_attribute(textH, TEXT_TYPE);
```

db_get_transform_angle()

Returns the angle of a transform object as returned from the function *db_get_instance_attribute()*. The return value is in thousandths of a degree, not in degrees. For example, a 90 degree angle is returned as 90000.

Syntax:

db_get_transform_angle(transformHandle);

where

transformHandle is the handle to an instance's transformation record.

Example:

```
decl angle;
angle = db_get_transform_angle(instTrans);
```

db_get_transform_mirror_x()

Returns the x axis mirror flag of a transform object, as returned from the function *db_get_instance_attribute()*, where: 0 = not mirrored around the x axis, and 1 = mirrored around the x axis.

Syntax:

db_get_transform_mirror_x(transformHandle);

where

transformHandle is a handle to an instance's transformation record.

Example:

```
decl mirx;
mirx = db_get_transform_mirror_x(instTrans);
```

db_get_transform_mirror_y()

Returns the y axis mirror flag of a transform object, as returned from the function *db_get_instance_attribute()*, where: 0 = not mirrored around the y axis, and 1 = mirrored around the y axis.

Syntax:

db_get_transform_mirror_y(transformHandle);

where

transformHandle is a handle to an instance's transformation record.

Example:

```
decl miry;
miry = db_get_transform_mirror_y(instTrans);
```

db_get_transform_x()

Returns the x coordinate of an instance's transformation record.

Syntax:

db_get_transform_x(transformHandle);

where

transformHandle is a handle to an instance's transformation record.

Example:

```
decl x;
x = db_get_transform_x(instTrans);
```

db_get_transform_y()

Returns the y coordinate of an instance's transformation record.

Syntax:

db_get_transform_y(transformHandle);

where

transformHandle is a handle to an instance's transformation record.

Example:

```
decl y;
```

```
y = db_get_transform_y(instTrans);
```

db_get_wire_attribute()

Returns a wire attribute, given a handle to the wire and an attribute key. Wires and Traces are the same object. A trace is simply a wire with a width > 0. Traces are used in layout to represent actual physical connectivity, while wires are used in both layout and schematic to represent ideal electrical connectivity. Width, bend and miter radius are only relevant to traces.

Syntax:

db_get_wire_attribute(wireHandle, wireAttribute);

where

wireHandle is a handle to a wire.

wireAttribute is the wire attribute, where:

WIRE_WIDTH = Width of the wire/trace in user units

WIRE_ID = Unique wire ID number

WIRE_BEND = Type of wire corner; returns one of these types:

DB_MITERED_CORNER
DB_SQUARE_CORNER
DB_CURVED_CORNER

WIRE_MITERRADIUS = If curved, the curve radius angle*1000

WIRE_SEGMENT = Handle to segment list

Example:

```
decl id;
id = db_get_wire_attribute(wireH, WIRE_ID);
```

db_get_x()

Returns an integer, the x coordinate in data base units.

Syntax:

db_get_x(coordHandle);

where

coordHandle is a handle to a coordinate, as returned from *db_get_coord()*.

Example:

```
decl pointHandle, coordHandle, x;
coordHandle = db_get_coord(pointHandle);
x = db_get_x(coordHandle);
```

db_get_y()

Returns an integer; the y coordinate in data base units.

Syntax:

db_get_y(coordHandle);

where

coordHandle is a handle to a coordinate, as returned from *db_get_coord()*.

Example:

```
decl pointHandle, coordHandle, y;
coordHandle = db_get_coord(pointHandle);
y = db_get_y(coordHandle);
```

db_instance_next_pin()

Returns a pin handle, the next pin in the instance's pin list; if last pin, returns NULL.

Syntax:

db_instance_next_pin(pinHandle);

where

pinHandle is a handle to an instance pin, as returned from a previous call or a call to *db_instance_first_pin()*.

Example:

```
decl pinHandle, instHandle;
pinHandle = db_get_instance_attribute(instHandle,INST_PIN_HEAD);
while(pinHandle);
{
pinHandle = db_instance_next_pin(pinHandle);
}
```

db_next_dg()

Returns a handle to the next data group (shape or text) belonging to a given mask; NULL if end of the list.

Syntax:

db_next_dg(prevDG);

where

prevDG is a data group handle as returned from this function or *db_first_dg()*.

Example:

```
decl dgHandle, instHandle;
dgHandle = db_first_dg(maskHandle);
while (dgHandle)
{
process_dgs(dgHandle);
dgHandle = db_next_dg(dgHandle);
}
```

db_next_instance()

Returns a handle to the next instance belonging to a given design representation; NULL if end of the list.

Syntax:

db_next_instance(instHandle);

where

instHandle is a handle to an instance as returned from this function or *db_first_instance()*.

Example:

```
decl instHandle, repHandle;
instHandle = db_first_instance(repHandle);
while (instHandle)
{
process_insts(instHandle);
instHandle = db_next_instance(instHandle);
}
```

db_next_mask()

Returns a handle to the next mask belonging to a given design representation; NULL if end of the list.

Syntax:

db_next_mask(maskHandle);

where

maskHandle is a handle to a mask as returned from this function or *db_first_mask()*.

Example:

```
decl maskHandle, repHandle;
maskHandle = db_first_mask(repHandle);
while (maskHandle)
{
process_masks(maskHandle);
maskHandle = db_next_mask(maskHandle);
}
```

db_next_node()

Returns handle to the next node of a given representation; NULL if end of the list.

Syntax:

db_next_node(nodeHandle);

where

nodeHandle is a handle to a node, as returned from this function or a call to *db_first_node()*.

Example:

```
decl nodeHandle;
nodeHandle = db_next_node(nodeHandle);
```

db_next_parm()

Returns a handle to the next parameter of the instance. Takes as an argument a previous handle to a parameter. NULL if end of the list.

Syntax:

db_next_parm(parmHandle);

where

parmHandle is a handle to a parameter as instance.

Example:

```
decl parmH;
db_next_parm(parmH);
```

db_next_point()

Returns a handle to the next point belonging to a given data group shape. NULL if end of the list.

Syntax:

db_next_point(pointHandle);

where

pointHandle is a handle to a point as returned from this function or *db_first_point()*.

Example:

```
decl pointHandle, dgHandle;
pointHandle = db_first_point(dgHandle);
while (pointHandle)
{
process_points(pointHandle);
pointHandle = db_next_point(pointHandle);
}
```

db_next_port()

Returns handle to the next port on the symbol for traversing the symbol; NULL if end of the list.

Syntax:

db_next_port (portHandle);

where

portHandle is a handle to a port. Note that a pin and port Handle are interchangable. Use *db_get_pin_attribute()* to retrieve attributes relevant to pins or ports.

Example:

```
decl pHandle;
pHandle = db_get_symbol_attribute( symHandle, SYMBOL_PORT_HEAD);
while( pHandle )
{
// do something with the handle
pHandle = db_next_port(portHandle);
}
```

db_next_property()

Returns a handle to the next property in a property list, given a handle to any property in the list; NULL if end of the list.

Syntax:

db_next_property(propertyHandle);

where

propertyHandle is the property handle, returned from a function such as *db_first_property()*, *db_next_property()*.

Example:

```
decl propHandle;
propHandle =  db_get_instance_attribute(instHandle, INST_PROPERTY);
while( propHandle )
{
// do something with it
propHandle = db_next_property(propHandle );
}
```

db_next_segment()

Returns a handle to the next segment of a shape data group; NULL if end of the list.

Syntax:

db_next_segment(segHandle);

where

SegHandle is a handle to a segment, as returned from a previous call or a call to *db_first_segment()*.

Example:

```
decl segH;
```

```
segH = db_next_segment(segH);
```

db_node_first_pin()

Returns a handle to the first pin of a given node.

Syntax:

db_node_first_pin(nodeHandle);

where

nodeHandle is a handle to a node, as returned from *db_first_node()* or *db_next_node()*.

Example:

```
decl pinH, nodeHandle;
pinH = db_node_first_pin(nodeHandle);
```

db_node_next_pin()

Returns a handle to the next pin of a given node; NULL if no more pins in node.

Syntax:

db_node_next_pin(pinHandle);

where

pinHandle is a handle to a pin, as returned from a previous call or a call to *db_node_first_pin()*.

Example:

```
pinH = db_node_next_pin(pinH);
```

db_num_parms()

Returns the number of parameters in a parameter list.

Syntax:

db_num_parms(parmHandle);

where

parmHandle is a handle to a parameter list.

Example:

```
decl num, repH;
num = db_num_parms(db_first_instance(repH));
```

db_segment_to_points()

Allocates and returns a handle to a list of points owned by a segment. After using this function, the point list must be freed with *db_free_points()*.

Syntax:

db_segment_to_points(segHandle);

where

segHandle is a handle to a segment.

Example:

```
decl pnts;
pnts = db_segment_to_points(segHandle);
```

db_set_map()

Sets the current transformation mapping. Returns: none.

Syntax:

db_set_map (mapHandle);

where

mapHandle is a handle to transformation mapping, as returned from *db_get_map()*.

Example:

```
decl saveMap, instTransH;
saveMap = db_get_map();
db_setup_transform(instTransH);
process_instances();
db_set_map(saveMap);
```

db_setup_map()

Modifies the current transformation mapping set by *db_set_map()*, *db_setup_transformation()*, or *db_setup_map()* by applying new mapping information

with a cascading effect, with the effect of applying both maps in sequence. Returns: none.

Syntax:

db_setup_map(map);

db_setup_map([x, y, ang, xs, ys, mirX, mirY]);

where

In the first form:

map is a map as retreived by *db_get_map()* is the only argument.

In the second form:

x is optional; the value of horizontal translation in database units.

y is optional; the value of vertical translation in database units.

ang is optional; the value of rotation angle in 0.001 degrees, rotation around x,y.

xs is optional; the value of horizontal scale factor, a real number, scaled about x/y.

ys is optional; the value of vertical scale factor, a real number, scaled about x/y.

mirX is optional; the flag for mirror about horixontal line through x/y, after rotation.

mirY is optional; the flag for mirror about vertical line through x/y, after rotation.

Example:

```
db_setup_map( 100, 0, 90000, 1.0, 1.0, 0, 0 );
```

db_setup_transform()

Sets up the instance transformation mapping (rotation, translation and mirror). Returns: none.

Syntax:

db_setup_transform(instTransH);

where

instTransH is a handle to an instance transformation, as returned by *db_get_instance_attribute()*.

Example:

```
decl saveMap, instTransH;
saveMap = db_get_map();
db_setup_transform(instTransH);
process_instances();
db_set_map(saveMap);
```

db_total_points()

Returns an integer, the total number of points for a shape.

Syntax:

db_total_points(pointHandle);

where

pointHandle is the handle to the first point in a list of points, as returned from *db_first_point()*.

Example:

```
decl dgHandle, pntHandle, totalPoints;
pntHandle = db_first_point (dgHandle);
totalPoints = db_total_points (pntHandle);
```

db_transform_angle()

Transforms an angle value by rotation and mirror values from the current transformation as set by *db_set_map()*, *db_setup_transformation()*, or *db_setup_map()*. Returns a modified value of angle.

See also *db_transform_points()* and *db_transform_coord()*.

Syntax:

db_transform_angle(angle);

where

angle is the value of angle in 0.001 degrees.

Example:

```
decl a;
a = db_transform_angle( a );
```

db_transform_bbox()

Transforms a bounding box by rotation and mirror values from the current transformation as set by *db_set_map()*, *db_setup_transformation()*, or *db_setup_map()*. Returns a modified bounding box value.

Syntax:

db_transform_bbox(bBoxHandle);

where

bBoxHandle is the handle of a bounding box.

Example:

```
decl bboxH, newBBoxH;
bboxH = db_get_rep_bbox(repH);
newBBoxH = db_transform_bbox(bboxH);
```

db_transform_coord()

Transforms a single coordinate value using the current transformation as set by *db_set_map()*, *db_setup_transformation()*, or *db_setup_map()*. Returns a modified coordinate value.

See also *db_transform_points()* and *db_transform_angle()*.

Syntax:

db_transform_coord(coord);

where

coord is the coordinate value, in database units.

Example:

```
decl pnt, coord;
pnt = db_first_point( dgHandle );
coord = db_get_coord( pnt );
coord = db_transform_coord( coord );
```

db_transform_points()

Transforms the given point list using the current instance transformation, set with *db_setup_transform()*. Returns a handle to the transformed point list.

Syntax:

db_transform_points(pointListHandle);

where

pointListHandle is the handle to a point list, as returned from *db_segment_to_points()*.

Example:

```
decl trans, instH, points, segH, coord, x, y;
trans = db_get_instance_attribute(insH, INST_TRANSFORM);
db_setup_transform(trans);
points = db_segment_to_points(segH);
points = db_transform_points(points);
while(points)
{
   x = db_get_x(coord);
   y = db_get_y(coord);
   points = db_next_point(points);
}
```

format_instance_data()

Creates custom netlist formats. The formatting codes used in the dataFormat string are described under *create_item()*. Returns a completely formated string with all formating codes replaced by their appropriate values.

Syntax:

format_instance_data(instHandle, designName, designCode, dataFormat);

where

instHandle is the handle of an instance of an component.

designName is the name assumed for design for format purposes.

designCode is the type of code for design.

dataFormat is the format string to display the data in a schematic. Refer to the section “Format Strings” on page 4-5.

Example:

```
format_instance_data(Inst, "test", 1, "%n %t");
```

Chapter 15: Component Definition Functions

This chapter describes each Component Definition function in detail. The functions are listed in alphabetical order.

create_compound_form()

Creates a new compound form and stores the form in the dictionary for the current simulator type. Compound forms represent parameter values more complex than just strings. A compound form contains one or more parameter values. Returns: none.

Syntax:

create_compound_form(formName, label[, dialogDataStr], attribute, netlistFormat, displayFormat, parameter1, parameter2,..., parameterN);

where

formName is a string, the unique form name.

label is a string, a descriptive string for the form.

dialogDataStr is optional; a string, that can be any dialog data. For the standard Edit Component dialog box, this field can be used as the name of the table entry field. This argument indicates to the dialog the gui component to use to breakdown the parm value input. For example, the standard Edit Component dialog supports these gui configurations:

"StdForm"
"StringAndReference"
"SingleTextLine"
"InstSelectionForm"
"Node SetForm"
"FileBasedForm"
"ReadFileForm"

Note: This argument can be omitted if dialogDataStr is the same as formName. For complete listing, see *$HPEESOF_DIR/de/ael/pde_gemini.ael*.

attribute is not used in this release.

netlistFormat is the format string to netlist the parameter value as. Refer to "Format Strings" on page 4-5.

displayFormat is the format string to display in a schematic. Refer to "Format Strings" on page 4-5.

parameterN is one or more parameters, created with *create_parm()*.

Example:

This example creates a form for describing variables. The variable has a default name *X* and a default value *1.0*.

```
create_compound_form("VarFormStdForm", "Standard",
   "StdForm", 0, "%0s=%1s", "%0s=%1s",
   create_parm("VarName", "Variable Name", 0, "VarNameForms",
      UNITLESS_UNIT), prm ("VarNameForm", "X")),
   create_parm("VarValue", "Variable Value", 0, "VarValueForms",
      UNITLESS_UNIT, prm("editcompPowerVar", "1.0")));
```

create_constant_form()

Creates a new constant form and stores the form in the dictionary for the current simulator type. Constant forms are used to describe one or more constant values. A parameter using constant form displays a list of values from which to choose, rather than a field for typing in the value. This is used by components with parameters whose allowable values are a list of constants. Returns: none.

Syntax:

create_constant_form(name, label, attribute, netlistFormat, displayFormat);

where

name is a string representing the form name.

label is a descriptive label for the form.

attribute is an integer, usually 0 for this type of form.

netlistFormat is the format string to netlist the parameter value as. Refer to "Format Strings" on page 4-5.

displayFormat is the format string to display in a schematic. Refer to "Format Strings" on page 4-5.

Example:

This example creates forms for a parameter that has either *yes* or *no* values. *Yes* is netlisted as a 1 and *no* as a 0, but the strings *yes* and *no* are displayed in the

schematic and the dialog box. The forms are combined into a single formset with the *create_form_set()* function.

```
create_constant_form("y_n1", "YES", 0, "1", "yes");
create_constant_form("y_n0", "NO", 0, "0", "no");
create_form_set("yes1_no0", "y_n1", "y_n0");
```

create_design_default_item()

Defines default component template used to represent parametric subnetworks of the current design type. Current design type can be set using *set_design_type()*. Returns: none.

See also: *create_item()* and the section "Format Strings" on page 4-5.

Syntax:

create_design_default_item(registration function, library, prefix, attrib, priority, dialogCode, dialogData, netlistFormat, netlistData, displayFormat, symbolName, artworkType, artworkData, extraAttribute[parameterN]);

where

registration function is an AEL function to set library group association when a subnetwork component is created.

library is a string; the default library associated with new subnetwork component.

prefix is a string; the prefix for the instance name (e.g., "TL" for name "TL1").

attrib is an integer; attribute code, where:

0 = no special characteristics
8 = ITEM_UNIQUE
16 = ITEM_DESIGN_INST

priority is an integer; almost always –1.

dialogCode is a string; name of the dialog describing which Edit Component dialog box to use. The variable *standard_dialog* should be used for most components.

dialogData is a description string; passed to the dialog creation function and displayed in the *Component Description* field of the Edit Component dialog box.

netlistFormat is a string; used to control the netlist format for the component. Usually specified with the variable standard_netlist or ComponentNetlistFormat.

netlistData is a string used in conjunction with the netlist format string.

displayFormat is a special string used to control the display of the component parameters in the schematic. Usually specified with the variable *standard_symbol* or *ComponentAnnotFont*. Currently this format string also dictates how on-screen editing works.

symbolName is a string; the name of the schematic symbol for the component.

artworkType is an integer; the type of artwork for the component, where:

0 = no artwork
1 = fixed artwork
2 = ael generated artwork
3 = synchronized

artworkData is the name of the artwork function, or design containing the artwork.

For *artworkType*=0, set to NULL.

For artworkType=fixed artwork, set to the name of the design file containing the artwork.

For artworkType=ael generated, set to the name of the AEL function used to generated the artwork.

extraAttribute is optional; an extension to the attribute code. extraAttribute choices are listed in Table 15-2.

parameterN is optional; the list of parameters. Parameters are generated with the *create_parm()* command.

Example:

```
create_design_default_item (register_net_item, "*", "X", 16, NULL,
standard_dialog, NULL, ComponentNetlistFmt, NULL, ComponentAnnotFmt, NULL,
3, NULL);
```

create_design_default_parm()

Defines a default parameter template for new parameters added to a parametric subnetwork of the current design type. Current design type can be set using set_design_type().

See also: *create_design_default_item()*, *create_parm()*.

Syntax:

create_design_default_parm(name, label, attrib, formSet, unitCode, defaultValue);

where

name is a string, name of the parameter.

label is a descriptive string for the parameter.

attrib is an integer indicating special parameter, usually 0.

formSet is the name of the form set used for this value.

unitCode is the choice for units, where:

STRING_UNIT = –2
UNITLESS_UNIT = –1
FREQUENCY_UNIT = 0
RESISTANCE_UNIT = 1
CONDUCTANCE_UNIT = 2
INDUCTANCE_UNIT = 3
CAPACITANCE_UNIT = 4
LENGTH_UNIT = 5
TIME_UNIT = 6
ANGLE_UNIT = 7
POWER_UNIT = 8
VOLTAGE_UNIT = 9
CURRENT_UNIT = 10
DISTANCE_UNIT = 11
TEMPERTURE_UNIT = 12
DB_GAIN_UNIT = 13

defaultValue is optional; a real number or a value returned from the *prm()* function. The *prm()* function generates an acceptable default value for parameters with different form sets.

Example:

```
create_design_default_parm("X", "", 0, "StdFormSet", -1, "");
```

create_form_set()

Creates a set of forms for use by the *create_parm()* function. A form set describes the set of allowable forms the value of a parameter may take on. A number of predefined form sets exist. Returns: none.

See also: Chapter 2, Using AEL Database Retrieval Functions

Syntax:

create_form_set(name, formName1, formName2,.... formNameN);

where

name is the name of the form set. Used by the *create_parm()* function.

formName is one or more form names.

Note If you redefine an existing format to include different forms, then there may be backwards-compatibility problems when you open a design with components whose parameters were created with the old form set definitions; for example:
(1) Going from a non-compound to a compound form (by way of the *create_compound_form()* function)
(2) Going from a compound form to a non-compound form

Example:

This example creates a form set composed of the *y_n1* and *y_n0* forms. These allow a parameter value to take on a *yes* or *no* value.

```
create_constant_form("y_n1", "YES", 0, 1, "YES");
create_constant_form("y_n0", "NO", 0, 0, "NO");
create_form_set("yes1_no0", "y_n1", "y_n0");
```

create_item()

Creates a new component definition and stores it with the current dictionary. This is the central component definition function. Components are defined for the current simulator, which is established with a prior call to *set_simulator_type()*. Components are then organized into palette and library groups, which can be placed in a Schematic or Layout window (see *library_group()* and *de_define_palette_group()* functions in this chapter). Returns: none.

See also: Chapter 2, Using AEL Database Retrieval Functions

Syntax:

create_item(name, label, prefix, attrib, priority, iconName, dialogCode, dialogData, netlistFormat, netlistData, displayFormat, symbolName, artworkType, artworkData[, extraAttribute, cbList, parameterN]);

where

name is a string; a unique name of the component.

label is a string; a descriptive label for the component. This is also used as the balloon help for the component palette selection. The label should not be longer than 80 characters (a Win32 restriction).

prefix is a string; the prefix for the instance name (e.g., “TL” for name “TL1”) used when the component is placed.

attrib is an attribute code—the attribute choices are listed in Table 15-1.

priority is an integer; indicates special netlist handling. Normally specified as NULL or -1. Not used.

iconName is a string; name of the bitmap file used for component button in a palette.

dialogCode is a string; name of the dialog, usually *standard_dialog*. The dialog is used to edit the parameters and/or some other attributes of the component.

dialogData is a string; passed to the dialog creating function. Usually “*”. Not really used.

netlistFormat is a string; used to control the netlist format for the component. Usually specified with the variable *standard_netlist* or *ComponentNetlistFmt*.

netlistData is a string; used in conjunction with the netlist format string. %d.

displayFormat is a string; used to control the display of the component annotation in the schematic. Usually specified with the variable *standard_symbol*. Currently this format string also dictates how on-screen editing works.

symbolName is a string; the name of the schematic symbol for the component.

artworkType is an integer; indicates the type of artwork for the component, where:

0 = no artwork
1 = fixed artwork
2 = ael generated artwork
3 = synchronized

artworkData is a string; the name of the artwork function or design containing the artwork.

For fixed artwork, it should be a layout artwork design name (without a *dsn* extension).

For macro artwork, it should be an AEL artwork generation function.

extraAttribute is optional; an extension to the attribute code. Use the function *dm_get_item_definition()* to retrieve this attribute. Attribute choices are listed in Table 15-2.

cbList is optional; the list of callbacks. For example,

list(dm_create_cb ("...", "..."), ...);

Currently supported callbacks are: ITEM_NETLIST_CB

parameterN is optional; the list of parameters. Return value from the *create_parm()* command.

Note If you redefine a component to use different parameters (by way of the *create_parm()* command), and then you open a design that contains components created with the older component definition, it will not work for the following cases:
(1) Going from a non-PARM_REPEATED to a PARM_REPEATED parameter
(2) Going from a PARM_REPEATED to a non-PARM_REPEATED parameter

---

Example:

```
/* TLIN */
create_item(    "TLIN",                                 // name
                "Ideal 2-Terminal Transmission Line",   // label
                "TL",                                   // prefix
                0,                                      // attribute
                NULL,                                   // priority
                "TLIN",                                 // iconName
                standard_dialog,                        // dialogName
                "*",                                    // dialogData
                ComponentNetlistFmt,                    // netlistFormat
                "TLIN",                                 // netlistData
                ComponentAnnotFmt,                      // displayFormat
                "SYM_TLin",                             // symbolName
                no_artwork,                             // artworkType
                NULL,                                   // artworkData
                ITEM_PRIMITIVE_EX,                      // extraAttrib
                create_parm ("Z", "Characteristic Impedance", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", RESISTANCE_UNIT, prm("StdForm","50.0")),
                create_parm ("E", "Electrical Length", PARM_OPTIMIZABLE | PARM_STATISTICAL,
"StdFileFormSet", ANGLE_UNIT, prm("StdForm","90")),
                create_parm ("F", "Reference Frequency for Electrical Length", PARM_OPTIMIZABLE |
PARM_STATISTICAL, "StdFileFormSet", FREQUENCY_UNIT, prm("StdForm","1 GHz")));
```

Table 15-1. create_item() Attribute Choices

Attribute	Numeric Equivalent	Description
ITEM_BEND	2048	Component is a transmission line bend.
ITEM_BOM_ITEM	262144	Component is included in a BOM.
ITEM_DESIGN_INST	16	Component refers to external design.
ITEM_DRAWSHEET	16384	Component is a drawing sheet or PCB outline.
ITEM_GLOBAL	512	Component has global scope, such as MSUB.
ITEM_GLOBAL_NODE	536870912	Component is a global node.
ITEM_GROUND	1	Component is a ground.
ITEM_NO_LAYOUT_ANNOT	131072	Component does not generate annotation.
ITEM_NOT_ALL_PARM	1073741824	Component does not have an “all parameters” selection in the standard dialog box.
ITEM_NOT_NETLISTED	268435456	Component is not netlisted.
ITEM_NOT_PLACABLE	134217728	Component is not placable.
ITEM_PORT	32	Component is a port.
ITEM_SCOPE_GLOBAL	16777216	Component has global scope.
ITEM_SCOPE_LOCAL	4194304	Component has local scope.
ITEM_SCOPE_NESTED	8388608	Component has nested scope.
ITEM_TEE	4096	Component is a transmission line connection tee.
ITEM_UNIQUE	8	Only one instance of this component can be placed in a design.
ITEM_VARIABLE	2	Component is an equation/variable definition.

Table 15-2. create_item() Extra Attribute Choices

Attribute	Numeric Equivalent	Description
ITEM_ANALYSIS_EX	16384	The component is an analysis item.
ITEM_CKT_MODEL_EX	64	Identifies the model as Analog/RF type.
ITEM_CROSS_EX	8	
Note: If none of the ITEM_INITIAL_ORIENTATION_* options are set, use the previously-selected orientation.		

Table 15-2. create_item() Extra Attribute Choices (continued)

Attribute	Numeric Equivalent	Description
ITEM_CURVE_EX	1	
ITEM_FILE_EX	4096	Identifies file component for the Edit Component dialog.
ITEM_FREQUENCY_EX	1024	Identifies frequency component for the sparam and freqplan dialog.
ITEM_GOAL_EX	262144	The component is a goal item.
ITEM_INITIAL_ORIENTATION_DOWN_EX	524288	Place -90.
ITEM_INITIAL_ORIENTATION_LEFT_EX	2097152	Place 180.
ITEM_INITIAL_ORIENTATION_RIGHT_EX	4194304	Place 0.
ITEM_INITIAL_ORIENTATION_UP_EX	1048576	Place 90.
ITEM_MEASUREMENT_EX	32768	The component is a measurement item.
ITEM_MSTRIP_EX	2	
ITEM_NO_SPL_CHAR_EX	268435456	Only allow alpha-numeric or underscore characters in the component instance name.
ITEM_NOT_AUTO_REPEAT_PLACEMENT_EX	8388608	Place one component at a time.
ITEM_PCB_EX	16	
ITEM_PRIMITIVE_EX	32	Identifies the associated item as a compiled component contained in the simulator.
ITEM_SPEC_EX	131072	The component is a yield specification item.
ITEM_STRIP_LINE_EX	4	

Note: If none of the ITEM_INITIAL_ORIENTATION_* options are set, use the previously-selected orientation.

create_parm()

Creates a parameter definition for a component. Returns a component parameter. The return value is used by *create_item()*.

See also: “Form Sets” on page 4-2.

Syntax:

create_parm(name, label, attrib, formSet, unitCode, defaultValue[, cbList]);

where

name is a string; name of the parameter.

label is a string; descriptive label for the parameter.

attrib is an integer; an attribute code. Choices for parameters are:

PARM_LAYOUT = Disable layout-related parameters if layout is not licensed.

PARM_NO_DISPLAY = Do not show parameter on schematic by default.

PARM_OPTIMIZABLE = Allow optimization entry page.

PARM_STATISTICAL = Allow statistical entry page.

PARM_DOE = Allow doe entry page.

PARM_REPEATED = This is a repeated parameter; s[1]=100; s[2]=200;

PARM_RIGHT_HAND_ONLY = Do not generate left hand side when netlisting. Can be tested with netlist format %30. Also, when this attribute is set, the onscreen edit can partition the parameter displayed in the annotation into separate fields according to the annotation fmtstring, such as: start=%1s step=%2s.

PARM_NOT_ON_SCREEN_EDITABLE = The parameter value is not on-screen editable.

PARM_REAL = For real parameter numeric value, the prompt in the standard component dialog box displays *Real*.

PARM_INT = For integer parameter numeric value, the prompt in the standard component dialog box displays *Integer*.

PARM_STRING = For string parameter numeric value, the prompt in the standard component dialog box displays *String*.

PARM_COMPLEX = For complex parameter numeric value, the prompt in the standard component dialog box displays *Complex, 1.2+j2.5*.

PARM_FIX = For fixed point parameter numeric value, the prompt in the standard component dialog box displays *Fixed Point*.

PARM_PRECISION = For precision parameter numeric value, the prompt in the standard component dialog box displays Precision, 8.24.

PARM_INTARRAY = For integer array parameter numeric value, the prompt in the standard component dialog box displays Array, 1 2 3 ....

PARM_FIXARRAY = For fixed point array parameter numeric value, the prompt in the standard component dialog box displays Array, 1.25 3.5 ....

PARM_REALARRAY = For real array parameter numeric value, the prompt in the standard component dialog box displays Array, 1.5 3.6 ....

PARM_COMPLEXARRAY = For complex array parameter numeric value, the prompt in the standard component dialog box displays Array, (1.5,3.6)(2.4,4.7).

PARM_STRINGARRAY = For string array parameter numeric value, the prompt in the standard component dialog box displays Array, (str1)(str2).

PARM_DISCRETE_VALUE = For discrete parameter numeric value. Refer to "Designing a Discrete Valued Parts Parametric Subnetwork" on page 4-20.

PARM_NOT_EDITED = The parameter value is not editable.

PARM_NOT_NETLISTED = The parameter should not be netlisted.

formSet is the name of the form set used for this value. This is the same as *name* in the *create_form_set()* function.

unitCode is the choice for units, where:

STRING_UNIT = -2
UNITLESS_UNIT = -1
FREQUENCY_UNIT = 0
RESISTANCE_UNIT = 1
CONDUCTANCE_UNIT = 2
INDUCTANCE_UNIT = 3
CAPACITANCE_UNIT = 4
LENGTH_UNIT = 5
TIME_UNIT = 6
ANGLE_UNIT = 7
POWER_UNIT = 8

VOLTAGE_UNIT = 9
CURRENT_UNIT = 10
DISTANCE_UNIT = 11
TEMPERATURE_UNIT = 12
DB_GAIN_UNIT = 13

defaultValue is optional; a value returned from the *prm()* function. The *prm()* function generates an acceptable default value for parameters with different form sets.

cbList is optional; the list of callbacks. For example,

list(dm_create_cb ("...", "..."), ...);. Currently-supported callbacks are:

PARM_DEFAULT_VALUE_CB
PARM_MODIFIED_CB

Example:

This example creates a parameter C representing capacitance, with attributes, using the StdFileFormSet form set, capacitance as the units and 1.0 pF as the default value. This might then be used by *create_item()* to create a capacitance with C as its parameter.

```
create_parm("C", "Capacitance", PARM_OPTIMIZABLE | PARM_STATIS
"StdFileFormSet", "CAPACITANCE_UNIT", prm( "StdForm", "1.0 pF"));
```

create_text_form()

Creates a form whose set of possible values are a set of strings. These can be either a simple list of strings or a list of strings dynamically determined when an item referencing the form is placed. Returns: none.

See also: "Format Strings" on page 4-5.

Syntax:

create_text_form(name, label[, dialogDataStr], attrib, netlistFormat, displayFormat[, optionFunction, validateFunction, dataValue, dataFunction]);

where

name is the unique name of the form.

label is a string describing this form.

dialogDataStr is optional; any dialog data. For the standard edit component dialog box, this argument indicates the gui component to use for the breakdown of the

parameter value input. For example, the standard Edit Component dialog supports these gui configurations:

"StdForm"
"StringAndReference"
"SingleTextLine"
"InstSelectionForm"
"Node SetForm"

This argument can be omitted if dialogDataStr is the same as formName. For the standard component dialog, this field may be used as the name of the table entry field. For a complete listing, see *$HPEESOF_DIR/de/ael/pde_gemini.ael*.

attrib is an integer, usually 0, describing the or'd attribute bits for this form.

netlistFormat is a string describing the netlisting of this form's value. Refer to "Format Strings" on page 4-5..

displayFormat is a string describing the display in the schematic of the values of this form.

optionFunction is optional; a function used to dynamically collect strings.

validateFunction is optional; a function to validate the strings collected by the option function.

dataValue is optional; a string; passed to the optionFunction, validateFunction, and dataFunction.

dataFunction is optional; a function to provide special control over netlist generation.

Example:

```
create_text_form("SingleTextLineInteger", "IntegerValue", "SingleTextLine",
0, "%v","%v", NULL, stdforms_validate_integer, NULL);
```

de_define_palette_group()

Defines a palette group. The palette group is displayed in the palette dialog. Bitmaps for the components are displayed in the active palette (if defined for the component, otherwise the component name appears). Returns: none.

Note This function should be called at bootup for the palette to be displayed in the palette dialog.

Syntax:

de_define_palette_group(winType, dsnType, groupName, groupLabel, position, item1name, item1label, item1bitmap name[, item2name, item2 label, item2 bitmap name, ...]);

where

winType is the type of window, where:

SCHEM_WIN = Schematic window
LAYOUT_WIN = Layout window

dsnType is the unique code for each design type, such as *analogRFnet* or *sigprocNet*.

groupName is the name of the palette group.

groupLabel is the descriptive label for the group.

position is an integer: -1 to alphabetically sort the list of palettes after adding the new palette, -2 to append the new palette at the end, any other integer (0 or greater) to insert the new palette in the specified position in the list of palettes.

item1name is the first component name.

item1label is the first component label.

item1bitmap name is the first component bitmap file name.

item2name is optional; the second component name.

item2label is optional; the second component label.

item2bitmap name is optional; the second component bitmap file name.

Example:

```
de_define_palette_group(
 SCHEM_WIN, "analogRF_net", "mypal", "My palette", -2, "R",
 "Resistor", "R", "C", "Capacitor", "C"
 );
```

dm_create_cb()

Defines a callback function. Callbacks can be registered with *create_parm()* and *create_item()* function calls. For example:

```
create_item( ...list(dm_create_cb(...), dm_create_cb() ...));
create_parm(...list (dm_create_cb(...), dm_create_cb() ...));
```

Also, the callback function is invoked in the event of component/parameter activities. The cbFunction is called with the following protocol:

```
cbFunction (cbP, clientDataStr, callData)
```

Returns a callback function.

Syntax:

```
dm_create_cb(cbType, cbFunctionNameStr, clientDataStr[, enableFlag]);
```

where

cbType is the type of callback function, where:

PARM_DEFAULT_VALUE_CB = the callback function is called when the parm definition default value is needed in the pde program.

PARM_MODIFIED_CB = the callback function is called when the parameter value is changed.

ITEM_NETLIST_CB = the callback function is called when the instance of the component is netlisted.

cbFunctionNameStr is the name of the cb function. This function takes three arguments: The first is the callback pointer (for program use). The second is the clientDataStr. The third is calldata (for program use).

clientDataStr is the client data in string format. This gets passed to the cb function as the second argument.

enableFlag is a flag, where TRUE = enable this callback function. If disabled, the callback function will not be invoked.

Example:

The example demonstrates how to specify a default value callback so the default value can be retrieved dynamically.

```
create_item( "myItem", "My Item", "MyItem", 0, "myitem", standard dialog, *,
ComponentNetlistFmt, "myItem", componentAnnotFmt, "SYM_MyItem", no_artwork,
NULL, ITEM_PREMITIVE_EX,
create_parm( "Len", "Length", 0, "StdFormSet", LENGTH_UNIT,
prm( "StdForm", "1.0 mil"),
list( dm_create_cb(PARM_DEFAULT_VALUE_CB,
"my_item_get_default_inductance_cb", "", TRUE))));

defun my_item_get_default_inductance_cb(
  cbP, // not used
  clientDataStr,  // not used
  callData) // the parmDef
{
 if(de_get_current_length_unit()=="mil")
  return prm("StdForm", "10 mil");
 if(de_get_current_length_unit()=="mm")
  return prm("StdForm", "2 mm");
 if(de_get_current_length_unit()=="um")
  return prm("StdForm", "1.0 um");
 return prm("StdForm", "1.0 mil");
}
```

library_group()

Defines a new library group composed of the listed components. Adds the group to the current component dictionary. Since groups are associated with the window by the type of design displayed in the window, the desired type of design must be indicated by a call to *set_design_type()* prior to calling this function. The library label appears in the library dialog, along with all the components in the group. Returns: none.

Syntax:

library_group(name, label[,bToFront], item1, item2, ..., itemN);

where

name is the name of the library group.

label is a descriptive label for the library.

bToFront is optional boolean flag. If TRUE, the group will be added to the front of the library group list. If FALSE (the default), the group will be added to the end of the library group list.

itemN is a defined component name; one or more can be given.

Example:

```
library_group("ckt ac sim group", "AC Simulation", "AC", "Options",
"SweepPlan", "ParamSweep", "NodeSet", "NodeSetByName");
```

prm()

Creates and returns a default parameter value. This is most frequently used in conjunction with *create_parm()* to assign a default value to an component parameter. Returns: A valid default parameter value for the given form.

Syntax:

prm(formName, paramValue1, paramValue2,..., paramValueN);

where

formName is a string; name of a form defined with *create_form()*. The form name must be one of the forms in the parameter's formset, specified in *create_parm()*.

ParamvalueN is one or more default values for the parameter, depending on the construction of the form:

For text forms, the value is a single string, in quotes (see *create_text_form*).

For constant forms, no value is required (see *create_constant_form*).

For compound forms, the values of each field are described hierarchically, as with the default value of the *create_parm*; that is, each field value requires a *prm()* function as is appropriate to the form set and desired form in the form set for the field (see *create_compound_form*).

Example:

```
prm("stdForm", "500");
```

Typically used as in:

```
create_parm("depVar","DependentVariable", PARM_NO_DISPLAY,
 "StdFileFormSet", UNITLESS_UNIT, prm("StdForm", ""));

create_parm("Variable Value", "Variable equation",
 PARM_OPTIMIZABLE|PARM_RIGHT_HAND_ONLY|PARM_REPEATED|
 PSTM_STATISTICAL, "VarEqnForms", UNITLESS_UNIT,
 list(prm("VarFormEditCompPowerVar", prm("VarNameForm","X"),
 prm("editcompPowerVar", "1.0"))));
```

reference_library_group()

Creates a new library group for the current design type by reference to an existing component group for another design type. Since groups are associated with the window by the type of design displayed in the window, the desired type of design must be indicated by a call to *set_design_type()* prior to calling this function. The new group will be an alias for the existing group, and not a separate group, so the list of components for the group is not duplicated. An attempt to add components using *library_group()* to a group created using this function is ignored, but adding components to the referenced group affects all groups referencing it as well. The group created by calling this function will be a library group, but may refer to either a library or palette group for another design type. Returns: none.

See also *reference_palette_group(), library_group()*, and *set_design_type()*.

Syntax:

reference_library_group(name, design_type, palette_flag[,bToFront]);

where

name is the name of referenced group.

design_type is the design type code for the referenced group.

palette_flag is a flag indicating whether the referenced group is a library or palette group, where:

0 = library (default)
1 = palette

bToFront is optional boolean flag. If TRUE, the group will be added to the front of the library group list. If FALSE (the default), the group will be added to the end of the library group list.

Example:

```
set_design_type( librabenlin );
reference_library_group( “Project Circuit”, libranet, 0 );
```

reference_palette_group()

Creates a new palette group for the current design type by reference to an existing component group for another design type. Since groups are associated with the window by the type of design displayed in the window, the desired type of design must be indicated by a call to *set_design_type()* prior to calling this function. The new

group will be an alias for the existing group, and not a separate group, so the list of components for the group is not duplicated. An attempt to add components using *de_define_palette_group()* to a group created using this function is ignored, but adding components to the referenced group affects all groups referencing it as well. The group created by calling this function will be a palette group, but may refer to either a library or palette group for another design type. Returns: none.

See also *reference_library_group()*, *de_define_palette_group()*, and *set_design_type()*.

Syntax:

reference_palette_group(name, design_type, palette_flag[,bToFront]);

where

name is the name of referenced group.

design_type is the design type code for the referenced group.

palette_flag is a flag indicating whether the referenced group is a library or palette group, where:

0 = library (default)
1 = palette

bToFront is optional boolean flag. If TRUE, the group will be added to the front of the palette group list. If FALSE (the default), the group will be added to the end of the palette group list.

Example:

```
set_design_type( librabenlin );
reference_palette_group( “Project Circuit”, libranet, 0 );
```

set_design_choices()

Defines choices for characteristics allowed for parametric subnetwork components. Returns: none.

Syntax:

set_design_choices(choice type, label, value ...);

where

choice type is the choice for parametric subnetwork components characteristics, where:

0 = artwork, such as “AEL macro”
1 = component attribute, such as “Layout Object”
2 = parameter attribute, such as “Not Edited”
3 = parameter form set, such as “SmtPADDataOnly (layout)”
4 = netlist model, such as “Subnetwork”
5 = symbol, such as “SYN_OPort”
6 = model parameter attribute, such as “Optimizable”
7 = parameter value type attribute, such as “Real”

label value pairs is one or more label strings paired with values, where the values are integers for types 0, 1, 2, 6, and 7 or strings for types 3, 4, or 5.

Example:

```
set_design_choices(3, "Optimizable", "StdFormSet",
 "Integer", "StdFormSet",
 "Real", "StdFormSet",
 "SmtPAD Data Only (layout),
 push in ‘not netlisted’ button", "StdFormSet");
```

set_design_sub_choices()

Defines sub-choices for each characteristic choice for parametric subnetwork components. Returns: none.

Syntax:

set_design_sub_choices(choice type, choice index, label, value ...);

where

choice type is a choice type, where:

0 = artwork
1 = component attribute
2 = parametric attribute
3 = parameter form set
4 = netlist model
5 = symbol
6 = model parameter attribute
7 = parameter value attribute

choice index is an Index indicating which label/value pair from *set_design_choices()* to attach sub-choices, where 0 is for the first pair.

label value pairs is one or more label strings paired with values, where the values are integers or strings.

Example:

```
set_design_sub_choices(0, 2, "conn", 0, "cpad2", 1, "cpad3", 2);
//to set macro artwork choices. In this case the value in the label
//value pairs is unimportant.
```

set_design_type()

Sets the current type of design for group definitions. Library and palette groups are associated with design windows according to current type of design. The design type must be set using this function before the function *library_group()* will work. Returns: none.

See also: *set_simulator_types()*.

Syntax:

set_design_type(type);

where

type is a unique code for each design type, as defined in stddefs.ael, such as *analogRFnet* or *sigprocNet*.

Example:

```
set_design_type(analogRFnet);
```

set_netlist_info()

Sets special netlisting characteristics for current type of design. Returns: none.

See also: "Format Strings" on page 4-5.

Syntax:

set_netlist_info(prefix format, suffix format, priority, in format, out format, input format);

where

prefix format is a special format string for beginning of a subnetwork netlist.

suffix format is a special format string for end of a subnetwork netlist.

priority is an integer priority code used in ordering design netlist in netlist output; usually set to 1.

in format is a string, not used.

out format is a string, not used.

input format is a string, not used.

Example:

```
/* set the format string for subnetwork netlist generation */
set_netlist_info("define %n (%@ %32?%M%:_net%m%; %e) \n%37?parameters%:;
parameters%; %b %k=%p %e", "end %n", 1, "", "", "");
```

set_simulator_type()

Sets the current simulator. Used to associate commands and creation functions with a simulator type. It must be set before any *create_item()* or *create_*_form()* function is called to correctly associate a simulator type with a component, form, or formset definition. Returns: none.

Note: In the *create_*_form()* function, where * can be one of these: *create_compound_form()*, *create_text_form()*, or *create_constant_form()*.

Syntax:

set_simulator_type(simulator_type);

where

simulator_type is the choice for simulator type, where:

1 = Hptolemy
–1= Hptolemy or Analog/RF

Example:

```
set_simulator_type(1);
```

Chapter 16: Component Definition Query Functions

This chapter describes each Item Definition Query function in detail. The functions are listed in alphabetical order.

dm_find_form_definition()

Returns a handle to a form definition given the form name and, optionally, a simulator ID. Then this handle can be used by the function *dm_get_form_definition_attribute()* to retrieve other attributes of the form.

Syntax:

dm_find_form_definition(name [, simulator]);

where

name is the name of the form.

simulator is the optional simulator ID code for the dictionary where the form is stored. The current simulator set by *set_simulator_type()* is assumed if this parameter is omitted.

Example:

```
decl formH, netFmt;
formH = dm_find_form_definition("stdForm", 1);
netFmt = dm_get_form_definition_attribute(formH, DM_FORM_NET_FORMAT);
```

dm_find_item_definition()

Returns an handle to an item definition given the item's name. The current design type simulator dictionary is used to find the item definition. Current design type can be set by *set_design_type()*.

Syntax:

dm_find_item_definition(name);

where

name is the name of the item.

Example:

```
decl item;
```

```
item = dm_find_item_definition("R");
```

dm_first_parm_definition()

Returns the handle to the first parameter definition of an item, given a parameter definition handle as returned from *dm_get_item_definition_attribute()*.

Syntax:

dm_first_parm_definition(parmHandle);

where

parmHandle is the handle to a parameter definition.

Example:

```
decl parmH, parmList, itemH;
itemH = dm_find_item_definition("R");
parmList = dm_get_item_definition_attribute(itemH, ITEM_PARMS);
parmH = dm_first_parm_definition(parmList);
```

dm_get_design_class_code()

Returns a code representing the design class; where: 0 = network design.

Syntax:

dm_get_design_class_code(designHandleOrDesignCode);

where

designHandleOrDesignCode is a handle to a design, as returned from *db_get_design()* or a design type code as required by *set_design_type()*, such as *analogRFnet* or *sigprocNet*.

Examples:

```
decl code, designHandle;
code = dm_get_design_class_code(analogRFnet);

designHandle = db_get_design("myDesign");
code = dm_get_design_class_code(designHandle);
```

dm_get_design_name()

Returns a string, the generic group name defined in AEL for a particular type of design.

Syntax:

dm_get_design_name(designHandleOrDesignCode);

where

designHandleOrDesignCode is a handle to a design, as returned from *db_get_design()* or a design type code as required by *set_design_type()*.

Example:

```
decl type;
type = dm_get_design_name(analogRFnet); // returns string analogRFnet
type = dm_get_design_name(sigprocNet); // returns string sigprocNet
```

dm_get_form_definition_attribute()

Returns the value of a form definition attribute, given a handle to the form definition. These attributes are defined when the form is created with *create_compound_form()*, *create_text_form()*, or *create_constant_form()*.

Syntax:

dm_get_form_definition_attribute(form, attribute [,index]);

where

form is a form handle, as returned from *create_*_form()*.

Note: In the *create_*_form()* function, where * can be one of these: *create_compound_form()*, *create_text_form()*, or *create_constant_form()*.

attribute is the data for the form selected from this list:

DM_FORM_NAME is the unique name of the form.

DM_FORM_LABEL is the descriptive label for the form.

DM_FORM_NET_FORMAT is the format string to netlist the parameter value.

DM_FORM_DISP_FORMAT is the format string to display the parameter value in a schematic.

DM_FORM_FIELDS are the labels for the form fields.

DM_FORM_ATTR are the attribute codes for the form. See *create_*_form()* for more information.

DM_FORM_NUMFIELDS is the number of fields in the form.

DM_FORM_CLASS is the general class of the form.

DM_FORM_PARMS is the list of parameters defining a form. Used in compound forms.

index is optional; the index specifying which form, in a complex form.

Example:

```
decl Label, formH;
formH = dm_find_form_definition("StdForm");
Label = dm_get_form_definition_attribute(formH, DM_FORM_LABEL);
```

dm_get_item_definition_attribute()

Returns the value of an item definition attribute given a handle to the item.

Syntax:

dm_get_item_definition_attribute(item, attribute);

where

item is a handle to item definition, as returned by *dm_find_item_definition()*.

attribute is the type of data to retrieve from the item definition. These attributes are described in the item's AEL definition created with the *create_item()* function. The meaning of these fields is documented in existing AEL documentation under *create_item()*.

ITEM_NAME is a unique name of the item.

ITEM_LABEL is a descriptive label for the item.

ITEM_ATTR is an integer, attribute code. See *create_item()* for more information.

ITEM_PRIORITY is the item netlisting priority.

ITEM_PREFIX is the instanceName prefix, such as the C in C12.

ITEM_DIALOG_CODE is the dialog box used to edit item.

ITEM_DIALOG_DATA is the descriptive string displayed in dialog.

ITEM_NET_FORMAT is a netlist formating string.

ITEM_NET_DATA is the data used by the netlist format.

ITEM_SCHEM_FORMAT Like a netist formating string describes the format of parameter display on schematic.

ITEM_SCHEM_NAME is the name of a design containing the schematic symbol.

ITEM_ART_TYPE is the type of artwork, where: 0 = no artwork; 1 = fixed artwork; 2 = AEL macro artwork.

ITEM_ART_DATA is the name of a layout artwork design for fixed artwork and is an ael artwork generation function name for AEL macro artwork.

ITEM_ICON is the name of the icon bitmap file for the palette. If NULL, use ITEM_NAME.

ITEM_PARMS is the parameters created with *create_parm()*.

ITEM_SIMTYPE is the simulator code (simcode). Simcode is a unique code for each simulator, such as circuit or system.

ITEM_ATTR_EX is an integer; extra attribute code. See *create_item()* for more information.

Example:

```
decl attr, itemH;
itemH = dm_find_item_definition("MLIN");
attr= dm_get_item_definition_attribute(itemH, ITEM_ATTR);
```

dm_get_parm_definition_attribute()

Returns a value of a parameter definition attribute, as defined with *create_parm()*. The meaning of these fields are documented under *create_parm()*.

Syntax:

dm_get_parm_definition_attribute(parmHandle, attribute);

where

parmHandle is the handle to a parameter definition.

attributes is the type of data to retrieve from the parameter definition, where:

DM_PARM_NAME is the name of the parameter.

DM_PARM_LABEL is a descriptive label for the parameter.

DM_PARM_ATTR is an integer, attribute code. See *create_parm()* for more information.

DM_PARM_UNIT is the unit for the parameter.

DM_PARM_FORMSET is the name of the formset used for this parameter value.

DM_PARM_DEFVALUE is the default value of the parameter, returned from the *prm()* function. See *prm()* for more information.

Example:

```
decl itemH, parmList, parmH, parmName;
itemH = dm_find_item_definition("R");
parmList = dm_get_item_definition_attribute(itemH, ITEM_PARMS);
parmH = dm_first_parm_definition(parmList);
parmName = dm_get_parm_definition_attribute(parmH, DM_PARM_NAME);
```

dm_get_simcode_from_designcode()

Returns a simcode, which is required by functions such as *dm_find_form_definition()*, given a design code, which is returned from functions such as *db_get_design_code()*.

Syntax:

dm_get_simcode_from_designcode(designCode);

where

designCode is the unique code for each type of design, such as *analogRFnet* or *sigprocNet*.

Example:

This example sets simCode to analogRFnet.

```
decl simCode;
simCode = dm_get_simcode_from_designcode(analogRFnet);
```

dm_index_parm_definition()

Returns a handle to the indexed parameter definition.

See also: *dm_first_parm_definition(), dm_next_parm_definition()*.

Syntax:

dm_index_parm_definition(parmHandle, indx);

where

parmHandle is the handle to a parameter definition.

indx is the indexed parameter to retrieve.

Examples:

```
decl itemH, parmList, parmH;
itemH = dm_find_item_definition("R");
parmList = dm_find_item_definition_attribute(itemH, ITEM_PARMS);
// retrieve the 2nd parameter of the item
parmH = dm_index_parm_definition(parmList, 2);

// retrieve the 3rd parameter of the item
parmH = dm_first_parm_definition(itemH);
parmH = dm_index_parm_definition(parmH, 3);
```

dm_next_parm_definition()

Returns the next handle to the parameter definition of an item, given a parameter handle as returned from *dm_first_parm_definition()*.

Syntax:

dm_next_parm_definition(parmHandle);

where

parmHandle is the handle to a parameter definition.

Example:

```
decl itemH, parmH;
itemH = dm_find_item_definition("R");
parmH = dm_first_parm_definition(itemH);
parmH = dm_next_parm_definition(parmH);
```

dm_num_parm_definitions()

Returns the number of parameter definitions in a parameter list, given a parameter definition handle.

Syntax:

dm_num_parm_definitions(parmHandle);

where

parmHandle is the handle to a parameter definition.

Example:

```
decl num, parmH, itemH;
itemH = dm_find_item_definition("MLIN");
parmH = dm_first_parm_definition(itemH);
num = dm_num_parm_definitions(parmH);
```

Chapter 17: Simulator Command Functions

This chapter describes each Simulator Command function in detail. The functions are listed in alphabetical order.

clear_server()

Clears the busy status state of the active server (simulator). This command helps to clear the server after a communications error. Returns: none.

Syntax:

clear_server(server_handle);

where

server_handle is a handle to simulator/server.

Example:

```
clear_server(serverHandle);
```

create_server()

Creates a handle to a server process that can be controlled through AEL. If the server process is not currently running, the server process is spawned automatically when it is sent a message. That is, you do not need to invoke the server process explicitly, but can simply start sending commands and data to it through AEL and you can be assured the process will be up and running. The server process receives commands and data and returns information through AEL.

Typically the server program is an HP EEsof program that communicates dynamically through a special message protocol, but it can be an ordinary program communicating using stdin and stdout. For non-HP EEsof programs, every data message sent to the server is received by the program as a line to stdin and every line printed to stdout by the program is returned as a data message. Returns the handle of the server.

See also: *send_server_command(), send_server_data(), send_server_interrupt(), send_server_kill(), start_server(), push_message_handler(), and pop_message_handler()*.

Syntax:

create_server(name, label, hostMachine [, params, nonEmx, paramAelFunc, createUnique, fReinstate]);

where

name is the name of the server program.

label is the description of program.

hostMachine is the name of host machine.

params is optional; the parameter string to be passed to program when spawned.

nonEmx is optional; the flag indicating whether server will be non-HP EEsof program, which cannot take advantage of special communications protocol, where:

0 = (default) for HP EEsof programs
1 = non-HP EEsof program

paramAelFunc is optional; the name of an AEL function which can be used to generate information to be passed to server when it is spawned. The name of the AEL function is provided as a string (enclosed in quotes). The function will be passed the original parameter string, and should return a new parameter string.

createUnique is optional; the flag indicating that more than 1 server process with the same name can be invoked: 0 = (default) for returning existing server with the same name, and 1 = create another server process even if one with the same name is already running

fReinstate is optional; the flag indicating whether to notify other processes that a new server process has been invoked, where:

0 = (default) Invoke a new server process and do not notify the other proceeses in the session

1 = Invoke a new server process and notify other processes in the session

If the server process was spawned previously, but has died since, the reinstate flag indicates whether or not other processes in the session get notified of the newly-spawned server process. Notification ensures that the newly-spawned server will be able to communicate with these processes.

Example:

```
decl surHandle=create_server("myserver", "MyServerProcess", “”);
```

de_analyze()

Invokes the simulator and sends it a new, updated netlist of the current design. Options specified in the simulation setup dialog will be used for the simulation. Returns: none.

Syntax:

de_analyze();

Example:

```
de_analyze();
```

de_analyze_tune()

Invokes the simulator, sends the simulator a new, updated netlist of the current design, and prepares the simulator to receive de_tune() commands. Options specified in the simulation setup dialog are used for the simulation. Returns: none.

See also: *de_tune()*, *de_tune_deinit()*.

Syntax:

de_analyze_tune();

Example:

```
de_analyze_tune();
```

de_close_all_datadisplay()

Hides all data display windows that have been opened. Returns: none.

See also: *de_restore_all_datadisplay()*.

Syntax:

de_close_all datadisplay();

Example:

```
de_close_all_datadisplay();
```

de_open_datadisplay()

Opens a saved data display file. Sends the *OPEN_FILE* command, with the name of the file to load, to the data display server. Returns: none.

See also: *de_new_datadisplay()*.

Syntax:

de_open_datadisplay(fileName[, addcwd]);

where

fileName is the name of the saved graphics state file.

addcwd is the flag to indicate whether the current working directory should be pre-pended to the fileName.

Example:

```
de_open_datadisplay("mydata");
```

de_restore_all_datadisplay()

Opens all data display windows that have been hidden, using *de_close_all_datadisplay()*. Returns: none.

See also: *de_close_all_datadisplay()*.

Syntax:

de_restore_all_datadisplay();

Example:

```
de_open_datadisplay("usr/test_prj/plot1.dds");
de_open_datadisplay("usr/test_prj/plot2.dds");
de_close_all_datadisplay();
de_restore_all_datadisplay();
```

de_restore_status()

Opens the closed status server window. Returns: none.

Syntax:

de_restore_status();

Example:

```
de_restore_status();
```

de_tune()

Activates a tune analysis after updating parameters to the values specified in the paramList. The function de_analyze_tune() must be called (once) to set up the

conditions for tuning before the function de_tune() is called. Options specified in the simulation setup dialog are used for the simulation. Returns: none.

See also: *de_analyze_tune()*, *de_tune_deinit()*.

Syntax:

de_tune(paramList);

where

paramList is a list of parameter names and values. The parameter name and value pairs are separated by the bar character (|), and the list is terminated by a bar.

Example:

```
de_tune("C2.C 5.131 pF|SRC3.Vdc 1.168 V|");
```

de_tune_deinit()

Terminates the tuning operation in the simulator. This function should be called after all tuning analyses have been performed. Returns: none.

See also: *de_analyze_tune()*, *de_tune()*.

Syntax:

de_tune_deinit();

Example:

```
de_tune_deinit();
```

de_turn_off_trace_history()

Deactivates trace history mode. Returns: none.

See also: *de_turn_on_trace_history()*.

Syntax:

de_turn_off_trace_history()

Example:

```
de_turn_off_trace_history();
```

de_turn_on_trace_history()

Activates trace history mode, and sets the number of traces to be displayed. Returns: none.

See also: *de_turn_off_trace_history()*.

Syntax:

de_turn_on_trace_history(numTraces)

where

numTraces is the number of history traces to be displayed.

Example:

```
de_turn_on_trace_history(7)
```

de_update_optimization_values()

Requests updated optimization/yield results from any optimization or yield analysis performed by the current server (simulator). All designs referenced by the DUT are updated. Returns: none.

Syntax:

de_update_optimization_values();

Example:

```
de_update_optimization_values();
```

pop_message_handler()

Restores a saved message handler for a server to the state previous to the last call to *push_message_handler()*. See also *push_message_handler()*. Returns: none.

See also: *push_message_handler()*, *create_server()*.

Syntax:

pop_message_handler(server_handle);

where

server_handle is a handle of simulator/server returned by *create_server()*.

Example:

```
pop_message_handler( sh );
```

push_message_handler()

Installs a message handler function to receive messages from a server. When a message is received from the server, the handler function is called and passed the server handle, type code for the message, and the message text. The message type codes are:

0 = command
1 = data
2 = begin data block
3 = continue data block
4 = end data block
5 = done (server acknowledge of command)
6 = error
7 = server died

For non-HP EEsof server programs, only data messages may be exchanged. Returns: none.

See also: *pop_message_handler()*, *create_server()*.

Syntax:

push_message_handler(messageFn, server_handle);

where

messageFn is the name of an AEL function to receive messages from server.

server_handle is a handle of simulator/server returned by *create_server()*.

Example:

```
decl my_server=create_server("myserver", "MyServerProcess", NULL);
defun print_message( sh, type, msg )
{
 if (type == 0)
  fputs( stderr, strcat( "Command:",msg ));
 if (type == 1)
  fputs( stderr, strcat( "Data:",msg ));
 if (type == 2)
  fputs( stderr, strcat( "Start data:",msg ));
 if (type == 3)
  fputs( stderr, strcat( "Cont data:",msg ));
 if (type == 4)
  fputs( stderr, strcat( "End data:",msg ));
 if (type == 5)
  fputs( stderr, strcat( "Done:",msg ));
```

```
 if (type == 6)
  fputs( stderr, strcat( "Error:",msg ));
 if (type == 7)
  fputs( stderr, strcat( "Server died:",msg ));
}
push_message_handler( print_message, my_server );
```

send_server_command()

Sends a command to the server identified by server_handle. Prefix with the CMD command protocol statement. Returns: none.

See also: *create_server()*.

Syntax:

send_server_command(command, server_handle, [wait, invokeDir]);

where

command is the command string to send to simulator.

server_handle is a handle to simulator/server, returned by *create_server()*.

wait is optional; the flag indicating whether the function should return immediately or wait for the “DONE” mesage from the server, where:

0 = (default) for immediate return
1 = wait for “DONE” mesage

invokeDir is optional; the directory from which to invoke the server if necessary, where:

NULL = (default) for using current working directory
DirName = for using the given directory

Example:

```
decl sh = create_server("myServer", "myServerProcess", NULL);
send_server_command("MAP_WIN", sh, 1);
```

send_server_data()

Sends data statement to the server identified by server_handle. Prefix data with DATA statement. Returns: none.

See also: *create_server()*.

Syntax:

send_server_data(dataStatement, server_handle [, invokeDir]);

where

dataStatement is the data statement to send to the simulator.

server_handle is a handle to simulator/server, returned by *create_server()*.

invokeDir is optional; the directory in which to invoke the server, if necessary, where:

NULL = (default) for using current working directory
DirName = for using the given directory

Example:

```
decl sh = create_server("myServer", "myServerProcess", NULL);
send_server_data("new data", sh);
```

send_server_interrupt()

Sends an interrupt to the current server identified by server_handle. Returns: none.

See also: *create_server()*.

Syntax:

send_server_interrupt(server_handle[, message]);

where

server_handle is a handle to simulator/server, returned by *create_server()*.

invokeDir is optional; an interrupt message.

Example:

```
decl sh = create_server("myServer", "myServerProcess", NULL);
send_server_interrupt(sh);
```

send_server_kill()

Sends the special kill command (“KILL”) to the server identified by server_handle. Returns: none.

See also: *create_server()*.

Syntax:

send_server_kill(server_handle);

where

server_handle is a handle to simulator/server returned by *create_server()*.

Example:

```
decl sh = create_server("myServer", "myServerProcess", NULL);
send_server_kill(sh);
```

server_running()

Returns the status of the server identified by serverHandle, where: 1 (True) = the server identified by serverHandle is running, and 0 (False) = the server identified by serverHandle is not running.

See also: *create_server()*.

Syntax:

server_running(serverHandle);

where

serverHandle is the pointer for specific server to be evaluated, returned by *create_server()*.

Example:

```
decl running, SHandle=create_server("myserver", "MyServerProcess", NULL);
running = server_running(SHandle);
```

start_server()

Causes the server identified by serverHandle to be spawned. Returns: none.

See also: *create_server()*.

Syntax:

start_server(server_handle [, mapStderr, invokeDir]);

where

server_handle is a handle of simulator/server, returned by *create_server()*.

mapStderr is optional; sets stderr remap, where:

0 = (default) Do not remap stderr.

1 = Remap stderr to a pipe. A line written to stderr is returned as a DATA message.

invokeDir is optional; the directory in which to invoke the server, if necessary:

NULL = (default) for using current working directory
DirName = for using the given directory

Example:

```
decl hpeesofsim = create_server("simulator", "HPEESOF simulator", NULL);
start_server( hpeesofsim);
```

Chapter 18: User Interface Functions

This chapter describes each User Interface function in detail. The functions are listed in alphabetical order.

add_menu()

Adds an item, menuItemName, to the user-defined menu and associates the AEL function aelFuncName with the item. When you select the item, the function will be called. The user-defined menu appears in the menu bar of the current window and the menu name is set with *set_user_menu_label()*. Returns: none.

See also: *add_separator(), set_user_menu_label(), check_user_menu(), clear_user_menu()*.

Syntax:

add_menu(menuItemName, aelFuncName, menuName);

where

menuItemName is the name to display as a selectable item in the menu.

aelFuncName is a string. AEL function to call when item is selected (should have no arguments).

menuName is one of these pre-defined internal names. "User", "User2", "User3", "User4", "User5". It is also the name displayed on the top of the menu if it hasn't been changed by a call to set_user_menu_label().

Example:

This example only adds a menu to the first user menu and does not check first to see whether a menu slot has been used. For a complete example, which checks first for an empty slot, refer to “Creating a Custom Menu” in the Customization manual.

```
defun my_func()
{
  fputs(stderr, "my_func_ael");
}
add_menu("myfunc", "my_func_ael", "User");
```

add_separator()

Adds a separator in the menu. Returns: none.

See also: *add_menu(), set_user_menu_label(), check_user_menu(), clear_user_menu()*.

Syntax:

add_separator();

Example:

```
add_menu(“Command 1”, “default_cb”, NULL);
add_menu(“Command 2”, “default_cb”, NULL);
add_separator();
add_menu(“Command 3”, “default_cb”, NULL);
```

api_get_current_window()

Returns the handle to the currently active window.

Syntax:

api_get_current_window();

Example:

```
decl currentWindowHandle;
currentWindowHandle = api_get_current_window();
```

api_get_graph_color_index_by_name()

Returns the index of the color for the given color name; null if not found.

See also: *api_get_graph_color_name_by_index()*, *api_get_current_window()*, *api_get_window_graph()*.

Syntax:

api_get_graph_color_index_by_name(graphH, colorName);

where

graphH is a handle to the window graphics area, as returned from api_get_window_graph().

colorName is the color name; a string. A list of available strings can be found in the $HPEESOF_DIR / config directory in the hpeecolor.cfg file.

Example:

```
decl winInst, graphH, colorIndex;
winInst = api_get_current_window();
graphH = api_get_window_graph (winInst);
colorIndex = api_get_graph_color_index_by_name (graphH, "black");
```

api_get_graph_color_name_by_index()

Returns the name of the color for the given color index; null if not found.

See also: *api_get_graph_color_index_by_name()*, *api_get_current_window()*, *api_get_window_graph()*.

Syntax:

api_get_graph_color_name_by_index(graphH, colorIndex);

where

graphH is a handle to the window graphics area, as returned from api_get_window_graph().

colorIndex is the color index (integer >= 0).

Example:

```
decl winInst, graphH, colorName;
winInst = api_get_current_window();
graphH = api_get_window_graph (winInst);
colorName = api_get_graph_color_name_by_index (graphH, 1);
```

api_get_graph_pattern_index_by_name()

Returns the index of the pattern for the given pattern name; null if not found.

See also: *api_get_graph_pattern_name_by_index()*, *api_get_current_window()*, *api_get_window_graph()*.

Syntax:

api_get_graph_pattern_index_by_name(graphH, ptnName);

where

graphH is a handle to the window graphics area, as returned from api_get_window_graph().

ptnName is the pattern name; a string. A list of available strings can be found in the $HPEESOF_DIR / config directory in the hpeefill.cfg file.

Example:

```
decl winInst, graphH, ptnIndex;
winInst = api_get_current_window();
graphH = api_get_window_graph (winInst);
ptnIndex = api_get_graph_pattern_index_by_name (graphH, "zigzag_1");
```

api_get_graph_pattern_name_by_index()

Returns the name of the pattern for the given pattern index; null if not found.

See also: *api_get_graph_pattern_index_by_name()*, *api_get_current_window()*, *api_get_window_graph()*.

Syntax:

api_get_graph_pattern_name_by_index(graphH, ptnIndex);

where

graphH is a handle to the window graphics area, as returned from api_get_window_graph().

ptnIndex is the pattern index (integer >= 0).

Example:

```
decl winInst, graphH, ptnName;
winInst = api_get_current_window();
graphH = api_get_window_graph (winInst);
ptnName = api_get_graph_pattern_name_by_index (graphH, 1);
```

api_get_window_graph()

Returns a handle to the graphics area of the given window.

Syntax:

api_get_window_graph(windowHandle);

where

windowHandle is a handle of a window to retrieve the graphics area from.

Example:

```
decl windowHandle, graphHandle;
windowHandle = api_get_current_window();
graphHandle = api_get_window_graph(windowHandle);
```

api_set_current_window()

Sets a window to be currently active. Returns: the handle of the window replaced by the window specified by this function.

See also: *api_get_current_window()*, *api_set_current_window_by_seq_num()*.

Syntax:

api_set_current_window([windowType | windowHandle]);

where

windowType is the type of window, where:

MAIN_WIN = Main window
SCHEM_WIN = Schematic window
LAYOUT_WIN = Layout window

OR

windowHandle is a handle of a window to retrieve the graphics area from.

Example:

```
decl oldWinHandle;
oldWinHandle = api_set_current_window(SCHEM_WIN);
```

OR

```
oldWinHandle = api_set_current_window(windowHandle);
```

api_set_current_window_by_seq_num()

Makes the window instance having the given sequence number as the current window. This function can be used to select the exact window instance of windows that can have multiple instances. Returns: The handle to the previous window instance.

See also: *de_create_window()*, *api_set_current_window_by_seq_num()*.

Syntax:

api_set_current_window_by_seq_num(seqNo);

where

seqNo is the window sequence number as the order of when the window was opened.

Example:

```
// Closes the current window instance
api_set_current_window_by_seq_num (1);
de_close_window();
```

check_user_menu()

Five user menus are available in each of the three ADS windows: Main, Schematic, and Layout. This function can be used to determine if a user menu is being used by another application. It returns the current label of the specified user menu. If the label has not be reset by a user or application, the default name will be returned. If a different label is returned, the label has been reset, and the menu is probably in use by another application.

This function takes an integer 1-5 and returns the default names of “User”, “User2”, “User3”, “User4”, or “User5”. These predefined AEL constants may be used instead: deUserMenuName, deUser2MenuName, deUser3MenuName, etc. For a complete example, which uses this function to find the first available user menu, refer to “Creating a Custom Menu” in the Customization manual.

See also: *add_menu(), add_separator(), set_user_menu_label(), clear_user_menu()*.

Syntax:

check_user_menu(menuIndex);

where

menuIndex is an integer 1-5

Example:

```
fputs(stderr, check_user_menu( 1 ));
```

clear_user_menu()

Removes and clears all entries in the user-defined menu. Returns: none.

See also: *add_menu(), add_separator(), set_user_menu_label(), check_user_menu()*.

Syntax:

clear_user_menu();

Example:

The following command clears the second user menu, named “User2”.

```
clear_user_menu(2);
```

de_data_dialog()

Displays a scrollable multiline text editor. Returns: none.

Syntax:

de_data_dialog (title, dataStr, editable, okCB, [userData]);

where

title is the dialog box title.

dataStr is the initial text to display in the dialog box.

editable indicates whether text is editable, where:

0 = text is not editable
1 = text is editable

okCB is the AEL function to execute when OK is pressed.

userData is optional; the user-defined data.

Example:

```
defun register_date_time(newDateTime)
{
  fputs(stderr, newDateTime);
}
de_data_dialog("Date/Time Editor", "Date:12/12/94\n Time: 12:00PM\n", 1,
"register_date_time");
```

de_info()

Displays an information dialog box. Returns: none.

Syntax:

de_info (infoMsg, timeoutFlag);

where

infoMsg is the information to display in the dialog box.

timeoutFlag is the setting to dismiss the dialog box after a timeout interval. The timeout interval is set via DIALOG_TIME_OUT environment variable, where:

0 = do not dismiss automatically
1 = dismiss automatically

Example:

```
de_info("Save completed", 1);
```

de_open_check_rep_dialog()

Opens dialog box for displaying check representation report in the current window. Returns: none.

Syntax:

de_open_check_rep_dialog();

Example:

```
de_open_check_rep_dialog();
```

de_open_hierarchy_dialog()

Opens dialog box to display hierarchy of current design. Returns: none.

Syntax:

de_open_hierarchy_dialog();

Example:

```
de_open_hierarchy_dialog();
```

de_open_info_dialog()

Opens dialog box for displaying information. Returns: none.

Syntax:

de_open_info_dialog();

Example:

```
de_open_info_dialog();
```

de_print_info()

Prints the contents of the following dialog boxes: data, new features, info, hierarchy, editor, DSE schematic status, DSE layout status, and check rep report. Returns: none.

de_print_info(dialogType);

where

dialogType is one of the these dialog types:

DE_DATA DIALOG
DE_NEW_FEATURES_DIALOG
DE_INFO_DIALOG
DE_HIERARCHY_DIALOG
DE_EDITOR2_DIALOG
DE_DSE_SCHEM_STATUS_DIALOG
DE_DSE_LAYOUT_STATUS_DIALOG
DE_REP_REPORT_DIALOG

Example:

```
de_print_info(DE_DATA_DIALOG);
```

de_prompt()

Invokes the prompt dialog box in the current window with user-supplied prompt string and sets a call back AEL function (*aelFuncName*) to be called when the OK button is selected. Dialog box has a single text field in which to type a response to the prompt string. Returns: none.

Syntax:

de_prompt(title, promptLabel, defaultAnswer, okCB, aelFuncName);

where

title is a dialog box title.

promptLabel is a string; label.

defaultAnswer is the initial text displayed in the text field. This string can be set to NULL.

okCB is the OK callback which will override the default OK callback. NULL if not used. This callback function should take three arguments. For example, my_prompt_ok_cb (okH, dlgH, winInst), where:

okH is the handle to the OK button

dlgH is the handle to the prompt dialog

winInst is the current window instance

aelFuncName is a string. AEL function to call when OK is pressed. This function takes 1 string argument which is the response text in the text field. The response text is passed to the AEL function when OK is pressed.

Example:

```
defun my_prompt_cb(input_str)
 {
   fputs(stderr, input_str);
 }
 de_prompt("my prompt", "please enter answer", "yes", NULL, "my_prompt_cb");
```

de_question()

Invokes question dialog box in the current window. Returns: none.

Syntax:

de_question (title, questionStr, yesAelFuncName, noAelFuncName);

where

title is the dialog box title.

questionStr is the question to display in the dialog box.

yesAelFuncName is a string. AEL function to call when yes is selected (should have no arguments).

noAelFuncName is a string. AEL function to call when no is selected (should have no arguments).

Example:

```
defun yes_cb()
{
  fputs(stderr, "Answer yes");
}
defun no_cb()
{
  fputs(stderr, "Answer no");
}
de_question("Save As", "Save File", "yes_cb", "no_cb");
```

install_get_xy()

Installs a user-defined event handler for prompting in the current window. Returns: none.

Syntax:

install_get_xy (aelFuncName[, autoRepeatable, prmStr]);

where

aelFuncName is a string. AEL function to call when (x,y) location is entered by clicking the left mouse button. The (x,y) coordinate of the pointer location is passed in to the AEL function.

autoRepeatable is optional; sets repeat capability, where:

0 = one-shot (default). The routine is not re-installed after the (x,y) location is entered.

1 = auto-repeatable. The event handler will be invoked every time the mouse is clicked until either the End Command button is pressed or another command is invoked.

prmStr is optional; the parameter to override the default event prompt.

Example:

The Example: prints out point each time mouse is clicked.

```
defun my_print_xy(x,y)
{
  fputs(stderr, fmt_token(list (x,y)));
}
install_get_xy("my_print_xy", 1, "enter the (x,y) location");
```

install_get_xy_pair()

Installs a user-defined event handler for prompting in the current window. Returns: none.

Syntax:

install_get_xy_pair (aelFuncName[, autoRepeatable, prm1, prm2, trackingStyle]);

where

aelFuncName is a string. AEL function to call when two points are entered through left mouse clickings. The AEL function receives X1, Y1, X2, Y2 as its parameters.

autoRepeatable is optional; sets repeat capability, where:

0 = one-shot (default). The routine is not re-installed after the (x,y) location is entered.

1 =auto-repeatable. The event handler will be invoked every time the mouse is clicked until either the End Command button is pressed or another command is invoked.

prm1 is optional; the parameter to override the default event prompts. This is the prompt, if defined, for the 1st (x,y).

prm2 is optional; the parameter to override the default event prompts. This is the prompt, if defined, for the 2nd (x,y).

trackingStyle is optional; sets the tracking style, where:

TRACKING_RECT = rectangle rubberbanding lines between 1st and 2nd points.

TRACKING_LINE = (default). straight rubberbanding line between 1st and 2nd points.

TRACKING_NONE = no rubberbanding line between 1st and 2nd points.

Example:

```
defun my_print_2_points(x1,y1,x2,y2)
{
  fputs(stderr, fmt_tokens(list (x1,y1,x2,y2)));
}
install_get_xy_pair("my_print_2_points", 1, "enter the first point", "enter
the second point", TRACKING_RECT);
```

list_select()

Pops up a list selection dialog box. Returns: none.

Syntax:

list_select(title, selectionList, aelFuncName);

where

title is the dialog box title.

selectionList is the list of selections to display in the dialog box.

aelFuncName is a string. AEL function to call when component selected.

Example:

The example prints the string for the selected component after dialog box is displayed and OK is pressed.

```
defun item_selected(item)
{
  fputs(stderr, strcat(item,"was selected"));
}
```

```
list_select("Items",list("one","two","three"),"item_selected");
```

set_user_menu_label()

Sets the label for the user menu. Returns: none.

See also: *add_menu()*, *add_separator()*, *check_user_menu()*, *clear_user_menu()*.

Syntax:

set_user_menu_label(menuLabel, menuName);

where

menuLabel is the name to display on the user menu.

menuName is one of the following internal labels for the user menus: “User”, “User2”, “User3”, “User4”, or “User5”.

Example:

For a complete example, which checks first for an empty slot, refer to “Creating a Custom Menu” in the Customization manual.

```
// changes the label on the 2nd user menu
set_user_menu_label("My Menu", “User2”);
```

Index